Detección estática de propiedades de ejecución de programas concurrentes con *locks* utilizando SACO

Trabajo de fin de grado del Grado en
Doble Grado en Ingeniería Informática - Matemáticas,
Universidad Complutense de Madrid

Facultad de Informática
Departamento de Sistemas Informáticos y Computación

Realizado por:

Alicia Merayo Corcoba

Dirigido por:

Elvira María Albert Albiol
Samir Genaim

Curso 2016-2017

A ti.
A todas esas personas que
solo con las dos primeras palabras
saben que esto va por ellas.

Agradecimientos

En el fondo, son las relaciones con las personas lo que da valor a la vida.
Wilhem von Humboldt

Unas 80 páginas después de que leas esta, estaremos ante el final de una etapa. Cinco años después de llegar a la Universidad, seis después de la PAU o 22 después de que abriese un ojo por primera vez. Se cierra otro ciclo y se abren nuevas posibilidades. Y, como en cada momento, lo bonito de todo esto es que no lo he recorrido yo sola.

Papá, mamá, María, Diego; sois casa. Sois ese lugar seguro al que me ha gustado siempre escaparme, donde parece que lo malo es menos malo y lo bueno es aún mejor. Esto es mío, pero también es vuestro, porque la persona que lo ha escrito es quien es gracias a vosotros.

Tíos y Ana, gracias por darme un trocito de la familia que había dejado en Ponferrada, por hacer que Madrid fuese más casa. Por quererme y apoyarme, y por disfrutar cada logro como si fuese el mejor triunfo.

A todas esas personas que, compartiendo sangre o no, han sido familia. La famila es un vínculo, y muchas veces no tiene por qué conllevar genes. Y a todos los que ahora no están pero me gustaría que estuviesen y les habría gustado estar, os echo de menos.

A Mª Angeles, por haber puesto la semilla para que esta experiencia comenzase. Tú fuiste esa primera ilusión por las matemáticas, y ese empuje que me hacía falta para, después, tomar este camino.

Elvira y Samir, gracias. Por haberme dado esta oportunidad y por haberme ayudado y guiado mientras aprendía y desarrollaba este trabajo. Que esto sea solo el comienzo.

A todos los del aula 16, lo siento pero no voy a seguir la tradición. No voy a decir que me habéis hecho perder el tiempo, porque muchas veces un buen café a tiempo es necesario.

Gabo, Miguel, vosotros habéis conseguido que esto mereciese más la pena. Y todas esas personas con las que he disfrutado estos años, aunque ahora nos veamos menos, también sois un trocito.

Y a ti, Jesús. No te dejo para el final porque me importes menos, sino todo lo contrario. Quería que este agradecimiento tuviese algo simbólico, empezamos esto juntos y hemos conseguido terminarlo. E, igual que hemos cerrado la carrera, podemos cerrar estos

agradecimientos. A partir de ahora, todo es nuevo, pero eso no tiene por qué dar miedo. Si hemos podido con un doble grado, podemos con todo. Ya sabes: “salta, y deja que te crezcan las alas en el camino hacia el suelo”.

Índice general

Resumen vii

Summary ix

Palabras clave xi

Keywords xiii

1. Introducción 1

2. El lenguaje imperativo 11
- 2.1. Tipos 11
- 2.2. Identificadores 11
- 2.3. Asignación a enteros 12
- 2.4. Instrucciones 12
- 2.5. Estructura de un programa 14
- 2.6. Semántica 15
- 2.7. Noción de deadlock 18

3. El lenguaje ABS 21
- 3.1. Diseño en ABS 21
- 3.2. Tipos 22
- 3.3. Sincronización en ABS 22
- 3.4. Sintaxis 23
- 3.5. Semántica operacional 24
- 3.6. Deadlock 27
  - 3.6.1. Análisis de deadlock 27

4. Traducción 29
- 4.1. Visión global de la traducción 29
  - 4.1.1. Variables globales 30

4.1.2. Mútex 31
4.1.3. Procesos 32
4.1.4. Main 33
4.2. Función de traducción 33
4.3. Equivalencia semántica entre sentencias en el lenguaje imperativo y sentencias en ABS 36

5. Implementación **55**
5.1. Traducción 55
5.2. Estructuras y campos auxiliares 56
5.3. Ejemplos 58
5.4. Uso de la herramienta 62

6. Conclusiones y resultados **65**

Resumen

Actualmente, la ejecución concurrente es una de las bases de la informática. Este tipo de ejecución ha llevado a tener que resolver problemas complejos como la sincronización de las variables compartidas entre programas, para evitar las *carreras* de datos. Las carreras de datos ocurren cuando varios procesos acceden simultáneamente a datos globales, y como consecuencia de los accesos simultáneos, el estado de las variables no es el esperado (o el correcto), es decir, los datos quedan "corrompidos".

Refiriéndonos a estas partes que queremos proteger como *secciones críticas* de nuestros programas, se han desarrollado múltiples mecanismos para poder ejecutarlas sin riesgo. El modelo más usado para esta protección de los datos es el uso en programas en lenguajes imperativos de primitivas *lock* y *unlock* para realizar la entrada y salida de la sección crítica, protegida por diferentes *cerrojos* o *mútex*.

Sin embargo, aunque este modelo proporciona un mecanismo para evitar la corrupción de los datos, un uso incorrecto de los cerrojos no garantiza que no exista corrupción, ni tampoco ausencia de bloqueos (o *deadlocks*).

Llegamos así al problema del *deadlock*, que sucede cuando varios programas están esperando a recursos que mantienen otros programas, de forma que ninguno puede avanzar. Este problema ha sido estudiado en numerosos lenguajes, muchos basados en paso de mensajes asíncrono, intentando encontrar un método para saber si un programa va o no a bloquearse.

Un ejemplo de esto es ABS, un lenguaje basado en actores en el que se han desarrollado herramientas de análisis estático para detectar deadlock en programas basados en paso de mensajes y que se encuentran a nuestra disposición.

La motivación de este trabajo es unir el modelo de concurrencia basado en lenguajes imperativos y *locks* y el uso de las herramientas de ABS en programas asíncronos basados en paso de mensajes para, así, beneficiarnos de ambos aspectos. Para ello, se plantea como objetivo implementar una traducción de programas en lenguaje imperativo que utilizan mútex a programas escritos en ABS, desarrollando un modelo apropiado que represente los cerrojos y demostrando la equivalencia entre ambos.

Por ello, en el trabajo se desarrollan los siguientes puntos:

- Planteamiento del mecanismo equivalente a los locks en ABS.

- Desarrollo de un lenguaje imperativo genérico en el que poder implementar los programas que vamos a traducir. Se desarrollarán las posibles instrucciones, la estruc-

tura que deberá tener el programa, la semántica de las instrucciones, y la noción de deadlock en este lenguaje.

- Introducción del lenguaje ABS, su semántica y el análisis de deadlock.
- Traducción de las instrucciones, mediante la definición de la función apropiada. Demostraremos formalmente, utilizando mecanismos matemáticos, tanto la equivalencia de instrucciones en ambos lenguajes como la implicación de que si en el lenguaje imperativo hay deadlock, entonces lo habrá en ABS.
- Implementación de la traducción mediante estructuras adecuadas y traducción de ejemplos utilizando la herramienta desarrollada.
- Conclusiones y resultados del trabajo.

Summary

Nowadays, concurrent computing is one of the bases of computer science. Despite of its important benefits, this type of execution forced us to seek solutions for related complex problems, such as synchronization of accesses to shared resources in order to avoid *data races*. Data races occur when various processes simultaneously access global variables, which might result in unexpected results, e.g., the corresponding data remains corrupted.

Calling these parts of the program that we want protect (to avoid accessing global data simultaneously) as *critical sections*, there are many techniques that can be used to execute them without any risk. The most used model to achieve such protection in imperative programming languages is based on the use of the *lock* and *unlock* primitives to enter and exit critical sections that are protected by different *locks* and *mutex*.

However, even though this model provides a mechanism to avoid data races, and thus avoiding corruption of data, an incorrect use of it does not guarantee absence of data races and might lead to *deadlocks*.

This lead us to the *deadlock* problem, which occurs when two or more processes are waiting to obtain resources held by others without the ability to make progress in any of them. This problem has been studied in different programming languages, many of them are based on asynchronous message passing, trying to develop algorithms (a.k.a., analysis) to check if a program can reach a deadlock state. An example of such language is ABS, an actor-based language for which several deadlock analysis tools have been developed, and, moreover, they are publicly available.

The motivation of this work is to use of existing deadlock analysis tools of ABS, to analyze the deadlock behavior of imperative programs that are based on the use of locks, and thus taking advantage of these tools instead of developing them from scratch for the imperative language. This is challenging both settings are based on different concurrency models.

Our approach is based on translating imperative programs with locks to ABS programs, while keeping the deadlock behavior. The challenge is how to mode locks in an appropriate way in ABS such that the corresponding analysis detect, or prove absence of, deadlock behavior. To achieve this, in this work we have done the following:

- Development of a mechanism to model locks in ABS.

- Definition of a simple generic imperative language with locks, in order to implement programs that we want to translate to ABS. We formally define the syntax,

semantics, and the notion of deadlock in this language.

- Overview of the ABS language: its syntax, semantics, the notion of deadlock, and corresponding deadlock analyses.
- Definition of a function to translate programs in our imperative language to corresponding ABS programs, and formally proving, using mathematical techniques, the equivalence of these programs in particular when considering the deadlock property, i.e., if there is an execution in the imperative program that leads to a deadlock then there a corresponding execution in ABS that leads to a deadlock.
- Implementation of the translation and its application to several examples.
- Conclusions and results of this work.

Palabras clave

- ABS
- Lock
- Unlock
- Deadlock
- Bloqueo de procesos
- Equivalencia semántica
- Concurrencia
- Paralelismo
- Cerrojos
- Modelos de concurrencia basados en actores

Keywords

- ABS
- Lock
- Unlock
- Deadlock
- Process blocking
- Semantic equivalence
- Concurrency
- Parallelism
- Locks
- Actor-based concurrency model

CAPÍTULO 1

Introducción

Supongamos que tenemos el proceso *O1* y el mútex *m1*, con un esquema de código en un lenguaje imperativo como el que sigue:

```
process O1{
    NoSC 1
  lock(m1);
    SC
  unlock(m1);
    NoSC 2
}
```

Este mecanismo basado en *lock/unlock* es uno de los mecanismos de concurrencia más utilizados. Con él, podemos proteger los datos en el acceso concurrente de forma que sólo un proceso los pueda usar a la vez, evitando así el acceso simúltaneo y por tanto, su corrupción. Un uso correcto de los cerrojos garantiza la protección de los datos, pero introduce un nuevo problema: el bloqueo o deadlock.

Nuestro proyecto está motivado por la existencia de un analizador de deadlock desarrollado para el lenguaje ABS basado en actores. En este lenguaje se han desarrollado de hecho varias herramientas destinadas a detectar situaciones de deadlock o bloqueo mediante análisis estáticos. Sin embargo, de lo que no disponemos en este caso es del mecanismo de concurrencia de *lock/unlock*, ya que en ABS no existen estas primitivas.

En ABS, el modelo está basado en invariantes de monitores. Es decir, cada objeto concurrente (que es así como se llama a las unidades de ejecución en este lenguaje) tiene un lock implícito. En cada uno de estos objetos, tendremos métodos declarados y múltiples tareas que intentan ejecutar estos métodos. Sólo una de estas tareas podrá ejecutarse a la vez en el objeto, por lo que a nivel local sí que tendremos protección de los campos de los objetos. El problema viene cuando queremos una protección de datos entre procesos, ya que cada objeto concurrente funciona como una unidad independiente.

La primera idea que podríamos plantear en ABS sería utilizar uno de estos objetos

concurrentes para realizar la función que, en los programas concurrentes, les corresponde a los mútex. Si sólo podemos ejecutar una tarea a la vez en cada objeto concurrente, podemos verlo también como que sólo un proceso externo puede ejecutar a la vez un método de la codificación del mútex. Es decir, haciendo un símil con el lenguaje imperativo, tenemos que sólo un proceso puede adquirir el mútex a la vez, mientras el resto esperan. La espera de los procesos al mútex también la tendremos mediante esta codificación, ya que en cada objeto concurrente habrá a lo sumo una tarea ejecutándose a la vez mientras el resto esperan en una cola para adquirir el lock del objeto. Será en esta cola donde esperen nuestros múltiples procesos a ejecutar el método/adquirir el mútex. Cuando uno de los procesos libere el lock del objeto, será porque ha terminado de ejecutarse o lo ha soltado voluntariamente, ya que no tenemos una ejecución especulativa, por lo que no tendremos problemas de que otro proceso le quite el mútex. Al ser liberado el lock, se observa la cola de tareas y se escoge una entre todas para ejecutarse. No podemos escoger qué tarea se va a ejecutar, es una decisión no determinista, pero esto no supondrá un problema con respecto al modelo *lock/unlock* debido a que en un lenguaje imperativo tampoco tenemos asegurado a priori que el mútex vaya a ser asignado en orden de intento de adquisición por parte de los procesos.

Por tanto, tenemos un primer código de aproximación a los mútex y los objetos concurrentes de la forma:

```
1  class M1(){
2    Unit p1(Parametros){
3      // Codigo p1;
4    }
5  }
6
7  class O1(M1 m1){
8    Unit main(){
9      NoSC 1
10     m1!p1(Argumentos);
11     NoSC 2
12   }
13 }
```

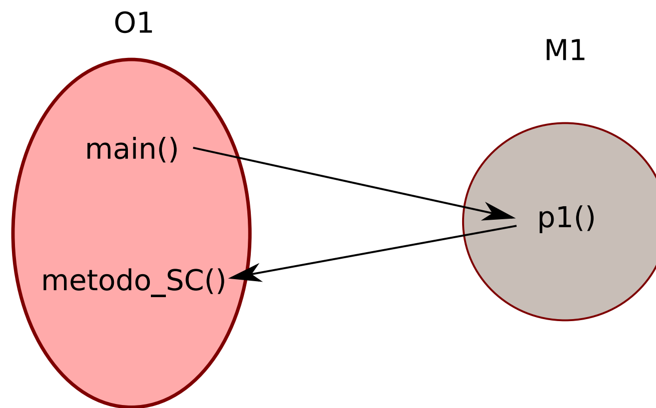
Las partes que no están protegidas por mútex tienen una traducción directa, en el método principal del objeto; tenemos que ver ahora qué va a englobar el código del método *p1* del mútex. La aproximación más lógica es que tengamos un método para cada posible llamada en el objeto del mútex que contenga la codificación de la sección crítica correspondiente. Sin embargo, esta versión da problemas de implementación y escalabilidad, ya que necesitaríamos pasar a cada método del mútex las variables y el estado del proceso *O1* con lo cual obtendríamos, para empezar, una lista de parámetros/argumentos sin longitud fija.

Dado que esta primera opción parece no ser viable, vamos a intentar la segunda aproximación codificando la parte de la sección crítica en un segundo método del proceso *O1*. De esta forma no tendremos que pasar el estado del proceso ni las variables a otro objeto concurrente. Sin embargo, tenemos el problema de la llamada a este método manteniendo

el invariante de los mútex cuando hay más de un proceso que los puede utilizar. Llegamos así al estudio y uso de las dos instrucciones de sincronización más comunes en ABS: *await/get*, basadas en variables asignadas a la ejecución de procesos (que llamaremos futuras) y que sólo se resuelven cuando el proceso al que están asignadas se termina de ejecutar. En el caso de *await*, liberamos el lock implítico del objeto mientras se resuelve la variable futura; en la instrucción *get*, por el contrario, esperamos a que la variable futura se resuelva manteniendo mientras tanto el lock.

Lo primero que hay que remarcar en esta versión es que el uso de alguna de estas instrucciones es necesario para que el flujo de ejecución no continúe, ya que en ABS las llamadas que realizaremos a los métodos serán en su mayoría asíncronas, por lo que de no poner estas instrucciones se lanzaría la tarea correspondiente al método de la llamada y se continuaría con el flujo del programa sin esperar a que termine de ejecutarse el método llamado.

Tenemos, por tanto, un esquema de objetos concurrentes y llamadas de la forma



donde, para que *M1* pueda realizar la llamada a *metodo_SC*, necesita conocer el objeto *O1*. Esto se puede hacer pasando *this* desde *main* al hacer la llamada a *p1*, necesitando entonces *p1* tener un parámetro de tipo *O1*. Observamos que, como la sección crítica ha sido codificada en el propio proceso *O1*, el método de *M1* sólo necesita conocer el objeto al que tiene que llamar, por lo que este será el único parámetro.

Como hemos explicado anteriormente con los procesos, sólo podremos ejecutar una tarea a la vez en cada objeto concurrente, por lo que en particular para que se ejecute *metodo_SC* en *O1* necesitamos que la tarea que ejecuta *main* haya soltado el lock mientras no se resuelve la llamada a *p1*, pero para esto basta usar una instrucción *await*. Por otra parte, en el caso del mútex no podemos soltar el lock hasta que se haya terminado de ejecutar el código asociado a la traducción de la sección crítica, por lo que después de la llamada asíncrona necesitaremos indicar al objeto concurrente *M1* que tiene que esperar manteniendo el lock del objeto, para lo que bastará usar una instrucción *get*.

Tenemos entonces un esquema de la forma

```
1 class O1(M1 m1){
2   Unit main(){
3     NoSC 1
4     await m1!p1()?;
5     NoSC 2
```

```
6     }
7     Unit codigo_SC(){
8       SC
9     }
10  }
11
12  class M1(){
13    Unit p1(O1 o){
14    get o!codigo_SC();
15    }
16  }
```

Tenemos, por tanto, un objeto concurrente para la codificación de cada proceso y un objeto concurrente para la codificación de cada mútex. De esta forma, podemos usar el lock implícito del objeto del mútex para controlar la concurrencia y así permitir más de un objeto que use el mismo mútex, y codificar la sección crítica en el objeto que tiene la traducción correspondiente al código del proceso, pudiendo acceder a ella luego debido a que se ha soltado el lock implícito del objeto al realizar la llamada al mútex.

Lo único que nos faltaría por ver es cómo resolver los nombres de los métodos asignados a cada llamada, ya que para unos pocos mútex no tener una construcción genérica de los nombres de los objetos se convierte en una opción inviable. Por ello, supongamos que a cada aparición de lock a un mútex en el lenguaje imperativo en un proceso le asignamos su orden de aparición. En ese caso, podríamos llamar al método del mútex de la forma m_n, siendo n el número de la aparición del *lock*, y al método del objeto que codifica el proceso $m1_n$ suponiendo que el mútex se llame *m1*. Además, a la hora de tener varios procesos y varios mútex, para poder hacer más genérica la construcción de las clases de los mútex, vamos a introducir los "tipos" de las clases en ABS, que vienen dados por interfaces que implementan las distintas clases.

Esta última idea, sin introducir el código correspondiente a los interfaces, que se explicarán más adelante, se traduce en

```
1   class O1(MutexInterface m1) implements ProcessInterface{
2     Unit main(){
3       NoSC 1
4       await m1!m_1(this) ?;
5       NoSC 2
6     }
7
8     Unit m1_1(){
9       SC
10    }
11  }
12
13  class M1() implements MutexInterface{
14    Unit m_1(ProcessInterface o){
15      get o!m1_1();
16    }
```

```
}
```

Con esta idea de implementación, vamos a ver distintos casos que podrían codificarse de esta forma. Además, tendremos que introducir una condición sobre los locks para que se puedan traducir a ABS mediante este método, que llamaremos *buen orden*.

Definición 1.1 (Buen orden).
Definiremos, para empezar, un operador $<<$, de forma que $a << b$ indica que el evento a se produce antes en el tiempo que el evento b.

Diremos que un proceso P mantiene el *buen orden* si para todos los mútex m_i, m_j usados en P se cumple que, si $lock(m_i) << lock(m_j)$, entonces se da una de las dos siguientes posibilidades:

- $unlock(m_i) << lock(m_j)$
- $unlock(m_j) << unlock(m_i)$

Podríamos traducir la definición anterior en "para todos m_i, m_j se tiene que cumplir que, si cogemos el mútex m_i antes que el mútex m_j, o bien soltamos m_i antes de coger m_j, o bien soltamos m_j antes de soltar m_i.

Observación 1.1.

1. Necesidad del buen orden: permite crear bloques para separar el código en función de *lock/unlock*, de forma que podemos dividir la traducción del código en múltiples métodos.
2. NO tendremos buen orden en P si existen dos mútex m_i, m_j tales que

$$lock(m_i) << lock(m_j) << unlock(m_i) << unlock(m_j)$$

   Necesitamos este buen orden para garantizar que los bloques que traducimos sean "atómicos", en el sentido de que se puedan traducir dentro del código de un procedimiento y no haya que intercalarlos.

Observación 1.2. Aunque puede parecer que imponemos demasiadas restricciones a los programas mediante la condición de buen orden, en realidad nos estamos adaptando al comportamiento de otros métodos de sincronización en lenguajes conocidos, como es el caso de *synchronized* en Java.

