UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Trabajo final de máster

2015/2016

MODELADO Y VERIFICACIÓN DE PROGRAMAS *EVENT-B* USANDO *MAUDE*

Autora: Francisca Ruiz Pérez

Director: Santiago Escobar Román

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ÍNDICE

1) Introducción

La mayoría de herramientas de modelado en Ingeniería del Software carecen de mecanismos de verificación de propiedades basados en métodos formales, aunque existen muchas alternativas que incluyen métodos formales ligeros (que no requieren aprender formalismos matemáticos complejos para el modelado).

Event-B es un método formal de modelado y análisis de sistemas líder en Europa en el ámbito de las herramientas de modelaje en Ingeniería del Software pues utiliza teoría de conjuntos como notación, refinamiento como representación de los diferentes niveles de abstracción de los sistemas y demostraciones matemáticas para comprobar la consistencia entre dichos niveles. Es una extensión del lenguaje *B*, desarrollado en la década de los 90 del siglo pasado, y lo emplean diversas compañías como *SAP*, *Bosch*, *Systérel*, *Alstom*, *Thales* o *InnoQ* para diseñar sus sistemas.

Aunque este lenguaje emplea internamente métodos formales, queda mucho trabajo por realizar para integrarlos de una forma simple que potencie su eficacia. Hoy en día existen herramientas que traducen programas *Event-B* a *Java* o *Prolog*; no obstante, nadie ha definido su semántica de forma rigurosa. Con esta tesis de máster se ha buscado estudiar y comprender la semántica de los modelos descritos en *Event-B* y vislumbrar un mecanismo de traducción de programas de *Event-B* a *Maude*.

Maude es un lenguaje basado en la lógica de reescritura que, dada su potencia, es posible utilizar no sólo como lenguaje de programación declarativo sino también como lenguaje de especificación formal ejecutable y como un sistema de verificación formal. Además, la simplicidad de su lógica y el empleo de técnicas de semicompilación permiten que sus implementaciones sean eficientes.

Como se verá, durante el trayecto hemos encontrado que sí es posible trasladar programas *Event-B* a *Maude*, aunque de momento ha sido un trabajo artesano, sobre todo en lo que respecta al armazón matemático del primero. Es un primer paso (no se ha tratado el refinamiento y se ha simulado de manera primitiva el paso de parámetros a un evento) que podría ser seguido por la exploración de *Maude* como metalenguaje para

este caso.

Por capítulos, en el segundo se explica el lenguaje *Maude*: unas pinceladas de reescritura seguidas por aquellas características relevantes para el desarrollo de este trabajo final de máster. En el tercero pasa a describirse *Event-B* y los dos elementos básicos de sus modelos (contextos y máquinas). En el cuarto se explica el proceso seguido en las traducciones que se detallan en el capítulo quinto, para concluir con algunos apuntes de hacia dónde se podría continuar con el camino emprendido aquí.

2) El lenguaje *Maude*

2.1. Apuntes sobre reescritura

Se asume una signatura de tipos ordenados $\Sigma = (S, \leq, \Sigma)$, con un conjunto de tipos parcialmente ordenado $(S, \leq)$ y una familia de variables ordenadas por los tipos S definida de la siguiente forma: $\chi = \{\chi_s\}_{s \in S}$, y basada en conjuntos disjuntos de variables, con cada conjunto χ_s siendo infinitamente contable. El conjunto $T_\Sigma(\chi)_s$ denota los términos de tipo s, y $T_{\Sigma,s}$ los términos sin variables del tipo s. Escribiremos $T_\Sigma(\chi)$ y T_Σ para las correspondientes álgebras de términos de tipos ordenados. Dado un término t, *Var (t)* representa el conjunto de variables de t.

Las posiciones de un término se representan con secuencias de números naturales. Muestran un camino de acceso en el término cuando éste se ve como un árbol. El tope o posición raíz se denota con la secuencia vacía Λ. La relación $p \leq q$ entre posiciones se define como:

- $p \leq p$, para toda posición p.
- $p \leq p.q$ para todas las posiciones p y q.

Dado un conjunto $U \subseteq \Sigma \cup \chi$, $Pos_U(t)$ indica el conjunto de posiciones del término t encabezadas por símbolos o variables de U. El conjunto de posiciones de un término t se escribe $Pos(t)$ y el de posiciones no variables, $Pos_\Sigma(t)$. El subtérmino de t en la posición p se escribe $t|_p$ y $t[u]_p$ representa el término t donde el subtérmino $t|_p$ ha sido reemplazado por u.

Una sustitución $\sigma \in Subst(\Sigma, \chi)$ es un mapeo ordenado de variables en χ hacia términos en $T_\Sigma(\chi)$ que es casi siempre la identidad salvo un conjunto finito $X_1,..., X_n$ de variables de χ, denominado el dominio de σ. Las sustituciones se escriben $\sigma = \{X_1 \rightarrow t_1,..., X_n \rightarrow t_n\}$, donde el dominio de σ es $Dom(\sigma) = \{X_1,..., X_n\}$ y el conjunto de variables introducidas por $t_1,..., t_n$ se indica con $Ran(\sigma)$. La sustitución identidad es id. Las sustituciones se extienden homomórficamente al conjunto de términos $T_\Sigma(\chi)$. La aplicación de una sustitución σ a un término t se representa como $t\sigma$ o $\sigma(t)$. Por simplicidad, se asume que todas las sustituciones son idempotentes, es decir, cada

sustitución σ satisface $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. La idempotencia de las sustituciones asegura la siguiente propiedad: $t\sigma = (t\sigma)\sigma$. La restricción de σ a un conjunto de variables V es $\sigma|_V$. La composición de dos sustituciones σ y σ' se denota como $\sigma\sigma'$. La combinación de dos sustituciones σ y σ' tal que $Dom(\sigma) \cap Dom(\sigma') = \emptyset$ se escribe $\sigma \cup \sigma'$. Una sustitución idempotente σ es un renombramiento si existe otra sustitución idempotente σ^{-1} tal que $(\sigma\sigma^{-1})|_{Dom(\sigma)} = id$.

Una Σ-ecuación es un par no ordenado $t = t'$, donde $t, t' \in T_\Sigma(\chi)_s$ para un tipo $s \in S$. Dada la signatura Σ y un conjunto E de Σ-ecuaciones, la lógica ecuacional de tipos ordenados induce una relación de congruencia $=_E$ sobre los términos $t, t' \in T_\Sigma(\chi)$. Una teoría ecuacional (Σ, E) es un par consistente en la signatura de tipos ordenados Σ y un conjunto E de Σ-ecuaciones.

El preorden de E-subsunción $\sqsupseteq_E$ (o simplemente $\sqsupseteq$ si E se sobreentiende) se satisface entre dos términos $t, t' \in T_\Sigma(\chi)$, denotado $t \sqsupseteq_E t'$ (entendiéndose que t es más general que t módulo E), si existe una sustitución σ tal que $t\sigma =_E t'$; dicta sustitución σ se denomina un E-emparejamiento entre t y t'. La relación de E-renombrado $t \approx_E t'$ se satisface si existe una renombramiento de variables θ tal que $t\theta =_E t'$. Dadas dos sustituciones σ, ρ y un conjunto de variables V, se satisface $\sigma|_V =_E \rho|_V$ si $x\sigma =_E x\rho$ para todas las variables $x \in V$; $\sigma|_V \sqsupseteq_E \rho|_V$ si existe una sustitución η tal que $(\sigma\eta)|_V =_E \rho|_V$; y $\sigma|_V \approx_E \rho|_V$ si existe un renombramiento η tal que $(\sigma\eta)|_V =_E \rho|_V$.

Un E-unificador para una Σ-ecuación $t = t'$ es una sustitución σ tal que $t\sigma =_E t'\sigma$. Dado el conjunto de variables W tal que $Var\ (t) \cup Var\ (t') \subseteq W$, un conjunto de sustituciones $CSU^W_E\ (t = t')$ se dice que es un conjunto completo de unificadores para la igualdad $t = t'$ *módulo E* fuera del conjunto W de variables si y sólo si:

i. cada $\sigma \in CSU^W_E\ (t = t')$ es un E-unificador de $t = t'$;
ii. para cada E-unificador ρ de $t = t'$ existe un $\sigma \in CSU^W_E\ (t = t')$ tal que $E\ \sigma|_W \sqsupseteq_E \rho|_W$;
iii. para cada $\sigma \in CSU^W_E\ (t = t')$, $Dom(\sigma) \subseteq (Var\ (t) \cup Var\ (t'))$ y $Ran(\sigma) \cap W = \emptyset$.

Si el conjunto de variables W es irrelevante o se sobreentiende del contexto, escribiremos $CSU_E(t = t')$ en vez de $CSU^W_E\ (t = t')$. Un algoritmo de E-unificación es completo si para cada ecuación $t = t'$ genera un conjunto completo de E-unificadores. Un algoritmo de

unificación se dice que es finito y completo si siempre termina generando un conjunto finito y completo de soluciones.

Una regla de reescritura es un par orientado $l \rightarrow r$, donde $l \notin \chi$, $Var\ (r) \subseteq Var\ (l)$, y $l, r \in T_\Sigma(\chi)_s$ para un tipo $s \in S$. Una teoría incondicional de reescritura para tipos ordenados es una tripleta (Σ, E, R) con Σ una signatura de tipos ordenados, E un conjunto de Σ-ecuaciones y R un conjunto de reglas de reescritura.

La relación de reescritura entre términos $T_\Sigma(\chi)$, escrita $t \rightarrow_R t'$ o $t \rightarrow_{p,R} t'$ se satisface entre dos términos t y t' si y sólo si existe una posición $p \in Pos_\Sigma(t)$, una regla l → r ∈ R y una sustitución σ, tal que $t|_p = l\sigma$ y $t' = t[r\sigma]_p$. El subtérmino $t|_p$ se denomina redex. La relación $\rightarrow_{R/E}$ sobre $T_\Sigma(\chi)$ es $=_E ; \rightarrow_R ; =_E$. Nótese que la relación $\rightarrow_{R/E}$ sobre $T_\Sigma(\chi)$ induce una relación $\rightarrow_{R/E}$ sobre el (Σ, E)-álgebra libre $T_{\Sigma/E}(\chi)$ de la forma $[t]_E \rightarrow_{R/E} [t']_E$ si y sólo si $t \rightarrow_{R/E} t'$. El cierre transitivo (resp. transitivo y reflexivo) de $\rightarrow_{R/E}$ se denota $\rightarrow^+_{R/E}$ (resp. $\rightarrow^*_{R/E}$). Un término t se denomina $\rightarrow_{R/E}$ -irreducible (o simplemente R/E-irreducible) si no existe ningún término t' tal que $t \rightarrow_{R/E} t'$.

Dada una regla de reescritura $l \rightarrow r$, se dice que es decreciente en tipo si para cada sustitución σ, se tiene que $r\sigma \in T_\Sigma(\chi)_s$ implica $l\sigma \in T_\Sigma(\chi)_s$. Una teoría de reescritura (Σ, E, R) es decreciente en tipo si todas las reglas R lo son. Dada una Σ-ecuación $t = t'$, se dice que es regular si $Var\ (t) = Var\ (t')$, y se dice que es decreciente en tipo si para cada sustitución σ, se tiene que $t\sigma \in T_\Sigma(\chi)$ implica $t'\sigma \in T_\Sigma(\chi)_s$ y viceversa.

Dadas dos sustituciones σ, ρ y un conjunto de variables V, la siguiente relación de reescritura para sustituciones $\sigma|_V \rightarrow_{R/E} \rho|_V$ se satisface si existe $x \in V$ tal que $x\sigma \rightarrow_{R/E} x\rho$ y para todas las demás variables $y \in V$ se tiene que $y\sigma =_E y\rho$. Una sustitución σ se denomina R/E-irreducible (o normalizada) si $x\sigma$ es R/E-irreducible para toda variable $x \in V$.

