UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

FormaLex: Mejorando el poder expresivo del lenguaje FL para la detección de defectos normativos

Tesis de Licenciatura en Ciencias de la Computación

María Celeste Gunski - Melisa Gabriela Raiczyk

Director: Fernando Schapachnik

Buenos Aires, Marzo 2016

FORMALEX: MEJORANDO EL PODER EXPRESIVO DEL LENGUAJE FL PARA LA DETECCIÓN DE DEFECTOS NORMATIVOS

Una de las primeras etapas del desarrollo de un programa informático es la especificación, donde se define qué tiene que hacer el programa, indicando los comportamientos permitidos y los prohibidos. Existen técnicas y herramientas que de manera formal analizan estas especificaciones para buscar inconsistencias o huecos, que son complejos de encontrar manualmente.

A pesar de que existen fuertes similitudes entre la especificación de software y la de las normas legales, los intentos de automatizar el análisis de dichas normas, hasta ahora, se han basado en la creación de nuevas herramientas totalmente desde cero. Para poder aprovechar la experiencia en el terreno informático en esta área se desarrolló `FL`, un lenguaje basado en lógica temporal lineal, y FormaLex, que consiste en un grupo de herramientas centrado en el análisis de coherencia de documentos deónticos escritos en `FL`.

Nuestra tesis consiste en extender el lenguaje `FL` y sus herramientas asociadas, agregando funcionalidades para permitir dar soporte a un conjunto más amplio de situaciones expresadas en los textos normativos y adaptar la herramienta para que funcionen todas estas nuevas características. Este trabajo se engloba dentro de uno más amplio, cuyo objetivo es proveer de herramientas prácticas que sirvan de ayuda para detectar errores a la hora de crear normas y reglamentos.

Palabras claves: lógica deóntica, FL, textos normativos, detección errores, FormaLex

AGRADECIMIENTOS

El primer agradecimiento es para nuestro director de Tesis, Fernando, no por formalidad sino porque realmente lo sentimos así: el tiempo que nos has dedicado, las veces que nos has empujado cuando las ganas flaqueaban y principalmente la voluntad de siempre responder en el menor tiempo posible nuestras dudas y consultas. Ha sido un gusto hacer este trabajo junto con vos.

Más allá de la Tesis, a la Facultad y a todos los docentes que nos hemos cruzado durante toda la vida universitaria, sin el trabajo de ellos no hubiéramos adquirido los conocimientos necesarios para poder llegar hasta aquí.

A Juan Pablo, predecesor de este trabajo y a Carlos, sucesor de este trabajo, por sus colaboraciones desinteresadas.

A las personas que han corregido nuestro trabajo: gracias por el tiempo y por las ganas.

A todos los que han estado al lado nuestro todo este tiempo: gracias por ayudarnos a no bajar los brazos y permitirnos llegar hasta aquí, creannos que sin ustedes esto no hubiese sido posible.

Agradecimientos de Melisa

Quiero agradecer en líneas generales a todos los que me han ayudado a transitar este largo camino que en un momento se hizo muy cuesta arriba pero ahora llega a un buen final. Creo que no hay persona que conozca a la que no le haya hablado en algún momento de la Universidad y toda palabra de aliento que me han dado en su momento me ha ayudado a seguir en el camino, por eso: gracias.

A mi familia, inagotable, incondicional: mis papás July y Guille, mis hermanos Ale y Marian, mis tíos y primos. Y a la familia que me regaló la vida: Pau, Ricky, San, Any, Juan y Palo. Pero una especial mención para mi hermano: gracias por ser mi consejero número uno, por decirme mil veces que sí podía y por enseñarme a amar esta profesión!

A mis amigas de siempre: Andre y Vale, nada para reprocharme, siempre comprensión.

A los amigos que me fuí cruzando en el camino: Gasty, Nat, Vale, Jime, Chuny, Romi. Y a todos los que en el trabajo/facultad han escuchado mis historias y me han tirado buena onda para seguir: Zul, Marie, Juan, Alex, Fede, Tom, Mati, Cari, Dami.

A mi compañera aquí presente, Celeste, y compañera es la mejor palabra que pude encontrar: si no nos hubiésemos acompañado como lo hicimos, esto hubiese sido imposible.

Y por último, y no menos importante, a mi compañero de vida, Ety. Por la comprensión y la paciencia, por la cena a las 12 de la noche cuando recién llegaba de la Facultad, por compartir los llantos de los exámenes fallidos y la euforia de cada paso bien dado, por tenerme fé hasta cuando ni yo misma la tenía. Gracias por enseñarme a creer en mi.

Agradecimientos de Celeste

Estos años de arduo trabajo finalmente llegan a su final y por eso quiero dar gracias a todos los que en este período estuvieron al lado mío, formando ese soporte tan importante para no darse por vencido y seguir hasta cumplir el objetivo.

A mi familia, incondicionales siempre, empujando y con palabras de aliento en los momentos necesarios: a mis papás la Beauty y el Richi, por enseñarme con sus propias experiencias y darme todo lo necesario, en lo material pero principalmente en lo personal y humano para llegar hastá acá hoy; a mis hermanos Nata, Carito, Matu, Agus, Lu, Jose, Anita y Cami, a los más grandes por estar siempre conmigo y ayudarme en todo y a los mas chiquitos porque también aprendí mucho de ellos; a Mari, Mati, Javi y Marco por junto a mis hermanos/as darme a los amores de mi vida Valen, Emma, Bauti, Titi, Sari, Cata y la pequeña Mor; a mis abus Clara, Tata, Goguyta y Bocha, pendientes de cada escaloncito superado; a mis primis Dai y Maia, que más que primas son mis amigas desde siempre; a los tíos Paty, Marce y Max; a Ana, por darme a mis pequeñitos amores y ser una buena compañera tanto de papá como de cada uno de nosotros. A todos y cada uno: los amo!

A Dami M. por presentarme esta carrera y por ser mi compañero de estudios y de vida durante casi toda su duración, gracias por esos años compartidos. A su familia, por los momentos compartidos y por ser también una segunda familia durante muchos años.

