UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Modelado y verificación del protocolo de exclusión mutua con tiempo real Fischer usando Maude

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Víctor García Valero

Tutor: Santiago Escobar Román

Curso: 2020/2021

Resumen

En el siguiente trabajo modelamos el protocolo de exclusión mutua con tiempo real Fischer como un módulo de sistema mediante el lenguaje de especificación y verificación Maude. Hacemos uso del alcance de la generación de estados y del *model checker* de Maude para verificar las propiedades de exclusión mutua, *deadlock freedom* y vivacidad.

Para conseguir este objetivo, en la primera parte enseñamos los conceptos básicos de dos lenguajes de especificación y verificación, Maude y CafeOBJ. Para apoyar la explicación, presentamos un ejemplo de especificación en Maude de un cruce con dos semáforos con vehículos, y hacemos un ejemplo de verificación de una propiedad de seguridad en él.

Luego hacemos un estudio del recorrido del protocolo Fischer, desde sus orígenes hasta la versión propuesta más reciente, para comprender las implicaciones que tiene sobre el uso del tiempo.

A continuación, analizamos una versión propuesta en el lenguaje CafeOBJ. Estudiamos la construcción del modelo, así como las asunciones temporales que se hacen sobre él. Además, mostramos la verificación de la propiedad de exclusión mutua realizada mediante *proof score method*, un método de verificación propio de CafeOBJ.

Luego definimos un modelo, en el lenguaje de especificación Maude, para el protocolo de exclusión mutua en tiempo real Fischer. Sobre este modelo comentamos las decisiones de diseño del tiempo y sus límites, tanto globales como específicos.

Con el modelo definido, empleamos la herramienta de *model checking* disponible en el lenguaje Maude, para definir las propiedades de exclusión mutua, *deadlock freedom* y vivacidad, como fórmulas en la lógica conocida como *linear temporal logic*. Hacemos uso de las fórmulas sobre dos estados que definimos en el *model checker,* y conseguimos verificar la propiedad de exclusión mutua para nuestro modelo. Para las otras dos propiedades explicamos las razones por las cuales no han podido ser verificadas y proponemos una solución para la propiedad *deadlock freedom*.

Palabras clave: Maude, Fischer, CafeOBJ, verificación, *model checking*.

Abstract

In the following work we model Fischer's real-time mutual exclusion protocol as a system module using the Maude specification and verification language. We make use of the scope of the generation of states and Maude's model checker to verify the properties of mutual exclusion, deadlock freedom and starvation freedom.

To achieve this goal, in the first part we teach the basics of two specification and verification languages, Maude and CafeOBJ. To support the explanation, we present an example of a specification in Maude of a crossing with two traffic lights with vehicles, and we do an example of verification of a safety property on it.

We then do a study of the journey of the Fischer protocol, from its origins to the most recent proposed version, to understand the implications it has on the use of time.

Next, we analyze a version proposed in the CafeOBJ language. We study the construction of the model, as well as the temporal assumptions that are made about it. In addition, we show the verification of the mutual exclusion property carried out using the proof score method, a verification method of CafeOBJ.

We then define a model, in the Maude specification language, for the Fischer real-time mutual exclusion protocol. On this model we discuss the design decisions of time and its limits, both global and specific.

With the model defined, we use the model checking tool available in the Maude language, to define the properties of mutual exclusion, deadlock freedom and vivacity, as formulas in the logic known as linear temporal logic. We make use of the two-state formulas that we define in the model checker, and we manage to verify the mutual exclusion property for our model. For the other two properties we explain the reasons why they could not be verified and propose a solution for deadlock freedom.

Keywords: Maude, Fischer, CafeOBJ, verification, model checking.

Tabla de contenidos

Listado de ilustraciones

1. INTRODUCCIÓN

Desde el nacimiento del *software* en los años 60 y al periodo conocido como crisis del *software* que siguió, la velocidad con la que el *software* es desarrollado se ha visto propulsada exponencialmente gracias al surgimiento de estándares, metodologías y herramientas de desarrollo. Con la gran cantidad de sistemas, protocolos y algoritmos desarrollados surge la necesidad de crear un modelo previamente a su producción, con el objetivo de verificar propiedades de interés sobre él.

Además, el creciente uso de la tecnología en mayores y más diversos sectores hace que la demanda de *software* de calidad sea cada vez mayor. Como respuesta a la demanda, sobre los años 80 y 90, son desarrollados métodos y lógicas que forman la base de los métodos formales.

Las áreas de aplicación de los métodos formales abarcan aplicaciones de seguridad crítica en transportes como aviones o barcos, aplicaciones con restricciones de tiempo como son las transacciones bancarias o el área de las telecomunicaciones, aplicaciones en el área de la privacidad y la seguridad como es el comercio electrónico, y el modelado y verificación de estándares y protocolos.

Hoy en día, el modelado de sistemas y la verificación automática de *software* son áreas en continuo estudio y desarrollo. Existen múltiples herramientas para modelar sistemas y verificar propiedades sobre ellos como son Spin, NuSMV, Promela, Maude o CafeOBJ.

Es en los métodos formales, concretamente en el campo de los protocolos, donde tiene lugar este trabajo fin de grado. El protocolo escogido es el conocido protocolo de exclusión mutua en tiempo real Fischer.

1.1. Motivación

La elección de los métodos formales surge fruto de la importancia que tienen estos en el campo del modelado de sistemas *software* y el uso de herramientas para verificar propiedades. El área de investigación es además muy amplia, tratando temas actuales como ingeniería del *software*, compiladores, privacidad y seguridad, biología y programación cuántica y sus aplicaciones.

La elección del protocolo Fischer viene caracterizada por tratarse de un protocolo simple pero complejo a la hora de tratar aspectos de tiempo real. Además de ser un protocolo para sistemas concurrentes sirve como concepto para otros nuevos protocolos o forma parte de otros nuevos como pueden ser TLS o *blockchain*.

Obviamente hubo alternativas de sistemas y protocolos a elegir antes que Fischer. Las alternativas principales fueron el protocolo de autenticación Kerberos o el protocolo de convenio para transacciones en tecnologías *blockchain*, como el que usa el sistema Ethereum. Pero al final estas alternativas fueron descartadas debido a su complejidad.

Además, otro de los motivos principales es aprender el *model checker* del lenguaje Maude, y lenguajes como CafeOBJ, un lenguaje de especificación cercano a Maude.

Finalmente, es [1] el que confirma el interés por elegir el protocolo Fischer y entender como lo han modelado y verificado en CafeOBJ mediante conceptos y métodos desarrollados por ellos en trabajos anteriores.

1.2. Objetivos

El principal objetivo de este trabajo es analizar el protocolo de exclusión mutua en tiempo real Fischer para poder construir un modelo en sintaxis de Maude sobre el cual verificar tres propiedades de sistemas concurrentes utilizando el *model checker* de Maude.

Para lograr el objetivo principal definimos los siguientes objetivos específicos:

- Aprender la sintaxis de los lenguajes Maude y CafeOBJ.
- Comprender el funcionamiento del protocolo Fischer.
- Entender el modelo construido en [1] con CafeOBJ y la verificación de la propiedad de exclusión mutua.
- Definir un modelo para el protocolo Fischer en Maude.
- Verificar mediante fórmulas y *model checking* en Maude tres propiedades sobre el modelo construido.

1.3. Proceso

Para el desarrollo de este trabajo fin de grado hemos optado por una metodología de trabajo basada en las fases de análisis, modelado y verificación, y por último razonamiento de los resultados.

Con la primera fase estudiamos y tratamos de entender el problema a través de la búsqueda de trabajos relacionados. En nuestro caso es el análisis del protocolo Fischer a través de la literatura disponible. A través de esta literatura conseguimos información en forma de decisiones y asunciones hechas sobre el modelo construido. Son de especial interés el manejo del tiempo que realizan las versiones de otros autores.

La segunda fase consiste en construir una solución a partir de la información conseguida en la primera fase. Dada la naturaleza de este trabajo, en esta segunda fase modelamos en Maude una versión del protocolo Fischer e intentamos verificar tres propiedades mediante su *model checker*.

La tercera fase es donde consideramos si los resultados son satisfactorios y si es necesario volver a la fase de análisis o de construcción. Si se vuelve a la fase de análisis, buscamos nuevas versiones propuestas o revisamos las que tenemos en busca de algún fallo de comprensión. En caso de volver a la construcción del modelo, son aplicados los cambios pertinentes y verificadas de nuevo las propiedades sobre el nuevo modelo.

1.4. Estructura

La estructura de este trabajo sigue una distribución por capítulos. A continuación, mostramos en orden los capítulos junto a una pequeña explicación de su contenido para saber que nos vamos a encontrar.

Capítulo 2: Explicación y descripción de la sintaxis de los lenguajes Maude y CafeOBJ en las versiones utilizadas durante el desarrollo del trabajo.

Capítulo 3: Explicación de las diferentes versiones del protocolo Fischer desde su concepción hasta la versión en forma de autómata.

Capítulo 4: Análisis y entendimiento del modelo propuesto y la verificación realizada por el artículo [1].

Capítulo 5: Construcción del modelo de sistema para el protocolo Fischer usando el lenguaje Maude.

Capítulo 6: Verificación mediante *model checking* de tres propiedades sobre el modelo construido.

Capítulo 7: Recopilación de las conclusiones extraídas tras el análisis, modelado y verificación del protocolo Fischer y su evaluación para cumplir los objetivos propuestos.

Capítulo 8: Propuestas de mejora para trabajos futuros con base en este trabajo o sobre el mismo protocolo.

2. CONOCIMIENTOS PREVIOS

Para la comprensión de este trabajo fin de grado es necesario conocer y entender, en la medida de lo posible, las herramientas utilizadas en la versión disponible durante la realización del trabajo. Es por ello, que a continuación veremos en primer lugar el lenguaje de especificación Maude, pilar principal en el modelado del protocolo de este trabajo. Además, veremos también un lenguaje cercano a este llamado CafeOBJ, que es el lenguaje utilizado en [1].

2.1. Maude

Es un lenguaje de programación declarativo que hace uso de módulos y reglas de reescritura para el modelado de sistemas y la verificación de propiedades. La versión utilizada en el desarrollo de los modelos es Maude 3.1 disponible en [2]. A continuación, veremos los conceptos básicos de Maude para entender los módulos desarrollados.

Como primer paso, veremos el concepto de tipo y cómo declarar subtipos. Después, estudiaremos cómo crear nuestra propia sintaxis mediante la definición de operadores para luego ver el uso de axiomas sobre los operadores para aplicar propiedades o atributos matemáticos. Veremos luego como definir variables para almacenar valores de diferentes tipos predefinidos y definidos por nosotros. Finalmente, aprenderemos sobre la unidad básica de programación, el módulo, en sus dos tipos: módulos de sistema y módulos funcionales, junto con el comando de interés.

2.1.1. *Tipos y subtipos*

En Maude lo primero que hacemos al crear un módulo, ya sea funcional o de sistema, es definir los tipos que se usarán en los operadores y variables que queramos usar. Para definir un tipo debemos hacer uso de la palabra reservada *sort* seguido de un identificador. Un identificador es una cadena de caracteres no reservados que usa Maude para identificar todo tipo de elementos, desde el nombre de un módulo hasta el de una variable, o en este caso, un tipo.

Veamos un ejemplo de como declarar tipos para así introducir el concepto de subtipo. La sentencia «*sort Coche .*» nos define un tipo con identificador Coche. El uso del punto en Maude es para terminar las sentencias, parecido a los punto y coma de otros lenguajes como Java, Python o C. Si quisiéramos definir varios tipos a la vez podemos usar la palabra reservada *sorts*. Por tanto, la sentencia «*sorts Coche Vehículo Bicicleta .*» nos declara los tipos *Coche*, *Vehículo* y *Bicicleta*.

El concepto de subtipo es una relación entre diferentes tipos de manera que un tipo A que es subtipo de un tipo B es también de tipo B. Para aclarar este concepto usaremos los tipos que ya hemos definido antes de *Coche*, *Vehículo* y *Bicicleta*. Para declarar una relación de subtipos existe en Maude la palabra reservada *subsort*. Por tanto, si queremos declarar que el tipo Coche es un subtipo de Vehículo escribimos la sentencia «*subsort Coche < Vehículo .*» donde el símbolo < es el operador reservado para indicar la dirección de la relación. Podemos hacer lo mismo para Bicicleta con «*subsort Bicicleta < Vehículo .*».

Obtenemos con las dos sentencias anteriores una relación que se asemeja a la herencia de otros lenguajes. Como vemos en la Ilustración 1, tenemos los tres tipos de *Coche*, *Bicicleta* y *Vehículo*, siendo *Coche* y *Bicicleta* subtipos de *Vehículo*.

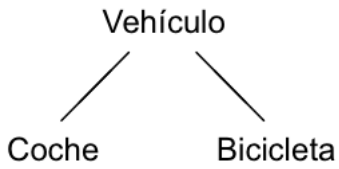

Ilustración 1. Relación de subtipo entre los tipos definidos.

2.1.2. *Operadores*

Los operadores nos permiten definir la sintaxis que tendrán los tipos que hemos definido, es decir, la notación que expresará aquello que necesitamos. De esta manera podemos modelar los elementos de un sistema o función con gran libertad de expresión.

Los operadores en Maude los definimos mediante la palabra reservada *op* seguido de la definición de la notación. Los operadores pueden estar parametrizados haciendo uso del carácter _ para indicar la posición de cada uno de los parámetros dentro de la notación. También los podemos parametrizar si omitimos cualquier _, haciendo que sea prefijo, es decir, que los parámetros se los pasaremos haciendo uso de los paréntesis como si de una función Java o C se tratara. Una vez puestos los parámetros indicaremos, tras los dos puntos, los tipos de cada parámetro en orden de aparición y finalmente el tipo devuelto por el operador tras una flecha.

Si seguimos con el ejemplo anterior de los tipos de Vehículo podemos definir un operador para cada tipo de vehículo, teniendo cada cual, como parámetro un número identificativo, bien pudiendo ser su matricula en caso de ser un coche o un número de serie en el caso de tratarse de una bicicleta. Además, para las bicicletas podemos definir un tipo *TipoBicicleta* con notaciones para los tipos montaña y ciudad. Hacemos uso de Maude y obtenemos el siguiente fragmento de código en la Ilustración 2.

```
*** Definición de los tipos de bicicleta (mo : Montaña, ci : Ciudad)
ops mo ci : -> TipoBicicleta .

*** Definición de un Coche con un número como matrícula identificativa
op c _ : Nat -> Coche .

*** Definición de una Bicicleta con un tipo y un número de serie
op b _ t _ : Nat TipoBicicleta -> Bicicleta .
```

Ilustración 2. Definición de los operadores para los tipos de bicicleta, coche y bicicleta.

En la Ilustración 2 tenemos la primera sentencia para la definición de los tipos de bicicleta como *mo* y *ci*, que serán posteriormente usados en la definición de la notación para las bicicletas en la última sentencia. Por tanto, una bicicleta con número

de serie uno y de tipo ciudad podemos escribirla como *b 1 t ci*. Para el coche tenemos la sentencia intermedia la cual recibe un número como matrícula.

2.1.3. *Axiomas*

Los axiomas son propiedades o atributos sobre conjuntos que se cumplen sobre un determinado operador cuando este se define en una secuencia de otro operador. Estos atributos son aplicados en Maude si los escribimos entre los símbolos *[* y *]* al final del operador secuencia, es decir, el operador que contiene una secuencia de otros operadores del mismo tipo. Los axiomas definidos por Maude que nos interesan son:

- *assoc* para indicar la propiedad asociativa.
- *comm* aplica la propiedad conmutativa.
- *id:e* para indicar que el operador *e* es el elemento identidad.