La relación de reescritura $\rightarrow_{R/E}$ se denomina terminante si no existe una secuencia infinita $t_1 \rightarrow_{R/E} t_2 \rightarrow_{R/E} \ldots t_n \rightarrow_{R/E} t_{n+1} \ldots$ Por otra parte, la relación $\rightarrow_{R/E}$ es confluente si cuando $t \rightarrow^*_{R/E} t'$ y $t \rightarrow^*_{R/E} t''$, existe un término t''' tal que $t' \rightarrow^*_{R/E} t'''$ y $t'' \rightarrow^*_{R/E} t'''$. Una teoría de reescritura de tipos ordenados (Σ, E, R) es confluente (resp. terminante) si la relación $\rightarrow_{R/E}$ es confluente (resp. terminante). En una teoría de reescritura de tipos ordenados que sea confluente, terminante y decreciente en tipo, para cada término $t \in$

$T_\Sigma(X)$, existe una única forma R/E-irreducible t' (módulo E-equivalencia) obtenida de t por reescritura hasta la forma canónica, la cual se denota como $t \to^{!}_{R/E} t'$, o $t\downarrow_{R/E}$ cuando t' es irrelevante.

La relación $\to_{R/E}$ es indecidible en general ya que las clases de E-congruencia pueden ser arbitrariamente extensas. Por lo tanto, la relación de reescritura $\to_{R/E}$ se implementa normalmente a través de la relación $\to_{R,E}$. La relación $\to_{R,E}$ sobre $T_\Sigma(\chi)$ se define de la siguiente forma: $t \to_{p,R,E} t'$ (o simplemente $t \to_{R,E} t'$) si y sólo si existe una posición $p \in Pos_\Sigma(t)$, una regla $l \to r$ en R y una sustitución σ tal que $t|_p =_E l\sigma$ y $t' = t[r\sigma]_p$. Nótese que si la relación de E-emparejamiento es decidible, la relación $\to_{R,E}$ también. Las nociones de confluencia, terminación y términos y sustituciones irreducibles se adaptan trivialmente para la relación $\to_{R,E}$. Si el conjunto de reglas R es confluente, terminante y decreciente en tipo, la relación $\to^{!}_{R,E}$ es decidible, ya que $\to_{R,E} \subseteq \to_{R/E}$. Se asumen las siguientes propiedades sobre R y E:

1. E es regular y decreciente en tipo; además, para cada ecuación $t = t'$ en E, todas las variables de *Var (t)* tienen un tipo máximo.
2. E tiene un algoritmo finito y completo de unificación.
3. Las reglas R son confluentes, terminantes y decrecientes en tipo módulo E.
4. $\to_{R,E}$ es localmente E-coherente, es decir, para todos los términos t_1, t_2, t_3 tenemos que $t_1 \to_{R,E} t_2$ y $t_1 =_E t_3$ implica que existe t_4, t_5 tal que $t_2 \to^*_{R,E} t_4$, $t_3 \to^+_{R,E} t_5$, y $t_4 =_E t_5$.

2.2. El lenguaje

Maude es un lenguaje de programación declarativo que permite modelar sistemas tanto desde el punto de vista estático (sus propiedades) como desde el punto de vista dinámico (las acciones que se pueden llevar a cabo en él y que son susceptibles de causar cambios en su estado). A esto se añade, por un lado, que la descripción de un sistema usando *Maude* es directamente ejecutable y, por el otro, que permite "razonar" sobre su comportamiento gracias a la fuerte base matemática que lo sustenta.

Utiliza reglas de reescritura como los lenguajes denominados *funcionales* (*Haskell*, *ML*, *Scheme* o *Lisp*). En concreto, está basado en la lógica de reescritura, lo que permite

definir multitud de modelos computacionales complejos tales como programación concurrente o la orientada a objetos (en adelante, POO). Por ejemplo, admite especificar objetos directamente en el lenguaje siguiendo una aproximación declarativa a la POO y que no está disponible ni en lenguajes imperativos como *C++* o *Java* ni en declarativos como *Haskell*.

El desarrollo de *Maude* nace de una iniciativa internacional cuyo objetivo es diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos. Se puede encontrar más información en:

http://maude.cs.uiuc.edu

La lógica de reescritura en la que se apoya no sólo permite representar un abanico amplio de sistemas, incluyendo concurrencia, algoritmos distribuidos, protocolos de redes, semánticas de lenguajes de programación o biología celular, sino que abre la puerta a aplicarla sobre el propio lenguaje, lo que proporciona a *Maude* la categoría de metalenguaje.

2.3. Un programa en Maude

El **módulo** es el concepto clave de *Maude*. Es, en esencia, un conjunto de definiciones de operaciones y la manera como interactúan, lo que matemáticamente se conoce como **álgebra** (un álgebra está formada por una serie de conjuntos de elementos y las operaciones permitidas sobre ellos). Incluye, además, la información necesaria para reducir y reescribir las expresiones que se introduzcan mediante el entorno del lenguaje.

Un programa en *Maude* está compuesto por diferentes módulos. Cada uno se define entre las palabras reservadas *mod* y *endm* –si es de sistema–, o entre *fmod* y *endfm* –si es funcional–. Incluye declaraciones de tipos y símbolos, que, junto con las reglas (encabezadas por *rl*), describen la lógica de algunos de los símbolos, las denominadas *funciones*.

Los símbolos y reglas definidos en un **módulo *de sistema*** tienen un comportamiento indeterminista y ejecuciones posiblemente infinitas en el tiempo (es decir, que tal vez no terminen nunca), mientras que los símbolos y reglas de un **módulo *funcional***, definidas éstas por ecuaciones (introducidas por *eq*), deben poseer un comportamiento determinista y acabar siempre su ejecución. La razón es que un módulo

de sistema permite reglas indeterministas y no terminantes porque modela un sistema de estados (o autómata) donde es posible que existan ciclos y varias posibles acciones a tomar para cada uno de sus estados. En cambio, un módulo funcional sólo admite ecuaciones (reglas deterministas y terminantes) pues representa un programa funcional y todo programa termina y debe devolver siempre el mismo valor. Por ejemplo, el siguiente módulo de sistema simula una máquina de café y galletas (todo un clásico):

```
mod VENDING-MACHINE is
   sorts Coin Coffee Cookie Item State .
   subsorts Coffee Cookie < Item .
   subsorts Coin Item < State .
   op null : -> State .
   op __ : State State -> State [assoc comm id: null] .
   op $ : -> Coin .
   op q : -> Coin .
   op a : -> Cookie .
   op c : -> Coffee .
   var St : State .

   rl St => St q .  --- Modela que se ha añadido un cuarto de dólar
   rl St => St $ .  --- Modela que se ha añadido un dólar
   rl $ => c .      --- Modela la obtención de un café por un dólar
   rl $ => a q .    --- Devuelve una galleta y un cuarto de dólar
   eq q q q q = $ . --- Cambia cuatro cuartos de dólar por un dólar
endm
```

Este sistema es indeterminista (obsérvese que para un dólar “$” hay dos posibles acciones) y no terminante (siempre se puede añadir más dinero a la máquina). Además, el módulo incluye una ecuación que hace que el cambio de cuatro cuartos de dólar por un dólar sea transparente, es decir, que no haya una transición entre dos estados.

Por otro lado, podemos especificar el siguiente módulo que simula la función factorial:

```
fmod FACT is
   protecting INT .
   op _! : Int -> Int .
```

```
   var N : Int .
   eq 0! = 1 . --- factorial de N=0 es 1
   eq N! = (N - 1)! * N [owise] . --- factorial de N>0
endfm
```

Este sistema es determinista y termina para cada posible ejecución. En resumen, un módulo de sistema admite reglas y ecuaciones mientras que en uno funcional sólo pueden aparecer ecuaciones.

2.4. Tipos de datos predefinidos

Maude dispone de varios tipos de datos predefinidos incluidos en el fichero *prelude.maude* de la instalación. En concreto:

- el tipo *Bool*, definido en el módulo *BOOL*.
- el tipo *Nat*, en *NAT*.
- el tipo *Int*, en *INT*.
- el tipo *Float*, en *FLOAT*.
- los tipos *Char* y *String*, en *STRING*.

Para usarlos hay que importar el módulo donde se encuentran con alguna de estas palabras reservadas: *including*, *protecting* o *extending*. Por ejemplo, el módulo *FACT* mostrado anteriormente importa el módulo *INT* de los números enteros.

2.5. Declaración obligatoria de variables

Se debe declarar el tipo de las variables antes de ser usadas en el programa, ya sea con el formato *var nombre_var : tipo* . (para una), con el de *var**s** nombre_var*$_1$ … *nombre_var*$_n$ *: tipo* . (para varias del mismo tipo) o añadiéndolo en el mismo momento de emplearlas, como en *X:Nat + Y:Nat*.

2.6. Declaraciones de tipos (sorts), símbolos constructores y variables

Una declaración de tipo presenta la forma *sort T* . –para un solo nuevo tipo de nombre *T*– o *sort**s*** T_1 … T_n . –para varios–. A continuación, se han de definir los constructores que formarán los datos asociados a ese tipo a través de la estructura *op C :* T_1 T_2 … $T_n \rightarrow T$. , donde T_1, T_2, …, T_n son los tipos de los parámetros de ese símbolo. De manera análoga, si se escribe *op**s*** C_1 … C_n *:* T_1 T_2 … $T_n \rightarrow T$. significa que los símbolos

C_1, ..., C_n poseen idéntica cantidad, tipo y posición de sus parámetros. Por ejemplo, las declaraciones de tipo:

```
sort Bool .
ops true false : -> Bool .
sort NatList .
op nil : -> NatList .
op _:_ : Nat NatList -> NatList .
```

introducen el tipo *Bool* con dos constantes (pues no presentan parámetros), *true* y *false*, y el tipo *Natlist* (listas cuyos elementos son números naturales, es decir, de tipo *Nat*).

Hay que tener en cuenta que *Maude* no admite tipos de datos paramétricos como *Haskell*; por lo tanto, no se pueden definir listas con parámetros sino que han de ser específicas para cada tipo, como en el caso de *NatList*. Sin embargo, es interesante fijarse en la forma de definir el operador binario infijo de construcción de una lista, ":", donde se indica que el primer argumento debe aparecer antes de los dos puntos y el segundo, detrás. En consecuencia, una lista de enteros se podrá definir con notación infija como *0 : (1 : (2 : nil))* en vez de la notación prefija :(*0,:(1,:(2,nil))*) simplemente indicando que el símbolo a utilizar es ":". Esto es muy práctico y versátil ya que simplemente se debe indicar con un "_" dónde va a aparecer el argumento:

op if_then_else_fi : Bool Exp Exp -> Exp .

op for (_ ; _ ; _) { _ } : Nat Bool Bool Exp -> Exp .

En concreto, en el ejemplo *VENDING MACHINE* se define `op _ _ : State State -> State .` que indica que el carácter " " (vacío) es un símbolo válido para concatenar estados. Y en el ejemplo *FACT* tenemos `op _! : Int ->Int .` , que declara el símbolo factorial en notación sufija.

2.7. Tipos de datos ordenados

En *Maude* se pueden crear tipos de datos ordenados o divididos en jerarquías. A continuación, se muestra cómo indicar que los números naturales se dividen en números naturales positivos y el cero mediante la palabra reservada *subsort*:

```
sorts Nat Zero NzNat .
subsort Zero < Nat .
```

```
subsort NzNat < Nat .
op 0 : -> Zero .
op s : Nat -> NzNat .
```

De esta forma, la expresión *s(0)* es de tipo *NzNat* y a la vez de tipo *Nat*, mientras que no es de tipo *Zero*. E, igualmente, la expresión *0* es de tipo *Zero* y *Nat* pero no *NzNat*.