A aquellos que compartieron conmigo la carrera y/o el trabajo: Gabi, Tom, Dami B., Caro, Nati, Brunis, Flor, Moski, Juan, Sammy, Fede, Tefu, Lean, Ruby. Con una mención especial para Mati R. que en el transcurso de estos años se convirtió en un soporte muy importante para mí, gracias por estar.

A los chicos de teatro, por bancarme y ser compañeros en mi elemento de descarga con sus locuras y sábados de ensayos y diversión.

A mi amiga, mi compañera, la principal responsable de que haya llegado tan lejos, Melisa, Mel, gracias por el empuje, por la compañía, por el soporte constante, lo logramos! Sin tu aguante esto no hubiese sido posible. A Ety, por aguantarnos tantas horas de estudios y TP, gracias por tu paciencia y por hacer feliz a mi coequiper.

A Ale, que a mi vida académica llegó un poco tarde, pero a la personal justo a tiempo. Gracias por el aguante, la compañía y el empuje de todos los días.

Índice general

1.. Introducción 1
  1.1. A qué nos enfrentamos 1
  1.2. La lógica como base de las verificaciones de textos normativos 1
  1.3. Haciendo uso de las herramientas para lograr nuestro objetivo 2
    1.3.1. Componentes de la Teoría Marco 3
      1.3.1.1. Acciones 3
      1.3.1.2. Roles 4
      1.3.1.3. Agentes 4
      1.3.1.4. Contadores 5
      1.3.1.5. Intervalos 5
      1.3.1.6. Timers 6
    1.3.2. Lenguaje de fórmulas 6
  1.4. La herramienta en ejecución 7
    1.4.1. Tecnologías utilizadas 7
    1.4.2. Algunos detalles del model checker 7
      1.4.2.1. Acciones 8
      1.4.2.2. Contadores 8
    1.4.3. Develando el misterio: el paso a paso del proceso 9
  1.5. Mejorando el poder semántico y expresivo de `FL` 10
    1.5.1. Incorporaciones en la nueva versión 10

2.. Mejoras a la Teoría Marco 11
  2.1. Contadores 11
    2.1.1. Motivación 11
    2.1.2. Implementación 11
  2.2. Roles y Subroles 15
    2.2.1. Motivación 15
    2.2.2. Implementación 15

3.. Mejoras realizadas en las Cláusulas 18
  3.1. Permisos Condicionales 18
    3.1.1. Motivación 18
    3.1.2. Implementación 18
  3.2. Permisos Permanentes 19
    3.2.1. Motivación 19
    3.2.2. Implementación 20
  3.3. Permisos Permanentes Condicionales 20
    3.3.1. Motivación 20
      3.3.1.1. Implementación 21
  3.4. Permisos Excepcionales 21
    3.4.1. Motivación 21
    3.4.2. Implementación 22

4.. Sintaxis de `FL` 26
  4.1. Teoría Marco 26
  4.2. Cláusulas 27
  4.3. Traducción a LTL 27

5.. Integración: roles, subroles, excepciones, cómo interpretarlos y no morir en el intento . . . 29

6.. Caso de estudio . . . 32
6.1. Términos y Condiciones del ingreso a la sala VIP de los Aeropuertos con la Tarjeta VIP . . . 32
6.2. Pasando los Términos y Condiciones a `FL` . . . 33
6.3. Verificando nuestro Caso de Estudio . . . 36

7.. Conclusiones y trabajo futuro . . . 37
7.1. Conclusiones . . . 37
7.2. Trabajo futuro . . . 38

1. INTRODUCCIÓN

1.1. A qué nos enfrentamos

Cuando encaramos un sistema de software queremos que el mismo sea correcto y poder verificar que tiene el comportamiento que queremos ante determinadas situaciones.

Para ello primero debemos describir justamente un marco de comportamientos para el mismo. Con los textos normativos se puede trazar una analogía: contienen una serie de reglas y especificaciones que determinan lo que se puede o no se puede hacer y lo que estamos obligados a hacer en determinadas situaciones.

Verificar estos textos normativos se puede entender como análogo a verificar formalmente un sistema de software y de hecho presenta las mismas complejidades y desafíos: corroborar que, efectivamente, el mismo es coherente, no presenta grises y que contempla todos los casos.

¿Cómo hacemos para entender que una norma es coherente? ¿Cómo garantizamos que ciertas reglas no se contradigan con otras que están escritas en la misma norma? Esta verificación es mucho más sencilla pensarla con la colaboración de herramientas.

FormaLex es el resultado de un amplio trabajo que viene atacando este problema, recostándose en el marco de las lógicas ya conocidas y en la utilización de herramientas que fueron creadas para dicho propósito.

1.2. La lógica como base de las verificaciones de textos normativos

Este proyecto está enmarcado en un área conocida como *Automated Legislative Drafting*. Su principal objetivo es brindar asistencia informática al legislador (entendido como cualquier persona responsable de redactar una norma, no sólo como un miembro del parlamento) a la hora de elaborar leyes o reglamentaciones.

El aporte comienza con el desarrollo del lenguaje `FL`, que intenta capturar las estructuras de razonamiento y representación más frecuentes en el ámbito legal. `FL` es un lenguaje de uso específico, en el sentido que, en él, los conceptos legales que usualmente se manipulan en documentos normativos son entidades de primer orden. Por ejemplo: la descripción de una falta a una regulación, el intervalo estipulado para la ejecución de una acción determinada, son algunos conceptos que tienen en `FL` una estructura sintáctica específica para modelarlos.

Este lenguaje fue construido de forma tal de poder generar una traducción eficiente a una de las lógicas temporales más difundidas: Linear Temporal Logic (LTL) [1]. LTL es una lógica modal temporal cuyos modelos son lineales. Es por esta razón que `FL` "hereda" varias de las características de dicha lógica como ser: temporalidad, noción de localidad, etc.