Para ver un ejemplo de como aplicar los atributos vamos a usar el tipo Vehículo para definir un operador que indique un conjunto de vehículos sobre el que aplicaremos las propiedades de interés. Primero creamos el tipo ConjuntoVehiculos, por ejemplo, utilizando la sentencia «*sort ConjuntoVehiculos .*». Posteriormente indicamos mediante una relación de subtipos, por ejemplo «*subsort Vehiculo < ConjuntoVehiculos .*», que un vehículo es un tipo de conjunto de vehículos, es decir, un conjunto donde solo hay un elemento. La siguiente Ilustración 3 muestra el código necesario para crear un conjunto de vehículos en Maude con las tres propiedades de conjuntos.

```
*** Declaración de relación de subtipo entre tipos
subsort Vehiculo < ConjuntoVehiculos .

*** Definición de el operador neutro de un conjunto de vehículos
op vacio : -> ConjuntoVehiculos .

*** Definición de un conjunto de vehículos
op __ : ConjuntoVehiculos ConjuntoVehiculos -> ConjuntoVehiculos [assoc comm
id: vacio] .
eq V:Vehiculo V:Vehiculo = V:Vehiculo .
```

Ilustración 3. Definición en Maude de un operador conjunto con propiedades asociativa, conmutativa y un elemento neutro.

En la segunda sentencia definimos el operador *vacio* que hará de elemento identidad para el conjunto de vehículos. Luego, en la tercera sentencia definimos el operador para un conjunto de vehículos como una secuencia de otros conjuntos de vehículos. Fijémonos como definimos al final los tres atributos para el nuevo operador, indicando la propiedad asociativa, conmutativa, y el elemento identidad *vacio*.

Es interesante que nos fijemos en la última sentencia donde hemos definido una ecuación para indicar que, si en un conjunto de vehículos hay dos vehículos idénticos,

uno de ellos sea eliminado del conjunto. De esta forma, definimos en Maude la eliminación de duplicados sobre conjuntos.

2.1.4. *Variables*

Las variables en Maude son definidas usando la estructura «*var* <*id*> *:* <*var-sort*> *.*», donde <*id*> es el nombre con el cual identificar la variable y <*var-sort*> el tipo que puede contener. Si queremos definir más de una variable del mismo tipo en una sola sentencia, podemos sustituir el *var* anterior por *vars*.

Como ejemplo podemos definir tres variables con identificadores *c, b, v*. El identificador *c* es para una variable de tipo *Coche*. Para *b* el tipo será *Bicicleta*. Mientras que para *v* será *Vehículo* como tipo. En cada variable puede guardarse un operador que devuelva el tipo especificado para la variable. Además, gracias a la relación de subtipos, la variable *v* puede almacenar también los contenidos de *c* o *b*.

2.1.5. *Módulos de sistema*

Los módulos de sistema nos permiten modelar sistemas en Maude, teniendo un comportamiento indeterminista con la posibilidad de ocurrir ejecuciones infinitas. Los definimos haciendo uso de la estructura *mod* <*ModuleName*> *is* <*DeclarationsAndStatements*> *endm*, donde <*ModuleName*> es el identificador del módulo en mayúsculas y <*DeclarationsAndStatements*> son sentencias. Como sentencias podemos hacer uso de las que ya hemos visto, como son los tipos y los operadores, aunque también existen las ecuaciones y las reglas.

Los módulos de sistema especifican teorías de reescritura. Las teorías son descritas con la forma $R = (\Sigma, E, \varphi, R)$. El elemento Σ representa el conjunto de todos los estados del sistema. El símbolo E representa la teoría de ecuaciones definida por las ecuaciones del módulo. El símbolo φ es una función que asigna un conjunto de números naturales a los estados de Σ según su número de argumentos. El último elemento, R, es el conjunto de reglas del sistema que especifican las transiciones entre los estados que hay en Σ.

Las ecuaciones son sentencias con la forma «*eq* <*Term-1*> = <*Term-2*> *.*», y transforman la parte izquierda del igual en la parte derecha, de forma que el *Term-1* pasa a ser *Term-2*. Una forma de ver las ecuaciones es como transformaciones deterministas que nos permiten reducir o simplificar expresiones.

Las reglas son como las ecuaciones a diferencia de que al aplicar estas estamos cambiando de estado dentro del sistema modelado, es decir, hay un incremento relativo de tiempo. Las reglas tienen la forma «*rl [id] : i => d .*», teniendo un identificador *id* opcional y convirtiendo la parte izquierda *i* en la derecha *d*. Veamos un ejemplo para explicar los módulos de sistema siguiendo con lo que ya hemos visto de los coches y bicicletas.

Supongamos que queremos modelar el funcionamiento de un cruce con dos semáforos, donde estarán los dos tipos de vehículos, coches y bicicletas. En este cruce a la carretera norte-sur llamaremos *cardo* y a la carretera este-oeste *decumano*. Los estados posibles de los semáforos serán verde o rojo. En los cruces podrá haber

vehículos, de los cuales tendremos coches y bicicletas tal cual los hemos definido en la sección de Operadores.

Definimos ahora lo que es la representación del sistema como un operador, para el cual hemos definido el tipo *CruceCon2Semaforos*. Dicho operador lo definimos mediante la sentencia:

«*op cardo [_] (_) [_] | decumano [_] (_) [_] : ConjuntoVehiculos Semaforo ConjuntoVehiculos ConjuntoVehiculos Semaforo ConjuntoVehiculos -> CruceCon2Semaforos .*»

La sentencia nos indica que las carreteras están separadas por el símbolo | y se identifican con *cardo* y *decumano* al principio. Los parámetros en cada carretera son, dos conjuntos de vehículos entre corchetes y un semáforo entre medias de los conjuntos de vehículos entre paréntesis.

Una vez tenemos la representación del sistema, podemos dar comportamiento gracias a las reglas. Para simplificar el modelo vamos a hacer que los vehículos solo puedan cruzar en línea recta. La Ilustración 4 muestra las reglas definidas siendo cuatro para cruzar vehículos y dos para cambiar el estado del semáforo.

```
*** Variables
vars ConjV1 ConjV2 ConjV3 ConjV4 : ConjuntoVehiculos .
var veh : Vehiculo .

*** Reglas del sistema
rl [CruzarCardo] : cardo [veh ConjV1] (verde) [ConjV2] | decumano [ConjV3]
(rojo) [ConjV4] => cardo [ConjV1] (verde) [veh ConjV2] | decumano [ConjV3]
(rojo) [ConjV4] .
rl [CruzarDecumano] : cardo [ConjV1] (rojo) [ConjV2] | decumano [veh ConjV3]
(verde) [ConjV4] => cardo [ConjV1] (rojo) [ConjV2] | decumano [ConjV3] (verde)
[veh ConjV4] .

rl [CruzarCardo2] : cardo [veh] (verde) [ConjV2] | decumano [ConjV3] (rojo)
[ConjV4] => cardo [vacio] (verde) [veh ConjV2] | decumano [ConjV3] (rojo)
[ConjV4] .
rl [CruzarDecumano2] : cardo [ConjV1] (rojo) [ConjV2] | decumano [veh] (verde)
[ConjV4] => cardo [ConjV1] (rojo) [ConjV2] | decumano [vacio] (verde) [veh
ConjV4] .

rl [CambioSemaforos1] : cardo [ConjV1] (rojo) [ConjV2] | decumano [ConjV3]
(verde) [ConjV4] => cardo [ConjV1] (verde) [ConjV2] | decumano [ConjV3] (rojo)
[ConjV4] .
rl [CambioSemaforos2] : cardo [ConjV1] (verde) [ConjV2] | decumano [ConjV3]
(rojo) [ConjV4] => cardo [ConjV1] (rojo) [ConjV2] | decumano [ConjV3] (verde)
[ConjV4] .
```

Ilustración 4. Reglas que modelan el comportamiento del sistema de un cruce con dos semáforos.

En primera instancia, definimos las variables que necesitaremos en las reglas. Hemos definido cuatro variables *Conj1, Conj2, Conj3* y *Conj4* como conjuntos de

vehículos y una variable *veh* de tipo vehículo para poder representar tanto coches como bicicletas.

Luego observamos las dos primeras reglas de cruzar, con identificadores *CruzarCardo* y *CruzarDecumano*, donde un vehículo podrá cruzar cuando su semáforo esté en verde. El mismo principio se aplica para las reglas con identificadores *CruzarCardo2* y *CruzarDecumano2*, pero en este caso se aplica cuando solo queda un vehículo, haciendo que el conjunto se quede vacío una vez cruce.

Finalmente, definimos dos reglas para modelar el cambio de estado de los semáforos. Ambas reglas hacen que se intercambien los estados de los dos semáforos entre sí.

En relación con las reglas, cabe destacar la existencia de las reglas condicionales. Las reglas condicionales son parecidas a las reglas normales solo que tienen la sintaxis «*crl [<label>] : <Term-1> => <Term-2> if <Condition-1> /\ … /\ <Condition-k> .*» donde las sentencias «*<Condition-1> /\ … /\ <Condition-k>*» son expresiones que devolverán false o true. Las condiciones de las reglas condicionales pueden optar tres formas:

1. Ser ecuaciones, con la forma *t = t'*.
2. *Matching equations*, con la forma *t := t'*.
3. Expresiones de reescritura, con la forma *t -> t'*.

Como comando de interés en los módulos de sistema tenemos el *search*, permitiéndonos buscar caminos de un estado a otro que le indiquemos. Este comando permite el uso de diferentes símbolos para indicarle los criterios para detener la búsqueda. Por defecto en Maude, las búsquedas son realizadas sobre un grafo de estados utilizando la aproximación *breadth first search*.

El comando *search* tiene la forma «*search <Term-1> <SearchArrow> <Term-2> .*», donde *<Term-1>* y *<Term-2>* son sentencias de estados válidos de nuestro sistema modelado. La palabra *<SearchArrow>* puede ser una las siguientes formas dando cada una un criterio distinto de proceder en la construcción del grafo de estados:

- =>*1* para hacer una búsqueda de un solo paso de ejecución, es decir se aplica solo una regla.
- =>+ para hacer una búsqueda de uno o más pasos de ejecución.
- =>* para hacer una búsqueda de ninguno, uno o más pasos de ejecución.
- =>*!* para hacer una búsqueda de un estado canónico, es decir, un estado al cual no se le puedan aplicar reglas que lleven a nuevos estados no explorados antes.

Por ejemplo, para el módulo de sistema del cruce con dos semáforos, podemos consultar si existe un camino desde un estado con varios coches en el *cardo*, hasta uno donde todos los coches en el *cardo* hayan cruzado el semáforo. Podemos añadir todas las especificaciones que queramos siempre que respetemos la sintaxis definida, es decir, que construyamos los estados como los hemos especificado en el módulo del sistema. Veamos algunos ejemplos de búsquedas.

Como primer ejemplo vamos a escribir un comando que nos busque si es posible llegar desde un estado con dos vehículos en el *cardo*, un coche y una bicicleta, y un coche en el *decumano*, hasta un estado con los dos vehículos en el *cardo*, pero ningún vehículo en el *decumano*. La sentencia se escribe como muestra la Ilustración 5, obteniendo por parte de Maude una solución a partir de aplicar cuatro reglas.

```
Maude> search cardo [c 1  b 1 t mo] (rojo) [vacio] | decumano [c 2] (verde) [vacio] ]
 =>* cardo [c 1  b 1 t mo] (verde) [vacio] | decumano [vacio] (rojo) [c 2] .
 search in CRUCE : cardo[c 1 b 1 t mo](rojo)[vacio]| decumano[c 2](verde)[vacio]
     =>* cardo[c 1 b 1 t mo](verde)[vacio]| decumano[vacio](rojo)[c 2] .

 Solution 1 (state 3)
 states: 4  rewrites: 4 in 0ms cpu (0ms real) (40816 rewrites/second)
 empty substitution

 No more solutions.
 states: 16  rewrites: 36 in 0ms cpu (0ms real) (129032 rewrites/second)
 Maude>
```

Ilustración 5. Ejecución del comando search para buscar un camino entre dos terminosen Maude.

Como segundo ejemplo, podemos probar algo más interesante. Vamos a ver si en el sistema que hemos modelado los dos semáforos pueden llegar a estar en verde simultáneamente. Las búsquedas de estados no deseados nos serán exitosas cuando Maude no encuentre ninguna solución, pues querrá decir que desde el estado inicial especificado no se llega a la situación que consideremos crítica en el sistema.

La Ilustración 6 muestra el uso de *search* con el mismo estado inicial anterior pero ahora tiene un estado final donde *cardo* y *decumano* tienen los semáforos en verde, situación que no debe ocurrir en un cruce.

```
Maude> search cardo [c 1  b 1 t mo] (rojo) [vacio] | decumano [c 2] (verde) [vacio]
=>* cardo [c 1  b 1 t mo] (verde) [vacio] | decumano [c 2] (verde) [vacio] .
search in CRUCE : cardo[c 1 b 1 t mo](rojo)[vacio]| decumano[c 2](verde)[vacio] =>*
    cardo[c 1 b 1 t mo](verde)[vacio]| decumano[c 2](verde)[vacio] .

No solution.
states: 16  rewrites: 36 in 0ms cpu (0ms real) (264705 rewrites/second)
Maude>
```

Ilustración 6. Ejecución del comando search para comprobar la seguridad del cruce.

La otra forma de especificar y verificar propiedades, de forma formal, sobre módulos de sistema es mediante el uso de la técnica de *model checking*. El *model checking* en Maude se lleva a cabo mediante la escritura de propiedades como fórmulas en *linear temporal logic* y al uso de módulos para describir predicados sobre los estados del sistema. Explicamos en mayor profundidad este concepto y lo ponemos en práctica en el Capítulo 6.

2.1.6. *Módulos funcionales*

Los módulos funcionales especifican funciones y se asemejan a los módulos de sistema con la diferencia de que solo pueden contener ecuaciones. Los módulos funcionales, a diferencia de los de sistema, son deterministas y finitos. Para definir un módulo funcional en Maude, lo hacemos entre las palabras reservadas *fmod* y *endfm*. Veamos un ejemplo de especificación de un módulo de tipo funcional.

Si nos fijamos en la siguiente Ilustración 7, tenemos un módulo funcional, definido entre *fmod* y *endfm*, el cual especifica mediante ecuaciones la función matemática factorial de un número *N*. En el módulo hemos definido el símbolo para dicha operación y las ecuaciones que transformarán la expresión hasta que sea irreducible, y así obtener el resultado.

```
fmod FACT is
    protecting NAT .
    op _! : Nat -> Nat .
    var N : Nat .
    eq 0 ! = 1 .                        --- factorial de N=0 es 1
    eq N ! = (sd(N,1))! * N [owise] . --- factorial de N>0 es (N-1)! * N
endfm
```

Ilustración 7. Código de un módulo funcional en Maude.

Siguiendo con la Ilustración 7, en la línea tres especificamos el símbolo de la función factorial, la cual obtiene un número natural en el sitio del símbolo _ y devuelve otro número natural, como indica la expresión *Nat -> Nat*.

Luego, en la línea cuatro declaramos la variable N, que usaremos en las siguientes ecuaciones de las líneas cinco y seis. Con la primera ecuación definimos que cuando Maude encuentre un *0 !* en la parte izquierda este sea traducida en el número uno. La segunda ecuación se ejecutará siempre que no se ejecute la primera gracias al atributo definido *owise*, y nos devolverá la función factorial de un número *N* como factorial de *N-1* multiplicado por *N*.

Además, cabe recalcar que al igual que hay reglas condicionales existen también ecuaciones condicionales. Estas se comportan de la misma forma que las ecuaciones descritas con la diferencia de que para poder aplicarse deben pasar una o varias condiciones, de la misma forma que las condiciones de las reglas. La sintaxis para las ecuaciones condicionales es «*ceq <Term1> = <Term2> if <Condition-1> /\ … /\ <Condition-k> .*», donde las condiciones <*Condition-k*> son expresiones que devolverán false o true. Las condiciones de las ecuaciones condicionales pueden optar dos formas:

1. Ser ecuaciones, con la forma *t = t'*.
2. *Matching equations*, con la forma *t := t'*.

Para finalizar, vamos a ver el comando *reduce*. Este comando nos es de gran utilidad para probar ecuaciones que hayamos definido, ya sea en módulos funcionales o en módulos de sistema. El comando reduce recibe una expresión y nos devuelve la máxima expresión reducida utilizando las ecuaciones que hayamos definido.