2.8. Propiedades avanzadas de los símbolos

El lenguaje *Maude* incorpora la posibilidad de especificar símbolos con propiedades algebraicas como la asociatividad, la conmutatividad, el elemento neutro, etc. que facilitan la creación de programas. Sea la siguiente redefinición del tipo de datos *lista de números naturales*:

```
sorts NatList ENatList NeNatList .
subsort ENatList < NatList .
op nil : -> ENatList .
subsort Nat < NeNatList < NatList .
op _:_ : NatList NatList -> NeNatList [assoc] .
```

El símbolo ":" es ahora asociativo, es decir, no le son necesarios los paréntesis para separar sus términos (nótese que sus dos argumentos han de ser del mismo tipo para poder indicar que el símbolo cumple esta propiedad). Ahora, el lenguaje entiende que las siguiente expresiones significan lo mismo:

```
s (0) : s ( s (0) ) : nil
s (0) : ( s ( s (0) ) : nil )
( s (0) : s ( s (0) ) ) : nil
```

Otra posibilidad es añadir un elemento neutro al operador asociativo:

```
sorts NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

En este momento, ":" es un símbolo asociativo y el término *nil* es el elemento neutro del tipo de datos; por tanto, se puede eliminar salvo cuando aparezca solo. Con este cambio, estas líneas son equivalentes :

```
s (0) : s ( s (0) ) : nil
```

```
s (0) : s ( s (0) )
nil : s (0) : nil : s ( s (0) ) : nil
```

También se puede agregar la propiedad conmutativa a la lista, creando el tipo de datos multiconjunto (o conjunto formado por conjuntos):

```
sorts NatMultiSet .
subsort Nat < NatMultiSet .
op nil : -> NatMultiSet .
op _:_ : NatMultiSet NatMultiSet -> NatMultiSet [assoc comm id: nil] .
```

La palabra *comm* indica que se puede intercambiar el orden de los elementos. Así, estas expresiones son semánticamente iguales:

```
0 : s (0) : s ( s (0) ) : s (0)
0 : s (0) : s (0) : s ( s (0) ) : nil
s (0) : 0 : s ( s (0) ) : s (0) : nil
nil : s (0) : nil : s ( s (0) ) : nil : s (0) : nil : 0 : nil
```

Para acabar, se puede incorporar la propiedad de que no pueden haber elementos repetidos, convirtiendo el multiconjunto en un conjunto:

```
sorts NatSet .
subsort Nat < NatSet .
op nil : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .
eq X:Nat : X:Nat = X:Nat .
```

La ecuación de la última línea elimina aquellas ocurrencias repetidas de un término, con lo que todo esto quiere decir lo mismo:

```
0 : s (0) : s ( s (0) )
0 : s (0) : s (0) : s (0) : s ( s (0) ) : nil
s (0) : 0 : s ( s (0) ) : s (0) : nil
nil : s (0) : nil : s ( s (0) ) : nil : s (0) : nil : 0 : nil
```

2.9. Declaración de funciones

Aquellos operadores o símbolos que dispongan de reglas o ecuaciones que los definan son denominados *funciones*, mientras que los que no las poseen son

constructores.

Las reglas de una función se definen con "*rl =>* ." y las ecuaciones con "*eq =* .", ambos operadores reservados. Aquí es obligatorio declarar el tipo de todo (funciones, variables, etc); en otros lenguajes funcionales, como *Haskell*, no es necesario aunque se recomienda. En particular, esto puede ayudar a detectar fácilmente errores en el programa, cuando se definen funciones que no se ajustan al tipo declarado.

Respecto a las reglas/ecuaciones que definen las funciones, éstas pueden ser de la forma *rl* $f(t_1, ..., t_n)$ *=> e* . o *eq* $f(t_1, ..., t_n)$ *= e* ., donde t_1, ..., t_n y *e* son términos. Las ecuaciones pueden etiquetarse con la palabra reservada *owise* (*otherwise*); en ese caso, se indica que sólo se utilizará si ninguna otra ecuación es aplicable para ese símbolo.

```
fmod FACT is
   protecting INT .
   op _! : Int -> Int .
   var N : Int .
   eq 0! = 1 .
   eq N! = (N - 1)! * N [owise] .
endfm
```

Únicamente se puede emplear en ecuaciones porque una regla tiene un significado indeterminista.

Las funciones se pueden definir también mediante reglas/ecuaciones condicionales: *crl* $f(t_1, ..., t_n)$ *=> e if c* . o bien *ceq* $f(t_1, ..., t_n)$ *= e if c* ., donde la condición *c* es un conjunto de emparejamientos de la forma *t := t'* separados por el operador `/\`.

Un emparejamiento *t := t'* indica que el término *t'* debe tener la forma del término *t*, instanciando las variables de *t* si es necesario ya que éstas pueden ser usadas en la expresión *e* de la regla para extraer información de *t'*.

Las ecuaciones condicionales sólo pueden aplicarse si la condición tiene éxito. También es posible definir una ecuación condicional en la que las guardas sean expresiones de tipo *Bool* en vez de *t := t'*; en ese caso, se interpretan como *true := t*. Seguidamente se muestra la anterior función *factorial* escrita de otra manera:

```
ceq N! = 1 if N == 0 .
ceq N! = (N - 1)! * N if N =/= 0 .
```

"==" se evalúa a *true* si ambas expresiones son iguales; de forma complementaria, "=/=" dará *true* si son distintas. En este caso, sería posible emplear el operador condicional *if then else fi*:

```
eq N! = if N == 0 then 1 else (N - 1)! * N fi .
```

Asimismo, debido a los tipos de datos ordenados, se permiten expresiones lógicas de la forma *t :: T*, que se evalúan a *true* si la expresión *t* es del tipo *T*. Esto puede ser útil en ecuaciones condicionales como:

```
op emptyList : NatList -> Bool .
eq emptyList (NL) = NL :: ENatList .
```

donde se dice que una lista *NL* está vacía, es decir que *emptyList* retorna *true*, si *NL* es del tipo *ENatlist*.

2.10. Búsqueda eficiente de elementos en listas y conjuntos

Una ventaja de disponer de listas, multiconjuntos y conjuntos, es que determinadas operaciones de búsqueda y emparejamiento de patrones resultan mucho más rápidas; de hecho, más que en otros lenguajes declarativos como *Prolog* o *Haskell*. Por ejemplo, la pertenencia de un elemento a una lista se realiza de forma secuencial en muchos lenguajes declarativos:

```
sort NatList .
op nil : -> NatList .
op _:_ : Nat NatList -> NatList .
op _in_ : Nat NatList -> Bool .
eq N:Nat in nil = false .
eq N:Nat in (X:Nat : XS:NatList)
 = (N:Nat == X:Nat) or-else (N:Nat in XS:NatList) .
```

Sin embargo, cuando disponemos de un operador asociativo con un elemento neutro, esta operación se hace de forma más elegante y eficiente:

```
sort NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

```
op _in_ : Nat NatList -> Bool .
eq N:Nat in (L1:NatList : N:Nat : L2:NatList) = true .
eq N:Nat in L:NatList = false [owise] .
```

La expresión '*1 in 1 : 2 : 3*' se puede ver como '*1 in nil : 1 : 2 : 3*' gracias a la propiedad del elemento neutro, donde '*L1:NatList*' se emparejará con '*nil*', '*N:Nat*' con '*1*' y '*L2:NatList*' con '*2 : 3*'. Ocurre algo parecido con '*2 in 1 : 2 : 3*' y '*3 in 1 : 2 : 3*'. La ventaja es que es el propio sistema el que decide la mejor técnica de búsqueda y no el/la programador/a.

En el caso de un conjunto (o multiconjunto) es aún más simple gracias a la conmutatividad:

```
sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .
op _in_ : Nat NatSet -> Bool .
eq N:Nat in (N:Nat : L:NatSet) = true .
eq N:Nat in L:NatSet = false [owise] .
```

2.11. La orden search

La orden *search* permite explorar, de diferentes maneras y siguiendo una estrategia de primero en amplitud, el espacio de estados accesibles.

Su sintaxis es:

```
search [ n, m ] in <Id_mód > : <Térm 1> <Flecha_búsq> <Térm 2>
such that <Condición> .
```

donde:

- *n* es un argumento opcional para proporcionar el límite de soluciones deseadas.
- *m* es otro argumento no obligatorio para establecer la profundidad máxima de búsqueda.
- *Id_mód* es el nombre del módulo donde se lleva a cabo la orden y que se puede omitir.
- *Térm 1* es el término de partida.

- *Térm 2* es el patrón a alcanzar.
- *Flecha_búsq* es una flecha que indica cómo se llevará a cabo el intento de demostración por reescritura de *Térm 1* a *Térm 2*:
  - =>1: exactamente un paso.
  - =>+: uno o más pasos.
  - =>*: 0, uno o más pasos.
  - =>!: sólo se admiten estados finales canónicos (estados que no pueden reescribirse más).
- La *condición* establece una propiedad opcional que debe ser satisfecha por el estado alcanzado.

3) El lenguaje *Event-B*

B es una aproximación formal basada en estados que promueve el paradigma de desarrollo de corrección por construcción y la verificación formal mediante demostración de teoremas. *Event-B* es una evolución de *B* que permite modelar sistemas reactivos. Aquí un modelo (o especificación del sistema) se define utilizando la noción de máquina abstracta de estados.

Una **máquina abstracta de estados** engloba el estado del modelo -representado como una colección de variables- y las operaciones definidas sobre él. Así pues, describe la parte dinámica, esto es el comportamiento, del sistema en cuestión. Normalmente una máquina viene con otro componente, el **contexto**, que incluye la parte estática del modelo, como puedan ser los conjuntos definidos por el usuario (*carrier sets*), las constantes y sus propiedades, expresadas como una lista de axiomas. *Event-B* emplea en el desarrollo del sistema una aproximación basada en el refinamiento de arriba a abajo. Dicho desarrollo empieza con una especificación abstracta del sistema que modela los requisitos funcionales más esenciales y que se va ajustando y ampliando, con nuevos eventos y variables, a medida que se capturan requerimientos más detallados.

Cada paso de refinamiento introduce nuevos eventos y variables en la especificación abstracta. Dichos eventos corresponden a pasos no visibles desde el nivel abstracto. Dentro de una cadena de refinamientos, las variables de un modelo más abstracto reciben el nombre de *variables abstractas*, mientras que aquéllas del siguiente modelo refinado son denominadas *variables concretas*.

El desarrollo formal de *Event-B* admite el refinamiento de datos y permite reemplazar algunas variables abstractas por sus correspondientes concretas. En ese caso, la invariante de la máquina refinada define la relación entre ambas.

Para verificar la corrección de un paso de refinamiento, se debe probar una serie de obligaciones de demostración (*proof obligations*) en el modelo subsiguiente que permiten demostrar que la máquina resultante (el modelo refinado) no introduce comportamientos observables nuevos y que todas las propiedades del modelo abstracto son automáticamente heredadas por el refinado. Además, los estados *concretos* están

unidos a los *abstractos* mediante la ***invariante cohesionadora*** (*gluing invariant*) del modelo refinado.

Recapitulando, los modelos en *Event-B* están fundamentados en dos pilares: los contextos, para su parte estática, y las máquinas, para la dinámica. Los primeros aíslan los parámetros y las propiedades que se cumplen en cualquier caso; los segundos encapsulan un sistema basado en transiciones donde el estado viene dado por un conjunto de variables y de transiciones modeladas mediante una serie de eventos con guarda.

3.1. Notación para el modelado en Event-B

Existen dos tipos de componentes en *Event-B*: los contextos y las máquinas.