Este lenguaje deóntico `FL`, presentado en [2, 3], se basa en las siguientes premisas:

- Tiene como objetivo principal encontrar *problemas de coherencia* en documentos normativos. Donde coherencia para nosotros significa: no puede haber comportamientos que estén a la vez permitidos y prohibidos para los mismos individuos: "Prohibido matar" y "Permitido matar en defensa propia", como tampoco estar prohibidos y ser obligatorios. Tampoco puede haber obligaciones lisas y llanas y a la vez obligaciones con reparaciones (CTDs, del inglés *Contrary-To-Duty Obligations*), ni puede haber CTDs prohibidas por otras reglas: "Prohibido estacionar en parada de colectivo" y "Si estacionás en una parada, tenés que pagar una mul-

ta", ¿está prohibido o puede hacerlo y pagar?, etc. La lista completa de casos que configuran un problema de coherencia se encuentra en [2].

- Otra de sus premisas de diseño es utilizar como motor de razonamiento a model checkers existentes. Esto se debe a que les tomó varias décadas a los model checkers temporales pasar de prototipos a herramientas capaces de empezar a manejar especificaciones de tamaño realista. Si una herramienta para análisis de normas se construyese desde cero muy probablemente debería pasar por un período similar de tiempo para ir incorporando mejoras, tanto en la teoría como en la implementación, que le permitiesen superar las especificaciones más simples. Además, los model checkers existentes cuentan con una activa comunidad que se encarga de ir mejorándolos.

El lenguaje se divide en dos partes:

- un conjunto de reglas, que son fórmulas con operadores deónticos para expresar permisos (P), prohibiciones (F) y obligaciones (O) siguiendo la noción clásica de que $\mathrm{F}(\varphi) \equiv \mathrm{O}(\neg\varphi)$ y que $\mathrm{P}(\varphi) \equiv \neg F(\varphi)$, por ejemplo:

  "Prohibido hablar dentro de la biblioteca"

  "Permitido hablar dentro de la sala parlante"

  "Es obligatorio devolver los libros antes de retirarse"

- y una *background theory*, o *teoría marco*, que provee mecanismos sencillos para describir la clase de modelos sobre los que predican las reglas. Allí se expresan cosas como precedencia de eventos (e.g., el día ocurre antes de la noche), unicidad (e.g., las personas nacen una sola vez), etc. utilizando construcciones para declarar acciones, roles (que luego dan origen a *agentes* que ejecutan las acciones), intervalos de tiempo, entre otras, que serán explicadas en detalle en 1.3.1.

Ambas partes tienen sus respectivas traducciones: en el caso de las reglas, las mismas se traducen formalmente a fórmulas de la lógica temporal LTL, por ejemplo si algo es obligatorio, entonces debe valer en todo modelo legalmente válido y por ende $\mathrm{O}(\varphi)$ se interpreta como la fórmula LTL $\Box\varphi$, es decir, la fórmula que dice que en todo estado del sistema φ es válida; por otro lado, la teoría marco es traducida automáticamente al lenguaje de especificación del model checker, usualmente mediante autómatas de Büchi.

Los modelos son denominados *trazas* ya que son conjuntos de comportamientos lineales. Cada uno de ellos describe un posible comportamiento *legalmente válido* de los agentes involucrados, es decir, los comportamientos que no cumplen con las reglas son descartados. Es importante observar que esto no impide comportamientos no determinísticos, los cuales son codificados por un conjunto asociado de trazas lineales.

Este lenguaje, combinado con *model checkers* como motores de inferencia para analizar y encontrar automáticamente problemas de coherencia, da como resultado *FormaLex*.

1.3. Haciendo uso de las herramientas para lograr nuestro objetivo

Como mencionamos anteriormente, lo que se desea hacer es tomar un documento normativo y analizarlo para encontrar incoherencias de forma automática. Dicho texto debe ser traducido al lenguaje `FL` y es el input de todo el proceso. No existe por el momento ninguna herramienta de automatización o pseudo automatización para este proceso, sino que se requiere de una o varias personas con cierto conocimiento de lenguajes formales que tomen el texto que se quiere tratar, lo interpreten y codifiquen en `FL`. (Ver sección 7.2 punto 2)

Veamos ahora los componentes de cada aspecto de `FL` con los que se cuentan para escribir estas especificaciones.

1.3.1. Componentes de la Teoría Marco

Como ya dijimos antes, `FL` está compuesto de dos partes. En esta sección describiremos los componentes que son utilizados en la teoría marco, la cual contiene las especificaciones de las características que dan origen a la clase de modelos que utilizaremos para verificar las reglas, es decir, el conjunto de trazas posibles.

1.3.1.1. Acciones

Un modelo lineal, o traza, consiste en una serie de estados relacionados secuencialmente mediante transiciones, en cada uno de estos estados pueden estar occurriendo o no un conjunto de acciones y los cambios de estado de estas acciones son los que van dando origen a las transiciones. Por esta razón, es fundamental contar con la declaración de este componente.

Las acciones en nuestro lenguaje pueden ser realizadas de forma impersonal (no precisamos alguien específico que esté realizando dicha acción, por ejemplo: llover)

```
impersonal action llover
```

o de forma personal (por ejemplo: el vendedor vende).

```
action vender
```

En este último caso, la acción debe ser llevada a cabo por un *agente*, que es una entidad que representa a las personas físicas o jurídicas que realizan las acciones, y que será presentadas en detalle en la sección 1.3.1.3

Un aporte de interés realizado en [5], fue el poder determinar que ciertas acciones pueden ser realizadas de forma sincronizada, por ejemplo, cada vez que se ejecuta la acción *pagar* es porque alguien más está ejecutando la acción *cobrar*. Entonces queremos que ambas estén sincronizadas ya que una no tiene sentido sin que suceda al mismo tiempo la otra.

```
action pagar synchronizes with cobrar
```

Además podemos especificar la cantidad de veces que se puede ejecutar una determinada acción y también quiénes tienen la posibilidad de hacerlo, por ejemplo, podemos decir que la acción *extraerDinero* sólo puede ser realizada por el *titularCuenta*, y solamente se pueden realizar 3 extracciones.

```
action extraerDinero ocurrences 3 only performable by titularCuenta
```

En el ejemplo, *titularCuenta* es lo que denominamos un *rol*. Los roles son formas de clasificar a los agentes en nuestro modelo, precisamente para determinar que ciertas acciones pueden o necesitan ser realizadas por algunos y no por otros. La forma de declarar estos roles se encuentra en la sección 1.3.1.2

También podemos escribir con `FL` acciones que generen un output, un resultado. Para ello debemos definir cuáles son esos posibles resultados que tiene asociados la acción, por ejemplo, si una compra se puede pagar en *efectivo* o con *cheque*, entonces podemos definir que la acción *cobrar* tiene esos posibles valores de salida.

```
action cobrar output values {efectivo, cheque} only performable by empleado
```

Nótese que todas estas posibilidades mencionadas arriba se van potenciando al combinarse entre sí, así como se combinan con los elementos que se desarrollan a continuación, para ir soportando de mejor forma las necesidades expuestas.