Como ejemplo vamos a utilizar la definición de la operación factorial en el módulo funcional *FACT* para probar diferentes expresiones. Por ejemplo, como muestra la Ilustración 8, probamos a reducir factorial de tres y factorial de cero. El primero nos devuelve seis y el segundo uno, los cuales son correctos. Además, como en el reduce no se aplican reglas y en los módulos funcionales no hay reglas podemos ver cómo en ambas ejecuciones el número de reescrituras es cero, indicando que no se aplicó ninguna regla y por tanto no se cambió de estado.

```
Maude> reduce 3 ! .
reduce in FACT : 3 ! .
rewrites: 10 in 0ms cpu (0ms real) (277777 rewrites/second)
result NzNat: 6
Maude> reduce 0 ! .
reduce in FACT : 0 ! .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result NzNat: 1
Maude>
```

Ilustración 8. Ejecución de dos expresiones factoriales con el comando reduce.

2.2. CafeOBJ

Es un lenguaje formal de especificación de sistemas y verificación de propiedades, que hace uso de lógica de ecuaciones y teorías de reescritura. Comparte muchas similitudes con Maude, pero con características propias. La versión que vamos a tratar, y que es usada en [1] ,base de este trabajo, es CafeOBJ 1.5.7 disponible en [3].

A fin de seguir una estructura similar ahora vamos a ver todos los conceptos que vimos de Maude, pero con la sintaxis propia de CafeOBJ. Primero veremos como declarar tipos y a definir la relación de subtipo entre diferentes tipos. Tras los tipos, veremos los operadores y las variables. Luego, aprenderemos a declarar ecuaciones, pues no existen las reglas en CafeOBJ. Por último, veremos que existe un tipo de módulo, el cual es similar a los módulos de sistema de Maude, y veremos brevemente como son verificadas las propiedades en CafeOBJ.

2.2.1. *Tipos y subtipos*

Los tipos son definidos entre los símbolos *[* y *]* y siguiendo la estructura *[<sort-name> … <sort-name>]*. La sentencia <*sort-name*> es el nombre del tipo a declarar. A diferencia de Maude, para declarar varios tipos escribimos en secuencia los nombres de los tipos. Por ejemplo, la línea *[Bicicleta Coche Vehiculo]* declara los tipos de vehículos, bicicletas y coches, y el tipo vehículo.

Para la declaración de subtipos, se sigue una estructura similar a la declaración de tipos con la estructura *[<sort-name> < <sort-name>]*. Las sentencias <*sort-name*> a cada lado del símbolo < indican los nombres de los tipos. El nombre a la derecha es el tipo padre, mientras que el tipo a la izquierda es el subtipo. Si queremos definir la misma relación de subtipos para *Bicicleta* y *Coche* con *Vehiculo*, igual que en la sección 2.1.1, escribimos las líneas *[Bicicleta < Vehiculo]* y *[Coche < Vehiculo]*.

2.2.2. *Operadores*

Los operadores son definidos con la palabra reservada *op*. La estructura completa es «*op <op-name> : <arity> -> <sort-of-op> { <operator-attribute> … <operator-attribute> } .*». Con <*op-name*> indicamos el nombre del operador, donde con los símbolos _ señalamos las posiciones de los parámetros. La parte siguiente de los dos puntos es conocida como rango. En el rango encontramos dos componentes.

Una componente es un conjunto de tipos especificados de forma secuencial y en orden de uso de los símbolos _ en <*arity*>. El otro componente *es* el tipo devuelto por el operador en <*sort-of-op*>.

En el caso de que varios operadores compartan rango pueden definirse en conjunto utilizando la estructura «*ops* <*op-name*> … <*op-name*> *:* <*arity*> -> <*sort-of-op*> *{* <*operator-attribute*> … <*operator-attribute*> *}*».

La sentencia que declara un operador para modelar las bicicletas con un identificador y el tipo de bicicleta es «*op b _ t _ : Nat TipoBicicleta -> Bicicleta*» donde podemos observar que es idéntica a la versión de la sección 2.1.1 utilizando Maude.

Al igual que con Maude, un operador puede no definir argumentos, es decir tener el conjunto <*arity*> vacío. En tal caso se denomina al operador constante. Operadores constantes son, por ejemplo, los declarados mediante la sentencia «*ops true false : -> Bool*», para definir los dos valores lógicos de veracidad y falsedad.

2.2.3. *Variables*

Las variables son declaradas de antemano a su uso en las ecuaciones. Su alcance reside dentro de la definición del módulo, por tanto, a menos de que el módulo sea importado en otro módulo, solo existen en el módulo donde son declaradas. La estructura de las variables es *var* <*var-name*> *:* <*sort-name*> donde <*var-name*> es el nombre identificativo de la variable y <*sort-name*> el nombre del tipo que puede contener.

En el caso de que queramos declarar muchas variables en una sentencia debemos seguir la estructura *vars* <*var-name*> … <*var-name*> *:* <*sort-name*>. Con esta estructura son declaras mas de una variable del mismo tipo.

Las variables declaradas con *var c : Coche*, *var b : Bicicleta* y *var v : Vehículo* pueden cumplir el propósito de almacenar diferentes valores en las ecuaciones. En el caso de *c* este representa un coche*, b* una bicicleta y *v* un vehículo, bien pudiendo ser un coche o una bicicleta.

2.2.4. *Ecuaciones*

Las ecuaciones son sentencias que especifican equivalencias entre términos. La estructura que tienen es *eq* <*lhs*> = <*rhs*>, seguido de un espacio y un punto para terminar la línea. El punto al final sigue el mismo propósito que el punto y coma de los lenguajes de programación Java o C. El comportamiento de las ecuaciones es transformar el término identificado con <*lhs*>, ubicado en la parte izquierda del igual, en el término <*rhs*>, encuentrado entre la derecha del igual y el punto del final de línea.

Podemos definir ecuaciones para dar comportamiento a los operadores. Con el fin de ver un ejemplo, modelamos un operador para representar una moto y dos ecuaciones. El operador para la moto es «*op m _ v _ : Nat Nat -> Moto .*» que representa una moto con primero un identificador y luego la velocidad a la que va. Las dos ecuaciones que vamos a definir van a realizar que una moto pueda acelerar o frenar. Pero recordémonos antes de definir una variable para cada parámetro que consideremos necesario. Nosotros definimos las variables *Id* y *Vel* de tipo *Nat* para

representar un valor numérico para el identificador de la moto y otro valor numérico natural para la velocidad en kilómetros por hora.

La ecuación para acelerar tiene la forma «*eq m Id v Vel = m Id v (Vel + 1)* .» donde la velocidad aumenta en una unidad. La ecuación para frenar tiene la forma «*eq m Id v Vel = m Id v (Vel - 1)* .» donde la velocidad es reducida en una unidad.

También existen las ecuaciones condicionales con la estructura «*ceq <lhs> = <rhs> if <condition> .*» donde *<condition>* es un término del tipo *Bool* perteneciente al módulo *BOOL*. Podemos ampliar las ecuaciones anteriores para que contengan una condición donde la moto no pueda superar un límite de velocidad. La ecuación condicional sería «*ceq m Id v Vel = m Id v (Vel + 1) if V < 89* .» de forma que la moto nunca superará los 90 kilómetros por hora.

2.2.5. *Módulos de sistema*

Los módulos de sistema son la unidad básica de modelado de sistemas. Los módulos son definidos mediante sentencias entre los símbolos corchetes. La estructura general es *mod <module-name> { <module-elements> }* donde *<mod-name>* es el nombre identificador del módulo, que suele escribirse en mayúsculas. Los elementos del módulo son el elemento *<module-elements>* y pueden tratarse de cualquiera de los conceptos que hemos visto hasta ahora. Desde declaraciones de tipos y variables, definición de operadores y la escritura de ecuaciones para dichos operadores.

Existe el concepto denominado *observable transition system*, que es explicado en [4]. Este concepto trata de la construcción de modelos mediante observadores. Los observadores son operadores cuyo comportamiento es definido mediante ecuaciones para observar un cierto aspecto del sistema.

Los observadores sirven el propósito de devolver valores del sistema, de tal forma que podemos usarlos para definir, por ejemplo, el estado inicial de un sistema, a partir de los valores iniciales que debe tener. Otra utilidad es poder definir transiciones en base a los valores de los observadores, antes y después de que ocurra una transición.

Siguiendo el ejemplo del cruce con dos semáforos definido con Maude en la sección 2.1.5, vamos a explicar como modelaríamos el sistema utilizando un *observable transition system*. En el módulo de sistema definimos cinco operadores.

Tenemos entonces un operador *init* definido a través de la sentencia «*op init : -> CruceCon2Semaforos .*» que es el encargado de representar el estado inicial del sistema. Los dos operadores transición *cruzarCardo* y *cruzarDecumano* reciben como argumentos un operador de tipo *Vehiculo* y un operador de tipo *CruceCon2Semaforos*. Por último, los operadores transición *cambioCardo* y *cambioDecumano* reciben como parámetro un operador de tipo *CruceCon2Semaforos*.

Estos últimos cuatro operadores son las transiciones que teníamos antes definidas en Maude mediante reglas, exceptuando CruzarCardo2 y CruzarDecumano2 que ya no son necesarias. Si nos fijamos en las etiquetas de las reglas de Maude estas empiezan en mayúsculas, y para distinguirlos en la versión de CafeOBJ ahora los cuatro operadores de transición empiezan en minúscula.

Ahora definimos los observadores mediante operadores y su comportamiento en base a los operadores, que acabamos de definir, mediante ecuaciones. Los observadores son *semaforoCardo, semaforoDecumano, conjV1, conjV2, conjV3* y *conjV4*. Tenemos entonces los dos semáforos del cruce y los cuatro conjuntos de vehículos, siendo los dos primeros para el *Cardo* y los dos últimos para el *Decumano*.

Los observadores *semaforoCardo* y *semaforoDecumano* reciben como argumento el estado del sistema como un operador de tipo *CruceCon2Semaforos*. En cambio, los observadores *conjV1, conjV2, conjV3* y *conjV4* reciben además un operador de tipo *SetVehiculos* que representa un conjunto de vehículos y que identificaremos como *setV1*, *setV2*, *setV3* y *setV4*.

Los dos observadores *semaforoCardo* y *semaforoDecumano* son los que nos indican en que estado se encuentra un semáforo. Mediante las ecuaciones «*eq semaforoCardo(init) = verde .*» y «*eq semaforoDecumano(init) = rojo .*» estamos definiendo los valores que tienen los semáforos en el estado inicial *init*.

Recordemos que en Maude tenemos las reglas *CambioSemaforos1* y *CambioSemaforos2* para intercambiar el valor de los semáforos entre sí. Ahora, en CafeOBJ, el equivalente es definir una ecuación condicional para cada semáforo. Las ecuaciones siguientes definen el comportamiento de los observadores *semaforoCardo* y *semaforoDecumano* cuando ocurre el operador transición *cambioCardo* y *cambioDecumano* sobre una variable S de tipo *CruceCon2Semaforos*.

ceq semaforoCardo(cambioCardo(S)) = (if semaforoCardo(S) = verde then rojo else verde fi) .

ceq semaforoDecumano(cambioDecumano(S)) = (if semaforoDecumano(S) = verde then rojo else verde fi) .

Como vemos, las ecuaciones comprueban mediante los mismos observadores, semaforo*Cardo* o *semaforoDecumano,* que estado tiene el semáforo y en base al resultado devuelven la constante contraria. De esta forma las reglas *CambioSemaforos1* y *CambioSemaforos2* de Maude se convierten ahora en los operadores transición *cambioCardo* y *cambioDecumano*.

Para modelar las anteriores reglas *CruzarCardo* y *CruzarCardo2* tenemos ahora el operador transición *cruzarCardo*. Lo mismo ocurre con *CruzarDecumano* y *CruzarDecumano2*, correspondiendo a *cruzarDecumano*. El comportamiento de estos operadores transición lo llevamos a cabo mediante los observadores de conjuntos correspondientes. Para ello definimos dos ecuaciones condicionales para cada operador transición.

Vamos a definir la transición *cruzarCardo* pues *CruzarDecumano* se hacen de forma muy similar. Debemos suponer que previamente hemos definido los operadores *add(V,C),* para añadir un vehículo *V,* y *remove(V,C)*, para quitar un vehículo *V,* de un conjunto *C* de tipo *SetVehiculos*.

La transición *CruzarCardo* que teníamos en Maude y que ahora es el operador transición *cruzarCardo* se define mediante las ecuaciones siguientes sobre los observadores *conjV1* y *conjV2*:

ceq conjV1(setV1, cruzarCardo(V,S)) = (if semaforoCardo(S) = verde then remove(V, setV1) fi) .

ceq conjV2(setV2, cruzarCardo(V,S)) = (if semaforoCardo(S) = verde then add(V,setV2) fi) .

Estas dos ecuaciones especifican el comportamiento de los observadores *conjV1* y *conjV2,* sobre los conjuntos de vehículos *setV1* y *setV2,* cuando ocurre el operador transición *cruzarCardo* para un vehículo *V* en un sistema *S*. En la primera ecuación, si el estado del semáforo del cardo es *verde*, el vehículo *V* es eliminado del conjunto de vehículos *setV1*. En la segunda ecuación, si el estado del semáforo del cardo es *verde*, el vehículo *V* es añadido al conjunto de vehículos *setV2*.

Por último, la forma de verificar propiedades sobre los módulos de sistemas se hace mediante *proof score method* utilizando *proof passages*, mostrados también en [4]. En CafeOBJ se suele utilizar el comando *reduce*, igual que el de Maude en la sección 2.1.6. Con este comando y junto a un termino *t* aplican las ecuaciones especificadas para desarrollar el sistema hasta un nuevo término *t'*. Es sobre este nuevo termino *t'* donde son comprobadas las propiedades mediante invariantes.

La definición de invariantes es llevada a cabo mediante la creación de módulos. Las invariantes son predicados que se cumplen en un estado del sistema. Además de las invariantes es necesario definir *induction steps* con los cuales comprobar que si un estado anterior cumple una propiedad el nuevo estado, que surge de aplicar una ecuación, también cumple la propiedad definida.

Una explicación exhaustiva se encuentra en el capítulo 4 sección 4.4. Ahora nos interesa explicar brevemente como haríamos para verificar la propiedad de que los dos semáforos nunca estarán a la vez en verde.

En nuestro caso podemos definir una invariante mediante una ecuación condicional donde especifiquemos que en un estado del sistema siempre que haya dos semáforos distintos en un cruce y uno de ellos esté en *verde*, el otro semáforo estará en el estado contrario *rojo* y viceversa.

Una vez tenemos la invariante definimos un *induction step* para comprobar que pase lo que pase en el sistema la invariante se sigue cumpliendo. En otras palabras, si en el estado inicial se cumple la invariante de los semáforos y ocurre un cambio de semáforo o un vehículo decide cruzar, entonces debemos ser capaces de inducir que la invariante no ha dejado de cumplirse.

Los *induction steps* son ecuaciones con implicaciones. De esta manera podemos definir una ecuación donde inducimos que, si la invariante de los semáforos se cumple en un estado y ocurre un cruce de un vehículo, entonces esto implica que la invariante se sigue cumpliendo, pues los semáforos no han cambiado durante la transición de estados.

Finalmente, con el estado inicial, las invariantes, los pasos de inducción y el comando *reduce* podemos comenzar la verificación. El objetivo sería verificar si el sistema modelado cumple la propiedad de que dos semáforos nunca estarán en verde al mismo tiempo para todos los estados alcanzables desde el estado inicial.

3. ANÁLISIS DEL PROTOCOLO FISCHER

Vamos a ver durante esta sección las diferentes versiones propuestas en la literatura sobre el protocolo de exclusión mutua en tiempo real Fischer. Destacaremos algunos trabajos previos sobre el protocolo, donde explicaremos su funcionamiento desde una primera versión algorítmica propuesta por primera vez en [5]. Luego pasaremos a una versión del protocolo donde usan una perspectiva algebraica de manera informal presentada en [6]. Continuaremos con la versión en forma de autómata propuesta por [7]. Finalmente, explicaremos el protocolo desde el punto de vista de [1] usando un autómata de estados.

El protocolo de exclusión mutua en tiempo real Fischer para múltiples procesos en sistemas concurrentes es un protocolo que hace uso del tiempo para conseguir la propiedad de exclusión mutua sobre una zona critica. Surgió en los años 90 propuesto por Michael J. Fischer en [8] y estudiado por primera vez por Leslie Lamport en [5].