Ejemplo 1.1 (Dos procesos, dos locks)**.**
Supongamos, en primer lugar, que tenemos el caso más básico: dos procesos y dos locks distintos, de forma que los procesos cogen los locks alternativamente:

```
process O1{
    NoSC 1.1
  lock(m1);
    SC 1.1
  unlock(m1);
    NoSC 1.2
```

```
7     lock(m2);
8       SC 1.2
9     unlock(m2);
10      NoSC 1.3
11  }
12
13  process O2{
14      NoSC 2.1
15    lock(m2);
16      SC 2.1
17    unlock(m2);
18      NoSC 2.2
19    lock(m1);
20      SC 2.2
21    unlock(m1);
22      NoSC 2.3
23  }
```

Vamos a transformar ahora este código a un código que siga el esquema de ABS. Además de lo visto antes, vamos a tener un *main* principal en ABS que nos sirva para iniciar las llamadas a los procesos.

Empezamos la implementacion desde los mútex, pasando por los procesos y llegando al *main* principal.

1. Mútex:

```
1  class M1() implements MutexInterface{
2    Unit m_1(ProcessInterface o){
3      get o!m1_1();
4    }
5  }
6
7  class M2() implements MutexInterface{
8    Unit m_1(ProcessInterface o){
9      get o!m2_1();
10   }
11 }
```

Como hay una única llamada al mútex en cada proceso, sólo necesitamos un método en cada objeto concurrente, ya que el código correspondiente a cada mútex en cada objeto estará implementado en los métodos correspondientes de los objetos concurrentes correspondientes a la traducción de cada proceso.

2. Procesos:

```
1  class O1(MutexInterface m1, MutexInterface m2)implements
      ProcessInterface{
2    Unit main(){
```

```
3      NoSC 1.1
4      await m1!m1_1(this) ?;
5      NoSC 1.2
6      await m2!m2_1(this) ?;
7      NoSC 1.3
8    }
9    Unit m1_1(){
10     SC 1.1
11   }
12   Unit m2_1(){
13     SC 1.2
14   }
15 }
16
17 class O2(MutexInterface m1, MutexInterface m2) implements
     ProcessInterface{
18   Unit main(){
19     NoSC 2.1
20     await m2!m2_1(this) ?;
21     NoSC 2.2
22     await m1!m1_1(this) ?;
23     NoSC 2.3
24   }
25   Unit m1_1(){
26     SC 2.2
27   }
28   Unit m2_1(){
29     SC 2.1
30   }
31 }
```

3. main():

```
1 {
2   MutexInterface m1 = new M1();
3   MutexInterface m2 = new M2();
4   ProcessInterface o1 = new O1(m1,m2);
5   o1!main();
6   ProcessInterface o2 = new O2(m1,m2);
7   o2!main();
8 }
```

Ejemplo 1.2 (Locks en buen orden intercalados, y llamada más de una vez a un lock en un proceso).

```
1 process O1{
2   NoSC 1.1
```

```
3    lock(m1);
4      SC 1.1:
5        Cod 1.1
6        lock(m2);
7        SC 1.2
8        unlock(m2);
9        Cod 1.2
10   unlock(m1);
11   NoSC 1.2
12 }
13
14 process O2{
15   NoSC 2.1
16   lock(m2);
17   SC 2.1
18   unlock(m2);
19   NoSC 2.2
20   lock(m2);
21   SC 2.2
22   unlock(m2);
23   NoSC 2.3
24 }
```

En el caso del código del proceso $O1$, se ha remarcado mediante un espacio el código que correspondería a la sección crítica *SC 1.1*, que se divide en múltiples sentencias. De esta forma, se observan los bloques que se derivan del buen orden y que permiten la traducción a ABS mediante el método desarrollado, que se verá en la traducción de los procesos.

Mostramos la traducción a código ABS, con el mismo orden usado en el ejemplo anterior:

1. Mútex:

```
1  class M1() implements MutexInterface{
2    Unit m_1(ProcessInterface o){
3      get o!m1_1();
4    }
5  }
6
7  class M2() implements MutexInterface{
8    Unit m_1(ProcessInterface o){
9      get o!m2_1();
10   }
11   Unit m_2(ProcessInterface o){
12     get o!m2_2();
13   }
14 }
```

En el caso de $m2$, necesitamos dos métodos en el objeto concurrente del segundo

proceso, ya que hay dos llamadas distintas a este mútex en el proceso.

2. Procesos:

```
class O1(MutexInterface m1, MutexInterface m2) implements
    ProcessInterface{
  Unit main(){
    NoSC 1.1
    await m1!m_1(this) ?;
    NoSC 1.2
  }
  Unit m1_1(){
    Cod 1.1
    await m2!m_1(this) ?;
    Cod 1.2
  }
  Unit m2_1(){
    SC 1.2
  }
}

class O2(MutexInterface m2) implements ProcessInterface{
  Unit main(){
    NoSC 2.1
    await m2!m_1(this) ?;
    NoSC 2.2
    await m2!m_2(this) ?;
    NoSC 2.3
  }
  Unit m2_1(){
    SC 2.1
  }
  Unit m2_2(){
    SC 2.2
  }
}
```

3. main():

```
{
  MutexInterface m1 = new M1();
  MutexInterface m2 = new M2();
  ProcessInterface o1 = new O1(m1,m2);
  o1!main();
  ProcessInterface o2 = new O2(m2);
  o2!main();
}
```

Nos planteamos como objetivo entonces encontrar una traducción que nos dé un programa ABS equivalente al que tenemos, para poder codificar los ejemplos de lenguajes imperativos y comprobar si tendrán o no deadlock, así como demostrar formalmente (utilizando herramientas de razonamiento matemáticas) la corrección de la traducción.

Para ello, primero se desarrollará un lenguaje imperativo genérico y se explicarán las bases de ABS. Después, desarrollaremos una función de traducción y demostraremos la equivalencia entre ambos programas, viendo que si uno se bloquea entonces el otro también lo hará. Para finalizar, realizaremos una implementación de la traducción y obtendremos las conclusiones del trabajo.

CAPÍTULO 2

El lenguaje imperativo

Para poder implementar una transformación genérica del lenguaje, vamos a desarrollar un lenguaje imperativo con una estructura fija. De esta forma, llegaremos a un convenio para que todos los programas estén estructurados de la misma forma.

2.1. Tipos

Los tipos aceptados por este lenguaje son:

- Int: el tipo base serán los enteros.
- Mutex: tipo de los locks.

$$\text{Tipo} ::= \text{Int} \mid \text{Mutex}$$

Las variables del lenguaje son las que aceptarán tipos. Los procesos escritos fuera del cuerpo principal pueden recibir parámetros y modificarlos, pero no devuelven ningún tipo.

2.2. Identificadores

Vamos a distinguir cuatro tipos distintos de identificadores, para hacer más legible la lectura de los programas. Estos tipos pueden tener después del símbolo inicial un número variable de letras minúsculas, mayúsculas y guiones bajos.

Los identificadores y símbolos de comienzo son:

- *Variables locales*: comienzan por letra minúscula.
- *Variables globales*: comienzan por un guión bajo.
- *Nombres de procesos*: comienzan por letra mayúscula.

- *Identificadores de procesos*: identifican unívocamente el proceso que se está ejecutando, comienzan por .

$$
\begin{aligned}
\text{Minuscula} &::= a|\ldots|z \\
\text{Mayuscula} &::= A|\ldots|Z \\
\text{Letra} &::= \text{Mayuscula} \mid \text{Minuscula} \\
\text{Digito} &::= 0|1|\ldots|9 \\
\text{Cualquiera} &::= \text{Minuscula} \mid \text{Mayuscula} \mid _ \\
\\
\text{IdenVarLocal} &::= \text{Minuscula}\ \{\ \text{Cualquiera}\ \} \\
\text{IdenVarGlobal} &::= _\{\ \text{Cualquiera}\ \} \\
\text{NomProceso} &::= \text{Mayuscula}\ \{\ \text{Cualquiera}\ \} \\
\text{IdenProceso} &::=\ .\{\ \text{Cualquiera}\ \}
\end{aligned}
$$

2.3. Asignación a enteros

El único tipo asignable que tenemos son los enteros, ya que los mútex sólo reciben operaciones *lock* y *unlock*.

Como la funcionalidad de nuestro programa es reflejar situaciones de deadlock, vamos a considerar que los valores de las variables enteras vienen dados por una expresión *AExpr* sin especificar. Esto no supondrá una gran pérdida de precisión, ya que se está desarrollando un lenguaje imperativo destinado a realizar una traducción a ABS a la que, posteriormente, se le aplicará un análisis estático. En este análisis estático, el valor de la expresión a la derecha no se conoce de antemano.

2.4. Instrucciones

Tenemos diferentes instrucciones que podremos utilizar:

- *Declaración de variables.*

  Declaramos las variables sin inicialización, se les asignará un valor por defecto. Los mutex serán globales a todos los procesos.

$$
\begin{aligned}
\text{DeclVar} ::= {} & \text{Tipo IdenVarLocal;} \mid \\
& \text{Tipo IdenVarGlobal;} \mid \\
& \text{NomProceso IdenProceso;}
\end{aligned}
$$

- *Asignación de variables.*

$$
\begin{aligned}
\text{AsignVar} ::= {} & \text{IdenVarLocal} = \text{Valor;} \mid \\
& \text{IdenVarGlobal} = \text{Valor;}
\end{aligned}
$$

- *Skip*

  Es una instrucción que no hace nada, pero que usaremos para las instrucciones que en lenguajes como *C++* o *Java* se representan con ;.

  Skip ::= skip;

- *If/else.*

  Vamos a usar la producción *Cuerpo*, que se definirá al final de esta sección y que englobará a todas las instrucciones definidas. Como sólo tenemos variables de tipo entero, usaremos *Aexpr* en la condición booleana.

  IfElse ::= if (Aexpr) \{ Cuerpo \} else \{ Cuerpo \}

- *While.*

  While ::= while (Aexpr) \{ Cuerpo \}

- *Lock/unlock*

  Para manipular los mútex, vamos a usar las instrucciones *lock* para coger el mútex y *unlock* para liberarlo. Los mútex tienen asignado un identificador de tipo *IdenVarLocal*. Se presupone que el programa que han pasado al parser tiene los tipos comprobados y este identificador se corresponde, en efecto, con el identificador de una variable de tipo *Mutex*.

  Lock ::= lock(IdenVarLocal);
  Unlock ::= unlock(IdenVarLocal);

- *New.*

  Esta instrucción se usa para crear nuevos procesos, en caso que se hayan definido aparte del cuerpo principal del programa. Los procesos se crean de manera similar a los threads de los procesos concurrentes. Podemos pasar argumentos en la llamada, en caso de que los procesos admitan parámetros en su declaración.

  Los argumentos de los parámetros son identificadores de variables, no se permite el paso de constantes. Además, estas variables serán de tipo local, ya que las globales son accesibles por todos los procesos sin necesidad de pasarlas. Los procesos en el *new* se asignan a un identificador de proceso, y podremos crearlos únicamente en el Main del programa, para evitar conflictos de nombres al crear procesos dentro de procesos y asignarlos a identificadores.

  Args ::= [IdenVarLocal {, IdenVarLocal }]

  New ::= IdenProceso = new NomProceso (Args);

- *Join.*

  Esta instrucción se usa para indicar a un proceso que espere a que otro acabe. Para poder hacer join sobre un proceso, *IdenProceso* debe estar previamente declarado y asignado a un proceso.

  Join ::= join (IdenProceso);

Y con esta definición de las instrucciones podemos definir la siguiente producción, que las engloba a todas:

Instruccion ::= DeclVar | AsignVar | IfElse | While |
Lock | Unlock | New | Join

Una secuencia (que puede ser vacía) de instrucciones viene específicada por

Cuerpo ::= {Instruccion}

2.5. Estructura de un programa

Un programa escrito en nuestro lenguaje imperativo tendrá primero una declaración del cuerpo principal y después las declaraciones de los procesos a los que llama en el programa.

```
//cuerpo principal
Main(){
  ...
}

//Declaraciones de procesos
process P(Params){
  ...
}
...
```

Definimos la declaración de parámetros como

Params ::= [TipoBase IdenVarLocal {, TipoBase IdenVarLocal }]

Con esto, la definición del *Main* y los procesos sigue las producciones

Main ::= Main () \{ Cuerpo \}
Process ::= process (Params) \{ Cuerpo \}

Por tanto, para acabar, tenemos que la estructura de nuestro programa es:

Programa ::= Main {Process}

2.6. Semántica

Para definir la semántica de nuestro lenguaje, vamos a definir primero una representación del estado del programa. En esta representación, guardaremos el estado global del programa, referido en particular al proceso en el que nos encontremos (*Main* u otros procesos).

Necesitamos guardar la información de:

- En qué proceso estamos ejecutando: $.p$.

- Qué procesos están activos en el programa. En P guardamos el conjunto de identificadores de procesos activos, definiendo las siguientes funciones auxiliares:

  - *isActive*, para saber si el proceso al que identifica $.p$ está activo

  $$isActive(.p) = \begin{cases} \text{true} & \text{si } .p \in P \\ \text{false} & \text{en caso contrario} \end{cases}$$

  - *quit*, para eliminar el proceso $.p$ del conjunto de procesos activos.

  $$quit(.p, P) \equiv P = P \setminus \{.p\}$$

  Devuelve el conjunto P modificado.

  - *add*, para añadir el proceso $.p$ al conjunto de procesos activos.

  $$add(.p, P) \equiv P = P \cup \{.p\}$$

  Devuelve el conjunto P modificado.

  A no ser que el programa haya acabado de ejecutarse, este conjunto nunca será vacío, ya que *Main* siempre estará en él.

- Qué mútex tenemos definidos en nuestro programa y su estado: M. Esta asignación de mútex, se representará en la semántica como un conjunto de variables, de forma que

  - $M[m] = .p$ si el mútex m está cogido por el proceso $.p$.
  - $M[m] = null$ si el mútex m está libre.

- Qué procesos hay esperando a cada mútex: W. Si m es un mútex, entonces $W[m]$ devolverá un conjunto de identificadores de procesos que están esperando para coger el mútex. Para controlar este conjunto, tendremos los métodos

  - *wait(m, .p, W)* para añadir al conjunto $W[m]$ el identificador $.p$. Devuelve el conjunto W modificado.
  - *select(m, W)* devuelve un identificador de proceso que esté esperando al mútex m o *null* en caso de que no haya ninguno esperando.

- *quitProc(m, .p, W)* elimina el identificador $.p$ del conjunto asociado a $W[m]$. Devuelve W modificado.

- Qué variables de tipo entero tenemos. En este caso tenemos que distinguir dos casos:
  - Variables globales: denotamos este conjunto por G.
  - Variables locales: denotamos este conjunto por L.
- Flujo de instrucciones que estamos ejecutando en el proceso en el que estemos, s.

Para hacer una representación más clara del estado del objeto, vamos a representar por separado las variables globales, que englobarán las variables enteras globales, los mútex, los procesos esperando por un mútex y los procesos activos, y el estado del propio proceso que se está ejecutando en la instrucción. Un estado de un programa en nuestro lenguaje imperativo vendrá dado por

$$\underbrace{[\ \mathrm{gl}\,(M, W, G, P)]}_{\text{vars. globales}} \quad \underbrace{\mathrm{ip}\,(.p, L, \{s\})}_{\text{información del proceso}}$$

de forma que, en los casos de la semántica en que no intervengan las variables globales, no hará falta representarlo.

Para las instrucciones definidas antes, definimos la siguiente semántica:

1. *Asignación*

   Vamos a definir Aexp como el tipo de las expresiones aritméticas que dan lugar a las asignaciones de los enteros en nuestro programa. Entonces, definimos la función $\mathcal{A}$ de la siguiente forma:

   $$\mathcal{A} : \mathrm{Aexp} \to \mathbb{Z}$$

   Distinguimos entonces entre la asignación en el caso de que x sea una variable local del proceso en el que estemos o que sea global a todo el programa:

   $$\left[\ \mathrm{assign}\ ^{L}\right] : \frac{x \in \mathrm{dom}(L)}{\mathrm{ip}\,(.p, L, \{x = a; s\}) \to\ \mathrm{ip}\,(.p, L\,[x \to \mathcal{A}\,[\![a]\!]]\,, \{s\})}$$

   $$\left[\ \mathrm{assign}\ ^{G}\right] : \frac{x \in \mathrm{dom}(G)}{\begin{array}{l}\mathrm{gl}\,(M, W, G, P)\ \ \mathrm{ip}\,(.p, L, \{x = a; s\}) \to \\ \qquad\qquad \mathrm{gl}\,(M, W, G\,[x \to \mathcal{A}\,[\![a]\!]]\,, P)\ \ \mathrm{ip}\,(.p, L, \{s\})\end{array}}$$

2. *Skip*

   $$\mathrm{ip}\,(.p, L, \{\ \mathrm{skip}\ ; s\}) \to\ \mathrm{ip}\,(.p, L, \{s\})$$

3. *If/else*

   Como en el lenguaje imperativo el único tipo que tenemos aparte de los mútex son los enteros, definimos primero la siguiente función:

   $$\mathcal{B} : \mathrm{Aexp} \to \{\ \mathrm{true}\ ,\ \mathrm{false}\ \}$$

de forma que

$$\mathcal{B}[\![x]\!] = \begin{cases} \text{true} & \text{si } \mathcal{A}[\![x]\!] \neq 0 \\ \text{false} & \text{si } \mathcal{A}[\![x]\!] = 0 \end{cases}$$

Con esta función, tenemos las siguientes reglas semánticas:

$$\left[\text{ if }^{\text{true}}\right] : \frac{\mathcal{B}[\![b]\!]}{\text{ip}\,(.p, L, \{\text{ if } b \text{ then } s_1 \text{ else } s_2; s\}) \rightarrow \text{ ip}\,(.p, L, \{s_1; s\})}$$

$$\left[\text{ if }^{\text{false}}\right] : \frac{\neg\mathcal{B}[\![b]\!]}{\text{ip}\,(.p, L, \{\text{ if } b \text{ then } s_1 \text{ else } s_2; s\}) \rightarrow \text{ ip}\,(.p, L, \{s_2; s\})}$$

4. *While*

$$\left[\text{ while }^{\text{true}}\right] : \frac{\mathcal{B}[\![b]\!]}{\text{ip}\,(.p, L, \{\text{ while } b \text{ do } s_1; s\}) \rightarrow \text{ ip}\,(.p, L, \{s_1;\text{ while } b \text{ do } s_1; s\})}$$

$$\left[\text{ while }^{\text{false}}\right] : \frac{\neg\mathcal{B}[\![b]\!]}{\text{ip}\,(.p, L, \{\text{ while } b \text{ do } s_1; s\}) \rightarrow \text{ ip}\,(.p, L, \{s\})}$$

5. *Lock/unlock*

   Para implementar la semántica operacional de estas dos instrucciones, vamos a considerar que se efectúan sobre un mútex m.

   - *Lock*: vamos a denotar las reglas por $\left[\text{ lock}^{\text{ true}}\right]$ si el lock está libre, y por $\left[\textit{lock}^{\text{ false}}\right]$ si está cogido.

$$\left[\text{ lock }^{\text{true}}\right] : \frac{M[m] = \text{null}}{\begin{array}{l}\text{gl}\,(M, W, G, P)\ \text{ ip}\,(.p, L, \{\text{ lock}(m); s\}) \\ \qquad \rightarrow \text{ gl}\,(M[m \mapsto .p], W, G, P)\ \text{ ip}\,(.p, L, \{s\})\end{array}}$$

$$\left[\text{ lock }^{\text{false}}\right] : \frac{M[m] \neq \text{null}}{\begin{array}{l}\text{gl}\,(M, W, G, P)\ \text{ ip}\,(.p, L, \{\text{ lock}(m); s\}) \rightarrow \\ \qquad \text{gl}\,(M, \text{wait(m,.p,W)}, G, P)\ \text{ ip}\,(.p, L, \{\text{ lock}(m); s\})\end{array}}$$

   - *Unlock*

     No se comprueba el caso del *unlock* si el mútex no está cogido, ya que se supone que sólo puede soltar un mútex el proceso que previamente lo haya cogido.

$$[\text{ unlock}] : \begin{array}{l}\text{gl}\,(M, W, G, P)\ \text{ip}\,(.p, L, \{\text{ unlock}(m); s\}) \\ \qquad \rightarrow \text{ gl}\,(M[m \rightarrow \text{null}], W, G, P)\ \text{ ip}\,(.p, L, \{s\})\end{array}$$

   - Además, vamos a crear una regla *acquire* que podrá aplicarse cuando haya procesos esperando en W y el lock m esté libre.

$$\left[\text{acquire}^{\text{ true}}\right] : \frac{M[m] = \text{null} \qquad W[m] \neq \varnothing \qquad id = \text{select(m,W)}}{\begin{array}{l}\text{gl}\,(M, W, G, P)\ \text{ ip}\,(id, L, \{\text{ lock}(m); s\}) \\ \qquad \rightarrow \text{ gl}\,(M[m \mapsto id], \text{quitProc(m,id,W)}, G, P)\ \text{ ip}\,(id, L, \{s\})\end{array}}$$

6. *New*

   Vamos a suponer que tenemos una función *cod(P)* que devuelve la secuencia de instrucciones en el cuerpo del procedimiento P, y que *init(P, args)* devuelve un conjunto L de variables, formado por las variables locales declaradas dentro del procedimiento P y los argumentos introducidos.