1.3.1.2. Roles

Como mencionamos brevemente en la sección anterior, en `FL` el rol nos da la posibilidad de rotular a agentes con ciertas características que tienen (o que no tienen), lo cual nos permite indicar cuáles de ellos pueden ejecutar ciertas acciones.

```
roles hombre, mujer
```

Contamos además con dos aportes adicionales para ajustar algunos comportamientos: a veces no es deseable que dos o más roles se combinen entre sí, es decir, que un mismo agente tenga esos roles al mismo tiempo, lo que queremos entonces es que sean *disjuntos*, la forma de escribirlo en nuestro lenguaje sería:

```
roles hombre, mujer disjoint
```

Con esto nos aseguramos que nunca haya un mismo agente con ambos roles.

Por último, a veces tenemos un conjunto de roles que cubren todas las posibilidades, siguiendo con el ejemplo anterior, si nos estamos refiriendo al ámbito de los sexos, con los roles *hombre* y *mujer* ya estamos cubriendo todas las posibilidades. En `FL` lo escribimos como:

```
roles hombre, mujer cover
```

Esto quiere decir que en nuestro modelo no tendremos nunca un agente que no sea ni *hombre* ni *mujer*, y por ende debe ser obligatoriamente alguno de los dos (o ambos, en caso de no indicar `disjoint`).

En el trabajo [5] se incorporó esta noción de rol a `FL` y en este trabajo hemos profundizado aún más en este tema, incorporando la posibilidad de especializar los roles en subroles (sin límites de niveles de profundidad) tal como se verá más adelante.

1.3.1.3. Agentes

Los agentes no cuentan con una declaración explícita en la teoría marco sino que son generados a partir de la información que le damos al sistema sobre los roles y sus condiciones: disjoint o cover. Su finalidad es ser las entidades del modelo que llevan adelante la acciones. Cada uno de ellos representa en general a una persona física, jurídica o incluso a un conjunto de éstas (por ejemplo, un agente podría ser "la sociedad").

Tal como lo indicamos en la sección anterior, se crearán tantos agentes como combinaciones posibles de roles tengamos. No queremos explayarnos mucho en este tema, se puede ver en detalle en la tesis predecesora [5], pero queremos que quede plasmada la idea de cómo funciona y lo haremos con dos simples ejemplos:

```
1. roles hombre, mujer disjoint
```

Esto genera tres agentes: uno con el rol hombre, otro con el rol mujer y el tercero que corresponde al agente sin roles, el cual debe existir dado que al no indicar `cover` no se tendría cobertura total.

```
2. roles comprador, vendedor
```

Aquí se generan 4 agentes ya que podemos combinar los dos roles porque no son disjuntos: agenteComprador, agenteVendedor, agenteCompradorYVendedor y agenteSinRoles.

Por último, la condición de cover en un conjunto de roles implica que alguno de esos roles debe estar en todos los agentes creados ya que, como cubren todos los posibles valores, el agente debe ser de alguno de ellos. En consecuencia, no es creado en este caso el agenteSinRoles.

1.3.1.4. Contadores

Los contadores, como su nombre lo indica, son variables que se utilizan para contabilizar ocurrencias de ciertas acciones y permiten establecer límites para las mismas. Cuando definimos un contador podemos especificarle:

- Alcance: local o global. Donde local significa que haya un contador por cada agente, y global, que haya un único contador compartido por todos los agentes.
- Valor inicial.
- Conjunto de acciones que lo incrementan y en cuánto lo incrementan.
- Conjunto de acciones que lo decrementan y en cuánto lo decrementan.
- Conjunto de acciones que lo regresan a su valor inicial.
- Conjunto de acciones que lo llevan a un valor específico y cuál es ese valor.
- Condición por la cual el contador es modificado.

Veamos con un ejemplo las opciones antes mencionadas:

```
local counter librosPrestados init value 0
decreases with action devolverLibro with 1,
increases with action retirarLibro with 1,
resets with action inicioCuatrimestre,
sets with action dejaFacultad to value 0
```

En el presente trabajo se agrega además la posibilidad de definir valores máximos y mínimos para cada contador y permitir un comportamiento determinado cuando llega a esas cotas máximas y mínimas, pero eso lo veremos más adelante cuando hablemos de las mejoras incorporadas en esta tesis.

1.3.1.5. Intervalos

Los intervalos nos sirven para incorporar la noción de un período temporal que muchas veces precisamos expresar. Se inician con una acción (o varias), en ese momento decimos que el intervalo está activo, y se finaliza con otra acción (o varias) y en ese momento el intervalo está inactivo.

Claramente las acciones que dan inicio y fin al intervalo no pueden ser ejecutadas en forma simultánea y tienen el comportamiento que esperamos: si se ejecuta una acción de inicio del intervalo, no se puede ejecutar cualquier otra de inicio (incluso ella misma) hasta que no se ejecuta una acción que da fin al intervalo y lo mismo sucede con las acciones de fin.