3.1. Algoritmo

En el artículo de Lamport nos es presentada la primera versión del algoritmo para su uso en procesadores con memoria compartida y que podemos ver a continuación en la Ilustración 9. En la Ilustración 9 vemos también que hay una sentencia *await b*, la cual interpretaremos como una instrucción *while* con condición $\neg b$ y cuerpo *skip*. Además, las sentencias enmarcadas entre los símbolos < y > son consideradas lecturas y escrituras atómicas. Así pues, Lamport nos comenta que el algoritmo asume un número positivo como identificador único para cada proceso. Finalmente, también se asume que el código externo al algoritmo que pueda estar en el programa no afecta a las variables usadas en el algoritmo.

$$
\begin{array}{l}
\textbf{repeat await } \langle\, x = 0 \,\rangle; \\
\qquad\qquad \langle\, x := i \,\rangle; \\
\qquad\qquad \langle\, delay \,\rangle \\
\textbf{until } \langle\, x = i \,\rangle; \\
critical\ section; \\
x := 0
\end{array}
$$

Ilustración 9. Primera versión en [5] del algoritmo propuesto por Fischer para un proceso i.

Todos los procesos, que identificaremos a cada uno con un número natural positivo *i,* tienen acceso a una variable compartida *x*. Cada proceso está en un *loop* donde espera que la variable compartida tome el valor cero, indicio de que no hay nadie en la zona crítica. Una vez se cumple la condición el proceso procede a escribir en la variable compartida el valor de su identificador. Luego con la sentencia *delay* ocurre una espera de un cierto tiempo para poder asegurar que cualquier otro proceso *i'* que también haya pasado la condición *x = 0* pueda escribir sobre la variable compartida su número identificativo antes de que el proceso *i* vuelva a comprobar el contenido de *x*. Gracias a esta espera, evitamos que el proceso *i* entre a la sección crítica, antes de que *i'* o cualquier otro proceso hayan podido sobrescribir *x*. A continuación, se comprueba si en efecto x contiene el valor del identificador del proceso que quiere entrar, y realiza las operaciones convenientes. Por último, se libera

la variable compartida x escribiendo el valor cero, valor que no puede tener ningún otro proceso como identificador.

3.2. Versión informal

A partir del algoritmo visto han surgido versiones informales como las que trata [6]. En este artículo el objetivo es verificar el protocolo de exclusión mutua Fischer utilizando *Timed Process Algebra*, que no describiremos en esta memoria porque no es necesario para la comprensión del protocolo Fischer.

El primer ejemplo informal es una versión parecida al algoritmo de Lamport en [5], pero especificando para dos procesos, no de forma genérica. Además, es añadida a la escritura final de x = 0 los corchetes que indican atomicidad. A nosotros nos es de especial interés el análisis de límites de tiempos que hace sobre la segunda versión informal. Esta segunda versión, que puede observarse en la Ilustración 10, es idéntica a la mostrada en la Ilustración 9, pero añadiendo límites relativos de tiempo, superiores e inferiores, a cada sentencia del algoritmo.

Component 1:

repeat
  repeat
    await $\langle x = 0 \rangle_0^\infty$;
    $\langle x := 1 \rangle_a^{a'}$;
    $\langle delay \rangle_d^{d'}$;
  until $\langle x = 1 \rangle_0^0$;
  critical section 1;
  $\langle x := 0 \rangle_a^{a'}$;
until false;

Component 2:

repeat
  repeat
    await $\langle x = 0 \rangle_0^\infty$;
    $\langle x := 2 \rangle_a^{a'}$;
    $\langle delay \rangle_d^{d'}$;
  until $\langle x = 2 \rangle_0^0$;
  critical section 2;
  $\langle x := 0 \rangle_a^{a'}$;
until false;

Ilustración 10. Versión informal en [6] del algoritmo Fischer para dos procesos.

Podemos ver en la Ilustración 10, que el algoritmo de Fischer ha sido concretado para dos procesos, uno con identificador uno y otro con identificador dos. A las operaciones de escritura y lectura sobre la variable compartida *x* le han sido añadidos dos límites, un límite inferior y uno superior, que son comunes para ambos procesos en cada una de las sentencias.

Al comenzar, la sentencia *await*, *es decir,* el proceso de espera a que la variable compartida esté libre, puede tomar entre cero e infinitas unidades de tiempo en llevarse a cabo. A continuación, la asignación del identificador propio *i* a la variable compartida mediante la sentencia *x:=i* puede tomar entre *a* y *a'* unidades de tiempo, que son parámetros del protocolo. De igual forma, la sentencia *delay* entre *d* y *d'* unidades de tiempo, que también son parámetros del protocolo.

Para todas las sentencias se supone que el límite inferior en el mismo proceso es inferior al superior. Además, se asume que el límite superior *a'* es menor al límite inferior *d,* de tal forma que la demora siempre toma más tiempo en llevarse a cabo que una asignación. Luego, la comprobación de que en efecto la variable compartida contiene el identificador del proceso actual que va a entrar en zona crítica se hace de forma instantánea. Con la sección crítica no hay límites establecidos, y por lo tanto se

asume que se tomara un tiempo indeterminado. Finalmente, la liberación de la variable compartida se hace con los mismos límites superior e inferior que cuando se escribió nada más estar libre.

Podemos deducir, como nos dicen en el artículo, que asumiendo el cumplimiento de $0 \leq a \leq a' < d \leq d' < 0$ el proceso dos tardará como máximo *a'* unidades de tiempo en completar *x:=2*. En caso de que el proceso uno haya sobrescrito *x* justo después de que el proceso dos haya leído su contenido, el proceso uno tardará como mínimo *d* unidades de tiempo en completar la demora. Por tanto, como $a' < d$, cuando el proceso uno llegue a la comprobación instantánea de *x=1*, el proceso dos habrá tenido el tiempo suficiente de escribir en *a'* unidades de tiempo su identificador en *x*. De esta forma la variable *x* ha tenido el tiempo suficiente para estabilizar su contenido evitando que el proceso uno haga una lectura incorrecta de *x* cuando el proceso dos aun no había escrito si identificador.

Como concepto final, es interesante ver como en esta nueva versión se aplican límites, superiores e inferiores, a cada proceso y que además son comunes entre ellos. Esto permite que veamos la implicación que tiene el tiempo real en el protocolo Fischer.

3.3. Versión formal

Ampliemos ahora esta versión del protocolo con una versión más formal como nos indica [7]. En esta tesis se nos presenta una versión del protocolo con fases teniendo que hacer acciones para pasar de una fase a otra, como nos muestra la Ilustración 11. Cabe añadir que a los procesos los llaman usuarios, y el identificador que tienen es un nombre. Además, sigue habiendo una única variable compartida, cuyo valor es cero cuando no hay ningún usuario en la zona crítica.

Shared variable: $x \in I \cup \{0\}$, initially 0.

(pc_i)

remainder	*** Remainder Region ***
	try_i
testing	wait until $x = 0$
set	$x \leftarrow i$
	pause
checking	if $x \neq i$ then goto test
leave-trying	crit_i
critical	*** Critical Region ***
	exit_i
reset	$x \leftarrow 0$
leave-exit	rem_i (goto *remainder*)

Ilustración 11. Fases en pseudocódigo del autómata Fischer.

Como podemos observar en la Ilustración 11, las sentencias que antes teníamos en el algoritmo ahora pasan a ser acciones. Dentro de estas acciones tenemos agrupaciones de acciones que conforman las fases del autómata. Por tanto, podemos

relacionar la Ilustración 11 con la siguiente Ilustración 12 donde se presentan los estados, acciones y límites temporales de cada acción del autómata Fischer. Debemos destacar las acciones, cada una de las cuales está compuesta por una precondición y un resultado de aplicar la acción. La precondición suele referirse a que el usuario debe encontrarse en un cierto estado

State
$pc_i \in \{remainder, testing, set, checking, leave\text{-}trying, critical, reset, leave\text{-}exit\}$ for $i \in I$, initially *remainder*
$x \in I \cup \{0\}$, initially 0

Actions

External try_i
Pre: $pc_i = remainder$
Eff: $pc_i \leftarrow testing$

Internal $test_i$
Pre: $pc_i = testing$
Eff: if $x = 0$ then $pc_i \leftarrow set$

Internal set_i
Pre: $pc_i = set$
Eff: $x \leftarrow i$
$pc_i \leftarrow checking$

Internal $check_i$
Pre: $pc_i = checking$
Eff: if $x = i$
then $pc_i \leftarrow leave\text{-}trying$
else $pc_i \leftarrow testing$

External $crit_i$
Pre: $pc_i = leave\text{-}trying$
Eff: $pc_i \leftarrow critical$

External $exit_i$
Pre: $pc_i = critical$
Eff: $pc_i \leftarrow reset$

Internal $reset_i$
Pre: $pc_i = reset$
Eff: $x \leftarrow 0$
$pc_i \leftarrow leave\text{-}exit$

External rem_i
Pre: $pc_i = leave\text{-}exit$
Eff: $pc_i \leftarrow remainder$

Tasks
Assume $a < b \leq c$
$\{try_i\}$: $[0, \infty]$
$\{test_i\}$: $[0, a]$
$\{set_i\}$: $[0, a]$
$\{check_i\}$: $[b, c]$
$\{crit_i\}$: $[0, a]$
$\{exit_i\}$: $[0, \infty]$
$\{reset_i\}$: $[0, a]$
$\{rem_i\}$: $[0, a]$

Ilustración 12. Estados, acciones y límites temporales de las acciones del autómata Fischer.

Al principio, el usuario se encuentra en la fase de espera, que en el caso del algoritmo es la sentencia *await*, y que identifican en el autómata como *reminder*. Una vez el usuario hace la acción *try*, su estado pasa a ser *testing* queriendo indicar que está intentando entrar en la zona crítica. En el estado *testing* espera a que la variable compartida esté libre y pasa a realizar la acción *set*, con la cual escribe su nombre en *x*.

Una vez escrito el nombre, pasa al estado *checking* donde comprobará si en efecto la variable *x* continúa con valor igual a su nombre. En este estado hay dos posibles alternativas. Si la condición *x=i,* siendo *i* el nombre del usuario, se cumple entonces pasamos a un nuevo estado denominado *leave-trying* donde se efectuará la acción *crit* con la que el usuario accederá a la sección crítica. En caso de no cumplirse la condición *x=i* el estado al que pasa el usuario es de nuevo *trying* para realizar un nuevo intento de acceso.

Finalmente, una vez el usuario ha terminado de usar la sección crítica, procede salir de ella realizando la acción *exit*, con la cual hará *reset* para poner en *x* el valor cero con el que indicar que vuelve a estar libre. Con esto, el último paso es volver al

estado de espera del principio, *reminder*, donde el usuario podrá volver a intentar entrar en la sección critica.

Es interesante destacar de esta especificación la ausencia de la demora justo antes de comprobar con *check* el valor de *x*. Esta demora faltante es el *delay* que tenemos en el algoritmo propuesto tanto en la Ilustración 9 como en la Ilustración 10. Esta ausencia es debida a la existencia de la asunción de dos propiedades sobre los límites de las acciones *set* y *check*. Como bien nos explica el artículo, para poder conseguir la propiedad de exclusión mutua sobre el autómata debe cumplirse que $upper(set_i) < lower(check_j)$. Por tanto, el límite superior de la acción *set* para un usuario *i* debe ser menor que el límite inferior de la acción *check* para un usuario *j*. En otras palabras, el usuario *i* deberá completar *set* antes de que se alcance el instante de tiempo donde podrá ocurrir la acción *check* del usuario *j*.

3.4. Versión autómata

Por último, vamos a ver el autómata propuesto en [1]. En el nuevo autómata tenemos seis estados, similares a los que teníamos con el anterior. Además, en lugar de tener usuarios tenemos procesos con un número positivo como identificador, de igual forma que en el algoritmo.

Los estados son *a*, *b*, *c*, *d*, *cs* y *e*. Podemos relacionar al estado *a* con el estado reminder. Los estados *b, c, y d* son los estados por los que pasa un proceso para intentar entrar en la sección crítica. El estado *cs* corresponde con el proceso estando en *critical section*, y el estado *e* para indicar la salida del proceso de sección crítica.

Cabe mencionar la existencia de una variable compartida entre procesos denominada *turn* para contener el identificador de un proceso. La variable *turn* cuenta con un valor especial *nop,* indicando así que ningún proceso tiene el turno. Todos los estados y las transiciones entre los estados las podemos ver en la Ilustración 13.

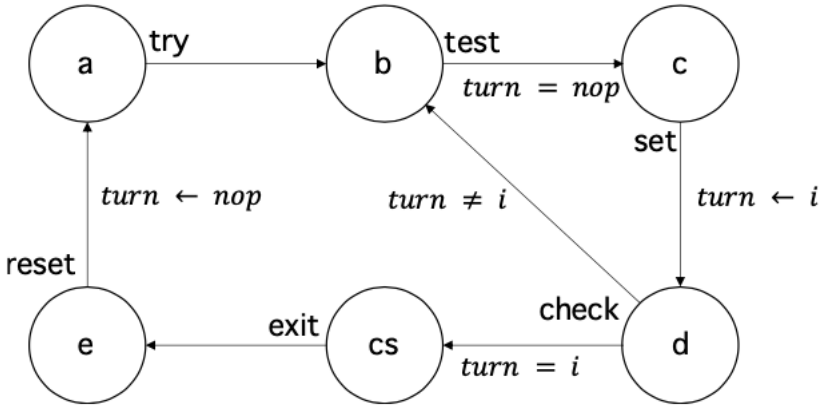

Ilustración 13. Autómata del protocolo Fischer con seis estados y una variable compartida turn.

El autómata de la Ilustración 13 presenta transiciones que podemos asociar a las acciones vistas en el autómata de [7], visible en la Ilustración 10. Vemos que para pasar del estado *a* al estado *b* hay una transición *try*, la cual solo se aplicará en el estado *a*. El mismo proceso se aplica para las demás transiciones y estados, pero teniendo algunas transiciones condiciones. Por ejemplo, la transición *test* solo puede

ocurrir cuando en b el valor de la variable *turn* sea *nop*, indicándonos que la sección crítica esta vacía pues nadie tiene asignado el turno. Otras transiciones como *set* y *reset* aplican una acción, en este caso una escritura sobre una variable. En la transición *set,* la variable *turn* se pone con el valor identificador del proceso. En la transición *reset* en cambio, la variable *turn* toma el valor *nop*.

Además de definir el autómata que representa el protocolo Fischer, los autores han definido dos variables, *u* y *l*, propias para cada proceso. Este par de variables se encargan de guardar los límites superior e inferior de las transiciones en cada estado del proceso. En igual forma, han definido dos constantes d_1 y d_2 globales del sistema. Estas constantes tienen el propósito de guardar dos valores que sirven al propósito de especificar los valores de los límites superior e inferior de las transiciones junto con el tiempo del sistema. Veamos como actúan estas constantes globales del sistema sobre las variables locales de cada proceso.

Cuando un proceso accede al estado *c*, el límite superior *u* del proceso toma como valor la suma del tiempo actual del sistema más el valor de la constante d_1. Con ello es establecido un límite superior para la transición *set*, transición que deberá ser completada por el proceso antes de que el tiempo global alcance dicho límite. Una vez realizada la transición *set*, el proceso llega al estado *d*. En el estado *d* se establece el valor del límite inferior *l* como la suma de el tiempo actual más el valor de la constante d_2. De esta forma se establece una restricción de comienzo para la transición *check* por la cual, una vez el tiempo global es mayor al límite inferior impuesto, el proceso en el estado *d* ya puede realizar la transición *check*.

Con estos límites impuestos y constantes definidas se consigue asegurar la propiedad de exclusión mutua cuando se cumple la condición $d_1 < d_2$. En otras palabras, se define para el conjunto de procesos que ningún proceso podrá realizar la transición *check* hasta que todos aquellos procesos que hayan entrado al estado *c* hayan realizado la transición *set*, llegando así al estado *d*.

4. ANÁLISIS ARTÍCULO BASE

En este capítulo estudiaremos como han modelado en [1] el protocolo de exclusión mutua en tiempo real Fischer utilizando el método *observable transition system* del lenguaje CafeOBJ. A continuación, veremos la especificación final del sistema con sus características. Después, descubriremos qué propiedades han escogido y cómo las han verificado. Finalmente comentaremos los resultados obtenidos.