3.1.1. Contextos

Un contexto describe la parte estática de un módulo. Su estructura completa es:

context
  identificador del contexto
extends
  identificador del contexto del que hereda
sets
  identificadores de sus conjuntos
constants
  identificadores de sus constantes
axioms / theorems
  con el formato *etiqueta : predicado*
end

Se compone de:

- la extensión o visibilidad de contextos.
- los tipos de datos definidos por el usuario (*carrier sets*).
- las constantes.
- los axiomas.

3.1.1.1 Extensión o visibilidad de contextos

Para que un contexto C_2 vea (o extienda) a otro C_1, se debe añadir su nombre (C_1) en la sección *Extends* del otro (C_2). El contexto resultante, C_2, se compondrá, pues, de sus axiomas y constantes enriquecidos con todos los de C_1 y de cualquier otro contexto que extienda, tratándolos como si fueran propios. Es por ello que no deben haber identificadores repetidos, pues producirá un error.

3.1.1.2 Tipos de datos definidos por el usuario (carrier sets)

Para declarar un nuevo tipo de dato, se añade su nombre (un identificador) en la sección *Sets*. Este identificador debe ser único, es decir, no debe haber sido usado como nombre de constante o de conjunto ni en el contexto actual ni en cualquier otro al que éste extienda. De forma implícita, el identificador es tomado como una nueva constante que representa al conjunto de todos los elementos del tipo.

Una forma habitual de declarar conjuntos enumerados (aquéllos donde se especifican de forma explícita todos sus elementos) es a través del operador *partition*. Por ejemplo, si queremos definir el conjunto $S = \{e_1, ..., e_n\}$, habrá que declarar $e_1, ..., e_n$ como constantes y añadir el axioma $partition(S, e_1, ..., e_n)$.

3.1.1.3 Constantes y axiomas

Event-B engloba bajo el epígrafe de *constantes* tanto aquellos elementos que no cambian nunca de valor como aquéllos que sí (lo que podríamos interpretar como variables) y las funciones. Todas ellas son declaradas estableciendo un identificador (único) en la sección *Constants*, al que se deben añadir más tarde axiomas a partir de los cuales inferir sus tipos.

Un axioma es una declaración que es asumida como cierta en el resto del modelo. Cada axioma se compone de una etiqueta y un predicado *P*. Todos los identificadores libres en *P* deben ser constantes.

Si un axioma se declara como teorema (con la palabra *theorem*), su validez debe poder ser probada por los axiomas declarados previamente.

El siguiente ejemplo, que será traducido más adelante a *Maude*, muestra las partes

enunciadas más arriba:

CONTEXT

agatha → **Nombre del contexto**

SETS

persons → **declaración de un conjunto que se llamará *persons***

CONSTANTS

Agatha
butler
Charles
hates
richer
killer

declaración de las constantes *Agatha*, *butler* y *Charles*, de los predicados *hates* y *richer* y de la variable *killer*.

AXIOMS

person_partition : partition(persons, {Agatha}, {butler}, {Charles}) → **definición del conjunto *persons*, e implícitamente, de *Agatha*, *butler* y *Charles*.**

hate_relation : hates ∈ persons ↔ persons → **definición de la función *hates***.

richer_relation : → **definición de la función *richer***.

richer ∈ persons ↔ persons ∧

richer ∩ id = ∅ ∧

(∀ x,y,z ·(x ↦ y ∈ richer ∧ y ↦ z ∈ richer) → x ↦ z ∈ richer) ∧

(∀x,y · x ∈ persons ∧ y ∈ persons ∧ x ≠ y → (x ↦ y ∈ richer ↔ y ↦ x ∉ richer))

killer_type : killer ∈ persons → **definición de la variable *killer***.

...

solution : theorem: killer = Agatha → **teorema que, por tanto, se ha de demostrar a partir de los axiomas dados previamente.**

END

3.1.2. Máquinas

Una máquina describe el comportamiento dinámico de un modelo mediante variables cuyos valores cambian debido a la ejecución de los eventos. Su estructura

genérica es:

machine
  identificador de la máquina
refines
  identificador de la máquina que refina
sees
  identificador del contexto
variables
  identificadores de sus variables
invariants
  con el formato *etiqueta : predicado*
theorems
  con el formato *etiqueta : predicado*
variant
  variantes
events
  initialisation
  $evento_1$
  …
  $evento_n$
end

Aquí existen dos elementos básicos que se deben probar:

1. La máquina debe ser consistente, o sea, nunca debe ser capaz de alcanzar un estado que viole alguna invariante.
2. La máquina es un refinamiento correcto, es decir, su comportamiento debe corresponderse con el de todas las otras máquinas que aquélla pudiera refinar.

3.1.2.1 Refinamiento y máquinas abstractas

Al igual que en el caso de los contextos, una máquina M_2 puede refinar a otra M_1 (como máximo). M_1 se conoce como máquina abstracta y M_2 como máquina concreta.

Este refinamiento consta de dos aspectos:

1. El estado de la máquina concreta está conectado con el de la abstracta mediante una invariante que relaciona las variables de ambas y que recibe el nombre de *invariante cohesionadora*.
2. Cada evento abstracto puede ser refinado por uno o más eventos concretos.

La invariante completa de la máquina la componen ambos tipos de invariantes, tanto la concreta como la abstracta, y se acumula durante los refinamientos.

Los refinamientos pueden ser empleados tanto para añadir complejidad al modelo (refinamiento por superposición u horizontal) como para agregar detalles a las estructuras de datos (refinamiento de datos o vertical).

3.1.2.2 Visibilidad de un contexto por una máquina

Si una máquina ve un contexto, los conjuntos y constantes declarados en éste son susceptibles de ser empleados por aquélla en todos sus predicados y expresiones.

3.1.2.3 Variables e invariantes

Para agregar una variable a una máquina, se emplea la sección *Variables* donde se añade, de forma análoga a la definición de constantes en una máquina, un identificador único. Su tipo debe poder ser inferido por las invariantes de la máquina.

Las variables de las máquinas abstractas $M_1,...,M_{n-1}$ se indican con una $\boldsymbol{v}$ y las de la máquina concreta con una $\boldsymbol{w}$.

Una invariante es una expresión que debe cumplirse en cada estado de la máquina. Se compone de una etiqueta y de un predicado y puede incluir constantes y variables de la máquina concreta y de todas sus abstractas. Aquéllas que sean marcadas como teoremas deducen su corrección a partir de la preservación de otras invariantes, por lo que la suya propia no requiere ser probada.

3.1.2.3.1. Variables comunes entre máquinas

Si una variable es declarada en una máquina M_j, es posible “redefinirla” en el refinamiento directo M_{j+1}. Entonces, se asume que los valores de la variable concreta y la abstracta son siempre iguales.

Si una variable desaparece en un refinamiento, o sea, no se declara, no es posible reintroducirla en un refinamiento posterior.

3.1.2.4 Variantes

Event-B permite probar la terminación de los eventos. Dicha terminación significa que un conjunto escogido de eventos son activados sólo un número finito de veces antes de que uno no marcado como terminante tenga lugar. Para permitir demostrarla, en un modelo se puede establecer una *variante*, una expresión numérica entera o de conjuntos finitos cuyas variables libres deben ser constantes o variables concretas.

Los eventos se etiquetan de tres maneras:

- *ordinary*: no se encuentran constreñidos por la limitación de la variante.
- *convergent*: deben decrementar la variante, que es obligatorio que aparezca.
- *anticipated*: deben no incrementar la variante.

De modo informal, la terminación se demuestra estableciendo que los eventos convergentes reducen la variante (que tiene un límite inferior). Si sólo hay eventos anticipados, es suficiente con crear una variante que ellos no aumenten y probar que es así.

En el ejemplo mostrado a continuación se describe una máquina de nombre *f_m2* que es un refinamiento de *f_m1* y que puede ver el contexto *f_c0*. Define las variables *r* y *s* (como números, si se observan las invariantes) y *d* y *g* (como conjuntos, pues en *f_c0* se estableció que *f* es una función que, dado un número entre 1 y n, devuelve un conjunto). Como el evento *send* es convergente, la máquina precisa de una variante estrictamente decreciente, que es $r + 1 - s$.

```
MACHINE
  f_m2
REFINES
  f_m1
SEES
  f_c0
VARIABLES
```

```
  g
  r
  s
  d
INVARIANTS
  inv1  :     s≤n+1
  inv2  :     s ∈ r..r+1
  inv3  :     s=r+1 ⇒ d=f(r)
VARIANT
  r+1−s
EVENTS
INITIALISATION  ≙
  STATUS
    ordinary
  BEGIN
    act1  :   g ≔ ∅
    act2  :   r ≔ 1
    act3  :   s ≔ 1
    act4  :   d :∈ D
END
final  ≙
  STATUS
    ordinary
  REFINES
    final
  WHEN
    grd1  :   r = n+1
  THEN
    skip
END
```

```
receive  ≙
  STATUS
    ordinary
  REFINES
    receive
  WHEN
    grd1  :  s=r+1
  THEN
    act1  :  g(r) ≔ d
    act2  :  r ≔ r+1
END
send  ≙
  STATUS
    convergent
  WHEN
    grd1  :  s = r
    grd2  :  r ≠ n+1
  THEN
    act1  :  d ≔ f(s)
    act2  :  s ≔ s+1
END
END
```

3.1.2.5 Eventos

Un evento define un posible cambio de estado para una máquina. La condición bajo la cual puede ser ejecutado recibe el nombre de **guarda**. Su **acción** describe cómo se relacionan entre sí el viejo y el nuevo estado. El **testigo** relaciona los parámetros abstractos con los concretos y permite así que desaparezcan los primeros.

El tiempo no se tiene en cuenta en su ejecución. Además, dos eventos nunca sucederán simultáneamente.

Un evento posee los elementos siguientes:

identificador del evento
status
{*ordinary*, *convergent*, *anticipated*}
refines
identificador del evento abstracto
any
identificadores de sus parámetros
where o bien *when*
guardas con el formato *etiqueta: predicado*
with
con el formato *etiqueta: testigo*
then
con el formato *etiqueta: acción*
end

Las máquinas presentan un evento especial, *initialisation*, que sirve para empezar la ejecución proporcionando un estado inicial. Siempre es el primero y sólo se activa esa vez; por eso no presenta guardas ni parámetros.

Como se ha mencionado anteriormente, un evento (evento concreto) puede refinar a uno o más eventos de una máquina abstracta (eventos abstractos).

En la máquina *mac1* descrita en la página 51 se encuentra *set_cars_colours*. Es un evento que refina otro, *set_cars*, que requiere de un parámetro denominado *new_value_colours* y que, si se cumplen las guardas dadas en el apartado *where* y la relación entre el parámetro del evento abstracto (*new_value*) y el del concreto dada en *with*, ejecuta *act1*:

set_cars $\triangleq$ → **evento abstracto**
any
new_value
where

```
    grd1 : new_value ∈ BOOL
    grd2 : new_value = TRUE ⇒ peds_go = FALSE
  then
    act1 : cars_go := new_value
end
```

```
set_cars_colours≙ → evento concreto
  refines
    set_cars
  any
    new_value_colours
  where
    grd1 : new_value_colours ⊆ COLOURS
    grd2 : green ∈ new_value_colours → peds_colour = red
    grd_y_r : cars_colours = {yellow} → new_value_colours = {red}
    grd_r_ry : cars_colours = {red} → new_value_colours = {red, yellow}
    grd_ry_g : cars_colours = {red, yellow} → new_value_colours = {green}
    grd_g_y : cars_colours = {green} → new_value_colours = {yellow}
  with
    new_value : new_value = TRUE ↔ green ∈ new_value_colours
  then
    act1 : cars_colours := new_value_colours
end
```

4) Codificación de *Event-B* en *Maude*

Dada la estructura de *Event-B* y de *Maude*, la intuición indicaba que los contextos, por ser la parte estática, serían traducidos como módulos funcionales y las máquinas, por ser la dinámica, como módulos del sistema.