Algunas cosas que podemos hacer con los intervalos son:

- Definir su alcance: local o global, con la mismo concepto que en los contadores
- Definir el conjunto de acciones que lo inicializan
- Definir el conjunto de acciones que lo finalizan
- Definir la cantidad máxima de veces que puede ocurrir el intervalo
- Definir intervalos infinitos
- Definir si ocurre siempre dentro de otro intervalo

Veamos la sintaxis en FL de lo mencionado anteriormente con algunos ejemplos:

- intervalo local acotado (con acciones de principio y fin)

```
local interval vacaciones defined by actions finClases - inicioClases
```

- intervalo dentro de otro intervalo

```
local interval deViaje defined by actions inicioViaje - finViaje
only occurs in scope vacaciones
```

- intervalo infinito y con cantidad máxima de ocurrencias

```
global interval eternidad defined by actions
  bigBang - infinite occurrences 1
```

Siempre que mencionamos un componente en FL es imposible no remitirnos a cómo se comportan las acciones con el mismo, no en vano dijimos que las acciones son el elemento más importante en el lenguaje.

Para el caso de los intervalos, podemos decir que una acción ocurre dentro de determinado intervalo, por ejemplo, la acción de realizarse la revisación médica para ser admitido en la pileta sólo se podrá realizar mientras la temporada de pileta esté activa:

```
global interval temporadaPileta defined by actions
  inicioVerano - finVerano
action realizarRevisacion only occurs in scope temporadaPileta
```

1.3.1.6. Timers

Tal como se explica en la tesis predecesora [5], para modelar el paso del tiempo usamos timers. El timer está compuesto por una serie de eventos que señalan el paso del tiempo. Por ejemplo, un timer puede ser: Primavera, Verano, Otoño, Invierno.

Notar que no se especifican duraciones. Se trata simplemente de eventos que deben sucederse en el orden indicado y que cumplen el rol de marcas temporales en los modelos sobre los que se trabaja.

Su sintaxis es la siguiente:

```
timer Primavera, Verano, Otoño, Invierno
```

1.3.2. Lenguaje de fórmulas

La otra parte principal de FL es el lenguaje con el cual se escriben las fórmulas, que como ya mencionamos es un lenguaje deóntico, por lo que precisamos expresar los operadores propios de este tipo de lenguajes.

Comencemos por el operador de obligación (en inglés Obligation) O(φ) significa que la fórmula φ se cumple siempre, es decir, en todos los estados de todos los modelos. Si queremos expresar que es oligatorio rendir un examen final, usamos este operador

```
O(rendir_examen)
```

Con este operador y utilizando la negación lógica se pueden definir los operadores de prohibición (Forbidden) F(φ) que significa que la fórmula φ está prohibida, por ejemplo, para expresar que está prohibido fijar carteles, escribimos

```
F(fijar_carteles)
```

y de permiso (Permission) P(φ) para representar que algo está permitido, por ejemplo hablar

```
P(hablar)
```

Se puede considerar a la prohibición como la como la obligación del contrario, por lo cual podemos decir que F(φ) es equivalente a O($\neg\varphi$). Al mismo tiempo decimos que algo está permitido si no está prohibido, entonces la fórmula P(φ) es equivalente a $\neg$ F(φ). Para un mayor detalle sobre las características del lenguaje, consultar [2].

Adicionalmente, contamos con los cuantificadores universal y existencial, la forma de utilizarlos es como sigue:

- FORALL(i:rol; φ) que indica que para todo agente con el rol indicado, se cumple la fórmula φ. Por ejemplo: `FORALL(i:alumno; O(rendir_examen))`
- EXISTS(i:rol; φ) que indica que existe un agente con el rol indicado, para el que se cumple la fórmula φ. Ejemplo: `EXISTS(i:cliente; F(ingresar))`

El detalle de la sintaxis para los cuantificadores se encuentra más adelante, en la sección 4.2

1.4. La herramienta en ejecución

En las secciones anteriores hablamos en detalle de los componentes principales que tenemos en `FL` pero no detallamos nada acerca del resto de las herramientas que componen *FormaLex*.

1.4.1. Tecnologías utilizadas

Puntualmente la herramienta que nosotras extendimos en esta tesis está basada en el lenguaje de programación *Java* y contamos con dos herramientas más de soporte:

- *JavaCC*[1], para poder generar código *Java* a partir del parser de `FL`, que es el que encarga de validar el archivo de input y poblar las entidades desde ese mismo archivo para continuar con el proceso, y
- *Velocity*[2], para escribir templates específicos de input para cada herramienta de *model checker* que querramos utilizar. En este trabajo puntual se usó *NuSMV*[3], pero si se quisiera modificar dicha herramienta lo único que deberíamos hacer es escribir un nuevo template con *Velocity* para el nuevo *model checker*.

Sabiendo ahora la tecnología que usamos veamos el proceso completo para entender cómo se usa la herramienta y qué sucede cuando ponemos en funcionamiento lo anterior.

1.4.2. Algunos detalles del model checker

NuSMV, en ó de tomar autómatas definidos como tales, tiene variables y un lenguaje que permite operar sobre ellas, de manera tal que el autómata se construye implícitamente a partir de un archivo de configuración. Su input son un conjunto de reglas ecuacionales que describen, incluso de manera no determinística, los posibles valores que puede tomar cada variable en el siguiente estado en base a los valores actuales de todas las variables.

[1] https://javacc.java.net
[2] http://velocity.apache.org
[3] http://nusmv.fbk.eu

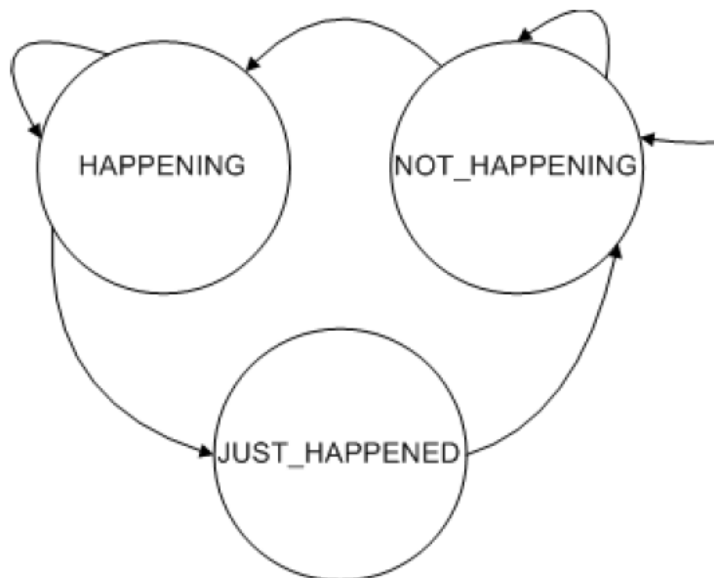
De esta forma, el model checker elige para cada traza los valores en el siguiente estado como alguna de las combinaciones que son válidas de acuerdo a todas las ecuaciones provistas. Eso permite en algunos casos agregar ecuaciones para brindar o restringir comportamiento, sin necesidad de modificar otras.