4.1. Modelado del sistema

Para el modelado del protocolo Fischer decidieron usar un *observable transition system*. El concepto de sistema de transiciones observable es introducido en [4]. Básicamente podemos entenderlo como un sistema de transiciones que utiliza observaciones. Las observaciones son llevadas a cabo mediante observadores que se encargan de consultar aspectos del sistema que observan. Los autores declaran tres observadores *u*, *l* y *loc* para los procesos, y dos observadores *turn* y *now* para el sistema. Los observadores se definen con operadores, pudiendo estar parametrizados. El comportamiento de los observadores se especifica mediante ecuaciones.

Los observadores *u* y *l* son los encargados de observar el valor del límite superior e inferior de un proceso. El observador *loc* en cambio, se encarga de observar en que estado se encuentra un proceso en el sistema. El observador definido como *turn* es el encargado de monitorizar el estado de la variable *turn* del sistema. Finalmente, el observador *now*, es utilizado para consultar el valor del tiempo global actual del sistema como si de un reloj se tratara.

Además, en los *observable transition system* se pueden definir dos tipos de transiciones, *discrete* y *continuous*. Las transiciones *discrete* son aquellas que ocurren en un momento dado. Por ejemplo, cuando un proceso esta en un cierto estado puede tomar transiciones que solo ocurran en ese estado, por ende, cuando cambie de estado ya no podrá hacerla hasta que vuelva a él. Dichas transiciones ocurren en un momento dado y no de forma continuada. Las transiciones *try*, *test*, *set*, *check*, *exit* y *reset* son transiciones de tipo *discrete*. Las transiciones *continuous*, por el contrario, son aquellas que pueden ocurrir de forma consecutiva. La transición *tick*, encargada de avanzar el tiempo en una cierta cantidad indicada por el parámetro *D*, puede ocurrir varias veces seguidas.

La especificación del sistema como un *observable transition system* es llevado a cabo mediante la especificación de ecuaciones. Las ecuaciones son las encargadas de definir el comportamiento y estado de los procesos en el sistema.

4.1.1. *Definición del estado inicial del sistema*

Para definir el estado inicial donde empieza el sistema, los autores definen en CafeOBJ el siguiente fragmento de código que podemos ver en la Ilustración 14. Primero definen el constructor para el estado inicial del sistema como el operador *init*. Luego declaran el operador para el observador *loc*, el cual recibirá el estado del sistema y el identificador del proceso a observar como parámetros. Como paso final

definen las ecuaciones de los observadores *loc* y *now*. El observador *loc* es definido en la primera ecuación para saber en qué estado el proceso *I* se encuentra en el estado inicial del sistema *init*. De esta forma definen el estado *a*, siguiendo la Ilustración 13, como el estado donde se encuentran todos los procesos. Para el observador *now* se define el valor cero como el valor inicial en que se encuentra el reloj del sistema. Con estas definiciones ya tenemos el estado inicial donde el sistema empieza.

```
 op init : -> Sys
bop loc : Sys Pid -> Loc
var I : Pid
 eq loc(init, I) = a .
 eq now(init) = 0 .
```

Ilustración 14. Fragmento de código para especificar el estado inicial del sistema Fischer.

4.1.2. *Transiciones del sistema*

Veamos ahora como definen las transiciones entre los estados *a*, *b*, *c*, *d*, *cs* y *e*. Para definir las transiciones entre estados en un *observable transition system* hacen uso de los observadores. Los observadores son los encargados de indicar qué valores tiene un proceso antes y después de una transición. Veamos la especificación de la transición *try* en las siguientes ilustraciones 15 y 16.

En la Ilustración 15 son declaradas las variables *I* y *J* para albergar identificadores de procesos. Además, es declarado el operador *c-try* para especificar que parámetros recibirá la condición que debe cumplirse en un proceso para pasar del estado *a* al estado *b*.

```
vars I J : Pid .
op c-try : Sys Pid -> Bool
```

Ilustración 15. Declaración de dos variables para identificadores de procesos y declaración del operador que modela la condición de la transición try.

En la Ilustración 16 es donde se define la condición de la transición *try* con la primera ecuación. Esta ecuación nos indica que la localización del proceso *I* debe ser el estado *a*, siguiendo la Ilustración 13. En la siguiente línea se define el operador para la transición *try*, recibiendo un sistema y un proceso como parámetros. Las dos últimas ecuaciones condicionales sirven para especificar la transición try.

La primera de las ecuaciones condicionales especifica que, si un proceso falla la condición para hacer la transición *try*, se quedará en el mismo sistema, es decir, se queda igual. La segunda ecuación condicional nos especifica que la localización de un proceso *J* en un sistema donde se aplica la transición *try* sobre el proceso *I*, será el estado *b*, siguiendo la Ilustración 13, si se cumplen dos condiciones.

La primera condición que cumplir es la más externa, indicando que debe cumplirse la condición *c-try* para el proceso *I*. Esto quiere decir que el proceso *I* debe estar en el estado *a*, siguiendo la Ilustración 13. La segunda condición, interna a la ecuación condicional, especifica que *I* y *J* sean el mismo proceso.

El propósito de la segunda condición es no aplicar la transición a procesos distintos. Esto es debido a que es posible que para varios procesos *I y J se* cumpla la condición *c-try*, pero solo debe aplicarse *try* sobre el proceso indicado por la transición con su segundo parámetro.

```
 eq c-try(S,I) = (loc(S,I) = a) .
bop try : Sys Pid -> Sys
ceq try(S,I) = S if not c-try(S,I) .
ceq loc(try(S,I),J) =
    (if I = J then b else loc(S,J) fi)
    if c-try(S,I) .
```

Ilustración 16. Definición del comportamiento de la transición try.

4.1.3. *Modelado de las restricciones de tiempo*

Veamos ahora la especificación de las restricciones de tiempo en el sistema modelado. Como bien vimos antes existen dos límites, uno superior y uno inferior. El límite superior *u* es aplicado sobre la transición *set*, indicando que un proceso tiene d_1 unidades de tiempo una vez se encuentra en el estado *c* para pasar al estado *d*. El límite inferior *l* es usado para que un proceso no pueda realizar la transición *check* hasta que no hayan transcurrido d_2 unidades de tiempo desde que llegó al estado *d*.

El límite superior es definido en la especificación del reloj global, concretamente en la ecuación *tick* encargada de hacer avanzar el tiempo en una unidad. Con la condición *now(S) + D < min(u(S))* el tiempo global del sistema no avanza más allá del límite superior menor de los procesos. En otras palabras, dada la naturaleza de la lógica de ecuaciones es posible que un proceso esté en tiempo para poder realizar la transición *set*, pues el tiempo global es menor que el límite superior *u*, pero no decida hacerlo pues otro proceso decidió tomar otra acción. En consecuencia, la única ecuación que se podría aplicar entonces para el proceso en *c* es la transición *set*, obligando al proceso a realizar la transición. Además, es necesario restablecer el límite superior mediante *set* para que el tiempo pueda seguir avanzando una vez un proceso cambia al estado *d*.

El límite inferior, en cambio, es especificado como una condición de la transición *check*. La siguiente Ilustración 17 muestra la asignación del límite inferior cuando se realiza la transición *set* y la definición de las condiciones para la transición *check*. La ecuación condicional indica que el límite inferior de un proceso J, al hacer la transición *set* para un proceso *I* cuando se cumple la condición *c-set* para el proceso *I,* pasa a valer el tiempo actual del sistema más el valor de la constante d_2. La ecuación, por otra parte, indica que para que un proceso *I* pueda realizar la transición *check*, este debe encontrarse en el estado *d* y además cumplir que el tiempo actual del sistema es mayor o igual al límite inferior.

```
ceq l(set(S,I),J) =
  (if I = J then now(S) + d2 else l(S,J) fi)
  if c-set(S,I) .
 eq c-check(S,I) =
   (loc(S,I) = d and l(S,I) <= now(S)) .
```

Ilustración 17. Definición del límite inferior y definición de la transición check.

4.1.4. *Módulos de datos*

Pasemos a ver brevemente los tres módulos de datos que han creado para definir tipos de datos, aunque no los mostramos con una ilustración. Estos módulos son necesarios para especificar el límite superior y modelar correctamente la condición sobre la transición *tick*. La idea principal es que el tiempo global no pase el menor de los límites superiores del conjunto de procesos en el estado *c*.

El primer módulo es *TIMEVAL* y extiende el módulo *RAT*, módulo predefinido para los números racionales. En él es donde se define el operador *oo* para representar el infinito. Adicionalmente, son ampliadas las operaciones como la suma, comparaciones o la obtención del mínimo para poder trabajar el operador infinito con ellas.

El segundo módulo es *PAIR*. En el módulo es declarado el tipo *Pair* representado por el operador *(i | x)* donde *i* es un identificador de proceso y *x* su límite superior. Este segundo módulo es usado en el tercer módulo *PSET*.

El tercer módulo *PSET* define el conjunto de procesos, declarando el tipo *Pair* como un subtipo del tipo *Pset*. El conjunto de procesos es definido como la secuencia de otros conjuntos de procesos. Sobre el conjunto de procesos se aplica la propiedad asociativa para conjuntos. Finalmente, sobre el conjunto de procesos son declaradas y definidas las operaciones *get*, *remove* y *min*. La operación *get* tiene el propósito de retornar el límite superior *u* de un *Pair* con el identificador de proceso *i*. La operación *remove* sirve para eliminar un *Pair* del conjunto de *Pset*. Y la operación *min* devuelve el *Pair* con el límite superior de menor valor del conjunto *Pset*.

4.2. Características propias del sistema

Daremos un vistazo a las características del sistema que han modelado a partir de los módulos antes definidos. Veremos primero la definición de las constantes d_1 y d_2 y como se define la condición $d_1 < d_2$. Luego veremos la especificación final del estado inicial *init* en base a sus observaciones. Por último, comprenderemos las acciones llevadas a cabo para manejar el límite superior *u*.

Las constantes d_1 y d_2 son declaras y su comportamiento definido en la siguiente Ilustración 18. Tres ecuaciones son definidas para establecer la propiedad de que $d_1 < d_2$. Con ello se asegura que las constantes solo puedan tomar ciertos valores.

```
ops d1 d2 : -> Rat+
eq  0 < d1 = true .
eq d1 < d2 = true .
eq d2 < oo = true .
```

Ilustración 18. Declaración y definición de las constantes de tiempo d_1 y d_2.

Para la especificación final del estado inicial han definido los observadores *turn*, *loc*, *now*, *u* y *l* como se muestra en la Ilustración 19. El observador *turn* contiene el valor *nop* al inicio del sistema. El estado inicial indicado con *loc* es el estado *a*. El reloj global indicado con el observador *now* al inicio del sistema vale cero, al igual que el límite inferior de todos los procesos. Finalmente, para el observador del límite superior *u* hay dos posibles valores siendo *nop* o el operador infinito *oo*, asegurando de esta forma que el tiempo global siempre pueda avanzar.

```
eq turn(init) = nop .
eq loc(init,I) = a .
eq now(init) = 0 .
eq l(init,I) = 0 .
eq u(init) = (nop | oo) .
```

Ilustración 19. Definición de los valores iniciales para los observadores del sistema.

Cabe destacar que el observador del límite superior no esta definido para un proceso concreto, pero sí para el sistema. Esto se debe al comportamiento especial que le han dado al límite superior. Como sabemos de antemano, cuando un proceso llega al estado *c* mediante la transición *test*, su límite superior es asignado como el tiempo actual más la constante d_1, pero además es creado un *Pair* con el identificador del proceso y su valor *u*. Este par de valores son añadidos a un conjunto que denominan también *u*. De esta forma, cuando hay que consultar el valor del límite superior para la transición *tick*, son consultados todos los pares, escogiendo el mínimo de los límites superiores pues fue el primero de los procesos que entró al estado *c*. Una vez la transición *set* tiene lugar, el par con el identificador del proceso y su límite superior son eliminados del conjunto. En conclusión, como vemos en la siguiente Ilustración 20, en la transición *set* el límite inferior *l* es establecido mientras que el límite superior *u* es eliminado mediante la operación *remove*.

```
ceq l(set(S,I),J) =
  (if I = J then now(S) + d2 else l(S,J) fi)
    if c-set(S,I) .
ceq u(set(S,I)) = remove(I,u(S))
    if c-set(S,I) .
```

Ilustración 20. Comportamiento de los observadores l y u cuando ocurre la transición set en el sistema.

4.3. Verificación formal

La propiedad que verifican sobre el modelo construido es la exclusión mutua. Para conseguir dicho objetivo, los autores utilizan el comando *red* para llevar a cabo razonamiento con ecuaciones. El funcionamiento del comando *red* es aplicar ecuaciones sobre un término *t* hasta alcanzar, si es posible, otro término *t'*.

La verificación de propiedades en CafeOBJ es realizada mediante el método *proof score method*, introducido en [4]. El método consiste en, dado un término *t* y un módulo *M* con un conjunto de ecuaciones E_M, si *t* se reduce a *true* mediante las ecuaciones E_M entonces podemos decir que la proposición *t* se cumple en el modelo *M*. En igual forma, en un módulo *M* definimos un operador *a* y una ecuación *eq r = l* con la cual un término *t* se termina conteniendo en la constante *a*. A partir de esta especificación, si el término *t* resulta ser verdadero mediante reducción, entonces la implicación $l = r \Rightarrow t$ se cumple para todo el modelo *M*.

Para la verificación de la exclusión mutua se ha construido un módulo *INV* donde definir las invariantes. La primera invariante es *inv1*, visible en la Ilustración 21. En este módulo vemos cómo se declara primero el operador *inv1*, que recibe un sistema y dos identificadores de procesos. Luego se define su ecuación como que la invariante declarada implica que, si dos procesos tienen el observador *loc* con valor *cs*, la zona crítica, entonces esto implica que ambos son el mismo proceso.

```
mod INV{
  pr(FISCHER)
  op inv1 : Sys Pid Pid -> Bool
  var S : Sys . vars I J : Pid .
  eq inv1(S,I,J) =
  (loc(S,I) = cs and loc(S,J) = cs
   implies I = J) .
}
```

Ilustración 21. Módulo para la definición de la primera invariante sobre el sistema Fischer.

La idea es ahora demostrar la proposición $\forall s \in RS.inv1(s)$, es decir, para todo estado *s* que pertenezca al conjunto de estados alcanzables *RS* se cumple la invariante *inv1*. Para ello son definidas pruebas, denominadas *proof passages*, con las demostrar el cumplimiento de la invariante. La primera prueba se realiza sobre el estado *init* con el comando «*red inv1(init, i, j) .*», cuyo resultado es *true*.

Una vez verificado el estado inicial queda comprobar si se cumple la invariante en todos los demás estados que surjan de aplicar las transiciones *try*, *test*, *set*, *check*, *exit* y *reset*. Para este propósito, son definidos módulos siguiendo un módulo base *ISTEP*. Con el módulo se define la ecuación «*istep1 = inv1(s,I,J) implies inv1(s',I,J) .*». Esto quiere decir que la transición cumple la invariante *inv1* si al aplicar la transición sobre el estado *s*, que cumple *inv1*, se cumple la invariante sobre el estado resultante *s'*. Con ello consiguen *proof passages* para todas las transiciones.

Pero surge un problema, y es que para tres procesos las invariantes devuelven false. Una primera solución es dividir los casos de prueba en dos, es decir, si tenemos los procesos *i, j y k*, entonces definimos un caso donde *i = k* y otro donde *(i = k) = false*. Aun así, sigue surgiendo false en algunas ocasiones. Para solucionar el problema, es definida una nueva invariante *inv2* a la que llaman *lemma*.

```
eq inv2(S,I,J)
   = ((loc(S,I) = cs or loc(S,I) = e)
      and not(I = J)
      implies
      turn(S) = I and not(loc(S,J) = c)) .
```

Ilustración 22. Segunda invariante para la demostración de la propiedad de exclusión mutua sobre el sistema modelado.

Como vemos en la Ilustración 22, la invariante *inv2* especifica que, si el estado del proceso *I* es la sección crítica *cs* o el estado de salida *e,* y el proceso *J* es otro proceso distinto de *I*, entonces el turno es del proceso *I*, porque está o bien en sección crítica o saliendo, y el estado del proceso *J* no es el estado *c*. Recordemos que cuando un proceso llega al estado *d* ningún otro proceso puede empezar un nuevo intento de entrar a sección crítica con la transición *test*, hasta que el turno sea liberado con la transición *reset*, tomando así el valor *nop*.