En aquellos casos donde lo que se define en el contexto es sencillo (conjuntos por enumeración, constantes enteras...), sí es cierto; sin embargo, el trabajo con el primer ejemplo escogido (ver página 33) mostró que si el contexto es complejo o presenta algo que demostrar (por existir el apartado *theorem*), esa aproximación ya no es válida.

Por otra parte, como se verá, dicho ejemplo contiene numerosas expresiones matemáticas, y su manipulación es un aspecto donde *Maude* destaca. Se buscó la forma de crear equivalencias aquí entre ambos lenguajes. No obstante, la cantidad y las dificultades surgidas llevaron a plantearnos un objetivo más modesto: averiguar si existía alguna manera de traducir el contexto y que funcionara. Una vez hallada, se tendría un punto de partida para generalizar y, más tarde, automatizar el proceso.

Para ampliar la visión de *Event-B*, seguimos con la traducción y buscamos un modelo que contuviera una máquina. De esta manera, ampliaríamos el tratamiento a los eventos y pasaríamos a poder manejar estados. Como ya se ha comentado, el estado de una máquina viene descrito por el contenido de sus variables: el evento *Initialisation* establece el punto de partida y el resto lo va modificando si procede.

Aquí surgió un nuevo interrogante: ¿cómo implementar los parámetros de un evento? Como en el caso del contexto, se optó por dar prioridad a encontrar una solución que funcionara, y ésta fue agregar al estado del sistema tantas variables extra como parámetros requirieran los eventos (en el ejemplo, una) y darle un valor en el *Initialisation* de *Maude*.

Para establecer un límite en el desarrollo de este trabajo final de máster, se procedió a traducir otro ejemplo que incluyera un contexto y una máquina, por un lado, y el refinamiento, por otro. Con eso en mente se eligió antes el modelado de semáforos binarios para ahora dar un paso más y sustituirlos por semáforos con colores.

El mecanismo de herencia entre máquinas y contextos, por separado, y el de visibilidad entre ambos tiene su equivalente en *Maude* en la importación de módulos y, a priori, no presenta mayor problema aquí (aunque es muy posible que sí los hubiera en una cadena de refinamientos, dado que si una variable desaparece en uno de sus pasos, no se la puede reintroducir en uno posterior).

Sin embargo, lo que nos obligó a tratar este caso de manera independiente al anterior fue la falta de tiempo para hallar formas de comunicar variables y eventos concretos con sus correspondientes abstractos. Así, siguiendo con la filosofía de averiguar primero si se puede traducir, se obvió este aspecto, pues tampoco invalidaba el modelo.

5) Casos de estudios

En este apartado pasan a explicarse, con detalle y de menor a mayor dificultad, tres traducciones de *Event-B* a *Maude* de ejemplos de contextos y máquinas, buscando generalizar al máximo para que pueda ser aplicado de manera sistemática.

Se empieza con un modelo que sólo incluye un contexto, rico en expresiones matemáticas. El segundo es únicamente una máquina y, con ella, la manipulación de eventos. El tercero es un refinamiento del segundo, con contexto y máquina, para empezar a tratar las relaciones entre ellos.

5.1. ¿Quién mató a Agatha?

5.1.1. Descripción

El contexto que se presenta a continuación implementa la solución al siguiente misterio: ¿Quién mató a la tía Agatha?

> Agatha, el mayordomo y Charles viven en la mansión Dreadsbury y son sus únicos habitantes. El asesino siempre odia a alguien y no es más rico que su víctima. Charles no odia a nadie que Agatha sí odie. Agatha odia a todo el mundo excepto al mayordomo. Éste, a su vez, odia a todos los que no son más ricos que Agatha y también a aquellos a los que Agatha odia. Para acabar, nadie odia a todos.

5.1.2. Implementación en *Event-B*

```
CONTEXT
  agatha
SETS
  persons
CONSTANTS
   Agatha
   butler
   Charles
   hates
```

```
  richer
  killer
AXIOMS
  person_partition : partition(persons, {Agatha}, {butler}, {Charles})
  hate_relation : hates ∈ persons ↔ persons
  richer_relation :
    richer ∈ persons ↔ persons ∧
    richer ∩ id = ∅ ∧
    (∀ x,y,z ·(x ↦ y ∈ richer ∧ y ↦ z ∈ richer) → x ↦ z ∈ richer) ∧
    (∀x,y · x ∈ persons ∧ y ∈ persons ∧ x ≠ y → (x ↦ y ∈ richer ↔ y ↦ x ∉ richer))
  killer_type : killer ∈ persons
  killer_hates : killer  ↦ Agatha ∈ hates
  killer_not_richer : killer  ↦ Agatha ∉ richer
  charles_hates : hates[{Agatha} ] ∩ hates[{Charles}] = ∅
  agatha_hates : hates[{Agatha} ] = persons \ {butler}
  butler_hates_1 : ∀ x·( x ↦ Agatha ∉ richer → butler ↦ x ∈ hates)
  butler_hates_2 : hates[{Agatha}] ⊆ hates[{butler}]
  noone_hates_everyone : ∀ x ·x ∈ persons ∧ hates[{x}]  ≠ persons
  solution : theorem: killer = Agatha
END
```

5.1.3. Cómo y por qué se tradujo así

La razón por la que se seleccionó este ejemplo fue la variedad de expresiones que lo componen. No obstante, su complejidad hizo que se optara por usar reglas en vez de ecuaciones, con lo que el tipo de módulo es de sistema y se pasó de tener un sistema estático en *Event-B* a uno dinámico en *Maude*.

En cada instante, el estado del sistema viene definido por una bolsa de hechos que empieza estando vacía (*op empty : -> PPys .*) y que, según avanza la secuencia de deducción, va acumulando predicados del tipo *X in hates(Y)*, *X !in hates(Y)*, *X in richer(Y)* y *X !in richer(Y)* mediante el operador +:

```
  op empty : -> PPys .
  op _+_ : PPys PPys -> PPys [assoc comm id: empty] .
```

Se considera que existe un fallo cuando dos o más de sus elementos se contradicen, es decir, en la bolsa se hallan de manera simultánea expresiones tales como *X in hates(Y)* y *X !in hates(Y)*.

```
--- Explicitamos que es un fallo si en la bolsa encuentra un hecho y
su contrario
  rl [fail] : {Prop + Y in A + Y !in A} => fail .
```

La introducción del conjunto *Persons* (declaración en la sección *Sets* combinada con las 3 constantes y el axioma *person_partition*) se implementó mediante *Sorts Persons* seguida de *ops charles butler agatha : -> Persons* ..

SETS

persons

CONSTANTS

Agatha

butler

Charles

hates

richer

killer

AXIOMS

person_partition : partition(persons, {Agatha}, {butler}, {Charles})

```
  sorts Persons PPy PPys Atom System .
--- Axioma 1 (person_partition): Persons = {Agatha, Charles, butler}
  ops charles butler agatha : -> Persons .
```

Las constantes *hates* y *richer* son en realidad relaciones entre *persons*, como se puede observar en *hate_relation* y *reacher_relation*, respectivamente. En la traducción se ha optado por tratarlos como operandos que, dado un elemento *P* de tipo *Persons*, retorna una estructura que incluye dicho operando, es decir, *hates(P)* o *richer(P)*.

hate_relation : hates ∈ persons ↔ persons

richer_relation : richer ∈ persons ↔ persons

```
--- Axiomas 2 (hate_relation) y 3 (richer_relation1):
  ops hates richer : Persons -> Atom .
```

Para este último, además, se ha desglosado cada una de sus condiciones en reglas:

➔ *richer_relation2* (richer ∩ id = Ø): El sistema acabará en fallo si encuentra una estructura que cumpla que *X:Persons in richer(X:Persons)*.

richer ∩ id = Ø

```
--- Axioma 4 (richer_relation2).
  rl [richer_relation2] : {Prop + X in richer(X)} => fail .
```

➔ *richer_relation3* (∀ x,y,z ·(x ↦ y ∈ richer ∧ y ↦ z ∈ richer) → x ↦ z ∈ richer): es la propiedad transitiva de ese operando: si se verifica que *Y:Persons in richer(X:Persons)* y *Z:Persons in richer(Y:Persons)*, entonces se añade al sistema que *Z:Persons in richer(X:Persons)*.

∀ x,y,z ·(x ↦ y ∈ richer ∧ y ↦ z ∈ richer) → x ↦ z ∈ richer)

```
--- Axioma 5 (richer_relation3).
  rl [richer_relation3] : {Prop + Y in richer(X) + Z in richer(Y)} =>
{Prop + Y in richer(X) + Z in richer(Y) + Z in richer(X)} .
```

➔ *richer_relation4* (∀x,y · x ∈ persons ∧ y ∈ persons ∧ x ≠ y → (x ↦ y ∈ richer ↔ y ↦ x ∉ richer): es la antisimétrica de *richer*: si *X* e *Y* son dos elementos distintos de tipo *Persons*, se cumple que si *Y in richer(X)*, entonces *X !in richer(Y)* y viceversa.

(∀x,y · x ∈ persons ∧ y ∈ persons ∧ x ≠ y → (x ↦ y ∈ richer ↔ y ↦ x ∉ richer))

```
--- Axioma 6 (richer_relation4).
 crl [richer_relation4a] : {Prop + Y in richer(X)} => {Prop + Y in
richer(X) + X !in richer(Y)} if X =/= Y .
 crl [richer_relation4b] : {Prop + X !in richer(Y)} => {Prop + X !in
richer(Y) + Y in richer(X)} if X =/= Y .
```

La constante *killer*, del tipo *Persons* según el axioma *killer_type*, ha sido transformada en una constante a la que se le asignará cada uno de los valores posibles

(*Agatha*, *Charles* y *butler*) antes de lanzar la búsqueda {*empty*} =>* *fail* que permita saber si se llega a una contradicción y, por tanto, la hipótesis es falsa. Las características dadas en *killer_hates* (*killer* ↦ *Agatha* ∈ *hates*) y *killer_not_richer* (*killer* ↦ *Agatha* ∉ *richer*) son implementadas como dos reglas que añaden esos hechos al sistema:

➔ *Agatha* ∈ *hates(Killer)*: *agatha in hates(killer)*).

killer_hates : killer ↦ Agatha ∈ hates

```
--- Axioma 8 (killer_hates).
  rl [killer_hates] : {Prop} => {Prop + agatha in hates(killer)} .
```

➔ *Agatha* ∉ *richer(Killer)*: *agatha !in richer(killer)*.

killer_not_richer : killer ↦ Agatha ∉ richer

```
--- Axioma 9 (killer_not_richer).
  rl [killer_not_richer] : {Prop} => {Prop + agatha !in
richer(killer)} .
```

La configuración de quién odia a quién ha sido traducida así:

➔ *charles_hates* (*hates[{Agatha}]* ∩ *hates[{Charles}]* = ∅): si *X:Persons in hates(agatha)*, entonces se añade *X !in hates(charles)* y viceversa. Además, se explicita que no es posible que *X in hates(charles)* + *X in hates(agatha)*.

charles_hates : hates[{Agatha}] ∩ hates[{Charles}] = ∅

```
--- Axioma 10 (charles_hates).
  rl [charles_hates_1] : {Prop + X in hates(agatha)}  => {Prop + X in
hates(agatha)  + X !in hates(charles)} .
  rl [charles_hates_2] : {Prop + X in hates(charles)} => {Prop + X in
hates(charles)  + X !in hates(agatha)} .
  rl [charles_hates_3] : {Prop + X in hates(charles) + X in
hates(agatha)} => fail .
```