En este trabajo se agregará y/o modificará el comportamiento de algunas de estas variables, por eso precisamos conocerlas un poco más en detalle, dejaremos fuera de esta sección aquellas variables a las que no le incorporamos modificaciones, si se las quiere conocer más en detalle, se encuentran descriptas en el trabajo predecesor a éste (Ver [5]).

1.4.2.1. Acciones

Una acción se representa con una variable enumerada. En cada transición puede cambiar de valor. Debe respetar las siguientes condiciones:

1. Puede tener sólo alguno de los siguientes valores: NOT_HAPPENING, HAPPENING o JUST_HAPPENED.
2. Su valor inicial es NOT_HAPPENING.
3. Puede cambiar de valor respetando el siguiente orden: NOT_HAPPENING > HAPPENING > JUST_HAPPENED > NOT_HAPPENING> ... y repite el ciclo.
4. Si su valor es JUST_HAPPENED en el siguiente estado tiene que cambiar de valor, en los demás casos puede conservar su valor indefinidamente.
5. Para poder transicionar de un estado a otro utiliza el operador `NEXT()`.



1.4.2.2. Contadores

Un contador se representa con una variable numérica. Debe respetar las siguientes condiciones:

1. El valor inicial de la variable es el indicado en su especificación. En caso de que no se indique este valor, es el default.

2. En cada transición puede cambiar de valor solamente si ocurre algunas de las acciones que lo modifican y estas acciones solamente pueden ocurrir de a una a la vez.

1.4.3. Develando el misterio: el paso a paso del proceso

Tenemos que decir que por cada especificación se hacen varios chequeos, que se describen en detalle en [2]. Uno de ellos es que todas las normas combinadas permitan al menos un comportamiento legal, ya que si no existiese sería porque hay normas que se contradicen entre sí. Para entender cómo funciona este chequeo debemos comprender mejor cómo funcionan los model checkers.

Su input consiste en un autómata A y una fórmula φ, y su misión consiste en verificar si existe algún modelo τ dentro del lenguaje de A, tal que $\tau \not\models \varphi$, es decir, si hay alguna corrida de A que viole φ. Si existe, el model checker nos la devolverá como un contraejemplo de la validez de φ.

En nuestro caso A se genera traduciendo la teoría marco. Las reglas deónticas se traducen a LTL y se unen mediante una conjunción en una fórmula que llamaremos φ'. Queremos ver si existe al menos una corrida de A que satisfaga φ'. Entonces, le proporcionamos al model checker la fórmula $\neg\varphi'$. Si encuentra un contraejemplo, ese contraejemplo es una violación de $\neg\varphi'$ y por ende satisface φ'. Es decir, se trata de un "testigo" de que existe al menos una forma de cumplir todas las reglas bajo la teoría marco y por ende de que no hay contradicciones.

Entendamos ahora cada uno de los pasos y los elementos necesarios para que esto funcione:

1. Archivo de configuración: actualmente se llama *config.ini* y contiene:
   - Ruta del *model checker*.
   - Ruta donde dejará los archivos de salida generados.
   - Ruta del Template de *Velocity*.
   - Ruta del archivo de entrada escrito en `FL`.
2. Archivo principal: como no podría ser de otra forma es el *Main.java* que básicamente se encarga de ejecutar los siguientes pasos:
   a) Parsea el archivo de entrada.
   b) Valida su sintaxis.
   c) De ser correcto, puebla todas las entidades de la Teoría Marco con los datos contenidos en dicho archivo.
   d) Se generan todos los agentes con los roles definidos.
   e) Se invoca a *Velocity* para que genere el autómata para el *model checker* correspondiente. (El autómata se genera en un archivo plano y se guarda en el directorio indicado como directorio de salida en el punto 1).
   f) Las fórmulas correspondientes a las normas escritas en `FL` se expanden y se instancian con los agentes generados en el punto (d) según corresponda.
   g) Las fórmulas se traducen de `FL` a LTL.
   h) Se crea una nueva fórmula que es la conjunción de todas las reglas (es decir, se dejan de lado los permisos) y se niega la misma (esto es importante: el *model checker* recibe la fórmula negada).

i) En este punto ya tenemos el autómata por un lado y la conjunción negada de todas las fórmulas por el otro. Ambos archivos son los que se envían al *model checker*.

j) El output del punto anterior es la validación arrojada por el *model checker* y queda también en el directorio de salida.

k) Nuestra herramienta analiza el resultado arrojado que puede ser que se encontró un comportamiento válido, es decir, no hay incoherencias en el archivo de entrada, o bien que no hay un comportamiento válido.

l) No nos olvidemos de los permisos: hasta ahora validamos las prohibiciones y obligaciones. A continuación se evalúan todos los permisos que tengamos, agregándolos de a uno a la conjunción de fórmulas negadas y repitiendo los pasos siguientes.

A pesar de que pueda resultar un poco extenso, consideramos que era importante detallar el proceso para entender la escencia de lo que está pasando en cada uno de los pasos.

1.5. Mejorando el poder semántico y expresivo de `FL`

Siguiendo el trabajo que presentó J. P. Benedetti en su tesis anterior [5], en el presente trabajo nos focalizamos en ampliar su poder expresivo.