Por tanto, para conseguir verificar todos los estados es necesario comprobar *inv1 /\ inv2* para todos los estados alcanzables desde el estado inicial *init*. Con ello crean 94 *proof passages* que devuelven *true*, verificando con este método la propiedad de exclusión mutua sobre el sistema modelado Fischer.

4.4. Conclusiones del artículo

Hemos visto como los autores del artículo modelan el protocolo Fischer mediante un *observable transition system,* descartando así la opción de usar la versión con tiempos. El concepto de sistema de transiciones observable con tiempos fue propuesto en [9]. En este artículo es donde a partir del concepto de *observable transition system* definen un nuevo tipo parametrizado con tiempo. Además, enseñan dos casos donde usan CafeOBJ para verificar propiedades temporales sobre los sistemas. Esta propuesta no fue llevada a cabo por los problemas que ocasiona el tratamiento del límite superior en sistemas con múltiples tareas.

También hemos podido entender las decisiones que hacen sobre los límites. Debemos quedarnos con las aplicaciones que realizan sobre las transiciones *set* y *check*, y del uso del límite superior para restringir el tiempo global. Estas decisiones tomadas nos serán de gran utilidad en el desarrollo de nuestro modelo.

En la transición *set*, a parte de hacer uso del mínimo de los límites superiores de todos los procesos, restablecen el límite al eliminarlo del conjunto. En la transición *check*, es el límite inferior quien es restablecido, tras ser comprobado, poniéndolo con valor cero al aplicar la ecuación.

La limitación del tiempo real global del sistema para que los procesos en el estado *c* sean obligados a aplicar la transición *set* también es interesante en dos sentidos. Por un lado, evita que los procesos pierdan la oportunidad de coger turno y se queden estancados. Pero, por el contrario, crea dependencias entre procesos, de tal manera que un proceso se ve limitado según el estado de otros procesos. Por ejemplo, si un proceso *A* está en el estado *c* y otro proceso *B* en el estado *a*, el proceso *B* se ve condicionado por el estado del proceso *A* si el tiempo global alcanza el límite establecido por el mínimo *u*, hacen que se bloquee *B* hasta que el proceso A realice la transición *set.*

Finalmente, nos quedamos también con el resultado positivo en la verificación de la propiedad de la exclusión mutua para el protocolo Fischer mediante *proof score method*.

5. ESPECIFICACIÓN CON MAUDE

En este capítulo vamos a modelar el protocolo de exclusión mutua en tiempo real Fischer, utilizando el lenguaje de especificación Maude. Luego, utilizaremos la herramienta Anima, disponible en [10], para ver el funcionamiento de nuestro modelo y analizar el comportamiento definido.

5.1. Modelado del sistema

Con el fin de modelar el sistema hemos definido un módulo con identificador *FISCHER*. Importamos el módulo NAT para poder hacer uso de los números naturales en los valores de los identificadores de los procesos y en la representación del valor del tiempo. El tiempo en nuestro sistema es un valor numérico natural con el propósito de representar el tiempo real de forma aproximada.

Los primeros conceptos que hemos definido son los tipos que vamos a tratar. Como se ve en la Ilustración 23, hemos definido los tipos *Process*, *PState*, *PSet*, *Turn* y *GlobalState*. El tipo *Process* lo definimos además como un subtipo de *PSet*, con lo cual un proceso es un conjunto de procesos de un solo elemento. Definimos también la relación de subtipo entre el tipo *Nat* y *Turn*, de tal forma que los números naturales también pueden ser turnos.

```
*** Declaración de los tipos Proceso y Estado del proceso
sorts Process PState .

*** Declaración del tipo set de procesos que es un supertipo de proceso
sort PSet .
subsort Process < PSet .

*** Declaración del tipo turno que es un supertipo de Nat
sort Turn .
subsort Nat < Turn .

*** Declaración del tipo estado global
sort GlobalState .
```

Ilustración 23. Declaración de los tipos del sistema Fischer.

Definimos a continuación, las mismas constantes utilizadas en la Ilustración 13 del capítulo 4 para los estados de cada proceso. Con los tipos definidos previamente, declaramos los operadores *empty*, los estados *a*, *b*, *c*, *d*, *cs* y *e* de un proceso y el operador para representar un proceso en el sistema. Estas declaraciones son visibles en la Ilustración 24.

El operador *empty* representa el conjunto vacío de procesos de tipo *PSet*. Los estados representan los valores de tipo *PState*. El operador para un proceso es la letra *p* seguida de un numero natural como identificador. Al igual que [1], nosotros vamos a definir el límite superior *u* y el límite inferior *l* como variables locales del proceso, como

especifica el protocolo, entre paréntesis y separadas por una coma. Por último, entre corchetes se encuentra el estado actual del proceso.

```
*** Definición del conjunto vacío de procesos
op empty : -> PSet .

*** Estados posibles
ops a b c d cs e : -> PState .

*** Definición de proceso con un id, límite superior, límite inferior y un
estado
op p_(_,_){_} : Nat Nat Nat PState -> Process .
```

Ilustración 24. Definición de los operadores para el conjunto vacío, los estados de un proceso y el propio proceso.

Luego, definimos el conjunto de procesos *PSet* como el operador secuencia de otros dos conjuntos de procesos *PSet*. Además, como atributos aplicamos las propiedades conmutativa y asociativa, y como elemento identidad el operador *empty* definido anteriormente. Esta definición se encuentra en la Ilustración 25, junto con la definición del operador *nop*. De igual forma que en [1] usando CafeOBJ, el operador *nop* es el valor especial de turno con el cual indicaremos que ningún proceso tiene asignado el turno.

```
*** Definición del conjunto de procesos
op __ : PSet PSet -> PSet [assoc comm id: empty] .
eq X:Process X:Process = X:Process .

*** Definición del operador nop como valor de turno
op nop : -> Turn .
```

Ilustración 25. Definición del conjunto de procesos y del operador nop.

El modelado del sistema global lo llevamos a cabo mediante la definición de un operador de tipo *GlobalState*. Este operador alberga la forma que podemos observar en la Ilustración 26, donde vemos cuatro zonas separadas por el símbolo |.

En el operador definimos en la primera zona las posiciones de las constantes *D1 y D2*. Estas constantes son declaras como variables de tipo *Nat*, y nos servirán para establecer los límites *u* y *l* de cada proceso en las transiciones oportunas. Luego, en la segunda zona, definimos la posición del reloj global del sistema entre los símbolos *[* y *]*. El reloj global es una variable, con identificador *GT* y de tipo *Nat,* con la que representar el tiempo de forma continua en el sistema. La tercera zona alberga el valor del turno, valor que será almacenado en la variable *Tn*. La cuarta zona es donde se encuentran todos los procesos del sistema como un conjunto de procesos *PSet*. Finalmente, declaramos las variables que vemos al final para usarlas en la definición de las reglas del sistema.

```
*** Definición del estado global del sistema
(D1,D2)|TiempoGlobal|Turno|Conjunto de procesos
op (_,_)|[_]|_|_ : Nat Nat Nat Turn PSet -> GlobalState .
var PS : PSet .
var Tn : Turn .
vars D1 D2 GT Id U L : Nat . --- GT es el tiempo global

vars u1 l1 u2 l2 : Nat .
```

Ilustración 26. Definición del estado global del sistema con un operador y declaración de las variables necesarias para las reglas.

Vamos ahora a modelar el comportamiento del sistema a través de la definición de reglas. Hemos definido una regla, identificada como *Tick*, para especificar la transición del tiempo global del sistema. Luego, hemos definido una regla por cada una de las transiciones entre los estados, excepto para la transición *Check* que tiene dos posibles caminos.

La regla condicional *Tick* hace avanzar el tiempo global del sistema en una unidad. Como observamos en la Ilustración 27, el tiempo global se identifica como la variable *GT* y se transforma en *(GT + 1)* cuando la regla es aplicada.

```
*** Transicion donde solo transcurre una unidad de tiempo
crl [Tick] : (D1,D2) | [GT] | Tn | PS => (D1,D2) | [(GT + 1)] | Tn | PS
     if GT < 100 .
```

Ilustración 27. Regla condicional para la especificación del transcurso del tiempo global del sistema.

Es importante destacar la condición de la regla *Tick* donde el tiempo *GT* no puede superar las 100 unidades de tiempo. Con la restricción *GT < 100* y la asunción de que el tiempo no puede ser negativo hacemos que el sistema exista entre los instantes de tiempo cero y 100. El propósito principal de la restricción es reducir el árbol de estados generado por el comando *search*, consiguiendo que no sea un número infinito de estados.

En una primera instancia pensamos en usar la operación modular sobre el tiempo, de forma que funcionase de forma parecida a un reloj. Esta opción acabó dando muchos problemas pues no representaba de forma fiel el tiempo sobre el modelo y, lo más importante, no recortaba el árbol de búsqueda. En definitiva, al hacer el tiempo modular, siempre surgían nuevos estados, muchos de ellos inconsistentes con los límites definidos. Por ejemplo, en algunos nodos del árbol el tiempo global era menor que el limite inferior de los procesos.

Como primera solución decidimos recortar el avance del tiempo global hasta las diez unidades, en lugar de hacer la operación módulo diez. Luego se aumentó a 20 unidades, para pasar finalmente a las 100 actuales. Asimismo, decidimos aplicar la restricción a las demás reglas también, pues las transiciones avanzan el tiempo global en una unidad.

El propósito de que todas las transiciones avancen el tiempo global en una unidad es evitar situaciones donde un proceso pueda aplicar transiciones importantes,

como *Test* o *Set*, y no transcurra el tiempo mediante la regla *Tick*. De esta forma el tiempo no ha avanzado, pero si el estado del proceso, situación que no es deseable ni real. Por tanto, definir el avance del tiempo en la definición de las demás reglas resulta en que evitemos el problema expuesto.

Las demás reglas condicionales son las transiciones entre los diferentes estados del sistema. La Ilustración 28 nos muestra las reglas para las transiciones *Try*, *Test* y *Set*. La regla *Try* implica a un proceso estar en el estado *a* para poder ejecutarse, además de que el tiempo global debe ser menor a 100 unidades. La condición para el estado *a* se consigue mediante *pattern matching* al escribir el operador *a* directamente en la posición del estado. De esta manera, si no se cumple la coincidencia del patrón de la regla y el patrón del estado global del sistema, no se procederá a comprobar la condición de la regla.

```
*** Transiciones
crl [Try] : (D1,D2) | [GT] | Tn | p Id (U,L) {a} PS => (D1,D2) | [(GT + 1)] |
Tn | p Id (U,L) {b} PS
    if GT < 100 .

crl [Test] : (D1,D2) | [GT] | Tn | p Id (U,L) {b} PS => (D1,D2) | [(GT + 1)] |
Tn | p Id (((GT + 1) + D1),L) {c} PS
    if Tn == nop /\ GT < 100 .

crl [Set] : (D1,D2) | [GT] | Tn | p Id (U,L) {c} PS => (D1,D2) | [(GT + 1)] |
Id | p Id (0,((GT + 1) + D2)) {d} PS
    if GT <= U /\ GT < 100 .
```

Ilustración 28. Reglas condicionales para las transiciones Try, Test y Set.

La regla condicional *Test* comprueba que el proceso se encuentre en el estado *b* y que el valor de la variable *Turn* sea igual al valor especial *nop*. Cuando procede a aplicarse la transformación, el nuevo estado será *c* donde se le habrá asignado al proceso un nuevo valor a su límite superior. El nuevo valor viene dado por la expresión *((GT + 1) + D1)*. Así pues, el límite superior es el tiempo global más uno más la constante *D1*.

La regla condicional *Set* tiene en cuenta este nuevo límite en su condición con *GT <= U /\ GT < 100* cuando el estado del proceso es el estado *c*. Asimismo, la regla asigna al límite inferior del proceso el valor dado por la expresión ((*GT + 1) + D2)*. Al mismo tiempo, tal y como realizan en el artículo base la transición *Set* elimina el límite superior *u* restableciéndolo con el valor cero.

La regla condicional *Check* tiene dos posibles salidas. Para su especificación hemos optado por definir dos reglas condicionales para diferenciar el comportamiento de la transición. Las dos reglas pueden verse en la Ilustración 29.

Ambas reglas, *CheckTrue* y *CheckFalse*, especifican que el estado del proceso debe ser el estado *d* y que debe cumplirse la condición *GT >= L /\ GT < 100*. La condición especifica que el tiempo global debe encontrarse por encima o igual al límite inferior y además ser menor a 100 unidades. La diferencia entre ambas reglas está en la primera condición que hemos omitido en la expresión anterior.

La primera condición para la regla *CheckTrue* es *Tn == Id*, mediante la cual se comprueba que el turno lo tiene el proceso actual con identificador *Id*. De esta forma entra a sección crítica.

La primera condición de la regla *CheckFalse* es la contraria, pues comprueba si el turno tiene otro valor distinto al identificador del proceso con *Tn =/= Id*. Si se cumple la condición, entonces el proceso vuelve al estado *b*, donde esperará para poder de nuevo intentar entrar a la zona crítica. Finalmente, la transición *Check* restablece el límite inferior *l* al asignarle valor cero.

```
crl [CheckTrue] : (D1,D2) | [GT] | Tn | p Id (U,L) {d} PS => (D1,D2) | [(GT +
1)] | Tn | p Id (U,0) {cs} PS
    if Tn == Id /\ GT >= L /\ GT < 100 .

crl [CheckFalse] : (D1,D2) | [GT] | Tn | p Id (U,L) {d} PS => (D1,D2) | [(GT +
1)] | Tn | p Id (U,0) {b} PS
    if Tn =/= Id /\ GT >= L /\ GT < 100 .
```

Ilustración 29. Reglas condicionales para representar la transición Check.

Las últimas dos reglas condicionales son *Exit* y *Reset, disponibles en la Ilustración 30*. La primera regla especifica que el estado del proceso debe ser *cs*. Con ello, y si el tiempo global es menor a 100 unidades de tiempo, el nuevo estado del proceso para a ser el estado *e,* saliendo de la zona crítica. La segunda regla sirve el propósito de modificar el sistema para que otro proceso pueda entrar a la zona crítica. La regla *Reset* hace que el nuevo estado del proceso pase de *e* al estado inicial *a*, igual que en la Ilustración 13. Además, restablece el valor de la variable compartida *Tn* con el operador *nop*. Con ello la regla *Test* puede volver a dejar pasar procesos a un nuevo intento de coger la sección crítica.

```
crl [Exit] : (D1,D2) | [GT] | Id | p Id (U,L) {cs} PS => (D1,D2) | [(GT + 1)]
| Id | p Id (U,L) {e} PS
    if GT < 100 .

crl [Reset] : (D1,D2) | [GT] | Id | p Id (U,L) {e} PS => (D1,D2) | [(GT + 1)]
| nop | p Id (U,L) {a} PS
    if GT < 100 .
```

Ilustración 30. Reglas condicionales para las transiciones Exit y Reset que especifican la salida de sección crítica y el restablecimiento del turno.

5.2. Funcionamiento del modelo usando Anima

La herramienta online Anima, disponible en [10], nos sirve de apoyo para mostrar el funcionamiento de Maude cuando usamos el comando *search*. El comando hace que Maude construya un árbol de búsqueda sobre el que comprobar si existe un camino de un termino a otro. Anima recibe un módulo Maude y un termino válido sobre el que empezar la construcción del árbol.

Nuestro término de inicio, correspondiente con la raíz del árbol, es el estado del sistema *(1,2) | [0] | nop | p 1 (0,0) {a} p 2 (0,0) {a}*. En este estado las constantes *D1* y *D2* valen uno y dos respectivamente, haciendo que se cumpla la condición *D1 < D2*. El tiempo global del sistema comienza en el instante cero, y el turno no está asignado a ningún proceso. En el sistema hay dos procesos, con identificadores uno y dos, límites con valor a cero y en el estado *a*.

El primer paso es cargar el fichero Maude que contiene el módulo de sistema *FISHCER*. Luego indicamos el término inicial antes descrito. Con ello Anima nos muestra el siguiente nodo amarillo, indicando que es la raíz del árbol, que podemos ver en la Ilustración 31.

Ilustración 31. Nodo raíz del árbol generado usando la herramienta Anima.