   Con estas dos funciones, la semántica del *new* es

$$\begin{aligned}[\text{ new}] : &\ \text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p1, L_1, \{.p2 = \ \text{new}\ P_2(args); s\}) \\ &\to\ \text{gl}\,(M, W, G, \text{add}(.p2, P))\ \ \text{ip}\,(.p1, L_1, \{s\})\ \ \text{ip}\,(.p2, \text{init}(P_2, args), \text{cod}(P_2))\end{aligned}$$

7. *Join*

   Describimos la regla en el caso de que el proceso al que espera haya acabado, ya que en caso contrario no se produce ningún cambio en el estado.

$$[\text{ join}^a] : \frac{\neg isActive(.p2)}{\text{ip}\,(.p1, L, \{\ \text{join}(.p2); s\}) \to\ \text{ip}\,(.p1, L, \{s\})}$$

8. *Secuencia vacía de instrucciones*

   En el caso de que se haya llegado al final del código de un proceso tenemos:

$$[\text{empty}] :\ \text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p, L, \{\}) \to\ \text{gl}\,(M, W, G, \text{quit}(.p, P))$$

A la hora de escoger un proceso para ejecutarse en el procesador, la elección será no determinista. Se puede usar de nuevo la función *select(q)*, usada en el caso de los mútex, para seleccionar un identificador de proceso al azar de un conjunto q de procesos listos para ejecutar.

2.7. Noción de deadlock

En el estudio del deadlock en concurrencia, se suele recurrir a grafos que reflejen dependencias entre procesos, de forma que un ciclo en estos grafos puede indicar una situación de deadlock en la ejecución del programa.

En el caso de nuestro lenguaje imperativo, vamos a dar una definición de deadlock que, veremos, que es equivalente a la aparición de ciclos en grafos de dependencia.

Definición 2.1 (Cambio de estado)**.** Dada la semántica del lenguaje imperativo descrita en la sección anterior, diremos que se produce un cambio de estado si, dado un resultado de la semántica operacional $a \to b$, se cumple que $a \neq b$.

Observación 2.1. Con las reglas de la semántica operacional que tenemos definidas, las dos únicas instrucciones que no dan lugar a un cambio de estado son

$$\left[\text{acquire}^{\text{ false}}\ \right] : \frac{M\,[m] = .p1}{\begin{aligned}&\text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p1, L_1, \{s_1\})\ \ \text{ip}\,(.p, L, \{\ \text{lock}(m); s\}) \\ &\quad \to\ \text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p1, L_1, \{s_1\})\ \ \text{ip}\,(.p, L, \{\ \text{lock}(m); s\})\end{aligned}}$$

$$[\text{ join}^{na}] : \frac{isActive(.p2)}{\text{ip}\,(.p1, L, \{\ \text{join}(.p2); s\}) \to\ \text{ip}\,(.p1, L, \{\ \text{join}(.p2); s\})}$$

que no se han definido previamente en la semántica por no producir ningún cambio.

Definición 2.2 (Deadlock)**.** Sea un programa en el lenguaje imperativo, donde P representa el conjunto de procesos activos. Dadas las operaciones semánticas descritas anteriormente, diremos que tenemos *deadlock* si y sólo si se dan las dos siguientes condiciones:

1. $P \neq \emptyset$

2. $\nexists . p \in P$ que pueda ejecutar alguna de las reglas semánticas descritas en la sección anterior de forma que se produzca un cambio de estado.

Con esta definición de deadlock podemos ver que, efectivamente, es equivalente a que haya ciclos en un grafo de dependencias, ya que para darse las dos situaciones iniciales a la vez tienen que haberse bloqueado al menos dos procesos mutuamente, bien sea esperando a que se libere un mútex o bien sea esperando a que acabe un proceso. Por tanto, si dibujásemos el grafo de dependencias de forma que $P_i \rightarrow P_j$ indica que P_i está esperando a P_j, tendríamos al menos un ciclo.

Tenemos por tanto dos posibles combinaciones que pueden dar lugar a *deadlock* en nuestro programa:

- lock/lock

- lock/join (o join/lock)

Observación 2.2. El caso de *join/join* no es posible por el hecho de que para que un proceso pueda esperar a otro, el segundo debe haber sido creado con anterioridad.

Ejemplo 2.1 (lock/lock)**.**

```
Main(){
  Mutex m1;
  Mutex m2;

  .p1 = new P1();
  .p2 = new P2();
}

process P1(){
  lock(m1);
  //posible deadlock
  lock(m2);
  unlock(m2);
  unlock(m1);
}

process P1(){
  lock(m2);
  //posible deadlock
  lock(m1);
  unlock(m1);
  unlock(m2);
}
```

donde efectivamente tenemos las dos condiciones de la definición de *deadlock*:

1. $P = \{\text{Main}, .p1, .p2\} \neq \emptyset$
2. Los procesos activos no tienen instrucciones para ejecutar y producir un cambio de estado:

$$\left[\text{acquire}_1^{\text{false}}\right] : \frac{M\,[m] = .p2}{\begin{array}{l}\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p2, L_2, \{s_2\})\;\;\text{ip}\,(.p1, L_1, \{\;\text{lock}(m2); s_1\})\\ \quad\rightarrow\;\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p2, L_2, \{s_2\})\;\;\text{ip}\,(.p1, L_1, \{\;\text{lock}(m2); s_1\})\end{array}}$$

$$\left[\text{acquire}_2^{\text{false}}\right] : \frac{M\,[m] = .p1}{\begin{array}{l}\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p1, L_1, \{s_1\})\;\;\text{ip}\,(.p2, L_2, \{\;\text{lock}(m1); s_2\})\\ \quad\rightarrow\;\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p1, L_1, \{s_1\})\;\;\text{ip}\,(.p2, L_2, \{\;\text{lock}(m1); s_2\})\end{array}}$$

Ejemplo 2.2 (lock/join)**.**

```
Main(){
  Mutex m;

  .p2 = new P2();
  .p1 = new P1();
}

process P1(){
  lock(m);
  //posible deadlock
  join(.p2);
  unlock(m);
}

process P2(){
  //posible deadlock
  lock(m);
  unlock(m);
}
```

Se cumplen las dos condiciones de la definición de *deadlock*:

1. $P = \{\text{Main}, .p1, .p2\} \neq \emptyset$
2. Los procesos activos no tienen instrucciones para ejecutar y producir un cambio de estado:

$$[\;\text{join}^{na}] : \frac{\mathit{isActive}(.p2)}{\text{ip}\,(.p1, L, \{\;\text{join}(.p2); s\}) \rightarrow\;\text{ip}\,(.p1, L, \{\;\text{join}(.p2); s\})}$$

$$\left[\text{acquire}^{\;\text{false}}\right] : \frac{M\,[m] = .p1}{\begin{array}{l}\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p1, L_1, \{s_1\})\;\;\text{ip}\,(.p2, L_2, \{\;\text{lock}(m); s_2\})\\ \quad\rightarrow\;\text{gl}\,(M, W, G, P)\;\;\text{ip}\,(.p1, L_1, \{s_1\})\;\;\text{ip}\,(.p2, L_2, \{\;\text{lock}(m); s_2\})\end{array}}$$

CAPÍTULO 3

El lenguaje ABS

ABS (*Abstract Behavioral Specification*) es un lenguaje destinado a especificar de forma abstracta el comportamiento de sistemas distribuidos orientados a objetos. Está basado en actores: los actores toman decisiones en base a mensajes recibidos. En este caso, los actores serán los objetos concurrentes.

Los modelos ABS se utilizan para la especificación de modelos formales ejecutables, reflejando el flujo de concurrencia a la vez que se abstraen de detalles de implementación como las representaciones internas o las propiedades del entorno.

3.1. Diseño en ABS

ABS está diseñado para ser cercano a la forma de pensar de un programador, con una sintaxis parecida a Java y un flujo de control parecido a una implementación real. La idea principal es que el control y los datos están ambos recogidos en el objeto concurrente. Además, dispone de semánticas definidas formalmente, que permiten abstraerse de detalles de implementación poco deseables.

El modelo de concurrencia en ABS se divide en dos capas, separando en la capa inferior memoria compartida local y síncrona, y en la capa superior comunicación asíncrona mediante paso de mensajes. Esta separación da lugar a una visión de la estructura que caracteriza a los programas en ABS:

- En la *capa inferior* tenemos los grupos de objetos concurrentes o *cogs*, que se asimilan a componentes de ejecución basados en objetos con memoria local compartida.
- En la *capa superior* nos encontramos con una comunicación asíncrona entre cogs, realizada mediante paso de mensajes. Entre grupos de objetos concurrentes no hay memoria compartida.

Cada objeto concurrente, como se verá posteriormente, sólo admite una tarea ejecutándose a la vez. Esto nos llevará a tener en cada objeto una cola de tareas pendientes

de ser ejecutadas. De esta forma, se permiten aproximaciones a la verificación del sistema basadas en el estudio de invariantes de monitores.

3.2. Tipos

En ABS no hay tipos primitivos para los valores básicos, pero se permite la definición de tipos algebraicos; soporta tipos definidos por el usuario con funciones de primer orden y ajuste de patrones. Hay tipos predefinidos como *Unit* (equivale a *void*), *Bool*, *Int* y *String*. Tanto los tipos de datos como las funciones son polimórficas (aceptan parámetros).

La abstracción introducida al principio del capítulo se ve reflejada en el tipo de las clases que representan la funcionalidad de los objetos, que viene dado por interfaces. Esta separación entre tipos e implementación da lugar a *modelos composicionales*, en los que diversos componentes realizan una tarea de manera coordinada. Los interfaces suponen el único método de encapsulación de ABS. Cada clase puede implementar más de un interfaz, suponiendo que soporte todos los métodos. Se acepta polimorfismo de tipos a nivel de interfaz, en el sentido de que un objeto que implementa un interfaz puede ser sustituido por otro objeto que soporte el mismo interfaz o un subtipo de este.

En ABS tenemos, además, un tipo de variables *futuras*, denotado por *Fut*, que tendrá especial importancia a la hora de la comunicación entre objetos. Una variable futura se declara y asocia a la ejecución de un método, de manera que en el momento de su declaración el valor de retorno no está disponible. Se utilizarán en la sincronización entre cogs, para separar la invocación del método del valor retornado por éste.

3.3. Sincronización en ABS

Las llamadas a métodos en ABS son asíncronas. Se planifican las activaciones de los métodos de manera cooperativa, de forma que se pueda controlar la intercalación de actividades en los grupos de objetos concurrentes. Sin embargo, este control no incluye la decisión de qué tarea ejecutar dentro de un objeto concurrente.

Cada objeto concurrente dispone, como se ha introducido previamente, de una cola de tareas pendientes de ejecutar, producto de distintas activaciones bien sea desde el propio objeto o desde objetos externos. En la implementación, cada cog en ABS tiene el diseño propio de un monitor, permitiendo únicamente la ejecución de una tarea a la vez. Cuando una tarea libera el lock que tenía cogido en el objeto, cualquiera de las otras tareas pendientes puede cogerlo para ejecutarse, pero el programador no puede decidir cuál se ejecutará.

De nuevo, podemos estudiar la sincronización según las capas de ABS:

- *Capa inferior*

  Cada cog tiene un procesador propio y se encuentra en un entorno distribuido, con un conjunto de objetos concurrentes.

  Dentro de cada cog nos encontramos con una construcción no determinista, que implica que podemos tener un conjunto finito de posibles estados a los que llegar tras aplicar una transición. Esto se debe a que, en cada cog, no decidimos cuál de

todas las tareas pendientes de ejecutarse va a obtener el procesador; en ABS no se asume nada sobre mecanismos de planificación.

- *Capa superior*

  Entre los diferentes cogs sólo se pueden realizar llamadas asíncronas. Para acceder a los métodos de otro objeto, tenemos que realizar llamadas a estos métodos. Cada objeto será el único que controle su propio estado.

De esta forma, el comportamiento de un cog se basa en una ejecución cooperativa de las activaciones en la cola del objeto. Estas activaciones vendrán dadas tanto por llamadas asíncronas externas o desde el propio objeto a métodos del mismo, como por llamadas síncronas internas realizadas en métodos del propio cog. Aunque pueden coexistir métodos síncronos y asíncronos, sólo se sincronizarán el objeto que invoca el método y el objeto invocado cuando sea necesario. Esta cooperación entre las tareas pendientes de ejecutar en un cog evita las carreras de datos.

Las llamadas a métodos de objetos están tipadas por interfaces. Nos referimos a llamadas asíncronas en el sentido de que el objeto que realiza la llamada puede decidir en tiempo de ejecución cuándo sincronizar con el resultado de la llamada. Una llamada a un método es la forma de introducir concurrencia en el sistema, incluyendo nuevos métodos que quieren ejecutarse en el objeto al que se ha llamado. La cantidad de concurrencia permitida en un modelo ABS se ve reflejada en el número de cogs introducidos en el modelo.

3.4. Sintaxis

En un modelo ABS tenemos definición de interfaces, clases, tipos de datos, funciones y un bloque *main*, que configura el estado inicial y constituye el punto de entrada al programa. Un programa, por tanto, está formado por un conjunto de clases, donde cada una de ellas puede definir un conjunto de campos y un conjunto de métodos.

Mediante la instrucción *new*, podemos crear objetos a partir de clases de manera dinámica, inicializando sus atributos a los valores por defecto del tipo que corresponda. Si no deseamos que estos valores sean los obtenidos por defecto, podemos declarar un método *init*, similar a los constructores en Java, que asigne los valores deseados.

ABS dispone de declaración de variables, asignación e instrucciones de control *if/else* y *while* con una sintaxis similar a la de los lenguajes de programación habituales.

Como se introdujo anteriormente, disponemos de un tipo de variables futuras para asignar a la ejecución de métodos. Suponemos que todos los métodos finalizan con una instrucción *return*. Hay dos instrucciones para interaccionar con este tipo de variables:

- *await*

  Esta instrucción comprueba si el resultado de la ejecución del método asociado a la variable futura está disponible y, en caso de estarlo, continúa la ejecución del método del objeto en el que se encontraba a partir de ese punto. Si el valor no está disponible, libera el lock del objeto, de forma que otras tareas pendientes en el objeto tengan la oportunidad de ejecutarse, y se coloca la tarea con este método en la cola de tareas pendientes de ejecución en el objeto. Es una instrucción que no bloquea el objeto

aunque, como veremos posteriormente, sí que puede estar presente en la aparición de deadlock.

- *get*

  Esta instrucción sirve para obtener el valor asociado al retorno del método, y que es accesible a través de la variable futura. En el caso de que el valor no esté disponible, espera a que lo esté manteniendo el lock del objeto y, por tanto, bloqueando la ejecución del resto de tareas en él. Será la instrucción que, en caso de aparecer sola (sin un *await* previo sobre la variable futura), inducirá deadlock en el sistema.

Tenemos dos tipos de comportamiento: el comportamiento activo y el comportamiento pasivo. El primero se dispara con un método *run*, que es opcional y que se ejecuta en la creación del objeto. El comportamiento pasivo, por otro lado, se debe a las llamadas asíncronas a métodos.

La comunicación en ABS se basa en llamadas síncronas y asíncronas a métodos, aunque las segundas son más comunes. Si o es el objeto en el que queremos ejecutar, m es el método que queremos ejecutar en el objeto y p son sus parámetros, realizamos una llamada *síncrona* mediante $[x =]o.m(p)$ y una llamada asíncrona mediante $[x =]o!m(e)$, donde x es una variable futura que puede omitirse.

Basándonos en llamadas asíncronas a métodos, *await* y *get*, podemos desarrollar el código correspondiente a cada una de las instrucciones anteriores.

```
1 //sincrona
2 x=o!m(e);
3 v=x.get;
4
5 //asincrona
6 x=o!m(e);
7 await x?;
8 v=x.get;
```

3.5. Semántica operacional

El estado de un programa viene dado por el conjunto de todos los objetos creados y el conjunto de todas las tareas activas, pendientes o terminadas.

Un *objeto* está representado por $ob\,(o, a, p)$, donde o es el identificador del objeto, a son los valores de los campos del objeto y p identifica la tarea activa ($\perp$ si el lock del objeto está libre). La representación de una *tarea* es un término $tsk\,(t, m, o, l, s)$ donde t es un identificador único de la tarea, m es el nombre del método que se está ejecutando, o identifica el objeto al que pertenece la tarea, l son los valores de las variables locales (incluidas futuras) y s es la secuencia de instrucciones que quedan por ejecutar ($s = \epsilon\,(\nu)$ indica que ha acabado la tarea y el valor ν está disponible).

Partiendo de un estado inicial, la ejecución se realiza aplicando reglas operacionales no deterministas. La semántica operacional está escrita en un estilo basado en la reescritura.

Cada transición de la forma

$$a\,b \rightarrow b'\,c$$

indica que parte de un estado representado por a y b, y que llega a un nuevo estado en el que sólo se indican los cambios b', que es el nuevo valor de b, y c, que son las cosas nuevas que se han añadido al estado. a no ha sufrido cambios, por lo que se omite.

Para poder definir las reglas de la semántica operacional, hay que describir previamente una serie de métodos auxiliares:

- *fresh(v)* devuelve cierto si el nombre v es globalmente único.
- *init_atts(B,z)* inicializa los atributos de un objeto de tipo B con los parámetros z.
- *buildLocals(z,m)* crea el estado inicial de una tarea en la llamada a un método m con parámetros z.
- *body(m)* devuelve el código correspondiente al cuerpo de un método m.
- *select(q)* devuelve una tarea t de la lista q de tareas listas para ejecutar en un objeto concurrente.
- $[\![e]\!]_{a \circ l}$ devuelve el valor correspondiente a la expresión e después de hacer el mapping correspondiente con los valores a de los campos del objeto y los valores l de las variables locales.

Usando estos métodos auxiliares, las reglas de la semántica operacional que tenemos son:

- *Asignación de variable local*

$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]_{a \circ l}}{\mathit{ob}\,(o,a,t)\,\mathit{tsk}\,(t,m,o,l,\{x = e; s\}) \rightarrow \mathit{tsk}\,(t,m,o,l[x \mapsto v],s)}$$

- *Asignación de campo del objeto*

$$\frac{x \in \mathrm{dom}(a) \quad v = [\![e]\!]_{a \circ l}}{\mathit{ob}\,(o,a,t)\,\mathit{tsk}\,(t,m,o,l,\{x = e; s\}) \rightarrow \mathit{ob}\,(o,a[x \mapsto v],t)\,\mathit{tsk}\,(t,m,o,l,s)}$$

- *skip*

$$\mathit{ob}\,(o,a,t)\,\mathit{tsk}\,(t,m,o,l,\{\mathit{skip}; s\}) \rightarrow \mathit{tsk}\,(t,m,o,l,s)$$

- *If/else*

$$\left[\text{ if }^{\text{true}}\right] : \frac{[\![e]\!]_{a \circ l}}{\begin{array}{c}\mathit{ob}\,(o,a,t)\,\mathit{tsk}\,(t,m,o,l,\{\text{ if } e \text{ then } \{s_1\} \text{ else } \{s_2\}\,; s\}) \\ \rightarrow \mathit{tsk}\,(t,m,o,l,\{s_1; s\})\end{array}}$$

$$\left[\text{ if }^{\text{false}}\right] : \frac{\neg\,[\![e]\!]_{a \circ l}}{\begin{array}{c}\mathit{ob}\,(o,a,t)\,\mathit{tsk}\,(t,m,o,l,\{\text{ if } e \text{ then } \{s_1\} \text{ else } \{s_2\}\,; s\}) \\ \rightarrow \mathit{tsk}\,(t,m,o,l,\{s_2; s\})\end{array}}$$

- *While*

$$[\text{ while }^{true}] : \frac{[\![e]\!]_{a\circ l}}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{\text{ while } e \text{ do } \{s_1\}\,;s\}) \\ \quad \to tsk\,(t,m,o,l,\{s_1;\ \text{while } e \text{ do } \{s_1\}\,;s\}) \end{array}}$$

$$[\text{ while }^{false}] : \frac{\neg\, [\![e]\!]_{a\circ l}}{ob\,(o,a,t)\, tsk\,(t,m,o,l,\{\text{ while } e \text{ do } \{s_1\}\,;s\}) \to tsk\,(t,m,o,l,s)}$$

- *Nuevo objeto*

$$\frac{\text{fresh}(o') \quad a' = \text{init_atts}\,(B,z)}{ob\,(o,a,t)\, tsk\,(t,m,o,l,\{x = \ \text{new}\, B\,(z)\,;s\}) \to tsk\,(t,m,o,l,s)\, ob\,(o',a',\bot)}$$

- *Activación*

$$\frac{t = \text{select}\,(q) \quad s \neq \epsilon\,(v)}{ob\,(o,a,\bot)\, tsk\,(t,m,o,l,s) \to ob\,(o,a,t)}$$

- *Llamada asíncrona*

$$\frac{o_1 = [\![x]\!]_{a\circ l} \neq null \quad l_1 = \text{buildLocals}(z, m_1) \quad \text{fresh}(t_1)}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{y = x!m_1(z);s\}) \\ \quad \to tsk\,(t,m,o,l\,[y \mapsto t_1]\,,s)\, tsk\,(t_1,m_1,o_1,l_1,\text{body}(m_1)) \end{array}}$$

- *Await*

$$[\text{await }^{\text{true}}] : \frac{[\![y]\!]_{a\circ l} = t_1}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{\text{await } y?;s\})\, tsk\,(t_1,m_1,o_1,l_1,\epsilon(v)) \\ \quad \to tsk\,(t,m,o,l,s) \end{array}}$$

$$[\text{await }^{\text{false}}] : \frac{[\![y]\!]_{a\circ l} = t_1 \qquad s_1 \neq \epsilon(v)}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{\text{await } y?;s\})\, tsk\,(t_1,m_1,o_1,l_1,s_1) \\ \quad \to ob\,(o,a,\bot) \end{array}}$$

- *Return*

$$\frac{v = [\![x]\!]_{a\circ l}}{ob\,(o,a,t)\, tsk\,(t,m,o,l,\{\text{return } x;s\}) \to ob\,(o,a,\bot)\, tsk\,(t,m,o,l,\epsilon(v))}$$

- *Get*

$$[\text{get }^{\text{true}}] : \frac{t_1 = [\![y]\!]_{a\circ l}}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{x = y.get;s\})\, tsk\,(t_1,m_1,o_1,l_1,\epsilon(v)) \\ \quad \to tsk\,(t,m,o,l\,[x \mapsto v]\,,s) \end{array}}$$

$$[\text{get }^{\text{false}}] : \frac{t_1 = [\![y]\!]_{a\circ l} \qquad s_1 \neq \epsilon(v)}{\begin{array}{l} ob\,(o,a,t)\, tsk\,(t,m,o,l,\{x = y.get;s\})\, tsk\,(t_1,m_1,o_1,l_1,s_1) \\ \quad \to tsk\,(t,m,o,l,\{x = y.get;s\}) \end{array}}$$

3.6. Deadlock

Si un programa concurrente alcanza un estado compuesto únicamente por tareas acabadas y por tareas que están esperando la terminación de otra y ninguna puede avanzar, diremos que hay *deadlock*. El conjunto de tareas acabadas puede ser vacío, pero el de tareas bloqueadas no.