1.5.1. Incorporaciones en la nueva versión

- Teoría Marco
  - Contadores: límites mínimos y máximos individuales por contador.
  - Contadores: posibilidad de establecer la suspensión del incremento/decremento sin impedir la ocurrencia de las acciones que modifican el límite.
  - Roles: Incorporación de la posibilidad de escribir toda una estructura relacional entre roles, respetando el comportamiento que ya tenían respecto a disjunción y cobertura total/parcial del dominio.
- Cláusulas
  - Permisos Condicionales: se incorpora la posibilidad de agregar condiciones para la aplicación de ciertos permisos.
  - Permisos Permanentes: se incorporan para poder expresar permisos que deseamos que se tengan de forma permanente.
  - Permisos Permanentes Condicionales: a los anteriores se le incorpora también la posibilidad de agregarle condiciones.
  - Identificación de reglas: poder identificar reglas con una etiqueta escrita en texto para realizar una identificación rápida. Si bien en sí mismo no parece muy útil, sí lo es cuando hay que identificar una regla para aplicar excepciones.
  - Poder identificar mediante una nueva funcionalidad si un agente tiene determinado rol a través de la palabra clave `belongsTo`.
  - Excepciones a las reglas (Permisos Excepcionales): se expande la sintaxis para expresar permisos que actúen como excepciones a reglas.

En los capítulos siguientes profundizaremos en cada una de estas incorporaciones, su motivación e implementación.

2. MEJORAS A LA TEORÍA MARCO

En esta sección se describen las mejoras al lenguaje realizadas en la Teoría Marco.

2.1. Contadores

2.1.1. Motivación

Hasta el momento los contadores podían tener establecidos valores para los límites máximo y mínimo definidos a nivel global en el template de *Velocity* y el valor era el mismo para todos los contadores declarados.

Además de tener esta restricción, que nos quita la posibilidad trivial de que dos contadores puedan tener diferentes valores límite y adicionalmente aumenta innecesariamente el espacio de estados que debe analizar el model checker, no estaba considerado el comportamiento de detener el incremento/decremento del contador así como tampoco el de no permitir la ocurrencia de una acción cuando se alcanza alguno de los valores límites.

2.1.2. Implementación

A partir de este trabajo contamos, entonces, con los siguientes agregados para los contadores, a saber:

- Posibilidad de definir valores **min value** y **max value** locales a cada contador.
- Control de los límites y suspensión del incremento/decremento pero sin impedir la ocurrencia de las acciones que modifican el valor del mismo.
- O, alternativamente, control de los límites impidiendo la ocurrencia de las acciones que lo modifican.

El primer agregado nos da la opción de definir valores límite que sean locales a cada contador y de manera explícita tener control sobre el rango en el que se permite que se muevan. Si bien el *model checker* trabaja naturalmente con rangos numéricos acotados, el hecho de no poder definir estos límites explícitamente puede generar que se trabaje con rangos innecesariamente grandes.

El segundo agregado mantiene la lógica de corte que se tenía hasta el momento pero introduce la validación de los diferentes valores para los mínimos y/o máximos en cada contador. Asimismo, cuando el contador alcanza uno de sus extremos, si bien se permite que la acción pueda ocurrir, el valor del mismo no se verá modificado.

Un buen ejemplo de la utilidad de esta opción es el modelado de lo que sucede con ciertos automóviles modernos que tienen la posibilidad de establecer una velocidad como máxima que el mismo automóvil no sobrepasa por más que se siga acelerando. Aplicando al concepto de contadores, si se establece el límite máximo en 100 kilómetros por hora, cuando el automóvil efectivamente llega a esa velocidad se le permite al conductor seguir ejecutando la acción *Acelerar* pero sin embargo el contador no se incrementa más allá del límite establecido.

Para mostrar cómo afectan estos agregados a los contadores veamos el siguiente ejemplo:

```
local counter libros_retirados init value 0
min value 0 max value 4
increases with action retirar_libro,
decreases with action devolver_libro
```

El contador representa la cantidad de libros que pueden ser retirados de una biblioteca. Inicialmente un socio no ha retirado ningún libro, por lo que el valor inicial del contador debería ser 0. Cada vez que se realice un retiro, el valor del contador debería incrementarse en uno y al devolver algún libro de los retirados este valor se decrementa.

Como podemos observar, se han utilizado los modificadores: `min value X max value Y`. Cabe mencionar que estas sentencias son optativas y se pueden ingresar ambas, ninguna de ellas o sólo una. Si no se declaran, se cuenta con valores *default* que actualmente son 0 y 20 respectivamente. Como dijimos antes, estos valores responden a la necesidad de darle un corte al *model checker* para buscar trazas viables.

Hasta aquí sólo contamos con los valores topes, pero supongamos que queremos modelar otra situación: dejar de ejecutar una acción ya sea porque se ha excedido el tope mínimo o máximo. En ese caso queremos que mientras no se alcance el valor máximo de libros retirados, el socio puede seguir retirando más libros, pero una vez alcanzado el límite, se evite que continúen los retiros hasta que no se devuelva alguno de los libros prestados anteriormente.

Para definir este nuevo comportamiento, modificamos la declaración del contador de la siguiente manera:

```
local counter libros_retirados init value 0
min value 0 max value 4
increases with action retirar_libro,
decreases with action devolver_libro,
reaching min impedes actions,
reaching max impedes actions
```

Es el mismo concepto que el anterior pero se agrega la "marca" de control de límites que, nuevamente, son valores opcionales. En este caso también se puede controlar solamente el límite superior, solamente el inferior, ambos o ninguno.

Nótese que el hecho de que todas las sentencias sean opcionales da origen a múltiples combinaciones, por ejemplo:

- Se elige controlar el límite superior pero no se establece en la definición del contador un valor máximo. Esto no tiene mucho sentido desde lo semántico pero llegado ese caso, el contador controlará el límite contra el valor máximo default establecido. Mismo caso para el límite inferior.

- Se establece un valor máximo o mínimo, pero no se impide la ejecución de la acción que modifica el contador. En este caso, si un alumno quiere retirar 5 libros podrá hacerlo pero el contador siempre quedará en el valor máximo que en este caso es 4. En este ejemplo particular no es deseable este comportamiento, en breve veremos un caso donde sí es útil representarlo.

- Se establece un valor máximo o mínimo y se impide la ejecución de la acción que modifica el contador. Es el caso que representamos con el código descripto arriba.