Si presionamos sobre la fecha roja inferior del nodo, hacemos que se expanda el árbol en un paso de ejecución al aplicar una regla. La Ilustración 32 muestra los tres nodos hijos surgidos de expandir el nodo raíz. Cuando un nodo es expandido, son mostrados los nuevos estados alcanzables al aplicar una regla. Para saber qué regla ha sido aplicada, miramos encima del nuevo nodo y vemos el identificador en rojo de la regla.

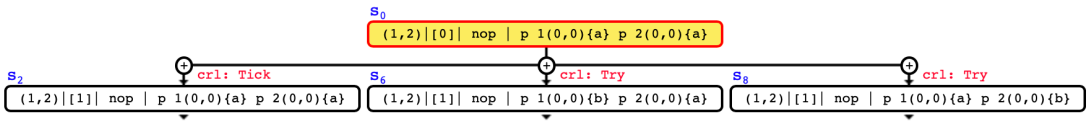

Ilustración 32. Nodo raíz y sus tres hijos en el árbol generado usando la herramienta Anima.

Cuando un nuevo nodo es creado es por la aplicación de una regla. La creación de nuevos nodos representa un nuevo estado en el sistema que es alcanzable desde uno de los nodos padres. Por ello, las reglas hacen el estado de un sistema avance, mientras que las ecuaciones representan equivalencias en un mismo estado. Si queremos ver qué ecuaciones y reglas han sido aplicadas sobre cada nodo, podemos pulsar sobre el símbolo más encima uno de los nodos. Los nodos verdes son los nodos intermedios como aparece en la siguiente Ilustración 33.

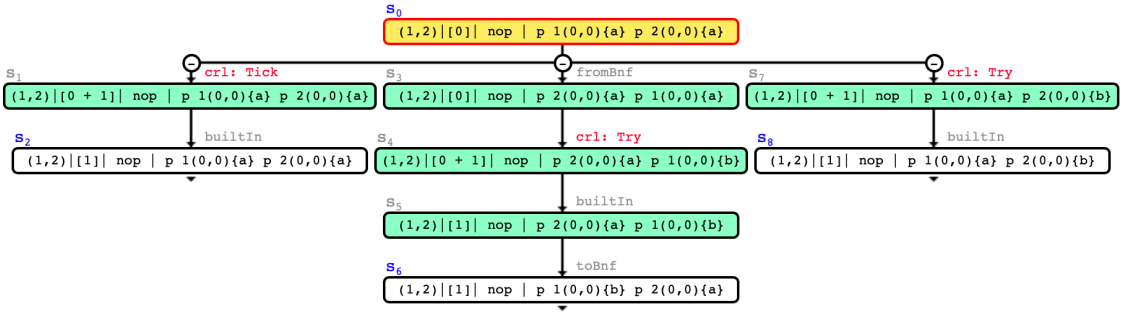

Ilustración 33. Nodos intermedios entre los nodos de estado.

Si observamos los nodos sobre los que pone *builtIn*, son los nodos a los que ha sido aplicada una ecuación del módulo *NAT* para simplificar la suma cero más uno en el valor uno. Por otra parte, vemos que el primer nodo central con la etiqueta *fromBnf* nos indica la aplicación de la propiedad conmutativa sobre el conjunto de procesos del estado padre.

Es interesante observar el patrón de expansión del árbol. Los nodos más a la izquierda son los nodos sobre los que siempre es aplicada la regla *Tick*. Los nodos más a la derecha aplican las reglas de transiciones entre estado, pero solo sobre el proceso dos. Mientras tanto, los nodos intermedios aplican de forma equitativa las transiciones entre los procesos uno y dos, de tal manera que avanzan a la vez, hasta que uno entre a zona crítica por supuesto. Esto es debido a cómo funciona Maude a la hora de aplicar reglas y ecuaciones, siendo comprobadas las que han sido definidas primero, es decir, las más arriba del módulo.

Con la herramienta Anima podemos hacer consultas de estados. Con el cuadro de búsqueda podemos hacer que Anima resalte en rojo el estado que hemos introducido. El problema es que el estado debe encontrarse en el árbol actualmente expandido, de lo contrario, no será resaltado. La siguiente Ilustración 34 muestra el resultado de consultar el término *(1,2) | [3] | nop | p 1 (3,0) {c} p 2 (0,0) {b}*.

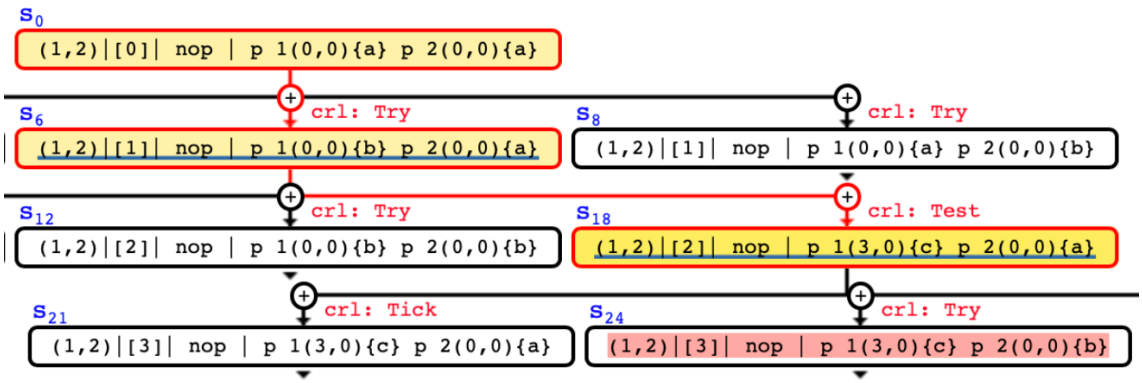

Ilustración 34. Resultado de consultar por un nodo existente en el árbol.

6. VERIFICACIÓN CON MAUDE

En este capitulo vamos a verificar las propiedades de exclusión mutua, *deadlock freedom* y vivacidad sobre el módulo de sistema construido. Para ello utilizaremos dos aproximaciones, una informal y otra formal.

En la aproximación informal hacemos uso del comando *search* para verificar la propiedad de exclusión mutua únicamente. Las propiedades de *deadlock freedom* y vivacidad no pueden ser verificadas de esta forma, pues requieren de mayor expresividad, la cual el comando *search* no nos la proporciona.

En la aproximación formal hacemos uso del *model checker* de Maude, definiendo dos módulos para poder definir propiedades sobre estados del árbol de estados alcanzables. Con estas propiedades construimos tres fórmulas en *linear temporal logic* para representar cada una de las propiedades.

6.1. Verificación informal

La propiedad que vamos a verificar sobre el sistema modelado en el módulo *FISCHER* es la exclusión mutua. Utilizamos el comando *search* para hacer una búsqueda de un camino entre dos términos en el árbol generado. Como bien explican en [5], cuando tratamos de verificar propiedades de seguridad mediante búsquedas, el objetivo es comprobar que un estado no deseado no es alcanzable desde un estado inicial. Al tratarse la exclusión mutua de una propiedad de seguridad queremos que la búsqueda no devuelva resultado, es decir, que no exista un camino entre el término inicial válido y el estado crítico final.

El término inicial que hemos usado es *(2,4) | [0] | nop | p 1 (0,0) {a} p 2 (0,0) {a}*. Este término representa un sistema con valores dos y cuatro para *D1* y *D2* respectivamente. El sistema empieza desde el instante de tiempo cero, y sin turno asignado. Existen en el sistema dos procesos en el estado *a*.

El término que representa el estado crítico del sistema es *(2,4) | [GT] | Tn | p 1 (u1,l1) {cs} p 2 (u2,l2) {cs}*. Este término es un sistema con un tiempo global dado por la variable definida en el módulo *FISCHER*, *GT* y el turno lo da la variable *Tn*. Por último, los dos procesos se encuentran ambos en la sección critica *cs*.

La regla search necesita de un símbolo intermedio entre los dos términos. Nosotros utilizamos =>* para que Maude, a la hora de construir el árbol, utilice en el proceso ninguno, uno o muchos pasos de ejecución. Utilizando los dos términos descritos y la flecha obtenemos el comando:

«*search (2,4) | [0] | nop | p 1 (0,0) {a} p 2 (0,0) {a} =>* (2,4) | [GT] | Tn | p 1 (u1,l1) {cs} p 2 (u2,l2) {cs} .*»

La ejecución del comando la mostramos en la Ilustración 35 en la cual Maude nos devuelve que no ha podido encontrar un camino desde el estado inicial hasta un estado crítico. En conclusión, la propiedad de exclusión mutua para dos procesos en el sistema *FISCHER* modelado se cumple.

```
[Maude> search (2,4) | [0] | nop | p 1 (0,0) {a} p 2 (0,0) {a} =>* (2,4) | [GT] |]
  Tn | p 1 (u1,l1) {cs} p 2 (u2,l2) {cs} .
search in FISCHER : (2,4)|[0]| nop | p 1(0,0){a} p 2(0,0){a} =>* (2,4)|[GT]| Tn
    | p 1(u1,l1){cs} p 2(u2,l2){cs} .

No solution.
states: 1184846  rewrites: 11245516 in 24513ms cpu (25367ms real) (458756
    rewrites/second)
Maude>
```

Ilustración 35. Ejecución del comando search para la comprobación de la propiedad de exclusión mutua sobre el módulo de sistema FISCHER.

6.2. Verificación formal

El *model checking* es un área en ciencias de la computación para la verificación de propiedades sobre modelos. El *model checker* de Maude utiliza *linear temporal logic*, una lógica para la especificación de propiedades como fórmulas temporales.

En esta sección vamos a ver los dos módulos definidos para el uso del *model checker* de Maude sobre el modelo *FISCHER*. Luego describimos las propiedades de exclusión mutua, *deadlock freedom* y vivacidad, esta última también conocida como *starvation freedom*. Para cada propiedad definimos una fórmula de la lógica de tiempo lineal y probamos su veracidad en la herramienta de *model checking* de Maude junto a los módulos definidos.

6.2.1. *Módulos para el model checking*

Gracias al módulo de sistema FISCHER obtenemos una teoría de reescritura para conseguir el nivel de especificación. Ahora nos queda alcanzar el nivel de verificación mediante la construcción de proposiciones atómicas y fórmulas de la lógica linear temporal. Las proposiciones atómicas son predicados o sentencias que se pueden cumplir sobre los diferentes estados de un sistema. Para poder definir predicados sobre nuestro módulo FISCHER necesitamos relacionar la teoría de reescritura del módulo de sistema con una estructura de Kripke.

Las estructuras de Kripke son grafos con estados etiquetados que sirven para representar sistemas reactivos y concurrentes. Una estructura de Kripke se puede definir como $M = (S, S_0, R, L)$ donde S es el conjunto de estados del grafo, S_0 los estados iniciales, R las relaciones entre estados que conocemos como transiciones y L es una función. La función L tiene la forma $S \rightarrow 2^{AP}$, y etiqueta cada estado en el conjunto S con las proposiciones, del conjunto de proposiciones atómicas AP, que se cumplen en él.

La relación entre teoría de reescritura y estructura de Kripke se consigue cuando son definidos dos conceptos. Primero definimos el *kind*, es decir, el tipo de los estados que se encuentran en el conjunto de estados Σ de la teoría de reescritura. Luego, definimos los predicados de estados relevantes, es decir, el conjunto de proposiciones atómicas AP.

Para conseguir definir los dos conceptos definimos un módulo identificado como *FISCHER-PREDS*. Este módulo importa el módulo de sistema *FISCHER* y el módulo *SATISFACTION* definido por el *model checker* de Maude.

Dentro del módulo, visible en la Ilustración 36, declaramos una relación de subtipo entre el tipo *GlobalState* del módulo *FISCHER* y el tipo *State* del módulo *SATISFACTION*. El tipo *GlobalState* es el *kind* que representará los estados de nuestro sistema como tipo *State* en la estructura de Kripke.

Siguiendo con la Ilustración 36, declaramos los operadores *criticalSection* y *wantsToEnter* de tipo *Prop* para modelar los predicados del conjunto de proposiciones atómicas *AP*. Los predicados en la lógica *Linear Temporal Logic* son constantes que no reciben parámetros, en cambio Maude sí permite definir predicados con parámetros. Por tanto, ambos predicados reciben como parámetro prefijo un número natural como identificador del proceso sobre el que es comprobado. Por ejemplo, la notación *criticalSection(3)* indica que el predicado *criticalSection* se aplica sobre el proceso del sistema con identificador tres.

```
*** Definición de GlobalState como subtipo de estado
subsort GlobalState < State .

*** Declaración del predicado para un proceso p (identificado por su id de
tipo Nat) en sección crítica
op criticalSection : Nat -> Prop .

*** Declaración del predicado para un proceso p (identificado por su id de
tipo Nat) en el estado b
op wantsToEnter : Nat -> Prop .
```

Ilustración 36. Definición del kind y declaraciones de los predicados del conjunto AP.

Luego es necesario definir variables para poder manejar los parámetros del sistema global. Estas variables las usaremos en la definición de los predicados que acabamos de declarar. Hemos definido un total de diez variables.

Las variables *N, uN, lN, D1, D2,* y *GT* son de tipo *Nat*. La primera sirve el propósito de contener el número identificador de un proceso. Las variables *uN* y *lN* representan los límites superior e inferior del proceso *N*. Las variables *D1* y *D2* son las constantes del sistema global, y la variable *GT* es el tiempo global del sistema. La variable *PSet* representa el conjunto de procesos que hay en el sistema global. El turno es contenido mediante la variable *Tn*. Por último, definimos las variables *GS* de tipo *GlobalState* y la variable *P* de tipo *Prop*.

Los predicados son definidos mediante ecuaciones con las que decidir cuándo son cumplidas. En la Ilustración 37 mostramos las dos ecuaciones para definir la veracidad de los predicados sobre un estado. La estructura de una proposición es <*Estado*> |= <*Predicado*>, indicando que la proposición dada por <*Predicado*> se cumple en el estado del sistema indicado con <*Estado*>.

La primera ecuación establece la veracidad del predicado *criticalSection(N)* si se cumple que en el estado global del sistema existe un proceso con identificador *N* cuyo

estado del proceso es *cs*. Con la ecuación estamos indicando que un nodo del árbol de estados posibles tiene un proceso en la sección crítica.

La segunda ecuación define la veracidad del predicado *wantsToEnter(N)* si en el estado global del sistema se encuentra un proceso *N* y el estado del proceso es *b*. Con esta ecuación indicamos que en un nodo del árbol de expansión un proceso quiere entrar a zona crítica.

Podemos ver una tercera ecuación cuyo propósito es definir la falsedad de los dos predicados. La definición estipula que para cualquier otro estado global *GS* la proposición *P* es falsa. Debido al atributo *owise* la ecuación será solo aplicada cuando no se cumplan las dos ecuaciones anteriores. De esta forma definimos para ambas proposiciones cuando no son cumplidas en un nodo del árbol.

```
*** Ecuaciones para los predicados de estado

*** Ecuación para decir que se cumple el predicado criticalSection(N), pues el
proceso N está en sección crítica
eq (D1,D2) | [GT] | N | p N (uN,lN) {cs} Pset |= criticalSection(N) = true .

*** Ecuación para decir que se cumple el predicado wantsToEnter(N), pues el
proceso N está en el estado b
eq (D1,D2) | [GT] | Tn | p N (uN,lN) {b} Pset |= wantsToEnter(N) = true .

*** Ecuación para cualquier otro caso
eq GS |= P = false [owise] .
```

Ilustración 37. Ecuaciones para definir los predicados del conjunto AP sobre los estados de tipo GlobalState del módulo FISCHER.

Ahora que tenemos la relación entre la teoría de reescritura y la estructura de Kripke, nos queda definir un estado inicial sobre el que probar fórmulas de la *linear temporal logic*. Para poder definir estados iniciales, estos deben cumplir dos condiciones.

La primera condición es que el conjunto de estados alcanzables desde el estado inicial a definir debe ser finito. Como segunda condición, la teoría de reescritura *R = (∑, E, φ, R)* más las ecuaciones *D,* que definen los predicados en el módulo *FISCHER-PREDS,* son tales que *E* y *E* $\cup$ *D* son deterministas y terminantes, y además *R* es *ground*. En otras palabras, la teoría de ecuaciones y su unión con las ecuaciones de los predicados cumplen la propiedad *Church-Rosser*, la cual especifica que el orden de aplicación de las ecuaciones no afecta al resultado obtenido. Que *R* sea *ground* quiere decir que el conjunto de reglas del sistema no contiene variables durante su funcionamiento, es decir, que los términos de entrada y salida tienen valores establecidos durante la ejecución del modelo.