➔ *agatha_hates* (*hates[{Agatha}]* = *persons* \ {*butler*}): es un aspecto a mejorar. De momento, para poder seguir adelante, se hizo una enumeración: *agatha*|*charles*|*butler in hates(agatha)*.

agatha_hates : hates[{Agatha}] = persons \ {butler}

```
--- Axioma 11 (agatha_hates).
  rl [agatha_hates_1] : {Prop} => {Prop + agatha in hates(agatha)} .
  rl [agatha_hates_2] : {Prop} => {Prop + charles in hates(agatha)} .
  rl [agatha_hates_3] : {Prop} => {Prop + butler !in hates(agatha)} .
```

➔ *butler_hates_1* ($\forall x \cdot (x \mapsto Agatha \notin richer \rightarrow butler \mapsto x \in hates)$): si se tiene que *agatha !in richer(X:Persons)*, entonces se añade que *X:Persons in hates(butler)*.

butler_hates_1 : $\forall x \cdot (x \mapsto Agatha \notin richer \rightarrow butler \mapsto x \in hates)$

```
--- Axioma 12 (butler_hates_1).
  rl [butler_hates_1] : {Prop + agatha !in richer(X)} => {Prop +
agatha !in richer(X) + X in hates(butler)} .
```

➔ *butler_hates_2* ($hates[\{Agatha\}] \subseteq hates[\{butler\}]$): se implementa la parte positiva (si *X:Persons in hates(agatha)*, entonces *X:Persons in hates(butler)*) y la negativa que va implícita (si *X:Persons !in hates(butler)*, entonces *X:Persons !in hates(agatha)*).

butler_hates_2 : $hates[\{Agatha\}] \subseteq hates[\{butler\}]$

```
--- Axioma 13 (butler_hates_2).
  rl [butler_hates_2_1] : {Prop + X in hates(agatha)}  => {Prop + X
in hates(agatha) + X in hates(butler)} .
  rl [butler_hates_2_2] : {Prop + X !in hates(butler)} => {Prop + X !
in hates(butler) + X !in hates(agatha)} .
```

➔ *noone_hates_everyone* ($\forall x \cdot x \in persons \wedge hates[\{x\}] \neq persons$): la forma escogida ha sido que el sistema falle si se encuentra que *agatha in hates(X:Persons)*, *charles in hates(X:Persons)* y *butler in hates(X:Persons)*.

noone_hates_everyone : $\forall x \cdot x \in persons \wedge hates[\{x\}] \neq persons$

```
--- Axioma 14 (no_one_hates_everyone).
  rl [no_one_hates_everyone] : {Prop + agatha in hates(X) + charles
in hates(X) + butler in hates(X)} => fail .
```

5.1.4. Implementación en *Maude*

Para llevar a cabo la traducción, fue necesario implementar también lo siguiente:

➔ los operadores de (no) pertenencia como (*!*)*in*:

```
    ops _in_ _!in_ : Persons Atom -> Ppy .
```

➔ el conjunto de proposiciones *PPys*, con una definición recursiva por acumulación y una ecuación para eliminar elementos repetidos:

```
    op empty : -> PPys .
    op _+_ : PPys PPys -> PPys [assoc comm id: empty] .
    eq P:PPy + P:PPy = P:PPy .
```

➔ la definición del sistema y del fallo:

```
    op {_} : PPys -> System .
    op fail : -> System .
```

El resultado completo se muestra a continuación:

```
mod AGATHA is
  sorts Persons PPy PPys Atom System .
  subsort PPy < PPys .

--- Axioma 1 (person_partition): Persons = {Agatha, Charles, butler}
  ops charles butler agatha : -> Persons .

--- Axiomas 2 (hate_relation) y 3 (richer_relation1): definimos hates
y richer como operandos que, dada una "Person", devuelve algo del
tipo "hates(B)"
  ops hates richer : Persons -> Atom .

--- Definimos ∈ y ∉
  ops _in_ _!in_ : Persons Atom -> PPy [prec 31] .

--- Construimos PPys como un conjunto de átomos, por lo que es
necesario el conjunto vacío...
  op empty : -> PPys .
--- ...y una definición recursiva por acumulación.
  op _+_ : PPys PPys -> PPys [assoc comm id: empty] .

--- Para evitar elementos repetidos...
```

```
  eq P:PPy + P:PPy = P:PPy .

--- Para simplificar el espacio de búsqueda, creamos un tipo System
  op {_} : PPys -> System .

--- Definimos la constante "fail" para que, cuando aparezca,
significique el fallo.
  op fail : -> System .

--- Axioma 7 (killer_type): definición de "killer" como una constante
de tipo Persons
  op killer : -> Persons .

--- El search se lanzará con {empty} =>* fail .

  var Prop : PPys .
  vars X Y Z : Persons .
  var A : Atom .

--- Axioma 4 (richer_relation2): richer ∩ id = ∅.
  rl [richer_relation2] : {Prop + X in richer(X)} => fail .

--- Axioma 5 (richer_relation3): ∀X, Y, Z·(X |-> Y) ∈ richer /\ (Y
|-> Z) ∈ richer => (X |-> Z) ∈ richer, o sea, ∀X,Y,Z: Y ∈
richer(X) /\ Z ∈ richer(Y) -> Z ∈ richer(X). En otras palabras, la
transitiva de richer.
  rl [richer_relation3] : {Prop + Y in richer(X) + Z in richer(Y)} =>
{Prop + Y in richer(X) + Z in richer(Y) + Z in richer(X)} .

--- Axioma 6 (richer_relation4): ∀X, Y·((X ∈ persons) /\ (Y ∈
persons) /\ X≠Y => ((X |-> Y) ∈ richer <=> (Y |-> X) ∉ richer)), o
sea, ∀X,Y ∈ persons, X≠Y: Y ∈ richer(X) <=> X ∉ richer(Y). En otras
palabras, la antisimétrica de richer.
 crl [richer_relation4a] : {Prop + Y in richer(X)} => {Prop + Y in
richer(X) + X !in richer(Y)} if X =/= Y .
```

```
 crl [richer_relation4b] : {Prop + X !in richer(Y)} => {Prop + X !in
richer(Y) + Y in richer(X)} if X =/= Y .

--- Axioma 8 (killer_hates): (Killer |-> Agatha) ∈ hates, o sea,
Agatha ∈ hates(Killer)
  rl [killer_hates] : {Prop} => {Prop + agatha in hates(killer)} .

--- Axioma 9 (killer_not_richer): (Killer |-> Agatha) ∉ richer, o
sea, Agatha ∉ richer(Killer)
  rl [killer_not_richer] : {Prop} => {Prop + agatha !in
richer(killer)} .

--- Axioma 10 (charles_hates): hates[{Agatha}] ∩ hates[{Charles}]=∅
  rl [charles_hates_1] : {Prop + X in hates(agatha)}  => {Prop + X in
hates(agatha)  + X !in hates(charles)} .
  rl [charles_hates_2] : {Prop + X in hates(charles)} => {Prop + X in
hates(charles)  + X !in hates(agatha)} .
  rl [charles_hates_3] : {Prop + X in hates(charles) + X in
hates(agatha)} => fail .

--- Axioma 11 (agatha_hates): hates[{agatha}] = persons \ {butler}
  rl [agatha_hates_1] : {Prop} => {Prop + agatha in hates(agatha)} .
  rl [agatha_hates_2] : {Prop} => {Prop + charles in hates(agatha)} .
  rl [agatha_hates_3] : {Prop} => {Prop + butler !in hates(agatha)} .

--- Axioma 12 (butler_hates_1): ∀X·(X |-> Agatha) ∉ richer => (butler
|-> X) ∈ hates, o sea, ∀X·Agatha ∉ richer(X) => X ∈ hates(butler)
  rl [butler_hates_1] : {Prop + agatha !in richer(X)} => {Prop +
agatha !in richer(X) + X in hates(butler)} .

--- Axioma 13 (butler_hates_2): hates[{Agatha}] ⊆ hates[{butler}]
  rl [butler_hates_2_1] : {Prop + X in hates(agatha)}  => {Prop + X
in hates(agatha) + X in hates(butler)} .
  rl [butler_hates_2_2] : {Prop + X !in hates(butler)} => {Prop + X !
in hates(butler) + X !in hates(agatha)} .
```

```
--- Axioma 14 (no_one_hates_everyone): ∀X·X ∈ persons => hates[{X}] ≠
persons.
  rl [no_one_hates_everyone] : {Prop + agatha in hates(X) + charles
in hates(X) + butler in hates(X)} => fail .

--- Explicitamos que es un fallo si en la bolsa encuentra un hecho y
su contrario
  rl [fail] : {Prop + Y in A + Y !in A} => fail .

endm
```

5.1.5. *Salida de la ejecución*

A continuación, se muestran los resultados de lanzar la búsqueda, partiendo de un conjunto vacío de hechos, con las tres opciones posibles para *killer*: que sea Agatha, que sea Charles y que sea el mayordomo. Como se puede observar, para el caso de Agatha no encuentra el fallo mientras que para Charles y el mayordomo sí.

```
mod AGATHA-A is
  including AGATHA .
--- Se prueba con que la asesina sea Agatha.
 eq [killer-agatha] : killer = agatha
endm

search {empty} =>* fail .
```

```
mod AGATHA-C is
  including AGATHA .
--- Se prueba con que el asesino sea Charles.
 eq [killer-charles] : killer = charles .
endm

search {empty} =>* fail .
```

```
show path 33 .
***(
state 0, System: {empty}
===[ rl {Prop:PPys} => {Prop:PPys + agatha in hates(killer)} [label
killer_hates] . ]===>
state 1, System: {agatha in hates(killer)}
===[ rl {Prop:PPys} => {Prop:PPys + agatha in hates(agatha)} [label
agatha_hates_1] . ]===>
state 7, System: {agatha in hates(agatha) + agatha in hates(killer)}
===[ rl killer => charles [label killer-charles] . ]===>
state 35, System: {agatha in hates(charles) + agatha in
hates(agatha)}
===[ rl {Prop:PPys + X:Persons in hates(charles) + X:Persons in
hates(agatha)} => fail [label charles_hates_3] . ]===>
state 124, System: fail
)***
```

```
mod AGATHA-B is
  including AGATHA .
--- Se prueba con que el asesino sea el mayordomo.
  eq [killer-butler] : killer = butler .
endm

search {empty} =>* fail .
show path 196 .
***(
state 0, System: {empty}
===[ rl {Prop:PPys} => {Prop:PPys + agatha in hates(killer)} [label
killer_hates] . ]===>
state 1, System: {agatha in hates(killer)}
===[ rl {Prop:PPys} => {Prop:PPys + agatha !in richer(killer)} [label
killer_not_richer] . ]===>
state 6, System: {agatha in hates(killer) + agatha !in
richer(killer)}
```

```
===[ rl {Prop:PPys} => {Prop:PPys + charles in hates(agatha)} [label
agatha_hates_2] . ]===>
state 26, System: {charles in hates(agatha) + agatha in hates(killer)
+ agatha !in richer(killer)}
===[ rl {Prop:PPys + agatha !in richer(X:Persons)} => {Prop:PPys +
X:Persons in hates(butler) + agatha !in richer(X:Persons)} [label
butler_hates_1] . ]===>
state 91, System: {charles in hates(agatha) + agatha in hates(killer)
+ killer in hates(butler) + agatha !in richer(killer)}
===[ rl {Prop:PPys + X:Persons in hates(agatha)} => {Prop:PPys +
X:Persons in hates(butler) + X:Persons in hates(agatha)} [label
butler_hates_2_1] . ]===>
state 264, System: {charles in hates(butler) + charles in
hates(agatha) + agatha in hates(killer) + killer in hates(butler) +
agatha !in richer(killer)}
===[ rl killer => butler [label killer-butler] . ]===>
state 648, System: {charles in hates(butler) + charles in
hates(agatha) + butler in hates(butler) + agatha in hates(killer) +
agatha !in richer(killer)}
===[ rl killer => butler [label killer-butler] . ]===>
state 1341, System: {charles in hates(butler) + charles in
hates(agatha) + butler in hates(butler) + agatha in hates(butler) +
agatha !in richer(killer)}
===[ rl {Prop:PPys + charles in hates(X:Persons) + butler in
hates(X:Persons) + agatha in hates(X:Persons)} => fail [label
no_one_hates_everyone] . ]===>
state 2340, System: fail
)***
```

5.2. Semáforo binario

5.2.1. Descripción

Se busca implementar un cruce simplificado vehículo–peatón regulado por semáforos. Las señales se modelan mediante valores booleanos: *false* para parar y *true*

para pasar.