Se distinguen dos situaciones: una tarea en espera, que podría estar esperando a obtener el lock del objeto o a leer una variable futura, y una tarea bloqueante que espera a leer una variable futura mientras mantiene el lock. Los posibles deadlocks van a aparecer como combinación de las dos primitivas de sincronización *await y?* e *y.get*.

3.6.1. Análisis de deadlock

Como se combinan mecanismos bloqueantes y no bloqueantes para acceder a las variables futuras, se necesita un análisis formal. En el caso de ABS el análisis de deadlock se basa en un grafo de dependencias, en el que se tienen tres tipos de arcos, uno para cada tipo de dependencia:

1. Dependencia *tarea-tarea*: una tarea espera al valor de una futura que otra deberá calcular.

2. Dependencia *tarea-objeto*: una tarea está esperando a coger el lock del objeto concurrente.

3. Dependencia *objeto-tarea*: una tarea espera al valor de una futura mientras mantiene el lock del objeto.

Cualquier tarea que no haya acabado y no tenga el lock del objeto concurrente tiene una dependencia con su objeto correspondiente.

Un estado S de un programa está bloqueado si y sólo si su grafo de dependencias contiene un ciclo.

Las instrucciones *await y?* sólo introducen una dependencia tarea-tarea. Para obtener ciclos en el grafo, y por tanto deadlocks, necesitaremos al menos una instrucción *y.get*; todos los ciclos buscados en el grafo durante el análisis deberán contener al menos una dependencia objeto-tarea.

El análisis de deadlock está hecho de forma que, si dice que un programa está libre de deadlock, entonces ninguna ejecución llegará a un estado de deadlock. En caso contrario, el programa devolverá un deadlock potencial que podrá ser cierto o estar derivado de que no puede probar que no hay deadlock en alguna ejecución.

Las tareas finalizadas se mantienen en el estado del programa, pero no se tienen en cuenta a la hora de analizar el deadlock, ya que no pueden conllevar ningún tipo de espera.

CAPÍTULO 4

Traducción

La base de nuestra traducción viene dada, en parte, por el número de procesos que vamos a poder ejecutar en paralelo en nuestro programa.

Con la traducción que introduciremos, cada nuevo proceso creado tendrá su propia ejecución en paralelo, de manera que tendremos un símil de tantos procesadores como procesos. Esto se debe a que, en ABS, cada cog tiene su propio procesador para una ejecución en paralelo.

Esta suposición sobre el número de procesadores a compartir entre procesos hace que tengamos el caso de estudio que más bloqueos por secciones críticas o carreras de datos puede producir: aquel en el que todos los procesos pueden intentar acceder concurrentemente a la parte protegida del código y, por tanto, pueden bloquear nuestro programa.

La siguiente base de la traducción está también centrada en los cogs, o grupos de objetos concurrentes. Como se ha visto previamente, un cog en ABS funciona como una unidad propia de ejecución, con un procesador que es compartido por las distintas tareas que queremos ejecutar en el objeto concurrente. Este funcionamiento, con un invariante similar al de un monitor, nos permitirá desarrollar la traducción de los mútex.

Tendremos una clase definida para cada mútex del programa, de forma que cada mútex será un cog independiente. Como cada cog tiene un lock propio para la ejecución de las distintas tareas, nos serviremos del lock de cada cog para gestionar las distintas secciones críticas, de forma que una llamada a un método del cog que codifica el mútex equivale a la ejecución del código que, en nuestro programa imperativo original, estaba entre dos instrucciones lock/unlock del mútex correspondiente.

4.1. Visión global de la traducción

Para empezar, vamos a desarrollar una visión global de la traducción. La creación de ciertas partes de las clases se explica de una forma abstracta para dejar abierta la posibilidad de introducir posteriormente código en ellas. Es decir, en parte en esta sección lo que creamos son los esqueletos. Cuando acabemos la construcción de todo el código

mediante la traducción que se explicará a lo largo del capítulo, se podría volcar a un fichero y dar por terminado el proceso.

Supongamos que se ha realizado una primera lectura del código para obtener:

- Un conjunto G formado por los nombres de variables globales de tipo entero.
- Un conjunto M con tuplas de la forma (m, n) donde m es el nombre del mútex y n el máximo número de apariciones del mútex en un mismo proceso. Además, contaremos con un número C dado por

$$C = \text{máx}\,\{n : (m, n) \in M\}$$

- Un conjunto P con los nombres de los tipos de procesos.
- Un conjunto H tal que $H[\text{ProcType}]$ nos devuelve los mútex usados en cada proceso de tipo *ProcType*.
- Un conjunto F tal que $F[\text{ProcType}]$ devuelve un conjunto de los identificadores de proceso sobre los que se hace *join*.
- Un conjunto L tal que $L\,[\text{ProcType}]$ nos devuelve el conjunto de variables locales declaradas en el proceso.

En la traducción de las variables globales y los mútex tenemos ya el resultado final, mientras que en los procesos y el main desarrollaremos el esqueleto previo a la traducción de instrucciones. Cabe destacar que, aunque se agrupe por tipos, en la implementación final de la traducción todas las interfaces se escribirán al principio y las clases a continuación, seguidas del cuerpo correspondiente al main.

4.1.1. Variables globales

Sea $G = \{_v1, \ldots, _vk\}$.

El interfaz de las variables globales viene dado por

```
interface GlobalsInterface{
  Int getV1();
  Unit setV1(Int a);
  ...
  Int getVk();
  Unit setVk(Int a);
}
```

Al igual que el interfaz, la clase que maneja las variables globales se puede crear completamente antes de realizar la traducción de las instrucciones internas. La clase *Globals* seguirá el esquema

```
class Globals() implements GlobalsInterface{
  Int v1;
```

```
3     ...
4     Int vk;
5
6     Int getV1(){
7       return v1;
8     }
9     Unit setV1(Int a){
10      v1 = a;
11    }
12    ...
13    Int getVk(){
14      return vk;
15    }
16    Unit setVk(Int a){
17      vk = a;
18    }
19  }
```

4.1.2. Mútex

El interfaz de los mútex tendrá C métodos, siendo C el máximo de apariciones de mútex definido previamente. Esto se debe a que, a lo sumo, un método de un mútex tendrá que implementar C métodos diferentes para gestionar las llamadas de otros procesos. Se introduce aquí la notación *ProcessInterface*, que se refiere al interfaz de los procesos que se introducirá en el apartado siguiente. De esta forma, tenemos

```
1  interface MutexInterface{
2    Unit m_1(ProcessInterface p);
3    ...
4    Unit m_C(ProcessInterface p);
5  }
```

En cada una de las clases de los mútex, tendremos que implementar este interfaz. Sin embargo, no en todos los mútex tenemos por qué alcanzar el máximo de apariciones, por lo que habrá ciertos métodos que se queden vacíos. Tenemos que implementar internamente tantos métodos como el máximo de ese mútex en todos los procesos, en aquellos procesos en los que no se llegue al máximo simplemente dejaremos vacío el método tras la traducción.

Para cada $(mj, n) \in M$, tendremos una clase Mj de la forma

```
1  class Mj() implements MutexInterface{
2    Unit m_1(ProcessInterface p){
3      Fut<Unit> f;
4      f = p!mj_1();
5      f.get;
6    }
7    ...
8    Unit m_n(ProcessInterface p){
```

```
9        Fut<Unit> f;
10       f = p!mj_n();
11       f.get;
12     }
13     Unit m_{n+1}(ProcessInterface p){}
14     ...
15     Unit m_C(ProcessInterface p){}
16   }
```

4.1.3. Procesos

En el caso de los procesos, en esta primera parte de la traducción podemos traducir completamente el interfaz. También crearemos las clases, dejando los cuerpos de los métodos vacíos para que se rellenen al aplicar la función τ de traducción que definiremos más adelante, en caso de que tengan código asociado en la traducción.

En el caso del interfaz, tenemos que incluir tanto el método *start* como los métodos $m_1, \ldots, m_n$ para cada $(m, n) \in M$. Esto se debe a que, como *ProcessInterface* será un interfaz común a todos los procesos, tendremos que tener suficientes métodos como para atender los casos de cualquier proceso. En el caso de que un proceso no necesite tantos métodos, como creamos el esqueleto de las clases dejándolo vacío, el resultado tras aplicar τ será que habrá código en los métodos correspondientes al programa original, y los que sobren en cada proceso estarán vacíos.

Vamos a suponer que tenemos los mútex $m1, \ldots, mj$ con máximos asociados $n1, \ldots, nj$. El interfaz resultante es

```
1  interface ProcessInterface{
2    Unit start();
3    Unit m1_1();
4    ...
5    Unit m1_n1();
6    Unit m2_1();
7    ...
8    Unit m2_n2();
9    ...
10   Unit mj_1();
11   ...
12   Unit mj_nj();
13 }
```

Supongamos que tenemos el conjunto $\{m_{i1}, \ldots, m_{ij}\} = H\,[\text{ProcType}]$. Obtenemos además los procesos sobre los que realizamos *join* en cada proceso, dados por $F\,[\text{ProcType}]$. Usando la función *fut(iden)* que concatena *iden* a *f_*, definimos el conjunto

$$\text{Fut} = \{\text{fut}(\text{iden}) : \text{iden} \in F\,[\text{ProcType}]\} = \{f1, \ldots, fs\}$$

Las variables locales del proceso son $\{l1, \ldots, lr\} = L\,[\text{ProcType}]$.

Para cada proceso ProcType $\in P$, siendo *(Int p1, ... Int pk)* los parámetros que acepta en el lenguaje imperativo, tenemos la clase

```
class PName(Int p1, ..., Int pk, Fut<Unit> f1, ...,Fut<Unit> fs,
    MutexInterface mi1,..., MutexInterface mij, GlobalsInterface
    globals) implements ProcessInterface{
  Int l1;
  ...
  Int lr;

  Unit start(){}
  Unit m1_1(){}
  ...
  Unit m1_n1(){}
  Unit m2_1(){}
  ...
  Unit m2_n2(){}
  ...
  Unit mj_1(){}
  ...
  Unit mj_nj(){}
}
```

4.1.4. Main

Por último, falta añadir a nuestro fichero el bloque correspondiente al *Main*, que simplemente será de la forma

```
{
GlobalsInterface globals = new Globals();
}
```

El código interno se rellenará posteriormente y se añadirá a partir de la línea de *Globals*. Por la forma en que se ha diseñado la traducción, en el caso del Main no podremos encontrarnos con uso de instrucciones *lock/unlock*, ya que no podemos crear métodos auxiliares para codificar el flujo de ejecución.

4.2. Función de traducción

Vamos a definir $I = (tokenImp)*$, que será el espacio de partida de nuestra función, donde los tokens representan las posibles estructuras semánticamente correctas en nuestro

lenguaje, y que vienen dadas por

< INT_TYPE :“Int” >
< MUTEX_TYPE :“Mutex” >
< IF :“if” >
< ELSE : “else” >
< WHILE : “while” >
< NEW : “new” >
< MAIN : “Main” >
< FOR : “for” >
< UNLOCK : “unlock” >
< LOCK : “lock” >
< PROCESS : “process” >
< JOIN : “join” >

< O_PAR :“(” >
< C_PAR :“)” >
< O_BRACE:“{” >
< C_BRACE:“}” >

< COMMA:“,” >
< SEMICOLON:“;” >
< ASSIGN:“=” >

< #DIGIT:[“0”-“9”] >
< #LETTER:[“A”-“Z”]|[“a”-“z”] >
< INT_CONST : [“-”] (<DIGIT>)+ >

< VAR_IDENT : [“a”-“z”] (<LETTER> | <DIGIT> | “_”)* >
< PROC_NAME : [“A”-“Z”] (<LETTER> | <DIGIT> | “_”)* >
< GLOBAL_VAR_IDENT : “_” (<LETTER> | <DIGIT> | “_”)+ >
< PROC_IDENT : “.” (<LETTER> | <DIGIT> | “_”)+ >

y $A = (tokenABS)*$, que será el espacio de llegada de la función, formado por las estructuras aceptadas en un programa ABS.

Necesitamos, además, ciertas funciones auxiliares:

- String FirstToUpper(String str) devuelve la cadena que recibe como parámetro con su primera letra en mayúsculas.

- writeProc(String nom, String[] codigo) rellena el método *nom* en el proceso en el que estamos con las instrucciones *codigo*.

- String newArgs(String args, String ProcName) añade a los argumentos *args* los argumentos referentes a mútex, futuras y variables globales que necesita el proceso, de acuerdo al orden que tienen en los parámetros. Como los nombres se traducen igual en parámetros e instrucciones, bastará usar el nombre de los parámetros en los argumentos.

Con estos dos espacios y las funciones auxiliares, podemos definir nuestra función de traducción. No se estudia la declaración de las variables locales ni globales, ya que se han usado ya previamente para la creación de los esqueletos. Sí que mantenemos la traduccioń de la declaración de mútex ya que, aunque hemos creado las clases, tenemos que crear un nuevo objeto concurrente.

Además, aunque en la práctica tendremos la semántica descrita para *lock/unlock*, en la función de traducción anotaremos con un superíndice la aparición correspondiente al

mútex en el proceso, para así hacer más sencilla la notación.

$$
\begin{aligned}
\tau : I &\rightarrow A \\
\emptyset &\mapsto \emptyset \\
\text{Mutex Var_ Ident; resto} &\mapsto \text{MutexInterface Var_Ident} \\
&\qquad\quad = \text{new } \textit{FirstToUpper(Var_Ident)}(); \\
&\quad \tau\,(\text{resto}) \\
\text{Var_Ident =AExpr; resto} &\mapsto \text{Var_Ident = AExpr;} \\
&\quad \tau\,(\text{resto}) \\
\text{Global_Var_Ident =AExpr; resto} &\mapsto \text{globals.setGlobal_Var_Ident}(\textit{AExpr}); \\
&\quad \tau\,(\text{resto}) \\
; &\mapsto \text{skip;} \\
&\quad \tau\,(\text{resto}) \\
\text{if (AExpr)\{codigoIf\}} & \\
\text{else \{codigoElse\} resto} &\mapsto \text{if } (\textit{AExpr} \neq 0)\{\ \tau\,(\text{codigoIf})\} \\
&\quad \text{else } \{\tau\,(\text{codigoElse})\} \\
&\quad \tau\,(\text{resto}) \\
\text{while(AExpr) \{codigoWhile\}} & \\
\text{resto} &\mapsto \text{while } (\textit{AExpr} \neq 0)\{\tau\,(\text{codigoWhile})\} \\
&\quad \tau\,(\text{resto}) \\
\text{lock}^n\text{(Var_Ident); codigoSC} & \\
\text{unlock}^n\text{(Var_Ident); resto} &\mapsto \textit{writeProc(Var_Ident_n, } \tau(\textit{codigo})\textit{)} \\
&\quad \text{Fut<Unit> y_n;} \\
&\quad \text{y_n = nom!m_n(this);} \\
&\quad \text{await y_n?;} \\
&\quad \tau\,(\text{resto}) \\
\text{Proc_Ident= new Proc_name(args); resto} &\mapsto \text{ProcessInterface Proc_Ident = new} \\
&\qquad\qquad \text{Proc_Name}(\textit{newArgs(args,Proc_Name)}); \\
&\quad \text{Fut<Unit> f_Proc_Ident = Proc_Ident!start();} \\
&\quad \tau\,(\text{resto}) \\
\text{join(Proc_Ident); resto} &\mapsto \text{await f_Proc_Ident?;} \\
&\quad \tau\,(\text{resto})
\end{aligned}
$$

donde el texto con caligrafía normal se refiere a la traducción en sí de la función en forma de instrucciones, y el texto en cursiva identifica funciones utilizadas para obtener parámetros auxiliares.

Se ve en la función de traducción anterior que las sentencias *lock/unlock* van unidas. Esto se debe a que, al mantenerse la propiedad de buen orden, estas dos sentencias funcionan, aparte de para indicar secciones críticas, para delimitar bloques de código. Es un comportamiento similar a las llaves en los bloques de los ámbitos en los lenguajes de

programación habituales.

4.3. Equivalencia semántica entre sentencias en el lenguaje imperativo y sentencias en ABS

Observación 4.1. En algunos casos, como en las sentencias *if/else* o *while*, no se ha adaptado la notación de las reglas semánticas completamente a la notación usada en el lenguaje imperativo o en ABS, introduciéndose palabras auxiliares como *then* o *do*. No significa que se haya cambiado la sintaxis de los programas, simplemente es una cuestión de aumentar la claridad en la demostración a nivel semántico.

En los capítulos 2 y 3 se han introducido los estados del lenguaje imperativo y ABS respectivamente. En el caso del lenguaje imperativo, tenemos que el *Main* representa el punto de comienzo y cada *new* lanza un nuevo thread, mientras que en su correspondiente traducción a ABS tenemos las estructuras del *main* y cada cog de codificación de un proceso con el método *start*.

A partir de este punto, vamos a considerar que T y S son las representaciones de dos estados en el lenguaje imperativo y ABS, respectivamente.

Definición 4.1 (**Estados equivalentes**). Diremos que T y S son equivalentes, y lo denotaremos por $T \sim S$, si se cumple:

- La configuración de L en T se corresponde con la configuración de $a_{.p}$ en S. Es decir, tendremos un contenido similar en las variables locales del proceso en el primer lenguaje y los campos del objeto que lo codifica en el segundo.
- De manera similar, tendremos una configuración similar para G en el lenguaje imperativo como para $Globals$ en ABS. En el caso de este punto y el anterior, es trivial ver si son o no equivalentes, ya que basta ver que tenemos los mismos identificadores y los mismos valores.
- Los mútex M y a quién están asignados en el lenguaje imperativo guardan una equivalencia con qué objeto concurrente ha realizado la llamada asíncrona para ejecutar la tarea actual en la codificación correspondiente del mútex en ABS.
- Los procesos que están esperando a coger un mútex en el lenguaje imperativo (W) se corresponden con los objetos concurrentes que han realizado una llamada asíncrona que ha creado una tarea que está esperando para coger el procesador en alguno de los objetos que codifican los mútex.
- Si s_T es la siguiente secuencia de instrucciones a ejecutar en un proceso en el lenguaje imperativo, existirá m tal que $s_S = \{s_{S1}, \ldots, s_{Sm}\}$ son las siguientes instrucciones a ejecutar en ABS, equivalentes a s_T.