Asumimos el cumplimiento de ambas condiciones y creamos el módulo con identificados *FISCHER-CHECK*. Este módulo importa el módulo anterior *FISCHER-PREDS,* el módulo *MODEL-CHECKER* y un módulo opcional que hemos decidido incluir denominado *LTL-SIMPLIFIER*.

La definición y declaración de dos estados iniciales la mostramos en la Ilustración 38. En las dos primeras líneas definimos dos operadores, *initial1* y initial2, de tipo *GlobalState*. Estos operadores son convertidos después mediante sus respectivas ecuaciones. La primera ecuación convierte *initial1* en un estado global donde las constantes *D1* y *D2* valen uno y dos respectivamente, el tiempo global comienza en cero, no está el turno asignado y hay dos procesos en el estado *a*. La segunda ecuación convierte el operador *initial2* en el estado global del sistema en el que las constantes siguen valiendo uno y dos, y el tiempo comienza en cero, pero uno de los dos procesos se encuentra en el estado *cs* y por tanto tiene el turno asignado, y el otro proceso se encuentra en el estado *b*.

```
*** Declaración de los estados iniciales
op initial1 : -> GlobalState .
op initial2 : -> GlobalState .


*** Definición de un estado inicial con dos procesos, 1 y 2, ambos en el
estado inicial a
eq initial1 = (1,2) | [0] | nop | p 1 (0,0) {a} p 2 (0,0) {a} .

*** Definición de un estado inicial con dos procesos,1 y 2, uno en cs y el
otro en b
eq initial2 = (1,2) | [0] | 1 | p 1 (0,0) {cs} p 2 (0,0) {b} .
```

Ilustración 38. Declaración y definición de dos estados iniciales para hacer model checking sobre FISHCER.

Con los módulos FISCHER-PREDS y FISCHER-CHECK estamos listos para definir fórmulas en lógica linear temporal y verificar propiedades sobre el módulo de sistema *FISCHER* con la técnica *model checking*.

6.2.2. *Propiedades comprobadas*

Las tres propiedades que hemos probado a verificar sobre el modelo construido *FISCHER*, que representa el protocolo Fischer, son la exclusión mutua, la ausencia de *deadlock freedom* y el cumplimiento de *vivacidad* para un proceso.

Vamos a ver una pequeña descripción de cada una, junto a su expresión en lenguaje natural. A partir de la expresión escribimos la fórmula correspondiente y la usamos en el *model checker* de Maude mediante el comando *reduce* y la función *modelCheck*. Esta función recibe un estado inicial sobre el cual construir el árbol de estados y una fórmula en *linear temporal logic* a comprobar.

6.2.2.1. *Exclusión mutua*

La propiedad de exclusión mutua establece que un recurso compartido que sea crítico no puede ser accedido al mismo tiempo por dos o más procesos. En el protocolo Fischer, esto se traduce a que dos procesos distintos no pueden acceder de

forma simultanea a la zona crítica. Ahora vamos a definir una fórmula en *linear temporal logic* con la descripción en lenguaje natural:

Para todos los posibles estados futuros siempre se cumple que ni el proceso uno ni el proceso dos están en la sección crítica.

A partir de esta descripción podemos asignar los operadores lógicos generales y específicos de la *linear temporal logic*. Recordemos que en esta lógica no existe el operador eventualidad *E*, y que el operador *A* viene implícito. Además, podemos agrupar las dos negaciones en una sola y juntarlas.

Con estas asunciones obtenemos la fórmula $G \neg (cs(1) \wedge cs(2))$. En ella *cs(1)* y *cs(2)* son los predicados correspondientes para nuestro preposición *criticalSection(N)*. La fórmula final para usar en la función *modelCheck* es *[ ] ~ (criticalSection(1) /\ criticalSection(2))*.

Cargamos en Maude los módulos definidos de *FISCHER, FISCHER-PREDS* y *FISCHER*-CHECK, en ese orden. Aplicamos la fórmula sobre los dos estados iniciales *init1* e *init2* haciendo uso de la función *modelCheck* en el comando *reduce* para obtener la siguiente Ilustración 39.

```
Maude> red modelCheck(initial1, [] ~(criticalSection(1) /\ criticalSection(2)))
.
reduce in FISCHER-CHECK : modelCheck(initial1, []~ (criticalSection(1) /\
    criticalSection(2))) .
rewrites: 12766588 in 23752ms cpu (24023ms real) (537473 rewrites/second)
result Bool: true
Maude>
>     red modelCheck(initial2, [] ~(criticalSection(1) /\ criticalSection(2))) .

reduce in FISCHER-CHECK : modelCheck(initial2, []~ (criticalSection(1) /\
    criticalSection(2))) .
rewrites: 12279978 in 21897ms cpu (22025ms real) (560791 rewrites/second)
result Bool: true
Maude>
```

Ilustración 39. Resultados obtenidos tras aplicar la fórmula de exclusión mutua sobre dos estados iniciales.

Como podemos ver en la Ilustración 39, la fórmula que especifica la propiedad de exclusión mutua se cumple para todos los estados alcanzables desde los dos estados iniciales *init1* e *init2*. Por tanto, podemos concluir que el modelo definido cumple la propiedad de exclusión mutua.

6.2.2.2. *Deadlock freedom*

La propiedad *deadlock freedom* especifica que, si un proceso quiere entrar en la zona crítica, entonces un proceso, no necesariamente el mismo que lo solicita, entra a la zona crítica. En el protocolo Fischer esto lo podemos comparar a que existe un proceso en el estado *b*, entonces eventualmente en el futuro un proceso, que puede ser el mismo o cualquier otro, consigue entrar a la sección crítica. La expresión en lenguaje natural es:

En todos los posibles futuros, siempre se cumple que, eventualmente en un instante dado un proceso desea entrar a la zona crítica, entonces existe un instante de tiempo posterior en el cual un proceso consigue entrar a la zona crítica.

A partir de esta descripción conseguimos la fórmula *AF (b(1) ∨ b(2)) → F(cs(1) ∨ cs(2)))* donde *b(1)* y *b(2)* son los predicados que indican que los procesos uno y dos están en el estado *b* y por tanto han empezado un intento de entrar en la zona crítica. A partir de esta fórmula, traducimos usando los operadores del *model checker* de Maude y obtenemos <> *(wantsToEnter(1) \/ wantsToEnter(2))* -> <> *(criticalSection(1) \/ criticalSection(2))*. Aplicamos la fórmula sobre los dos estados iniciales *init1* e *init2* y usamos la función *modelCheck* en el comando *reduce*.

Ocurre que para el estado *init1* Maude nos devuelve un contraejemplo conteniendo un camino donde al final ocurre la transición *try* para el proceso 1 y justo después el sistema no avanza más pues ha llegado al límite de 100 unidades de tiempo global. En cambio, para el estado inicial *init2,* donde ya hay un proceso en *b* y otro proceso en cs, si se cumple la propiedad. El segundo estado inicial debemos descartarlo pues no representa bien un estado inicial para la propiedad de *deadlock freedom*. La causa es que el estado inicial ya cumple la fórmula ya que hay un proceso en el estado *b y* el otro proceso se encuentra en el estado *cs*.

Por tanto, dado el modelo actual no podemos verificar el cumplimiento de la propiedad de *deadlock freedom*. Viendo el contraejemplo devuelto para *init1*, una posible solución podría ser modelar el tiempo real global de tal forma que no avance más allá del mínimo de los límites superiores de los procesos. Esta aproximación obliga a un proceso que está en c a realizar en algún momento la transición *set* acercando el proceso a la zona crítica y evitando que se quede bloqueado.

El problema es que aun modificando el modelo el *model checker* devolverá un contraejemplo. La causa principal reside en la construcción el árbol de estados alcanzables, donde la regla de paso del tiempo *Tick* al no tener más condición que ser menor que 100 generará una rama del árbol donde solo pase el tiempo.

Esta rama es un inconveniente para verificar propiedades en *linear temporal logic*. La lógica usada implica el uso del operador general lógico *A,* con el cual se especifica que la fórmula debe cumplirse en todos los caminos posibles, lo cual no es posible por la regla *Tick*.

Por tanto, concluimos que la propiedad de *deadlock freedom* no es verificable utilizando la fórmula construida sobre el modelo *FISCHER* que hemos definido. Sería pues necesario que definiéramos un nuevo modelo con características de tiempo extendidas.

6.2.2.3. *Vivacidad*

La propiedad de vivacidad, también conocida como *starvation freedom*, es parecida al *deadlock freedom*. La propiedad especifica que, si un proceso desea entrar a la zona crítica en un instante de tiempo *T*, existe un instante *T' > T* donde el proceso consigue entrar. Como vemos, la principal diferencia con la propiedad anterior es que ahora es el propio proceso quien debe entrar a la sección crítica y no puede ser cualquier otro, de esta forma se evita que se quede desatendido por el sistema. La correspondiente expresión en lenguaje natural es:

En todos los posibles futuros, siempre se cumple que, eventualmente en un instante dado, un proceso desea entrar a la zona crítica, entonces existe un instante de tiempo posterior en el cual ese proceso consigue entrar a la zona crítica.

La fórmula que deducimos a partir de la expresión es *AF b(1)* $\rightarrow$ *F cs(1)*. A partir de esta fórmula obtenemos la expresión <> *wantsToEnter(1)* -> <> *criticalSection(1)* con los operadores del *model checker*. Aplicamos la fórmula sobre solo el estado inicial *init1,* pues aplicarlo sobre el estado *init2* no tiene sentido al satisfacer ya la fórmula. Esta decisión la hemos aprendido tras analizar los resultados obtenidos de la verificación de la propiedad *deadlock freedom*.

Usamos la función *modelCheck* en el comando *reduce* con la fórmula y el estado *init1* y obtenemos la Ilustración 40. Al igual que para la propiedad anterior, podemos ver que ocurre exactamente lo mismo. Observamos como las dos últimas transiciones son *Tick* y *Try* de tal forma que, el tiempo llega a valer 99, luego un proceso entra al estado *b* haciendo que se cumpla la primera parte de la fórmula, y por último la ejecución termina en *deadlock* porque se ha llegado al límite del tiempo global.

```
p 1(0,0){a} p 2(0,0){a},'Tick} {(1,2)|[93]| nop | p 1(0,0){a} p 2(0,0){a},
'Tick} {(1,2)|[94]| nop | p 1(0,0){a} p 2(0,0){a},'Tick} {(1,2)|[95]| nop |
p 1(0,0){a} p 2(0,0){a},'Tick} {(1,2)|[96]| nop | p 1(0,0){a} p 2(0,0){a},
'Tick} {(1,2)|[97]| nop | p 1(0,0){a} p 2(0,0){a},'Tick} {(1,2)|[98]| nop |
p 1(0,0){a} p 2(0,0){a},'Tick} {(1,2)|[99]| nop | p 1(0,0){a} p 2(0,0){a},
'Try}, {(1,2)|[100]| nop | p 1(0,0){b} p 2(0,0){a},deadlock})
```

Ilustración 40. Final del contraejemplo proporcionado por el model checker para el estado init1 sobre la fórmula de vivacidad.

Podemos concluir entonces que la propiedad de vivacidad no se cumple para el modelo definido. La causa de que no sea conseguida la veracidad de la propiedad es debida a la especificación del protocolo Fischer.

Como bien comenta [5] en la sección dos, el protocolo cumple las propiedades de exclusión mutua y *deadlock freedom*, pero no puede asegurar que un proceso sea desatendido. Por tanto, confirmamos que el modelo definido no cumple la propiedad de vivacidad.

7. CONCLUSIONES

Como consecuencia del trabajo aquí expuesto, podemos concluir que el objetivo general se ha cubierto totalmente al conseguir construir un modelo que define aproximadamente al protocolo Fischer y verificar mediante la propiedad de alcance y la técnica de *model checking* la propiedad de exclusión mutua.

En concreto, gracias a que hemos aprendido la sintaxis de CafeOBJ hemos podido comprender el modelo propuesto, y las asunciones sobre el tiempo que realizan en [1]. Sobre el modelo que proponen hemos conseguido comprender la verificación que realizan para la propiedad de exclusión mutua.

También hemos entendido las diferentes versiones que hay del protocolo, así como sus características temporales, consiguiendo comprender su funcionamiento. A partir de estas características temporales, hemos definido nuestro modelo del sistema en Maude y hemos declarado una serie de módulos para el uso del modelo en el *model checker* de Maude junto a tres fórmulas.

En un futuro convendría definir un nuevo modelo o fórmulas con las cuales poder verificar la propiedad de *deadlock freedom* que no hemos conseguido verificar mediante el *model checker* de Maude.

8. TRABAJOS FUTUROS

Existen diferentes caminos para poder continuar o mejorar el trabajo aquí presente. También existen alternativas donde replicar los pasos seguidos en el trabajo para replicarlos sobre protocolos y sistemas existentes y nuevos.

Respecto a continuar o mejorar el trabajo, es posible ampliar el modelo propuesto para conseguir modelar un límite superior compartido entre los procesos. También pueden ser consideradas nuevas fórmulas para verificar, sobre el modelo existente, u otras propiedades que vayan surgiendo. En igual forma, se puede ampliar el trabajo con el uso de otras herramientas o métodos de verificación que ofrezcan nuevos puntos de vista.

Otra forma de ampliar el trabajo es hacer uso de objetos de Maude. La ventaja de realizar una aproximación con objetos es poder hacer uso de un objeto como reloj global para modelar el tiempo entre los procesos. Esta característica del reloj global puede aprovechar la inclusión de un objeto *Reloj* que los desarrolladores incluyeron a finales de 2020. De esta forma se pone a prueba la utilidad y funcionalidad de esta nueva característica.

Respecto al uso del modelo existente, es posible utilizar el modelo desarrollado como base para modelar el protocolo Fischer en otros lenguajes de modelado y especificación, como Spin o Promela. También es posible considerar los pasos tomados como una guía para el análisis y verificación de propiedades de protocolos que pueden ser modelados en Maude.

9. BIBLIOGRAFÍA

[1] M. Nakamura, S. Higashi, K. Sakakibara and K. Ogata, "Formal verification of Fischer's real-time mutual exclusion protocol by the OTS/CafeOBJ method," in *2020 59th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, Chiang Mai, Thailand, 2020.

[2] M. Clavel, F. Duran, S. Eker, S. Escobar, P. Lincoln, N. Marti-Oliet, J. Meseguer y C. Talcott, «Maude Manual (Version 3.1),» Octubre 2020. [En línea]. Available: http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html. [Último acceso: 15 Mayo 2021].

[3] T. Seino y M. Nakamura, «Introduction to CafeOBJ,» 1 Diciembre 2009. [En línea]. Available: https://cafeobj.org/intro/en/. [Último acceso: 3 Junio 2021].

[4] K. Ogata y K. Futatsugi, «Proof scores in the OTS/CafeOBJ method,» de *International Conference on Formal Methods for Open Object-Based Distributed Systems*, 2003.

[5] L. Lamport, «A fast mutual excusion algorithm,» *ACM Transactions on Computer Systems (TOCS),* vol. 5, nº 1, pp. 1-11, 1987.

[6] J. J. Vereijken, Fischer's protocol in timed process algebra, Eindhoven: Citeseer, 1994.

[7] V. Luchangco, *Using simulation techniques to prove timing properties,* Cambridge, 1995.

[8] M. J. Fischer, *Re: where are you? Electronic mail message from Michael Fischer to Leslie Lamport, Arpanet message sent on June 25, 1985 18:56:29 EDT, number 8506252257.AA07636@yale-bulldog.yale.arpa (47 lines),* 1985.

[9] K. Ogata y K. Futatsugi, «Modeling and verification of real-time systems based on equations,» *Science of computer programming,* vol. 66, nº 2, pp. 168-188, 2007.

[10] M. Alpuente, D. Ballis, F. Frechina y J. Sapiña, «Anima Online Stepper,» 2014. [En línea]. Available: http://safe-tools.dsic.upv.es/anima/. [Último acceso: 20 Junio 2021].