5.2.2. Implementación en *Event-B*

```
MACHINE
  mac
VARIABLES
  cars_go
  peds_go
INVARIANTS
  inv1 : cars_go ∈ BOOL
  inv2 : peds_go ∈ BOOL
  inv3 : ¬ (cars_go = TRUE & peds_go = TRUE)
EVENTS
Initialisation
  begin
    act1 : cars_go := FALSE
    act2 : peds_go := FALSE
  end
set_peds_go ≙
  when
    grd1 : cars_go = FALSE
  then
    act1 : peds_go := TRUE
  end
set_peds_stop ≙
  begin
    act1 : peds_go := FALSE
  end
set_cars ≙
  any
    new_value
```

```
 where
   grd1 : new_value ∈ BOOL
   grd2 : new_value = TRUE ⇒ peds_go = FALSE
 then
   act1 : cars_go := new_value
 end
END
```

5.2.3. Cómo y por qué se tradujo así

Se escogió este ejemplo para empezar a trabajar con los eventos. Aquí no hay contextos.

El control del semáforo de coches se lleva a cabo con la variable *cars_go* y el de peatones, con *peds_go*. Se establece su tipo mediante las invariantes *inv1* e *inv2*, respectivamente, y su traslación a *Maude* es inmediata.

```
INVARIANTS
 inv1 : cars_go ∈ BOOL
 inv2 : peds_go ∈ BOOL
```

```
--- Cars_go: Semáforo que regula el paso de los coches
--- Peds_go: Semáforo para los peatones.
  vars Cars_go Peds_go new_value : Bool .
```

Además, se impide que ambos estén abiertos con *inv3*, que no se explicita porque, dada la construcción del sistema, es un estado inaccesible.

El estado del sistema viene dado por el valor de ambas. Sin embargo, se ha introducido una tercera variable, *new_value*, para simular así el parámetro del evento *set_cars* pues guardará el nuevo valor que se querrá establecer para el semáforo de vehículos cuando dicho evento sea llamado.

```
  sort State .
--- El estado del sistema está formado por el valor de los dos
semáforos en un momento determinado.
  op s(_,_,_) : Bool Bool Bool -> State . --- Cars_go, Peds_go,
```

```
new_value
```

El evento *Initialisation* asigna a ambas el valor *false* y se traduce como un operando del mismo nombre al que se le ha añadido un argumento, *new_value*, que al lanzarlo contendrá el parámetro para *set_cars* antes mencionado:

```
Initialisation
 begin
  act1 : cars_go := FALSE
  act2 : peds_go := FALSE
 end
```

```
op initialisation : Bool -> State .
eq initialisation(new_value) = s(false,false,new_value) .
```

Para controlar el semáforo peatonal se tienen los eventos:

➔ *set_peds_stop*, que lo cierra estableciendo el valor de *peds_go* a *false*; para ello, se ha creado una regla del mismo nombre donde se cambia de estado poniendo el segundo argumento a falso:

```
set_peds_stop ≙
 begin
  act1 : peds_go := FALSE
 end
```

```
   rl [set_peds_stop] : s(Cars_go, Peds_go, new_value) => s(Cars_go,
false, new_value) .
```

➔ *set_peds_go*, que lo abre (*peds_go* = *true*) si el semáforo de los coches está cerrado (*cars_go* = *false*); se ha elaborado una regla homónima donde, si el primer argumento es falso, se realiza una transición a un estado donde éste se mantiene y el segundo pasa a ser cierto.

```
set_peds_go ≙
 when
  grd1 : cars_go = FALSE
 then
```

```
  act1 : peds_go := TRUE
 end
```

```
rl [set_peds_go] : s(false, Peds_go, new_value) => s(false, true,
new_value) .
```

El semáforo de vehículos se gestiona mediante un único evento, *set_cars(new_value)*, que lo cambia al contenido de *new_value* con la salvedad de que si dicho *new_value* es cierto, habrá modificación sólo si el de peatones es falso.

```
set_cars ≙
 any
  new_value
 where
  grd1 : new_value ∈ BOOL
  grd2 : new_value = TRUE ⇒ peds_go = FALSE
 then
  act1 : cars_go := new_value
 end
```

Una manera simple de traducirlo es crear una regla por cada valor del parámetro:

- si es falso, da igual cómo esté el semáforo de peatones pues el de coches puede ser falso sin peligro.
- si es cierto, habrá cambio de estado (a *s(true, false, true)*) si el segundo argumento, que recordamos que simboliza el semáforo de peatones, es falso.

```
   rl [set_cars_F] : s(Cars_go, Peds_go, false)=> s(false, Peds_go,
false) .
  rl [set_cars_T] : s(Cars_go, false, true) => s(true, false, true) .
```

5.2.4. Implementación en *Maude*

En este caso, el estado del sistema viene descrito por el valor de los dos semáforos y el del nuevo valor para el de vehículos. El ejemplo completo quedaría como sigue:

```
mod MAC is
  pr BOOL .
```

```
  sort State .

--- El estado del sistema está formado por el valor de los dos
semáforos en un momento determinado.
  op s(_,_,_) : Bool Bool Bool -> State . --- Cars_go, Peds_go,
new_value

--- Cars_go: Semáforo que regula el paso de los coches
--- Peds_go: Semáforo para los peatones.
  vars Cars_go Peds_go new_value : Bool .

--- El sistema arranca con el evento Initialisation, que pone los dos
semáforos a falso.
  op initialisation : Bool -> State .
  eq initialisation(new_value) = s(false,false,new_value) .

--- Evento set_peds_stop: Pone el semáforo peatonal a false.
  rl [set_peds_stop] : s(Cars_go,Peds_go,new_value) =>
s(Cars_go,false,new_value) .

--- Evento set_peds_go: Pone el semáforo peatonal a true si el
semáforo de los coches está a false.
  rl [set_peds_go] : s(false,Peds_go,new_value) =>
s(false,true,new_value) .

--- Evento set_cars(new value): Cambia el semáforo de los coches a
"new value" pero si "new value = true" sólo lo hará si el de peatones
es false.
  rl [set_cars_F] : s(Cars_go,Peds_go,false)=> s(false,Peds_go,false)
.
  rl [set_cars_T] : s(Cars_go,false,true) => s(true,false,true) .

endm
```

5.2.5. Salida de la ejecución

Para este caso, se ha buscado ver si existe alguna manera de alcanzar el estado prohibido, a saber, ambos semáforos a *true*, independientemente del valor del parámetro de *set_cars*:

```
search initialisation(true) =>* s(true,true,new_value) .
---No solution
search initialisation(false) =>* s(true,true,new_value) .
---No solution
```

5.3. Semáforo con colores

5.3.1. Descripción

En el caso anterior se dejó de lado la implementación de los colores. Aquí se retoma teniendo en cuenta que el semáforo de peatones consistirá en los dos colores habituales (*red* y *green*) y el de coches en los tres de siempre (*red*, *yellow* y *green*) pero con la secuencia existente en Estados Unidos: verde → amarillo → rojo → rojo-amarillo (por esta razón se representa *cars_colours* como un conjunto).

5.3.2. Implementación en *Event-B*

5.3.2.1. Contexto

El contexto *ctx1* lo definen así:

```
CONSTANTS
   red
   yellow
   green
SETS
   COLOURS
AXIOMS
   type : partition(COLOURS, {red} , {yellow} , {green})
```

5.3.2.2. Máquina

La máquina *mac1* la dejan de esta manera:

```
MACHINE
  mac1
REFINES
  mac
SEES
  ctx1
VARIABLES
  peds_colour
  cars_colours
INVARIANTS
  inv4 : peds_colour ∈ {red, green}
  inv5 : cars_colours ⊆ COLOURS
  gluing_peds : peds_go = TRUE ↔ peds_colour = green
  gluing_cars : cars_go = TRUE ↔ green ∈ cars_colours
EVENTS
Initialisation
  begin
    init4 : peds_colour := red
    init5 : cars_colours := {red}
  end
set_peds_green≙
  refines
    set_peds_go
  when
    grd1 : green ∉ cars_colours
  then
    act2 : peds_colour := green
  end
```

```
set_peds_red≙
  refines
    set_peds_stop
  begin
    act1 : peds_colour := red
  end
set_cars_colours≙
  refines
    set_cars
  any
    new_value_colours
  where
    grd1 : new_value_colours ⊆ COLOURS
    grd2 : green ∈ new_value_colours → peds_colour = red
    grd_y_r : cars_colours = {yellow} → new_value_colours = {red}
    grd_r_ry : cars_colours = {red} → new_value_colours = {red, yellow}
    grd_ry_g : cars_colours = {red, yellow} → new_value_colours = {green}
    grd_g_y : cars_colours = {green} → new_value_colours = {yellow}
  with
    new_value : new_value = TRUE ↔ green ∈ new_value_colours
  then
    act1 : cars_colours := new_value_colours
  end
END
```

5.3.3. Cómo y por qué se tradujo así

La complejidad aumenta: al hecho de tener un contexto y una máquina se añade el refinamiento, pues *mac1* "desciende" de *mac*. No obstante, hemos dejado esto último a un lado para centrarnos en la incorporación de contextos y en la descripción de eventos más completos que ofrece este caso.