Lema 4.1. Sean T y S dos estados equivalentes a nivel de estructuras (sin tener en cuenta las instrucciones asociadas al estado). Sea s_t una instrucción en el lenguaje imperativo, denotamos por $T[s_t]$ al estado en el lenguaje imperativo producto de mantener la configuración de T y poner s_t como el código a ejecutar. De manera similar, sustituimos el codigo

en S por $\tau(s_t)$. s_t y $\tau(s_t)$, en el contexto del proceso en que estamos, son equivalentes y obtenemos, por la definición de equivalencia de estados, $T[s_t] \sim S[\tau(s_t)]$. Si ejecutamos las instrucciones s_t y $\tau(s_t)$ en $T[s_t]$ y $S[\tau(s_t)]$, obtendremos dos estados T' y S' tales que $T' \sim S'$.

Demostración. En estas reglas, utilizaremos como identificadores de los objetos en ABS los mismos identificadores usados en el lenguaje imperativo para referirse a los procesos que codifican. En el caso de los nombres de los objetos concurrentes usados en ABS, utilizaremos el mismo identificador sin el punto. Es decir, tendremos identificadores de objetos de la forma $.p$ que mantienen el estado de objetos concurrentes de nombre p.

- $\emptyset \mapsto \emptyset$

  Es trivial ver la equivalencia en este caso, ya que no ejecutar nada en el primer lenguaje implica no ejecutar nada en el segundo.

- **Mutex Var_ Ident;**
  $\mapsto$ **MutexInterface Var_Ident= new *FirstToUpperCase(Var_Ident)*();**

  En el lenguaje imperativo, esta instrucción supone la creación de un mútex sobre el que podremos realizar las operaciones de *lock/unlock*.

  En ABS, no existe el concepto de mútex como variable para poder declararlo explícitamente. Sin embargo, los objetos tienen su propio lock para alternar la ejecución de las tareas, por lo que la equivalencia se puede encontrar, en este caso, teniendo un objeto para cada mútex, de forma que realizar *lock/unlock* sobre el mútex equivale a ejecutar una tarea en el objeto que lo codifica.

  Vamos a usar en este caso la regla semántica de creación de un objeto en ABS para comprobar que, efectivamente, al crearse el objeto tenemos una equivalencia con que el *lock* esté libre en el lenguaje imperativo, que es lo que sucede en general en los lenguajes imperativos y, en particular, en el nuestro. Para identificar los objetos concurrentes en el caso de los mútex, usamos el mismo nombre que el de la clase del objeto concurrente, ya que sólo tendremos un objeto concurrente de cada clase.

  Vamos a suponer, para hacer más ligera la notación, que el mútex que traducimos desde el lenguaje imperativo se llama *nom*, con lo que tendremos que el nombre de la clase, según la regla de traducción, es *Nom*.

$$\frac{\text{fresh}(Nom) \quad a_{Nom} = \text{init_atts}(Nom)}{\begin{array}{c} ob\,(o,a,t)\,tsk\,(t,m,o,l,\{\text{MutexInterface nom= new Nom(); } s\}) \\ \to tsk\,(t,m,o,l,s)\,ob\,(Nom,a_{Nom},\bot) \end{array}}$$

  Efectivamente, observamos que el nuevo objeto se crea sin ninguna tarea en ejecución, por lo que la primera tarea que quiera ejecutarse lo hará. Esto equivale en el lenguaje imperativo a que el primer objeto que intente adquirir el mútex lo consiga, ya que encontrará el , que es el comportamiento en caso de que esté libre.

- **Var_Ident =AExpr $\mapsto$ Var_Ident = AExpr;**

No se especifica la sintaxis de la expresión aritmética *Aexpr*, ya que no afecta a la situación de deadlock. Para reducir la notación vamos a hacer la equivalencia $e \equiv AExpr$. Las semánticas en este caso son:

- Lenguaje imperativo

$$\frac{x \in \text{dom}(L)}{\text{ip}\left(.p, L, \{x = e; s\}\right) \rightarrow \text{ ip}\left(.p, L\left[x \rightarrow \mathcal{A}\left[\!\left[e\right]\!\right]\right], \{s\}\right)}$$

- ABS

$$\frac{x \in \text{dom}(a_{.p}) \quad v = [\![e]\!]_{a.p \circ l}}{ob\left(.p, a_{.p}, t\right) tsk\left(t, m, .p, l, \{x = e; s\}\right) \rightarrow ob\left(.p, a_{.p}[x \mapsto v], t\right) tsk\left(t, m, P, l, s\right)}$$

Lo primero que haremos es bajar el cálculo del valor $v = [\![e]\!]_{a.p \circ l}$ al resultado de la regla. Además, tanto la función $[\![\cdot]\!]_{a.p \circ l}$ como $\mathcal{A}\left[\!\left[\cdot\right]\!\right]$ resuelven el valor de la variable de manera similar en base a los campos globales y variables locales respectivamente, por lo que devolverán un resultado equivalente y podremos intercambiarlas.

Lo último que tenemos que considerar para realizar las transformaciones en las precondiciones de la regla de forma que sean iguales es la equivalencia entre los conjuntos L y $a_{.p}$. Debido a la construcción del conjunto $a_{.p}$, explicada anteriormente en la traducción de declaraciones de variables locales, tenemos que ambos conjuntos tendrán en su inicialización conjuntos equivalentes. Probaremos en este caso la equivalencia de las dos reglas semánticas sobre las que estamos trabajando, ya que en caso de demostrarse para este caso base, los conjuntos seguirán siendo equivalentes y demostrar la ejecución de otra asignación más adelante será simplemente una generalización de esta regla, que parte con la premisa de la equivalencia de ambos conjuntos.

Por tanto, reflejando estas equivalencias en el árbol de derivación de ABS, tenemos

$$\frac{x \in \text{dom}(L)}{ob\left(.p, L, t\right) tsk\left(t, m, .p, l, \{x = e; s\}\right) \rightarrow ob\left(.p, L[x \mapsto \mathcal{A}\left[\!\left[e\right]\!\right]], t\right) tsk\left(t, m, .p, l, s\right)}$$

Aunque tanto en ABS como en el lenguaje imperativo se ha usado una forma de semánticas dividida en dos partes de forma que sea más fácil reflejar sólo la consecuencia en una de ellas, en las demostraciones será conveniente, en algunos casos, unir ambas bajo un mismo nombre para, así, poder trabajar de manera más sencilla. Debido a que esto sólo supone una reagrupación de la notación, no supondrá ningún tipo de contratiempo en la demostración de equivalencia. Para que sea más visual, vamos a adoptar el nombre *ip* en la agrupación en ABS. De esta forma la regla anterior se transforma en

$$\frac{x \in \text{dom}(L)}{\text{ip}\left(.p, L, t, m, l, \{x = e; s\}\right) \rightarrow \text{ ip}\left(.p, L[x \mapsto \mathcal{A}\left[\!\left[e\right]\!\right]], t, m, l, s\right)}$$

En el caso del lenguaje imperativo, $.p$ representa el identificador del proceso en el que nos encontramos. Son procesos sin tareas internas y con ellos como único método. En la traducción a ABS, dividimos lo que engloba $.p$ en el identificador de proceso $.p$,

la tarea t que se está ejecutando en ese momento y el método m que codifica la parte de la traducción correspondiente al nivel de anidamiento de mútex del programa del lenguaje imperativo en que nos encontramos. Estos tres parámetros, en el caso particular de la asignación, son equivalentes a $.p$ en el lenguaje imperativo, ya que en cada momento tenemos a lo sumo una tarea a ejecutar en el objeto con un cierto método. Esto se debe a que cada objeto concurrente codifica un proceso y mediante las llamadas a distintos métodos, cada uno ejecutado por una tarea, conseguimos reflejar el flujo de ejecución del programa original. De esta forma, tenemos que sólo hay una posible parte de $.p$ codificada en un conjunto $\text{ip}(\cdot)$ como el anterior y, por tanto, tenemos la equivalencia. Si abusamos de la notación, podemos agrupar en este caso $\{P, t, m\}$ en un identificador único $.p$.

Obtenemos entonces el árbol de derivación

$$\frac{x \in \text{dom}(L)}{\text{ip}(.p, L, l, \{x = e; s\}) \rightarrow \text{ip}(.p, L[x \mapsto \mathcal{A}[\![e]\!]], l, s)}$$

Lo único que falta para obtener el árbol que buscamos es introducir la notación $\{s\}$ para las sentencias de código en todos los casos (que es sólo una cuestión notacional) y eliminar la dependencia de l. Esta última dependencia se puede eliminar directamente, ya que por la forma de la traducción nunca habrá variables enteras locales en los métodos (todas se traducen en campos de los objetos), lo cual hará que en todos los casos $l \cap \{\text{Variables de tipo entero}\} = \emptyset$ y, por tanto, lo podemos eliminar de la regla al no ser influyente.

Hemos llegado así al árbol de derivación buscado:

$$\frac{x \in \text{dom}(L)}{\text{ip}(.p, L, \{x = e; s\}) \rightarrow \text{ip}(.p, L[x \mapsto \mathcal{A}[\![e]\!]], \{s\})}$$

- **Global_Var_Ident = AExpr; $\mapsto$ globals.setGlobal_Var_Ident(**$AExpr$**)**

  La llamada en este caso al método de *Globals* desde otro objeto es síncrona. Estudiamos por tanto el efecto que tendrá la ejecución en la clase *Globals*, ya que en el objeto concurrente el flujo de ejecución se detendrá hasta que el método retorne. Esto implica que basta ver que los valores se modifican de forma equivalente, para luego continuar el flujo del programa en el método del objeto concurrente llamante desde donde estaba previamente, ya que en ambos casos el cambio de estado tras la instrucción sólo se ve reflejado en el cambio de valor del campo asociado a la variable.

  De nuevo, usamos $e \equiv AExpr$. Las semánticas son:

  - Lenguaje imperativo

$$\frac{x \in \text{dom}(G)}{\begin{array}{l}\text{gl}(M, G, P)\ \ \text{ip}(.p, L, \{x = e; s\}) \rightarrow \\ \qquad\qquad \text{gl}(M, G[x \rightarrow \mathcal{A}[\![e]\!]], P)\ \ \text{ip}(.p, L, \{s\})\end{array}}$$

- ABS

$$\frac{x \in \text{dom}(a_{Globals}) \quad v = [\![e]\!]_{a_{Globals} \circ l}}{\begin{array}{r} ob\,(Globals, a_{Globals}, t)\, tsk\,(t, m, Globals, l, \{x = e; s\}) \rightarrow \\ ob\,(Globals, a_{Globals}[x \mapsto v], t)\, tsk\,(t, m, Globals, l, s) \end{array}}$$

En primer lugar, vemos que $[\![\cdot]\!]_{a_{Globals} \circ l}$ da el mismo valor que $\mathcal{A}\,[\![\cdot]\!]$. En este caso se ve la equivalencia entre G y a_G de forma más clara, ya que en ambos casos representan las variables globales (en el primer caso al programa en el lenguaje imperativo y en el segundo al objeto correspondiente en ABS). Además, vamos a agrupar la notación como se hizo anteriormente bajo el mismo nombre en ambas semánticas.

Adaptándonos al tipo de semántica que habíamos definido, podemos eliminar de la información del lenguaje imperativo los identificadores M, P y L, ya que en el caso de esta instrucción no sufren ningún cambio y, por tanto, son irrelevantes en la representación del cambio de estado del programa.

De esta forma, obtenemos

- Lenguaje imperativo

$$\frac{x \in \text{dom}(G)}{\text{ip}\,(.p, G, \{x = e; s\}) \rightarrow \ \text{ip}\,(.p, G\,[x \rightarrow \mathcal{A}\,[\![e]\!]]\,, \{s\})}$$

- ABS

$$\frac{x \in \text{dom}(G)}{\text{ip}\,(Globals, G, t, m, l, \{x = e; s\}) \rightarrow \ \text{ip}\,(Globals, G[x \mapsto \mathcal{A}\,[\![e]\!]], t, m, l, s)}$$

Vamos a transformar el árbol de derivación de ABS para obtener el árbol de derivación del lenguaje imperativo. Sin embargo, lo único que tenemos que observar es que, en el caso del objeto *Globals*, no hay variables locales a ningún método, luego $l = \emptyset$ y se puede prescindir de este identificador.

Además, en el caso del lenguaje imperativo tenemos $.p$ para referirnos al identificador del proceso, mientras que en ABS lo dividimos en su identificador (*Globals*), la tarea que se está ejecutando y el método que ejecuta esta tarea, que en este caso es *setGlobal_Var_Ident($\mathcal{A}\,[\![e]\!]$)*. En este caso, para ver la equivalencia, vamos a ver que se mantiene el flujo lineal de ejecución del programa. Esto se comprueba en los siguientes puntos:

- La llamada al método de la clase *Globals* desde el método correspondiente del objeto que codifica el programa es síncrona, lo cual evita que el programa pueda seguir avanzando mientras esta instrucción espera a ser ejecutada.
- Aunque puede haber varias tareas a la vez para ejecutarse en el objeto de las variables globales, no habrá bloqueos. Esto se debe a que sólo hay una instrucción en cada uno de los métodos del objeto y ninguna de ellas es bloqueante, por lo que las tareas a ejecutarse en el objeto en algún momento lo harán. Aunque puede que se escriba después de otras tareas de otros métodos y no en el momento de la llamada, es un comportamiento similar a las carreras de datos que se pueden producir en los lenguajes imperativos cuando no se utilizan métodos de control de concurrencia.

Tenemos por tanto que, en cierto estado del programa, $.p$ es equivalente a la representación que nos dan $Globals$, t y m, por lo que podemos sustituirlo.

Ajustando la notación y reflejando estos últimos cambios llegamos al árbol de derivación buscado:

$$\frac{x \in \mathrm{dom}(G)}{\mathrm{ip}\,(.p, G, \{x = e; s\}) \to \ \mathrm{ip}\,(.p, G[x \mapsto \mathcal{A}\,[\![e]\!]], l, s)}$$

Observación 4.2. En el caso de las variables globales, se ha mostrado el uso del *setGlobal_Var_Ident()* pero no del *getGlobal_Var_Ident()*, ya que se utiliza en la composición de las expresiones aritméticas, en las que no hemos centrado el análisis por no suponer ningún cambio en los resultados de deadlock.

- **; $\mapsto$ skip;**

  En este caso, la semántica asociada en el lenguaje imperativo sería la de *skip*.

  - Lenguaje imperativo

  $$\mathrm{ip}\,(.p, L, \{\text{ skip };s\}) \to \ \mathrm{ip}\,(.p, L, \{s\})$$

  - ABS

  $$ob\,(.p, a_{.p}, t)\, tsk\,(t, m, .p, l, \{\text{ skip };s\}) \to tsk\,(t, m, .p, l, s)$$

  Agrupamos los términos en ABS bajo el mismo nombre, y eliminamos en ambos lenguajes las referencias que, de forma clara, no sufren cambios, ya que no influyen en la equivalencia; estudiaremos el resto de identificadores. Tenemos así

  - Lenguaje imperativo

  $$\mathrm{ip}\,(.p, \{\text{ skip };s\}) \to \ \mathrm{ip}\,(.p, \{s\})$$

  - ABS

  $$\mathrm{ip}\,(.p, t, m, \{skip; s\}) \to \ \mathrm{ip}\,(.p, t, m, s)$$

  De nuevo tenemos que $.p$, t y m denotan en qué parte de $.p$ nos encontramos. Como sólo tenemos una tarea ejecutando a la vez en el objeto concurrente, un único método que se puede ejecutar en cada parte correspondiente de $.p$ y tanto la traducción como la ejecución son secuenciales, podemos sustituirlos.

  Ajustando la notación, llegamos en ABS al árbol de derivación buscado:

  $$\mathrm{ip}\,(.p, \{skip; s\}) \to \ \mathrm{ip}\,(.p, \{s\})$$

- **if (AExpr) {codigoIf} else {codigoElse}**
  $\mapsto$ if ($AExpr \neq 0$){ τ(codigoIf)} else {τ(codigoElse)}

  Tenemos que distinguir dos casos, en función del valor de $AExpr$. Es importante observar que, en estos casos, coinciden en ambos lenguajes la evaluación de la condición del if en la traducción. Es decir, que tendremos que evaluar la equivalencia de los casos de $\left[\text{ if }^{\text{true}}\right]$ por un lado y de $\left[\text{ if }^{\text{false}}\right]$ por el otro.

- $AExpr \neq 0$

  De nuevo usamos $e = AExpr$. Las semánticas que usaremos en este caso son:

  - Lenguaje imperativo

$$\frac{\mathcal{B}\,[\![e]\!]}{\text{ip}\,(.p, L, \{\text{ if } e \text{ then } s_1 \text{ else } s_2; s\}) \to \text{ ip}\,(.p, L, \{s_1; s\})}$$

  - ABS

$$\frac{[\![e \neq 0]\!]_{a.p \circ l}}{\begin{array}{c} ob\,(.p, a_{.p}, t)\, tsk\,(t, m, .p, l, \{\text{ if } (e \neq 0) \text{ then } \tau\,(s_1) \text{ else } \tau\,(s_2)\,; \tau\,(s)\}) \\ \to tsk\,(t, m, .p, l, \{\tau\,(s_1)\,; \tau\,(s)\}) \end{array}}$$

  Es equivalente poner $\mathcal{B}\,[\![e]\!]$ que $[\![(e \neq 0)]\!]_{a.p \circ l}$, podemos intercambiarlos para tener la misma precondición de las reglas. Vamos a quedarnos con $\mathcal{B}\,[\![e]\!]$ y e, para poder poner en el resultado de las reglas únicamente e y reducir su extensión. Podemos poner e indistintamente también en ABS porque va a ser evaluado dando el mismo resultado al aplicar la regla.

  Además, agrupamos términos como hemos hecho anteriormente. La regla del lenguaje imperativo no sufre cambios, la de ABS se transforma en:

$$\frac{\mathcal{B}\,[\![e]\!]}{\begin{array}{c} \text{ip}\,(.p, a_{.p}, t, m, l, \{\text{ if } e \text{ then } \tau\,(s_1) \text{ else } \tau\,(s_2)\,; \tau\,(s)\}) \\ \to \text{ ip}\,(.p, a_{.p}, t, m, l, \{\tau\,(s_1)\,; \tau\,(s)\}) \end{array}}$$

  A la hora de comprobar la equivalencia, en el caso del lenguaje imperativo L no sufre cambios, al igual que no sufren cambios $a_{.p}$ ni l en ABS. Podemos por tanto, estudiar la equivalencia en las reglas sin fijarnos en estos campos. Buscamos, por tanto, la equivalencia entre

$$\frac{\mathcal{B}\,[\![e]\!]}{\text{ip}\,(.p, \{\text{ if } e \text{ then } s_1 \text{ else } s_2; s\}) \to \text{ ip}\,(.p, \{s_1; s\})}$$

  y

$$\frac{\mathcal{B}\,[\![e]\!]}{\text{ip}\,(P, t, m, \{\text{ if } e \text{ then } \tau\,(s_1) \text{ else } \tau\,(s_2)\,; \tau\,(s)\}) \to \text{ ip}\,(P, t, m, \{\tau\,(s_1)\,; \tau\,(s)\})}$$

  Por tanto, lo único que tenemos que hacer para completar la demostración de la equivalencia es, como anteriormente, encontrar la equivalencia entre $\{.p, t, m\}$. Por la misma razón que en la regla anterior, podemos sustituirlo por $.p$, ya que el código de éste se traduce en los distintos métodos del objeto concurrente $.p$ en ABS, cada uno conteniendo la traducción de una parte disjunta del programa original. Además, sólo tendremos una tarea ejecutándose a la vez, que ejecuta el método correspondiente al flujo de ejecución actual del programa original. Por tanto, podemos sustituir las variables como hemos hecho en otros casos, obteniendo el árbol de derivación buscado a partir del de ABS:

$$\frac{\mathcal{B}\,[\![e]\!]}{\text{ip}\,(.p, \{\text{ if } e \text{ then } \tau\,(s_1) \text{ else } \tau\,(s_2)\,; \tau\,(s)\}) \to \text{ ip}\,(.p, \{\tau\,(s_1)\,; \tau\,(s)\})}$$

Cabe remarcar que, en el caso de la instrucción *if*, y tanto en este caso como el siguiente, podemos detener la demostración de equivalencia en este punto, aunque en el caso de un árbol tengamos s_1 y en el otro $\tau(s_1)$. Si demostramos la equivalencia de la traducción en los casos básicos, las traducciones que engloben más de una traducción se podrán descomponer, de forma que se puedan demostrar de forma individual como se está haciendo ahora mismo. Esta demostración no da problemas debido a que nuestra traducción es secuencial, por lo que basta aplicar el siguiente lema.

Lema 4.2. *Sean* $s1$, *y* $s2$ *dos sentencias en el lenguaje imperativo, cuyos homólogos en ABS son* $\tau(s_1)$ *y* $\tau(s_2)$ *respectivamente. Entonces, la traducción de* $s_1; s_2$ *será equivalente en ABS a* $\tau(s_1); \tau(s_2)$.