5.3.3.1. Contexto

En un principio, la traducción es sencilla:

```
CONSTANTS
  red
  yellow
  green
SETS
  COLOURS
AXIOMS
  type : partition(COLOURS, {red} , {yellow} , {green})
```

```
mod CTX1 is
  sort Colours  .

  ops red yellow green : -> Colours .
endm
```

Sin embargo, la implementación del semáforo peatonal (dos colores únicamente) complicaba el contexto en exceso y desviaba la atención del objetivo principal: encontrar una traducción que funcionara. Por ello, se optó por crear un subtipo para dicho semáforo y dejar para más tarde la búsqueda de una solución que pudiera ser aplicada de manera sistemática.

```
mod CTX1 is
  sorts Colours Ped_colours .
  subsort Ped_colours < Colours .

--- El conjunto Ped_colours = {red, green} -- No forma parte del
contexto original; se añade para cumplir inv4
  ops red green : -> Ped_colours .

--- El conjunto Colours = {red, yellow, green}
  op yellow : -> Colours .
endm
```

5.3.3.2. Máquina

El hecho de tener un semáforo (el de vehículos) que presenta más de un color simultáneamente significa añadir todo el tratamiento de conjuntos al sistema.

```
 --- Definimos el tipo ColourSet como un conjunto de colores.
  sort ColourSet .
  subsort Colours < ColourSet .
  op empty : -> ColourSet .
  op _,_ : ColourSet ColourSet -> ColourSet [assoc comm id: empty] .
  eq C:Colours , C:Colours = C:Colours .
--- Operador de pertenencia (∈).
  op _in_ : Colours ColourSet -> Bool  .
  eq C:Colours in C:Colours , CS:ColourSet = true .
  eq C:Colours in CS:ColourSet = false [owise] .
```

Como en el caso anterior, el estado del sistema se define por el valor de los dos semáforos y el de la variable extra que se emplea en *set_cars_colours*. Como el operador "," se ha utilizado en la construcción de *ColourSet*, se ha sustituido por ";".

```
  sort State .
--- Coches, peatones, new_value_colours
  op s(_;_;_) : ColourSet Ped_colours ColourSet -> State .
```

Las invariantes *inv4* e *inv5*, se traducen de manera implícita al declarar las variables así:

INVARIANTS

inv4 : peds_colour ∈ {red, green}

inv5 : cars_colours ⊆ COLOURS

```
---  Cars_c  (semáforo  que  regula  el  paso  de  los  coches)→inv5:
cars_colours ⊆ COLOURS .
--- Peds_c: Semáforo para los peatones.
  vars Cars_c New_value_c : ColourSet .
--- inv4 : peds_colour ∈ {red, green}. Definido como sort en CTXT1
  var Peds_c : Ped_colours .
```

El evento *Initialisation* pone los dos semáforos en rojo:

Initialisation
  begin
    init4 : peds_colour := red
    init5 : cars_colours := {red}
  end

```
  op initialisation : ColourSet -> State .
  eq initialisation(New_value_c) = s(red ; red ; New_value_c) .
```

El control del semáforo peatonal es similar al de *mac*, exceptuando que, dado que *Cars_c* es ahora un conjunto, se ha empleado una regla condicional al ponerlo en verde:

set_peds_green≙
  refines
    set_peds_go
  when
    grd1 : green ∉ cars_colours
  then
    act2 : peds_colour := green
  end
set_peds_red≙
  refines
    set_peds_stop
  begin
    act1 : peds_colour := red
  end

```
--- Evento set_peds_red: Pone el semáforo peatonal a false.
  rl [set_peds_red] : s(Cars_c ; Peds_c ; New_value_c) => s(Cars_c ;
red ; New_value_c) .
--- Evento set_peds_green: Pone el semáforo peatonal en verde si el
semáforo de los coches no lo está.
  crl [set_peds_green] : s(Cars_c ; Peds_c ; New_value_c) => s(Cars_c
; green ; New_value_c) if not(green in Cars_c) .
```

El manejo del semáforo para vehículos mediante el evento *set_cars_colours(new_value_c)* es el que sufre más modificaciones:

```
set_cars_colours≙
  refines
    set_cars
  any
    new_value_colours
  where
    grd1 : new_value_colours ⊆ COLOURS
    grd2 : green ∈ new_value_colours → peds_colour = red
    grd_y_r : cars_colours = {yellow} → new_value_colours = {red}
    grd_r_ry : cars_colours = {red} → new_value_colours = {red, yellow}
    grd_ry_g : cars_colours = {red, yellow} → new_value_colours = {green}
    grd_g_y : cars_colours = {green} → new_value_colours = {yellow}
  with
    new_value : new_value = TRUE ↔ green ∈ new_value_colours
  then
    act1 : cars_colours := new_value_colours
  end
```

```
crl [set_cars_colours] : s(Cars_c ; Peds_c ; New_value_c) =>
s(New_value_c ; Peds_c ; New_value_c)
(1)    if (green in New_value_c and Peds_c == red) or
(2)         (Cars_c == yellow and New_value_c == red) or
(3)         (Cars_c == red and New_value_c == red, yellow) or
(4)         (Cars_c == red, yellow and New_value_c == green) or
(5)         (Cars_c == green and New_value_c == yellow) .
```

- ✔ La condición *grd1* (*new_value_c* ⊆ *COLOURS*) se cumple con la definición de *new_value_c*.
- ✔ *grd2* (*green* ∈ *new_value_c* ⇒ *Peds_c* = *red*): se verifica en (1).
- ✔ *grd_y_r* (*Cars_c* = {*yellow*} ⇒ *new_value_c* = {*red*}): se comprueba en (2).

- ✔ *grd_r_ry* (*Cars_c* = {*red*} ⇒ *new_value_c* = {*red, yellow*}): se trata en (3).
- ✔ *grd_ry_g* (*Cars_c* = {*red, yellow*} ⇒ *new_value_c* = {*green*}): se testa en (4).
- ✔ *grd_g_y* (*Cars_c* = {*green*} ⇒ *new_value_c* = {*yellow*}): se ha traducido en (5).

Hay una diferencia fundamental entre el evento en *Event-B* y en *Maude*: en el primero se ha usado la implicación y en el segundo la conjunción. La razón es que "⇒" es en realidad un *if* y en *Maude* la implicación actúa siguiendo las leyes de De Morgan (a → b ≡ ¬a v b), lo que alteraba por completo el comportamiento del evento.

5.3.4. Implementación en *Maude*

5.3.4.1. Contexto

```
mod CTX1 is
  sorts Colours Ped_colours .
  subsort Ped_colours < Colours .

--- El conjunto Ped_colours = {red, green} -- No forma parte del
contexto original; se añade para cumplir inv4
  ops red green : -> Ped_colours .

--- El conjunto Colours = {red, yellow, green}
  op yellow : -> Colours .
endm
```

5.3.4.2. Máquina

```
mod MAC1 is
  pr CTX1 . --- Importa el contexto.
  pr BOOL .

  --- Definimos el tipo ColourSet como un conjunto de colores.
  sort ColourSet .
  subsort Colours < ColourSet .
  op empty : -> ColourSet .
  op _,_ : ColourSet ColourSet -> ColourSet [assoc comm id: empty] .
```

```
  eq C:Colours , C:Colours = C:Colours .
--- Operador de pertenencia (∈).
  op _in_ : Colours ColourSet -> Bool  .
  eq C:Colours in C:Colours , CS:ColourSet = true .
  eq C:Colours in CS:ColourSet = false [owise] .
--- El estado del sistema está formado por el valor de los dos
semáforos en un momento determinado y un argumento extra para los
eventos que lo requieran.
  sort State .
--- Coches, peatones, new_value_colours
  op s(_;_;_) : ColourSet Ped_colours ColourSet -> State .
--- Cars_c: Semáforo que regula el paso de los coches --> inv5:
cars_colours ⊆ COLOURS .
--- Peds_c: Semáforo para los peatones.
  vars Cars_c New_value_c : ColourSet .
--- inv4 : peds_colour ∈ {red, green}. Definido como sort en CTXT1
  var Peds_c : Ped_colours .
--- El sistema arranca con el evento Initialisation, que pone los dos
semáforos en rojo.
  op initialisation : ColourSet -> State .
  eq initialisation(New_value_c) = s(red ; red ; New_value_c) .
--- Evento set_peds_red: Pone el semáforo peatonal a false.
  rl [set_peds_red] : s(Cars_c ; Peds_c ; New_value_c) => s(Cars_c ;
red ; New_value_c) .
--- Evento set_peds_green: Pone el semáforo peatonal en verde si el
semáforo de los coches no lo está.
  crl [set_peds_green] : s(Cars_c ; Peds_c ; New_value_c) => s(Cars_c
; green ; New_value_c) if not(green in Cars_c) .
--- Evento set_cars_colours(new_value_c): Cambia el semáforo de los
coches a "new_value_c" con todas estas condiciones:
--- where
---   grd1 : new_value_c ⊆ COLOURS -> Se cumple con la definición de
new_value_c
---   grd2 : green ∈ new_value_c ⇒ Peds_c = red
---   grd_y_r : Cars_c = {yellow} ⇒ new_value_c = {red}
```

```
---   grd_r_ry : Cars_c = {red} ⇒ new_value_c = {red, yellow}
---   grd_ry_g : Cars_c = {red, yellow} ⇒ new_value_c = {green}
---   grd_g_y : Cars_c = {green} ⇒ new_value_c = {yellow}
--- with
---    new_value : new_value = TRUE ⇔ green ∈ new_value_c     -> De
momento no se pondrá por no estar implementado el refinamiento.
  crl [set_cars_colours] : s(Cars_c ; Peds_c ; New_value_c) =>
s(New_value_c ; Peds_c ; New_value_c)
     if (green in New_value_c and Peds_c == red) or
        (Cars_c == yellow and New_value_c == red) or
        (Cars_c == red and New_value_c == red, yellow) or
        (Cars_c == red, yellow and New_value_c == green) or
        (Cars_c == green and New_value_c == yellow) .
endm
```

5.3.5. Salida de la ejecución

Como en el caso anterior, se ha probado a ver si el sistema alcanza el estado erróneo *s(green ; green ; New_value_c)*:

```
Maude> search initialisation(green) =>* s(green ; green ;
New_value_c) .
search in MAC1 : initialisation(green) =>* s(green ; green ;
New_value_c) .

No solution.
states: 3  rewrites: 110 in 0ms cpu (0ms real) (~ rewrites/second)
```

```
Maude> search initialisation(yellow) =>* s(green ; green ;
New_value_c) .
search in MAC1 : initialisation(yellow) =>* s(green ; green ;
New_value_c) .

No solution.
states: 2  rewrites: 73 in 0ms cpu (0ms real) (~ rewrites/second)
```

```
Maude> search initialisation(red,yellow) =>* s(green ; green ;
New_value_c) .
search in MAC1 : initialisation(red,yellow) =>* s(green ; green ;
New_value_c) .

No solution.
states: 4  rewrites: 147 in 0ms cpu (0ms real) (~ rewrites/second)
```

```
Maude> search initialisation(red) =>* s(green ; green ;
New_value_c) .
search in MAC1 : initialisation(red) =>* s(green ; green ;
New_value_c) .

No solution.
states: 2  rewrites: 73 in 0ms cpu (0ms real) (~ rewrites/second)
```

6) Conclusiones y trabajos futuros

Este trabajo final de máster es un primer paso que parece indicar que se pueden traducir programas *Event-B* a *Maude*. Para ello, se sugiere:

- seguir con el proceso de encontrar maneras menos artesanales de traducir el andamiaje matemático que sustenta al primero.
- incluir en el tratamiento de los conjuntos la potencia de *Maude* como metalenguaje.
- tratar la implementación completa del refinamiento, las invariantes cohesionadoras y las obligaciones de demostración.
- buscar maneras de conseguir no fijar de partida el valor de los parámetros de los eventos e incluso de proporcionar argumentos a las reglas que traducen los eventos que no impliquen aumentar de manera artificial el estado del sistema.
- trabajar sobre el mecanismo de visibilidad de módulos en *Maude* para cumplir con las restricciones que presenta una cadena de refinamientos en *Event-B* por lo que respecta a la manipulación de las variables abstractas (si una variable desaparece en un refinamiento, no es posible recuperarla en uno posterior).

Bibliografía

- *All About Maude – A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer y Carolyn Talcott. 2007. Springer.
- *Maude 2.0 Primer. Version 1.0*. Theodore McCombs. Agosto 2003. http://maude.cs.illinois.edu/w/images/6/63/Maude-primer.pdf.
- *Industrial Deployment of System Engineering Methods*. Alexander Romanovsky y Martyn Thomas. 2013. Springer.
- *Rodin user's handbook (v. 2.8)*. Michael Jastram. http://handbook.event-b.org/current/html/ y también http://handbook.event-b.org/current/pdf/rodin-doc.pdf
- Sitio de *Maude* (universidad de Illinois en Urbana-Champaign): http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System
- *Extensiones a la comprobación de satisfacibilidad de restricciones*. TFM de Pablo Viciano Negre (director de tesina: Santiago Escobar). 17 de septiembre de 2012 .
- *System modelling using Event-B*. Neeraj Kumar Singh. 25 de marzo de 2014. http://imps.mcmaster.ca/courses/CAS-734-14/presentations/Event-B_Tutorial.pdf