Demostración. En ambos casos se parte de la premisa de dos estados de partida equivalentes

$$\langle S_{11}; S_{12}, s_1\rangle \to s_1'' \qquad \langle S_{21}; S_{22}, s_2\rangle \to s_2'', \qquad s_1 \equiv s_2$$

Podemos dar un paso en la semántica operacional, de forma que se ejecute la primera sentencia en ambos casos y, si se ha demostrado que los casos básicos son equivalentes, entonces tendremos

$$\langle S_{12}, s_1'\rangle \to s_1'' \qquad \langle S_{22}, s_2'\rangle \to s_2'', \qquad s_1' \equiv s_2'$$

Aplicando de nuevo la equivalencia de la traducción de casos básicos, llegamos a que $s_1'' \equiv s_2''$. ☐

- $AExpr = 0$

  Con $e \equiv AExpr$, las reglas semánticas son:

  - Lenguaje imperativo:

$$\frac{\neg\mathcal{B}[\![e]\!]}{\text{ip}\left(.p, L, \{\text{ if } e \text{ then } s_1 \text{ else } s_2; s\}\right) \to \text{ ip}\left(.p, L, \{s_2; s\}\right)}$$

  - ABS

$$\frac{[\![e=0]\!]_{a_p \circ l}}{\begin{array}{l} ob\left(.p, a_{.p}, t\right) tsk\left(t, m, .p, l, \{\text{ if } (e \neq 0) \text{ then } \tau(s_1) \text{ else } \tau(s_2); \tau(s)\}\right) \\ \quad \to tsk\left(t, m, .p, l, \{\tau(s_2); \tau(s)\}\right)\end{array}}$$

Con un razonamiento similar al realizado en el caso anterior del *if*, se pueden realizar los mismos pasos de transformación de un árbol en el otro. En un primer lugar unificamos, igualamos las precondiciones y eliminamos de la notación los términos que no influyen al reflejar el cambio de estado:

  - Lenguaje imperativo:

$$\frac{\neg\mathcal{B}[\![e]\!]}{\text{ip}\left(.p, \{\text{ if } e \text{ then } s_1 \text{ else } s_2; s\}\right) \to \text{ ip}\left(.p, \{s_2; s\}\right)}$$

◦ ABS

$$\frac{\neg\mathcal{B}\,[\![e]\!]}{\begin{array}{l}\mathrm{ip}\,(.p,t,m,\{\ \text{if } e \text{ then } \tau\,(s_1)\ \text{ else } \tau\,(s_2)\,;\tau\,(s)\}) \\ \quad\to\ \mathrm{ip}\,(.p,t,m,\{\tau\,(s_2)\,;\tau\,(s)\})\end{array}}$$

Basta entonces fijarnos en que la ejecución y codificación es igual al apartado anterior salvo por la evaluación de la condición del *if* para, así, poder sustituir en el segundo caso las variables $.p$, m y t por $.p$, de forma que llegamos al resultado buscado:

$$\frac{\neg\mathcal{B}\,[\![e]\!]}{\mathrm{ip}\,(.p,\{\ \text{if } e \text{ then } \tau\,(s_1)\ \text{ else } \tau\,(s_2)\,;\tau\,(s)\})\to\ \mathrm{ip}\,(.p,\{\tau\,(s_2)\,;\tau\,(s)\})}$$

- **while(AExpr) {codigoWhile}** $\mapsto$ **while** $(\mathcal{B}\,[\![AExpr]\!])$**{**$\tau$ (**codigoWhile**)**}**

  De nuevo, vamos a estudiar los dos casos de la precondición del while. Como siempre, fijamos $e \equiv AExpr$.

  - $e \neq 0$

    Las reglas semánticas en este caso son:

    ◦ Lenguaje imperativo

$$\frac{\mathcal{B}\,[\![e]\!]}{\mathrm{ip}\,(.p,L,\{\ \text{while } e \text{ do } s_1;s\})\to\ \mathrm{ip}\,(.p,L,\{s_1;\ \text{while } e \text{ do } s_1;s\})}$$

    ◦ ABS

$$\frac{[\![e\neq 0]\!]_{a_{.p}\circ l}}{\begin{array}{l}ob\,(.p,a_{.p},t)\,tsk\,(t,m,.p,l,\{\ \text{while }\ (e\neq 0)\ \text{ do } \tau\,(s_1)\,;\tau\,(s)\}) \\ \quad\to tsk\,(t,m,.p,l,\{\tau\,(s_1)\,;\ \text{while }\ (e\neq 0)\ \text{ do } \tau\,(s_1)\,;\tau\,(s)\})\end{array}}$$

    Vamos a proceder en principio como en el *if/else*, ya que la comprobación de condiciones y los campos modificados son similares. Empezamos sustituyendo $\mathcal{B}\,[\![\cdot]\!]$ en las precondiciones para ambas, unificando términos y evitando notación redundante de conjuntos que no sufren cambios y no tienen una importancia relevante en la representación del cambio de estado, como son L, $a_{.p}$ y l. De esta forma, tenemos las reglas

    ◦ Lenguaje imperativo

$$\frac{\mathcal{B}\,[\![e]\!]}{\mathrm{ip}\,(.p,\{\ \text{while } e \text{ do } s_1;s\})\to\ \mathrm{ip}\,(.p,\{s_1;\ \text{while } e \text{ do } s_1;s\})}$$

    ◦ ABS

$$\frac{\mathcal{B}\,[\![e]\!]}{\begin{array}{l}\mathrm{ip}\,(.p,t,m,\{\ \text{while } e \text{ do } \tau\,(s_1)\,;\tau\,(s)\}) \\ \quad\to\ \mathrm{ip}\,(.p,t,m,\{\tau\,(s_1)\,;\ \text{while } e \text{ do } \tau\,(s_1)\,;\tau\,(s)\})\end{array}}$$

    Aunque en este caso tenemos un bucle y por tanto el retorno al punto de comienzo, las condiciones en ambos casos son equivalentes, por lo que se puede considerar la ejecución como si fuera secuencial. Por tanto, volvemos a tener el caso

en que $.p$ nos da en el lenguaje imperativo el nombre del proceso ejecutándose, de manera similar a como lo refleja t en el programa en ABS, ayudándose del método m que codifica la parte del programa correspondiente y el objeto en que nos encontramos $.p$. Si los sustituimos, obtenemos el árbol de derivación que buscábamos salvo por la equivalencia entre s_1 y $\tau(s_1)$, pero para esta parte basta aplicar el lema 4.2.

$$\frac{\mathcal{B}[\![e]\!]}{\begin{array}{l}\mathrm{ip}\left(.p,\{\text{ while } e \text{ do } \tau(s_1);\tau(s)\}\right)\\ \qquad\qquad \rightarrow \ \mathrm{ip}\left(.p,\{\tau(s_1);\text{ while } e \text{ do } \tau(s_1);\tau(s)\}\right)\end{array}}$$

- $e = 0$

  Las reglas semánticas son

  - Lenguaje imperativo

  $$\frac{\neg\mathcal{B}[\![e]\!]}{\mathrm{ip}\left(.p,L,\{\text{ while } e \text{ do } s_1;s\}\right)\rightarrow \ \mathrm{ip}\left(.p,L,\{s\}\right)}$$

  - ABS

  $$\frac{[\![e=0]\!]_{a.p\circ l}}{\begin{array}{l} ob\left(.p,a_{.p},t\right) tsk\left(t,m,.p,l,\{\text{ while } (e\neq 0) \text{ do } \tau(s_1);\tau(s)\}\right)\\ \qquad \rightarrow tsk\left(t,m,.p,l,\{\tau(s)\}\right)\end{array}}$$

  De forma similar a como hicimos antes, eliminamos la información de variables que no cambian y por tanto no influyen en la configuración del estado, agrupamos bajo el mismo término e igualamos las precondiciones, obteniendo así

  - Lenguaje imperativo

  $$\frac{\neg\mathcal{B}[\![e]\!]}{\mathrm{ip}\left(.p,\{\text{ while } e \text{ do } s_1;s\}\right)\rightarrow \ \mathrm{ip}\left(.p,\{s\}\right)}$$

  - ABS

  $$\frac{\neg\mathcal{B}[\![e]\!]}{\mathrm{ip}\left(P,t,m,\{\text{ while } e \text{ do } \tau(s_1);\tau(s)\}\right)\rightarrow \ \mathrm{ip}\left(P,t,m,\{\tau(s)\}\right)}$$

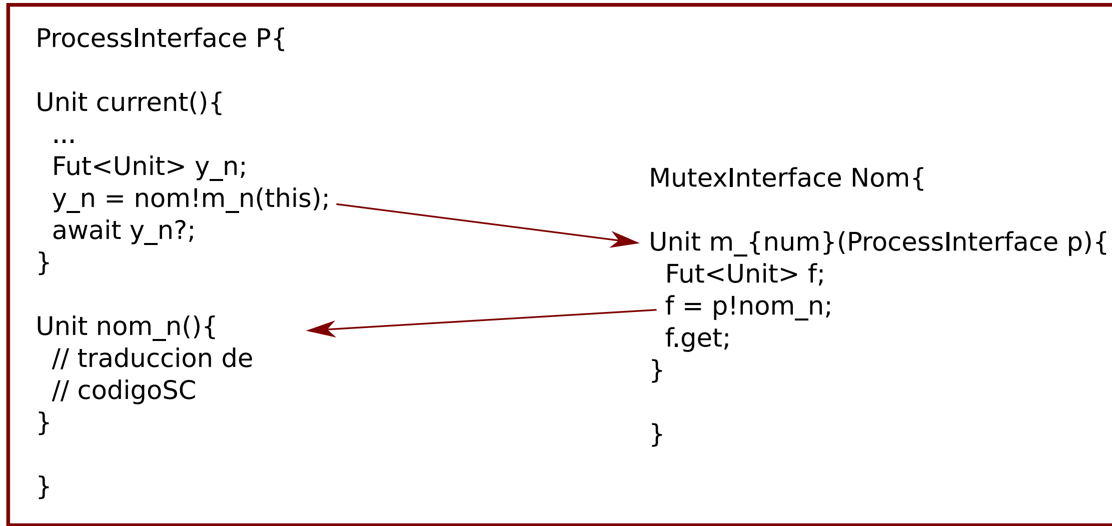
  y realizando el mismo razonamiento para los identificadores del proceso de los casos anteriores, tenemos el árbol de derivación

  $$\frac{\neg\mathcal{B}[\![e]\!]}{\mathrm{ip}\left(.p,\{\text{ while } e \text{ do } \tau(s_1);\tau(s)\}\right)\rightarrow \ \mathrm{ip}\left(.p,\{\tau(s)\}\right)}$$

  que, debido a la equivalencia de la traducción de las instrucciones del bloque interno, es también equivalente.

- **lock^n(Var_Ident); codigoSC unlock^n(Var_Ident);**
  $\mapsto$ ***writeProc(Var_Ident_n,*** τ(***codigo***)***)***
  Fut<Unit> y_n;
  y_n = nom!m_n(this);
  await y_n?;

  La función *writeProc* introduce en el método *nom_n* la traducción correspondiente a las instrucciones del bloque interno a *lock/unlock*. Tras la ejecución de este proceso, y la escritura de las correspondientes instrucciones en el método del proceso en el que estamos, tendremos un esquema de código con sus correspondientes llamadas de la forma



En la primera flecha, tenemos una llamada asíncrona al método del proceso *MutexInterface Nom* que es común tanto al caso del que en el objeto Nom se esté ejecutando otra tarea (traducción del caso de $\left[\text{ lock}^{\text{ false}}\right]$) como al caso en que no se esté ejecutando ninguna tarea (traducción de $\left[\text{ lock}^{\text{ true}}\right]$). Ambos casos esperarán en la llamada *await*, lo cual hace que se mantenga el flujo de ejecución del programa si consideramos todo el bloque de la sección crítica como una instrucción más. De esta forma, tenemos que pasar a demostrar que efectivamente en la llamada a *Nom* se traduce adecuadamente este bloque.

La regla semántica asociada a esta primera llamada asíncrona en ABS es

$$\frac{Nom = [\![nom]\!]_{a.p\circ l} \neq null \quad l_1 = \text{buildLocals}(this, m_n) \quad \text{fresh}(t_1)}{\begin{array}{l} ob\,(.p, a_{.p}, t)\, tsk\,(t, m, .p, l, \{y_n = nom!m_n(this); await\, y_n?; s\}) \\ \quad \to tsk\,(t, m, .p, l\,[y_n \mapsto t_1]\,, \{await\, y_n?; s\}) \\ \qquad\qquad tsk\,(t_1, m_n, Nom, l_1, \text{body}(m_n)) \end{array}}$$

Después de esto, mientras no se ejecute completamente el método de *Nom*, en este método se mantendrá en ABS la regla semántica de $\left[\text{await }^{\text{false}}\right]$, que viene dada por

$$\frac{[\![y_n]\!]_{a.p\circ l} = t_1 \qquad s_1 \neq \epsilon(v)}{\begin{array}{l} ob\,(.p, a_{.p}, t)\, tsk\,(t, m, .p, l, \{\text{await } y_n?; s\})\, tsk\,(t_1, m_1, o_1, l_1, s_1) \\ \qquad \to ob\,(.p, a_{.p}, \bot) \end{array}}$$

De esta forma, mediante *await* aseguramos que el *lock* del objeto se libera, de forma que se pueda ejecutar el método *nom_n* en la llamada de Nom al objeto concurrente $.p$, sin producirse por ello un bloqueo.

Tenemos que estudiar entonces los dos casos de la instrucción *lock* a partir de este punto en el método *Nom*. Estudiamos primero el caso equivalente a $\left[\text{ lock}^{\text{ false}}\right]$ ya que, como veremos luego, se necesita en la demostración de $\left[\text{ lock}^{\text{ true}}\right]$.

- $\left[\text{ lock}^{\text{ false}}\right]$

  La semántica en el lenguaje imperativo viene dada por

$$\frac{M\,[nom] \neq \text{null}}{\begin{array}{l}\text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p, L, \{\text{ lock}(nom); s\}) \rightarrow \\ \qquad\qquad \text{gl}\,(M, wait(nom, .p, W), G, P)\ \ \text{ip}\,(.p, L, \{\text{ lock}(nom); s\})\end{array}}$$

  y, por cómo se ha realizado la traducción, esto equivale en ABS a la llamada asíncrona a un método sobre el objeto *Nom* cuando ya hay otra tarea ejecutándose.

  Se intenta la activación de la tarea cuando ya hay otra activada, por lo que efectivamente no se produce progresión y el método se queda bloqueado en su ejecución hasta que se libere el mútex implícito de *Nom*. Aunque se libera el mútex del primer objeto mientras se espera, no se ejecutará nada que cambie el estado del objeto concurrente mientras no se empiece a ejecutar en *Nom* la tarea deseada, debido a que cada objeto concurrente que codifica un proceso está destinado a codificar un único flujo de ejecución y, por tanto, en cada instrucción del código original nos encontramos con una biyección con la instrucción y posición del código de ABS correspondiente.

  Podemos ver la regla semántica asociada al intento de activación cuando hay otro método ejecutándose para comprobar que, efectivamente, no se produce ningún cambio de estado y por tanto el programa se bloquea, con lo que finalizamos la desmotración de este caso

$$\frac{s \neq \epsilon\,(v)}{\begin{array}{l} ob\,(Nom, a_{Nom}, t_{Nom})\, tsk\,(t, m, Nom, l, s) \\ \qquad \rightarrow ob\,(Nom, a_{Nom}, t_{Nom})\, tsk\,(t, m, Nom, l, s)\end{array}}$$

  donde t_{Nom} identifica a la tarea que se está ejecutando en *Nom* en ese momento.

  En el caso del lenguaje imperativo, tendríamos el caso de *acquire* cuando el mútex aún no está libre, caso en el que tampoco tenemos progresión y en que tenemos un estado equivalente al de ABS.

$$\frac{M\,[nom] = .p1}{\begin{array}{l}\text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p1, L_1, \{s_1\})\ \ \text{ip}\,(.p, L, \{\text{ lock}(nom); s\}) \\ \qquad \rightarrow\ \text{gl}\,(M, W, G, P)\ \ \text{ip}\,(.p1, L_1, \{s_1\})\ \ \text{ip}\,(.p, L, \{\text{ lock}(nom); s\})\end{array}}$$

- $\left[\text{ lock}^{\text{ true}}\right]$

En este caso la semántica en el lenguaje imperativo viene dada por

$$\frac{M\,[nom] = \text{null}}{\begin{array}{l}\text{gl}\,(M, W, G, P)\;\; \text{ip}\,(.p, L, \{\; \text{lock}(nom); s\})\\ \qquad\qquad \to\;\; \text{gl}\,(M[nom \mapsto .p], W, G, P)\;\; \text{ip}\,(.p, L, \{s\})\end{array}}$$

y vamos a ver que esto es equivalente a la activación de la tarea correspondiente en el método *Nom*, que vendría dada por

$$\frac{t_1 = \text{select}\,(q) \qquad \tau(s) \neq \epsilon\,(v)}{ob\,(Nom, a_{Nom}, \bot)\, tsk\,(t_1, m_n, Nom, l, \tau(s)) \to ob\,(Nom, a_{Nom}, t_1)}$$

Aunque en este caso no hay consumición de ninguna instrucción en el caso de ABS, lo que nos importa es que comienza a ejecutarse la tarea del método, por lo que se adquiere el mútex implícito del objeto *Nom*. Como la ejecución no es especulativa, hasta que el proceso de *Nom* en que estamos no suelte el lock, ningún otro se podrá ejecutar. Esto, como veremos después, sólo sucederá cuando el método termine, por lo que podemos buscar la equivalencia con el lenguaje imperativo en la activación.

Vamos a estudiar esta equivalencia por el hecho de que, cuando se ejecuta la instrucción *lock* en el lenguaje imperativo y el mútex no está cogido, se indica en M que el mútex pasa a estar cogido, por lo que ningún otro proceso podrá cogerlo hasta que el que lo tiene lo libere explícitamente mediante una instrucción *unlock*. De nuevo nos encontramos ante una ejecución no especulativa.

Al contrario que en las otras demostraciones, en este caso no vamos a intentar demostrar que se puede obtener el árbol de derivación de un caso a partir del otro, sino que si intentamos ejecutar otra instrucción *lock* sobre el mismo mútex en el primer caso o intentamos ejecutar otra tarea en *Nom* en el segundo caso, en ambos nos quedamos bloqueados. Si comprobamos esto, bastará con comprobar más tarde que la liberación del mútex mediante *unlock* en el primer caso equivaldrá a liberar el mútex del objeto en el segundo caso. Sin embargo, este caso al que nos referimos es el de $\left[\; \text{lock}^{\,\text{false}} \right]$, que se ha probado previamente y que, efectivamente, no permite ejecutar ninguna instrucción.

Lo siguiente que se hace tras adquirir el lock del objeto es realizar una llamada asíncrona al método *nom_n* del objeto P.

$$\frac{.p = [\![p]\!]_{a_{Nom} \circ l} \neq null \quad l_1 = \text{buildLocals}(nom_n) \quad \text{fresh}(t_1)}{\begin{array}{l} ob\,(Nom, a_{Nom}, t)\, tsk\,(t, m_n, Nom, l, \{f = p!nom_n(); f.get;\})\\ \quad \to tsk\,(t, m, Nom, l\,[f \mapsto t_1]\,, \{f.get;\})\\ \qquad\qquad tsk\,(t_1, nom_n, .p, l_1, \text{body}(nom_n))\end{array}}$$

El método *nom_n* contendrá la codificación correspondiente a *codigoSC* que, como hemos referido anteriormente, es equivalente a la traducción de instrucciones más simples. Esto implica que, en el caso de la traducción del código interno, podemos asumir que la regla se ha aplicado a cada instrucción o bloque de instrucciones y la ejecución interna es correcta de acuerdo al flujo que

estamos demostrando. De esta forma, si no se produce ningún deadlock, llegará un momento en que el método termine.

Durante la ejecución del método, el mútex implícito de *Nom* no se libera debido a la instrucción *f.get*, por lo que efectivamente se deniega el acceso al recurso compartido al resto de procesos que intentan ejecutar. Este bloqueo viene reflejado en la semántica de $\left[\text{get}^{\text{ false}}\right]$, que es el caso con el que trabajamos hasta que *nom_n* no haya terminado

$$\frac{t_1 = [\![f]\!]_{a_{Nom}\circ l} \qquad s_1 \neq \epsilon(\varnothing)}{\begin{array}{l} ob\,(Nom, a_{Nom}, t)\, tsk\,(t, m_n, Nom, l, \{f.get\})\, tsk\,(t_1, nom_n, .p, l_1, s_1) \\ \quad \to ob\,(Nom, a_{Nom}, t)\, tsk\,(t, m_n, Nom, l, \{f.get\})\, tsk\,(t_1, nom_n, .p, l_1, s_1) \end{array}}$$

en donde efectivamente comprobamos que no avanzamos en el proceso ni liberamos el mútex.

Si todo funciona bien en la ejecución de la traducción de las instrucciones internas, llegará un momento en que en el proceso *nom_n* tendremos, usando la notación anterior, $s_1 = \epsilon\,(\varnothing)$. Ya habremos ejecutado todo el *codigoSC*, y pasaremos a tener el caso de $[\text{get}^{\text{ true}}]$, con la regla semántica

$$\frac{t_1 = [\![f]\!]_{a_{Nom}\circ l}}{\begin{array}{l} ob\,(Nom, a_{Nom}, t)\, tsk\,(t, m_n, Nom, l, \{f.get;\})\, tsk\,(t_1, nom_n, .p, l_1, \epsilon(\varnothing)) \\ \qquad \to tsk\,(t, m_n, o, l, \{\}) \end{array}}$$

donde cabe remarcar que, como *f.get* era la última instrucción del método, no nos queda código. El método es de tipo Unit, por lo que esto es equivalente a tener un return implícito

$$\begin{array}{l} ob\,(Nom, a_{Nom}, t)\, tsk\,(t, m_n, Nom, l, \{\}) \\ \qquad \to ob\,(o, a_{Nom}, \bot)\, tsk\,(t, m_n, Nom, l, \epsilon(\varnothing)) \end{array}$$

y volvemos a no tener ninguna tarea ejecutándose en Nom, por lo que de nuevo podríamos comenzar en $\left[\text{ lock}^{\text{ true}}\right]$ con otro proceso que estuviese esperando en $\left[\text{ lock}^{\text{ false}}\right]$ (mediante *acquire*) o que hiciese lock sobre *Nom* posteriormente (mediante *lock*). Qué proceso se ejecuta de los que estén esperando se decide de forma no determinista, se garantiza que si no hay bloqueo un proceso que esté esperando eventualmente entrará, pero no en qué orden lo hará.

Como tenemos una relación unívoca entre cada mútex y el objeto concurrente que lo codifica en la traducción, a lo sumo una tarea se puede ejecutar en el objeto concurrente y ya se ha visto no se deja de ejecutar la tarea en la que estemos hasta que se codifique todo el cuerpo correspondiente al bloque interno del *lock/unlock*, donde cada tarea implica la ejecución de un método llamado asíncronamente desde otro objeto concurrente. Hemos llegado, por tanto, a que efectivamente el comportamiento es equivalente al lenguaje imperativo, donde a lo sumo un proceso puede tener el lock cada vez.

Lo único que nos queda por remarcar en este caso para finalizar la demostración de equivalencia es ver qué sucede en el *await* que esperaba a la primera llamada

asíncrona, donde efectivamente podemos comprobar que el flujo del programa seguirá su ejecución normal.

$$\frac{[\![y_n]\!]_{a.p \circ l} = t_1}{ob\,(.p, a_{.p}, t)\, tsk\,(t, m, .p, l, \{\text{await } y_n?; s\})\, tsk\,(t_1, m_n, Nom, l_1, \epsilon(\emptyset)) \\ \to tsk\,(t, m, .p, l, s)}$$

Además, se puede observar en este caso que, como un objeto concurrente que codifica un proceso del lenguaje imperativo tendrá, a lo sumo, una tarea esperando a ejecutarse en cada momento, este flujo se retomará nada más terminar el método *m_n* en *Nom*.

En el caso del lenguaje imperativo, la semántica del *unlock* está dada por

$$\begin{aligned} &\text{gl}\,(M, W, G, P)\ \text{ip}\,(.p, L, \{\ \text{unlock}(nom); s\}) \\ &\qquad\qquad \to\ \text{gl}\,(M[nom \to \text{null}], W, G, P)\ \ \text{ip}\,(.p, L, \{s\}) \end{aligned}$$

donde poner *nom* a nulo equivale a liberar el mútex de *Nom* en ABS y la progresión de las instrucciones se corresponde con la continuación de la ejecución del flujo del programa tras el *await*.

- **Proc_Ident= new Proc_name(args);**
  ↦ ProcessInterface Proc_Ident =
  new Proc_Name(*newArgs(args,Proc_Name)*);
  Fut<Unit> futName = Proc_Ident!start();

  En el caso del lenguaje imperativo, el *new* está asociado a la creación de un nuevo proceso, mientras que en ABS está asociada a la creación de un nuevo objeto concurrente. Además, en ABS lanzamos explícitamente el método *start*, que es el punto de entrada a la ejecución de la traducción de las instrucciones del proceso. Veamos la sintaxis en ambos casos:

  - Lenguaje imperativo

$$\begin{aligned} &\text{gl}\,(M, G, P)\ \text{ip}\,(.p1, L_1, \{.p2 =\ \text{new}\ P_2(args); s\}) \\ &\qquad\qquad \to\ \text{gl}\,(M, G, \text{add}(.p2, P))\ \ \text{ip}\,(.p1, L_1, \{s\}) \\ &\qquad\qquad\quad \text{ip}\,(.p2, \text{init}(P_2, args), \text{cod}(P_2)) \end{aligned}$$

  - ABS. Vamos a denotar $ar = newArgs(args, Proc_Name)$ para aliviar la notación.

    - Creación de un nuevo objeto

$$\frac{\text{fresh}(.p2) \quad a_{.p2} = \text{init_atts}\,(P2, ar)}{ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{ProcessInterface}\, p2 =\ \text{new}\ P2\,(ar)\,; s\}) \\ \to tsk\,(t, m, .p1, l, s)\, ob\,(.p2, a_{.p2}, \perp)}$$

- Activación de un método

$$\frac{.p2 = [\![p2]\!]_{a_{.p2} \circ l} \neq null \qquad l_1 = \text{buildLocals}(start) \qquad \text{fresh}(t_1)}{\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{Fut<Unit> fn = p2!start(); s}\}) \\ \quad \to tsk\,(t, m, .p1, l\,[fn \mapsto t_1]\,, s)\, tsk\,(t_1, start, .p2, l_1, \text{body}(start)) \end{array}}$$

  Como además cada objeto concurrente sólo codifica un proceso y la llamada a *start* es el punto de comienzo, la tarea comenzará a ejecutarse al ser lanzada. Hemos sustituido *futName* por *fn* para aliviar la notación.

Vamos a ver que, tras ejecutar la instrucción en el lenguaje imperativo, se producen modificaciones similares en estructuras equivalentes a las modificadas en ABS tras la ejecución de las dos instrucciones de la traducción.

En la primera instrucción de ABS, lo que hacemos es crear un nuevo objeto. Para ello, primero inicializamos los atributos del objeto usando su tipo y los argumentos introducidos. En la sentencia del lenguaje imperativo, tenemos la misma instrucción en *init(P2,args)*, que inicializa el método, salvo la traducción de los argumentos. Ya hemos visto anteriormente la equivalencia de asignación de identificadores en variables enteras, por lo que también tenemos la equivalencia ahora, ya que en caso de haber argumentos se hará un mapping entre las declaraciones de los parámetros y sus valores asignados. En el caso de los argumentos que no son variables enteras, será un mapping entre variables futuras y objetos ABS que sólo pertenecen al segundo estado, luego no influyen en la equivalencia con el estado del lenguaje imperativo.

Como los identificadores $.p$ se han creado en el lenguaje imperativo para identificar unívocamente un proceso, tenemos que se va a dar trivialmente la condición de *fresh(.p2)*. Por ello, tenemos el siguiente árbol de derivación equivalente a la primera regla semántica de ABS:

$$\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{ProcessInterface}\, p2 = \text{ new}\, P2\,(ar)\,; s\}) \\ \qquad \to tsk\,(t, m, .p1, l, s)\, ob\,(.p2, \text{init_atts}\,(P2, ar)\,, \bot) \end{array}$$

Para la segunda regla, tenemos que .p2 no es nulo, ya que lo hemos inicializado en el paso anterior. El conjunto de variables locales se inicializará a vacío, y se irá rellenando con mútex o variables futuras si los hay. Además, podemos suponer que t_1 no ha sido usado, al estar demostrando esto de una forma local en donde no tenemos más identificadores con ese nombre, y *fn* es un nombre estándar que siempre se crea concatenando el identificador del proceso a "f_", por lo que también lo podemos resolver anticipadamente. Por ello, podemos evitar estas precondiciones, obteniendo el siguiente árbol de derivación equivalente a la segunda regla de derivación en ABS:

$$\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{Fut<Unit> fn = p2!start(); s}\}) \\ \quad \to tsk\,(t, m, .p1, l\,[fn \mapsto t_1]\,, s)\, tsk\,(t_1, start, .p2, l_1, \text{body}(start)) \end{array}$$

donde *body(start)* devuelve el código asociado a *start*, que se correspondería con la primera parte de la traducción del código del proceso. Las partes que no estén en este

código estarán en métodos auxiliares que se habrán creado de acuerdo a la traducción introducida para las instrucciones *lock/unlock*. Por ello, mediante esta instrucción y la traducción de las instrucciones del bloque interno del proceso, tendremos la equivalencia con *cod*, y lo único que nos faltaría por modificar es la estructura análoga a P en el lenguaje imperativo para reflejar que ha creado y se está ejecutando un nuevo proceso, pero eso es la equivalencia con la creación del objeto concurrente $.p2$ y la posterior llamada a su método *start*, que es el punto de comienzo del flujo traducido.

Tenemos, por tanto, que el comportamiento es de nuevo equivalente en ambos casos.

- **join(Proc_Ident); $\mapsto$ await f_Proc_Ident?;**

  De forma similar al caso del *lock/unlock*, tenemos que estudiar dos casos en el join, [join^a] y [join^{na}], y su correspondencia en ABS. Vamos a escribir $fn = f_Proc_Ident$ para aligerar la notación.

  - [join^{na}]
    En este caso, que join espere a por un proceso que no ha acabado equivale, en ABS, a que no se haya resuelto la futura por la que espera el *await*. Aunque en ABS el proceso en el que está libera su mútex para que lo pueda tomar otra tarea, anteriormente se ha comprobado que sólo vamos a tener a lo sumo una tarea que se pueda ejecutar cada vez en el objeto, debido a que cada uno de estos objetos traduce un único proceso y flujo de ejecución, por lo que efectivamente se detiene el flujo de ejecución. Las reglas semánticas en ambos lenguajes son:

    - Lenguaje imperativo

$$\frac{isActive(.p2)}{\text{ip}\,(.p1, L, \{\text{ join}(.p2); s\}) \rightarrow \text{ ip}\,(.p1, L, \{\text{ join}(.p2); s\})}$$

    - ABS

$$\frac{[\![y]\!]_{a_{.p1}\circ l} = t_1 \qquad s_1 \neq \epsilon(v)}{\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{await } fn?; s\})\, tsk\,(t_1, m_1, .p2, l_1, s_1) \\ \qquad \rightarrow ob\,(.p1, a_{.p1}, \bot) \end{array}}$$

    Para empezar, podemos ver que ambas precondiciones son equivalentes. En la primera, el método *isActive* devuelve cierto si el proceso está activo (no ha finalizado su ejecución), para lo que primero tiene que haber sido declarado. Esto se corresponde con las partes dos y uno de la precondición en ABS, respectivamente, unido a $tsk\,(t_1, m_1, .p2, l_1, s_1)$, que es la tarea que muestra que el flujo no terminado de instrucciones pertenece al método del objeto concurrente que estamos estudiando. Podemos sustituirlas en el árbol de derivación de ABS. Además, en este caso vamos a poner en la consecuencia de la regla también la información de la tarea t para que se vea más explícito el resultado, ya que se ha explicado anteriormente que el hecho de que no se ponga a la derecha implica que no ha sufrido cambios.

De esta forma, el árbol de derivación en ABS pasa a ser

$$\frac{isActive(.p2)}{\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{await } fn?; s\}) \\ \qquad \rightarrow ob\,(.p1, a_{.p1}, \bot)\, tsk\,(t, m, .p1, l, \{\text{await } fn?; s\}) \end{array}}$$

Observamos que, en el caso del lenguaje imperativo no se produce ningún cambio, mientras que en el caso de ABS se libera el mútex implícito del objeto. Sin embargo, esto no es un problema para ver la equivalencia en los comportamientos, ya que en ABS cada objeto concurrente está destinado a codificar un proceso y sólo puede ejecutarse una tarea cada vez, por lo que la única tarea disponible para coger de nuevo el mútex será t, que sigue en el mismo estado. Tenemos, por tanto, comportamientos equivalentes.

De nuevo, al igual que en el caso de $\left[\text{ lock}^{\text{ false}}\right]$, tenemos que en ABS tampoco se produce un cambio de estado al ejecutarse esta instrucción, debido a las dependencias con otros objetos y tareas. Por tanto, aquí tenemos la equivalencia de nuestro segundo caso de deadlock en el lenguaje imperativo.

- $[\text{ join}^{a}]$

  En este caso, que join espere a un proceso que ha acabado se compara, en ABS, con que la futura por la que espera el await se haya resuelto. Este es un caso que no induce bloqueos. Las reglas semánticas en ambos lenguajes son:

  - Lenguaje imperativo

    $$\frac{\neg isActive(.p2)}{\text{ip}\,(.p1, L, \{\text{ join}(.p2); s\}) \rightarrow \text{ ip}\,(.p1, L, \{s\})}$$

  - ABS

    $$\frac{[\![y]\!]_{a_{.p1}\circ l} = t_1}{\begin{array}{l} ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{await } fn?; s\})\, tsk\,(t_1, m_1, .p2, l_1, \epsilon(\varnothing)) \\ \qquad \rightarrow tsk\,(t, m, .p1, l, s) \end{array}}$$

    donde de nuevo podemos sustituir la precondición en ABS y la información de la tarea t_1 por la llamada al método *isActive*, que comprueba si el proceso ha acabado su ejecución. Obtenemos la regla

    $$\frac{\neg isActive(.p2)}{ob\,(.p1, a_{.p1}, t)\, tsk\,(t, m, .p1, l, \{\text{await } fn?; s\}) \rightarrow tsk\,(t, m, .p1, l, s)}$$

    y, de nuevo, vemos que no se producen más cambios en ninguno de los lenguajes que el avance del programa ejecutando la instrucción correspondiente. Tenemos, con ello, la equivalencia en ejecución.

□

Lema 4.3. Sea P_i un programa en el lenguaje imperativo y $P_{ABS} = \tau\,(P_i)$ el correspondiente programa traducido. Cada una de las ejecuciones de instrucciones en P_i equivale a ejecutar una serie de instrucciones en P_{ABS}, de forma que llegamos a estados equivalentes.

Demostración. Aplicamos la traducción global al programa. Por el lema 4.1, tenemos que la aplicación de τ a P_i nos dará un programa P_{ABS} de forma que se cumple el último apartado de la equivalencia de estados en el primer estado de cada programa.

- $n = 1$

  Por la aplicación del lema anterior, tenemos trivialmente que en un primer momento avanzamos a estados equivalentes.

- Supongamos cierto para n, lo comprobamos para $n + 1$.

  Tras la aplicación de las primeras n instrucciones tenemos que, por la hipótesis de inducción, estamos en estados equivalentes. Aplicar una instrucción, por el lema 4.1, nos llevará a estados equivalentes de nuevo.

□

Proposición 4.1. Sean T y S estados equivalentes. Entonces, T tiene deadlock si y sólo si S también lo tiene.

Demostración. En el caso del lenguaje imperativo, tener una situación de deadlock equivale a que no podemos producir un cambio de estado teniendo procesos activos. En ABS, tenemos la situación de deadlock cuando todas las tareas activas (debe haber tareas activas) están esperando la terminación de otra y ninguna puede avanzar, por lo que en el caso particular de nuestro ejemplo esto equivale a que tampoco podemos producir un cambio de estado en T.

Por la definición de equivalencia de estados, tenemos trivialmente por el último apartado que ambos estados serán equivalentes si, en el caso de no poder avanzar en uno, tampoco podremos avanzar en el otro. Aplicando esto a ambas definiciones de deadlock, llegamos al resultado. □

Teorema 4.1. Un programa P_i se bloqueará si y sólo si su homólogo $P_{ABS} = \tau(P_i)$ se bloquea.

Demostración. Por el lema 4.3, tenemos que cada uno de los estados de P_i tendrá su estado equivalente en P_{ABS}.

Si existe algún estado que tiene deadlock en el lenguaje imperativo, entonces por la proposición 4.1 tendremos que su estado equivalente en ABS también lo tendrá. De igual manera, para cada estado en el programa ABS que tenga deadlock, llegamos a que su estado equivalente deberá tenerlo. □

CAPÍTULO 5

Implementación

5.1. Traducción

Una primera aproximación de la traducción del programa, implementado en JavaCC, viene dada por la siguiente gramática BNF, siendo *Start* el símbolo de comienzo en la gramática. Se usan los tokens definidos en el capítulo anterior.

Para hacer más legible la gramática, se ha cambiado la sintaxis usada en JavaCC, de forma que todos los tokens correspondientes a símbolos terminales se han puesto en minúscula, estando los símbolos no terminales en mayúscula.

```
     Start ::= main o_par c_par o_brace Body c_brace (Processes)*
      Body ::= (Statement)*
 Processes ::= process proc_name o_par Params c_par o_brace Body c_brace
 Statement ::= int_type var_ident semicolon
             | int_type global_var_ident semicolon
             | proc_name proc_ident semicolon
             | mutex_type var_ident semicolon
             | var_ident asign AExpr semicolon
             | global_var_ident asign AExpr semicolon
             | semicolon
             | if o_par AExpr c_par o_brace Body c_brace else o_brace Body c_brace
             | while o_par AExpr c_par o_brace Body c_brace
             | lock o_par var_ident c_par semicolon Body
                   unlock o_par var_ident c_par semicolon
```

| proc_ident asign new proc_name o_par Args c_par
| join o_par proc_ident c_par semicolon
Params ::= [int_type var_ident (comma int_type var_ident)*]
Args ::= [var_ident (comma var_ident)*]

En la implementación del traductor, se leerá el código del lenguaje imperativo y se ejecutará la gramática descrita junto con acciones asociadas a cada token en la regla correspondiente. De esta forma, se irán rellenando estructuras auxiliares que, al finalizar el parser, se interpretarán para su posterior transformación a la estructura de un programa ABS, que se volcará en un fichero.

Hacemos la suposición de que el programa que recibe el parser escrito en el lenguaje imperativo es correcto sintácticamente, por lo que lo único que debe hacer el parser es traducir cada una de las sentencias al código que le correspondería en ABS.

5.2. Estructuras y campos auxiliares

En el parser se han usado los siguientes campos y estructuras auxiliares, que servirán para escribir el código posteriormente a la lectura del fichero:

- *String proc*. Es el nombre del proceso que se está traduciendo en este momento. Como en el caso de este lenguaje imperativo no hay anidamiento de procesos, no necesitamos una estructura más compleja que una cadena, ya que sólo necesitamos guardar la información del último proceso.

- *Stack<String> bloqueMutex*. Esta pila sirve para guardar la información de los bloques de mútex de cada proceso. Se inicializa a la pila vacía cuando comienza cada proceso, y se coloca en la base de la pila la cadena "start" para indicar que el código que viene a continuación se corresponde con el del ámbito más externo, el del método *Unit start()* de cada proceso. De esta forma, cada vez que hacemos un *lock*, colocamos en la cima de esta pila una referencia al ámbito de anidamiento en el que nos encontramos, que será de la forma "mut_n" donde *mut* es el nombre del mútex sobre el que se hace el lock y *n* el número de veces que se ha cogido este mútex en el proceso en el que nos encontramos (contando en la que estamos, se empieza en 1).

- *HashMap<String, Integer> numeroMutex*. Sirve, dentro de cada proceso, para guardar el número de veces que se ha hecho lock sobre el mútex. Se inicializa a una tabla vacía cada vez que se entra en un proceso nuevo.

- *HashMap<String, Integer> maxMutex*. Almacena el número máximo de apariciones de cada mútex en todos los procesos que lo usen. Se utiliza para:

    - Declarar la interfaz *MutexInterface* de los mútex con los métodos *Unit m_1(ProcessInterface p),...,Unit m_max(ProcessInterface p)* donde *max* es el máximo de todos los enteros almacenados en la tabla. Aunque en algunos mútex serán más bloques que los que tienen en el programa original, se dejarán los bloques sobrantes vacíos para poder tener el máximo de los bloques anidados en el mútex que lo necesite.

  - Declarar los métodos asociados a cada mútex en cada proceso en la interfaz *ProcessInterface*. En este caso se ve la necesidad de tener una estructura mayor que un entero global para almacenar el máximo de anidamiento, ya que no será necesario declarar siempre el máximo de apariciones de todos los mútex para cada mútex en cada proceso (que nos llevaría a un total de $k \cdot max$ suponiendo que haya k mutex), sino el máximo de cada mútex en todos los procesos, lo cual en ciertos casos puede reducir significativamente el número de procesos y, en el caso peor, tendrá el mismo número de métodos de antes.

- *HashMap<String, LinkedList<String>> paramsProcesos1*. Almacenan, para cada proceso, los parámetros enteros que deben recibir, en el orden que corresponda.

- *HashMap<String, LinkedList<String>> paramsProcesos2*. Almacenan, para cada proceso, los parámetros de mútex, variables globales y variables futuras que deben recibir.

- *HashMap<String, LinkedList<String>> argsProcesos1*. Almacenan los argumentos enteros que tienen que recibir los procesos en orden.

- *HashMap<String, LinkedList<String>> argsProcesos2*. Almacenan los argumentos que tienen que recibir los procesos en orden correspondientes a variables futuras, globales y mútex.

- *HashMap<String, String> tipoProceso*. Almacena, para cada identificador de proceso, el tipo que es. Se utilizará junto con los argumentos anteriores durante la escritura de creación del proceso, ya que en un primer momento no sabemos qué argumentos va a tener el proceso aparte de los enteros, por lo que debemos ejecutar el parser antes de escribirlo.

- *HashSet<String> globales*. Almacena las variables globales que se declaran a lo largo del programa. Se utilizará posteriormente para configurar la interfaz *GlobalsInterface* y la clase *Globals*.

- *HashMap<String, HashSet<String>> localesProcesos*. Para poder acceder a las variables locales de los procesos desde cada método, que representa un bloque asociado a un mútex, se necesita que estas variables se traduzcan en campos del objeto destinado a traducir el proceso. Por ello, se utiliza esta estructura para guardar las variables asociadas a cada proceso, de forma que en la escritura del código final se puedan incluir estos campos.

- *HashMap<String, HashMap<String, LinkedList<String>>> codigo*. Esta estructura es la más compleja de la traducción. En el HashMap más externo, la clave se refiere al nombre del proceso en el que nos encontramos, y en el valor tenemos otro HashMap que almacena el código correspondiente a cada bloque de anidamiento dentro del proceso.

  Esto es, en este segundo HashMap tenemos como clave un String de la forma “start” o “nomMutex_{num}”, que se corresponde con el nombre del método del proceso

que almacenará este código. El valor es una lista enlazada de String, donde cada una de las componentes es la traducción de una instrucción.

Con estas funciones, se estructura el código de la siguiente forma:

- Creación de la case CodeParser
    - Declaración de atributos
    - Función main
        - Llamada a la función de entrada al Parser, con la gramática
        - Escritura de la cabecera del fichero
        - Escritura de las interfaces
        - Escritura de las clases
        - Escritura del main
- Analizador léxico
- Analizador sintáctico

5.3. Ejemplos

Como ejemplo, traducimos los dos ejemplos introducidos en el capítulo dedicado al lenguaje imperativo, que podrían inducir deadlock en nuestro sistema.

Ejemplo 5.1.

```
Main(){
  Mutex m1;
  Mutex m2;

  P1 .p1;
  P2 .p2;

  .p1 = new P1();
  .p2 = new P2();
}

process P1(){
  lock(m1);
  //posible deadlock
  lock(m2);
  unlock(m2);
  unlock(m1);
}

process P1(){
  lock(m2);
  //posible deadlock
```

```
23    lock(m1);
24    unlock(m1);
25    unlock(m2);
26  }
```

El programa con dos locks intercalados se traduce como:

```
1   module Locklock;
2
3   interface Mi{
4     Unit m_1(Pi p);
5   }
6
7   interface Pi{
8     Unit start();
9     Unit m1_1();
10    Unit m2_1();
11   }
12
13  class M1() implements Mi{
14    Unit m_1(Pi p) {
15      Fut<Unit> f;
16      f = p!m1_1();
17      f.get;
18    }
19  }
20
21  class M2() implements Mi{
22    Unit m_1(Pi p) {
23      Fut<Unit> f;
24      f = p!m2_1();
25      f.get;
26    }
27  }
28
29  class P1(Mi m1,Mi m2) implements Pi{
30    Unit start() {
31      Fut<Unit> y;
32      y = m1!m_1(this);
33      await y?;
34     }
35    Unit m1_1() {
36      Fut<Unit> y;
37      y = m2!m_1(this);
38      await y?;
39     }
40    Unit m2_1() {
41    }
```

```
42   }
43 
44 class P2(Mi m2,Mi m1) implements Pi{
45   Unit start() {
46     Fut<Unit> y;
47     y = m2!m_1(this);
48     await y?;
49    }
50   Unit m1_1() {
51   }
52   Unit m2_1() {
53     Fut<Unit> y;
54     y = m1!m_1(this);
55     await y?;
56   }
57  }
58 
59 {
60 Mi m1 = new M1();
61 Mi m2 = new M2();
62 Pi p1 = new P1(m1,m2);
63 Fut<Unit> f_p1 = p1!start();
64 Pi p2 = new P2(m2,m1);
65 Fut<Unit> f_p2 = p2!start();
66 }
```

Ejemplo 5.2.

```
1  Main(){
2    Mutex m;
3 
4    P1 .p1;
5    P2 .p2;
6 
7    .p2 = new P2();
8    .p1 = new P1();
9  }
10 
11 process P1(){
12   lock(m);
13   //posible deadlock
14   join(.p2);
15   unlock(m);
16 }
17 
18 process P2(){
19   //posible deadlock
20   lock(m);
21   unlock(m);
```

```
}
```

En el caso del ejemplo con lock y join tenemos:

```
module Lockjoin;

interface Mi{
  Unit m_1(Pi p);
}

interface Pi{
  Unit start();
  Unit m_1();
}

class M() implements Mi{
  Unit m_1(Pi p) {
    Fut<Unit> f;
    f = p!m_1();
    f.get;
  }
}

class P1(Mi m,Fut<Unit> f_p2) implements Pi{
  Unit start() {
    Fut<Unit> y;
    y = m!m_1(this);
    await y?;
  }
  Unit m_1() {
    await f_p2?;
  }
}

class P2(Mi m) implements Pi{
  Unit start() {
    Fut<Unit> y;
    y = m!m_1(this);
    await y?;
  }
  Unit m_1() {
  }
}

{
Mi m = new M();
Pi p2 = new P2(m);
Fut<Unit> f_p2 = p2!start();
```

```
45 Pi p1 = new P1(m,f_p2);
46 Fut<Unit> f_p1 = p1!start();
47 }
```

5.4. Uso de la herramienta

La herramienta desarrollada en JavaCC está disponible en `http://costa.ls.fi.upm.es/th2abs/clients/web/`

En este enlace, se puede acceder al interfaz web *Easy Interface*, en el cual podemos probar los programas en el lenguaje imperativo. Al ejecutar la herramienta, el programa introducido será traducido y analizado para detectar posibles deadlocks. El resultado del análisis de deadlock se verá posteriormente en la parte inferior de la pantalla, tal y como se muestra en las siguientes imágenes.

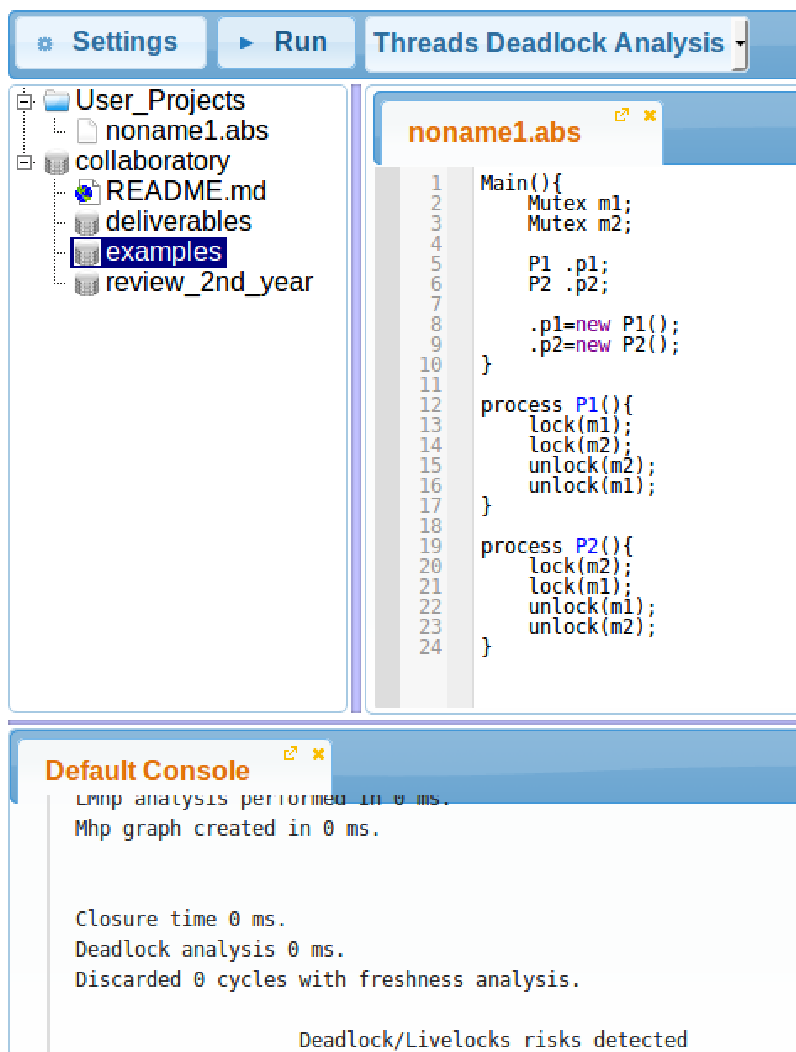


Figura 5.1: Deadlock: caso de lock-lock

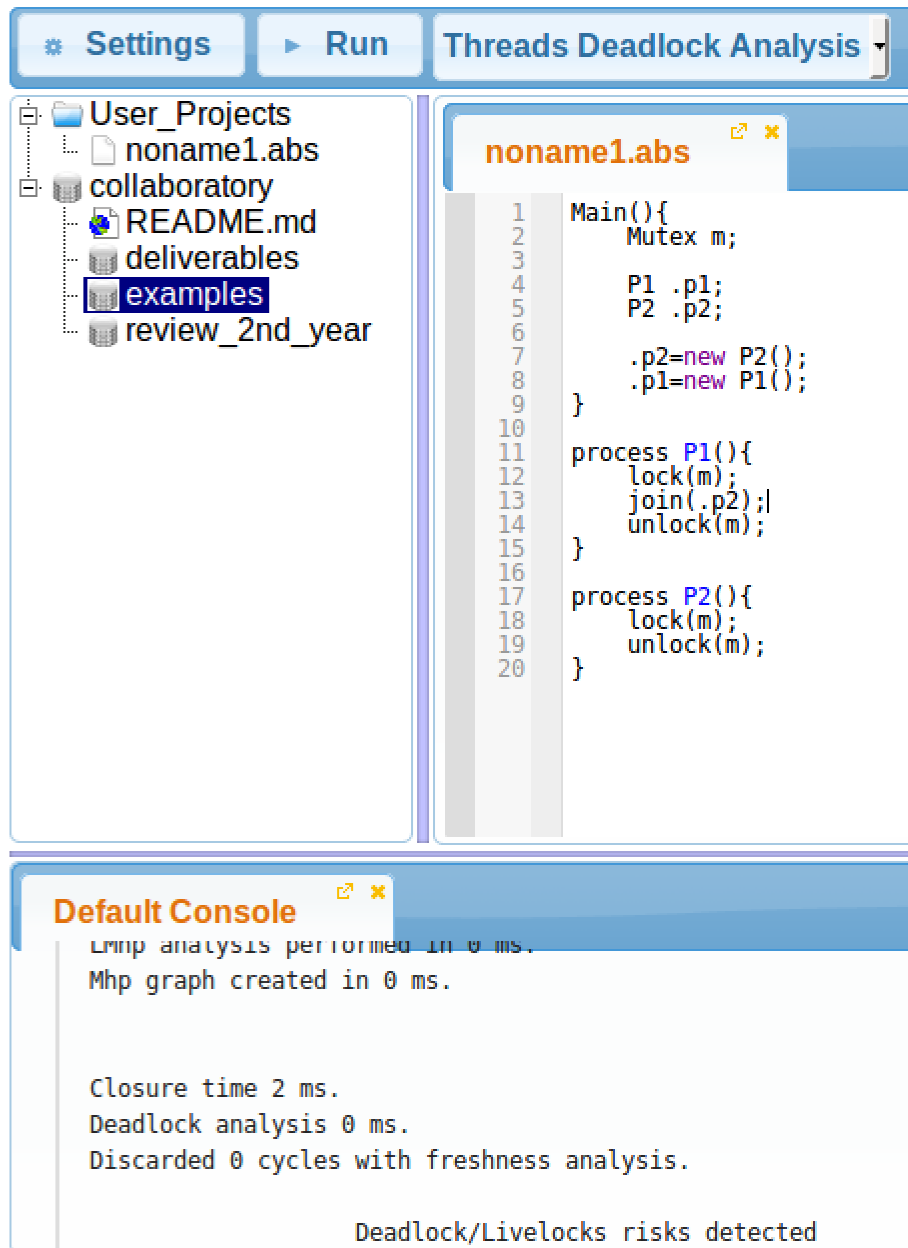


Figura 5.2: Deadlock: caso de lock-join

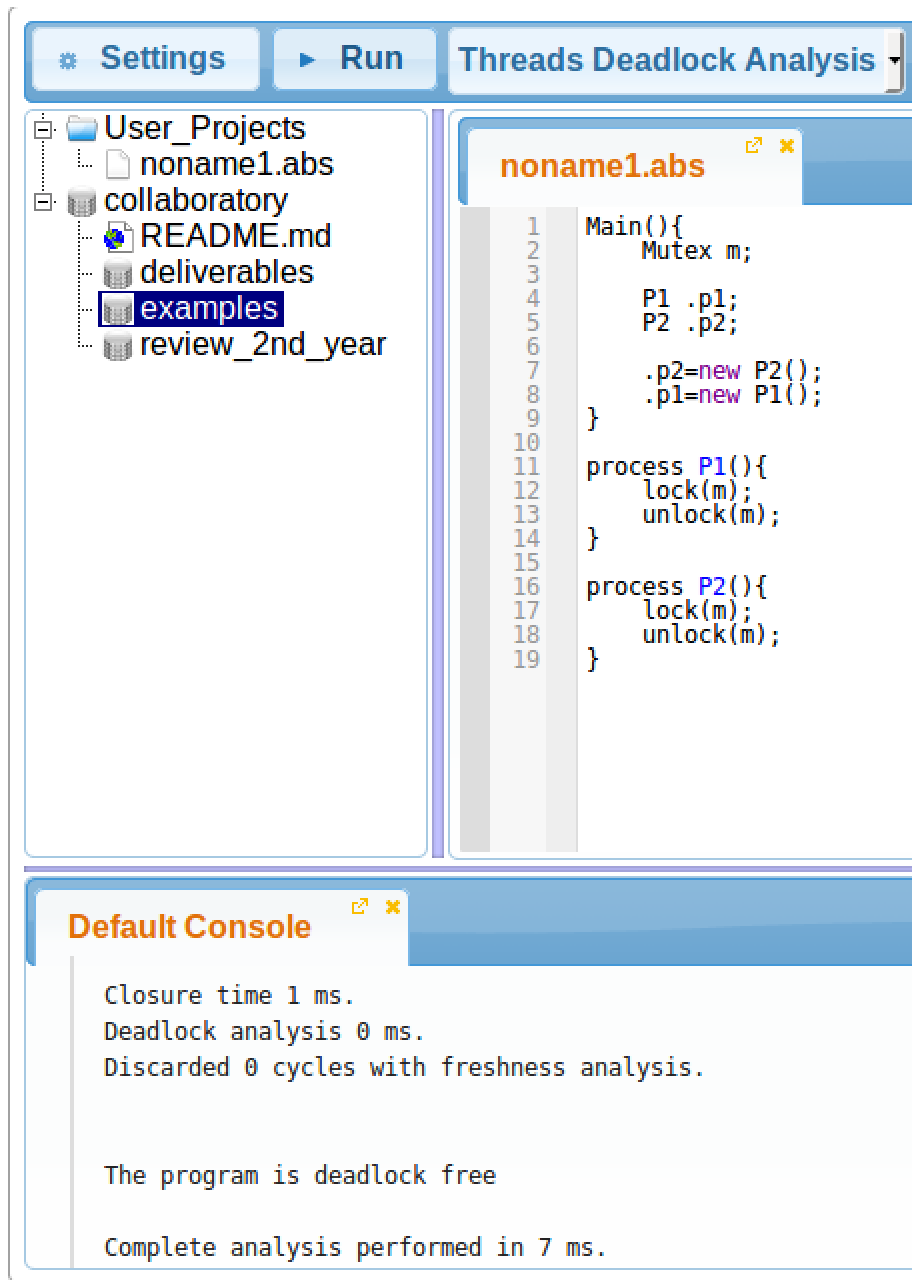


Figura 5.3: Programa sin deadlock

CAPÍTULO 6

Conclusiones y resultados

A lo largo del trabajo, se han realizado las transformaciones y pruebas pertinentes para conseguir una traducción de programas escritos en lenguajes imperativos que utilizan primitivas *lock/unlock* a programas ABS basados en actores. De esta forma, podemos beneficiarnos de las herramientas de control de concurrencia de unos y de los análisis de bloqueo de otros.

Lo primero que cabe remarcar de este trabajo es el lenguaje imperativo que se ha desarrollado. Es un lenguaje con una estructura fijada, similar a muchos lenguajes multi-threaded, por lo que no resulta difícil de entender o adaptar a otro tipo de programas. Además, en este lenguaje se han desarrollado las semánticas y transiciones de una forma estándar, basadas en cambios de estados, pero separando las representaciones de los estados en dos: ámbito global y ámbito local. Con esto, conseguimos una expresión más clara de los cambios, ahorrando notación innecesaria y consiguiendo más precisión en la representación que la que se tiene en otras semánticas de lenguajes *While*. Este desarrollo nos ha permitido un análisis a nivel teórico más extenso, lo cual se ha visto reflejado en una mayor facilidad para analizar los casos de deadlock y sus equivalencias.

En el caso del estudio del lenguaje ABS, se ha desarrollado principalmente como una recopilación de información de artículos que presentaban el lenguaje y explicaban el análisis de deadlock. Para poder mostrar después la equivalencia de una manera más exhaustiva, se han desarrollado ciertas reglas semánticas que no se mostraban en estos artículos, como es el caso de la instrucción *while*, utilizada posteriormente para ver la equivalencia entre el código en el lenguaje imperativo y el código traducido en ABS.

El capítulo más importante del trabajo y con una componente matemática grande es, sin duda, el referido a la traducción. Como se planteó en la introducción, mediante esta traducción estamos desarrollando un método novedoso y no existente hasta el momento de traducción de primitivas *lock/unlock* a ABS, mediante el estudio de los invariantes de monitores en el segundo lenguaje. Se plantea así una visión global de cómo van a ser los programas traducidos para, posteriormente, indicar qué traducción le corresponderá a cada instrucción. Para finalizar, desarrollamos las demostraciones de equivalencia entre

ambas traducciones, no sólo para comprobar la equivalencia de estados a nivel semántico, sino para comprobar la equivalencia en comportamiento, que es lo realmente importante en este estudio. Así, llegamos a la conclusión de que el programa en el lenguaje imperativo se bloqueará si y sólo si el correspondiente programa en ABS se bloquea. Esto, que era lo que estábamos buscando, nos proporciona la potente herramienta que queríamos: poder comprobar si un programa en un lenguaje imperativo se va a bloquear mediante un análisis estático realizado por herramientas que han sido previamente estudiadas, desarrolladas y mejoradas.

Para finalizar el trabajo, hemos querido no dejar sólo la invetigación en un nivel teórico, sino extenderla a los casos prácticos y de prueba. Por ello, se ha desarrollado una herramienta de traducción mediante el uso de JavaCC, que permite traducir un programa en lenguaje imperativo a su correspondiente programa ABS, al cual podremos aplicar las herramientas de análisis que se han desarrollado en la facultad y están fácilmente accesibles en el enlace proporcionado en este trabajo.

Bibliografía

[1] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2010), LNCS 6957. Springer 2012.

[2] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. Formal Techniques for Distributed Systems, pages 273-288. Springer 2013.

[3] Hanne Riis Nielson, and Flemming Nielson. Semantics with Applications: an Appetizer. Springer 2007.

[4] Gregory R. Andrews. Concurrent Programming: Principles and Practice. Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA 1991.

[5] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2015.

[6] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Combining Static Analysis and Testing for Deadlock Detection. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2016.

[7] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. «SOFTWARE AND SYSTEMS MODELING», 2016, 15, pages 1013 - 1048 [Scientific article]

[8] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock Analysis of Concurrent Objects: Theory and Practice. Integrated Formal Methods, pages 394-411, Springer 2013.

[9] Jesús Doménech, Samir Genaim, Einar Broch Johnsen, and Rudolf Schlatte. EasyInterface: A toolkit for rapid development of GUIs for research prototype tools. Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, lncs, pages 379-383. Springer 2017.