Pensemos ahora en la siguiente situación: "dada la alta demanda de cierta categoría de libros la administración de la biblioteca decidió que prestar este tipo de libros equivale a entregar dos libros de otra categoría", entonces precisamos modificar el contador anterior de la siguiente forma:

```
local counter libros_retirados init value 0
min value 0 max value 4
increases with action retirar_libro_categoria_A by 2,
increases with action retirar_libro_categoria_B by 1,
decreases with action devolver_libro_categoria_A by 2,
decreases with action devolver_libro_categoria_B by 1,
reaching min impedes actions,
reaching max impedes actions
```

Puede darse una situación donde nuestro contador esté en 3 entonces la acción *retirar_libro_categoria_A* no estará permitida y de hecho, como estamos controlando los límites no la vamos a poder ejecutar, pero, sin embargo, sí podremos ejecutar *retirar_libro_categoria_B* y este justamente es el comportamiento que queremos modelar. El disponible nos alcanza para llevar ciertos libros pero no otros.

Anteriormente mencionamos que muchas veces no tiene sentido establecer límites y no controlar el corte de la ejecución de las acciones, veamos ahora un ejemplo concreto donde sí queremos este comportamiento. Pensemos en un juego que involucre el uso de un tablero, supongamos que estamos en la posición 10 y la posición a la que queremos llegar para ganar es la número 14, tiramos el dado y obtenemos un 5, sólo podremos avanzar 4 casilleros ya que se nos termina el tablero y si no paramos nos caeremos del mismo. En este caso, modelaremos al contador sin control del tope máximo, esto hará que la acción *lanzar_un_5_en_dados* se pueda ejecutar sin problemas pero que el contador quede con el valor máximo que en este caso es 14 y no 15.

La definición del mismo quedaría como sigue:

```
local counter posicion_en_tablero init value 0
min value 0 max value 14
increases with action lanzar_un_as_en_dados,
increases with action lanzar_un_2_en_dados by 2,
increases with action lanzar_un_3_en_dados by 3,
increases with action lanzar_un_4_en_dados by 4,
increases with action lanzar_un_5_en_dados by 5,
increases with action lanzar_un_6_en_dados by 6
```

Generación del autómata considerando los cambios en los contadores

A los contadores que ya conocíamos (ver sección 3.2.8 de [5]) se agregan las siguientes variables:

1. min value: valor entero positivo que representa el límite mínimo que puede alcanzar el contador. (Por default: 0)

2. max value: valor entero positivo que representa el límite máximo que puede alcanzar el contador. (Por default: 20)

3. reaching min impedes actions: valor booleano que indica que se controla el límite mínimo. (Por default: false)

4. reaching max impedes actions: valor booleano que indica que se controla el límite máximo. (Por default: false)

Para que estas variables tengan el impacto deseado en el análisis del input, es necesario incorporar a la generación del autómata algunas fórmulas adicionales que reflejen lo mencionado anteriormente.

Sean:

- NOMBRE_ACCION y CONTADOR: templates que se instanciarán con las distintas acciones y contadores respectivamente.
- MIN_VALUE y MAX_VALUE: valores definidos con las sentencias

  ```
  min value MIN_VALUE
  max value MAX_VALUE
  ```

  (o 0 y 20 respectivamente si no se definió alguno de ellos)
- X: valor definido luego de la palabra reservada by para modificar el valor del contador. Por default toma el valor 1. Nótese que si está asociado a una acción de decrements será menor a 0 y en caso de corresponder a un increments será mayor a 0.
- REACHING_MIN_IMPEDES_ACTIONS y REACHING_MAX_IMPEDES_ACTIONS: indicadores de impedimento de ejecución de las acciones al llegar a los topes establecidos, definidos por las sentencias:
  reaching min impedes actions y reaching max impedes actions respectivamente.

Por cada acción que incrementa o decrementa el contador:

- La siguiente fórmula se encarga de permitir que los contadores se actualicen en caso de que no se excedan los límites:

  ```
  (next(NOMBRE_ACCIÓN) = JUST_HAPPENED & (CONTADOR + X <= MAX_VALUE)
  & (CONTADOR + X >= MIN_VALUE)
  -> next(CONTADOR) = CONTADOR + X) &
  ```

  Recordemos que el estado JUST_HAPPEND indica que una acción acaba de finalizar.
- Las fórmulas a continuación se encargan de permitir que los contadores tomen el valor del límite máximo o mínimo, respectivamente, en caso de que se exceda el límite correspondiente y no se impida la ejecución de la acción:

  ```
  #if(!REACHING_MAX_IMPEDES_ACTIONS)
  (next(NOMBRE_ACCIÓN) = JUST_HAPPENED & (CONTADOR + X > MAX_VALUE)
  -> next(CONTADOR) = MAX_VALUE ) &
  #end

  #if(!REACHING_MIN_IMPEDES_ACTIONS)
  (next(NOMBRE_ACCIÓN) = JUST_HAPPENED & (CONTADOR + X < MIN_VALUE)
  -> next(CONTADOR) = MIN_VALUE ) &
  #end
  ```
- Las siguientes fórmulas se agregan para impedir la ocurrencia de una acción que incrementa o decrementa al contador si fue definido este comportamiento con las sentencias mencionadas previamente y si se exceden los límites correspondientes:

```
#if(REACHING_MAX_IMPEDES_ACTIONS)
(NOMBRE_ACCIÓN = NOT_HAPPENING & (X > 0 & CONTADOR + X > MAX_VALUE))
 -> next(NOMBRE_ACCIÓN) = NOT_HAPPENING ) &
#end

#if(REACHING_MIN_IMPEDES_ACTIONS)
(NOMBRE_ACCIÓN = NOT_HAPPENING & (X < 0 & CONTADOR + X < MIN_VALUE))
 -> next(NOMBRE_ACCIÓN) = NOT_HAPPENING ) &
#end
```

Nótese que antes de realizar este cambio el autómata tenía un comportamiento no determinístico en este punto (pudiendo pasar al estado `HAPPENING` o quedándose en el estado `NOT_HAPPENING`), pero ahora se obliga a mantenerse, mientras duren las condiciones mencionadas, en el estado `NOT_HAPPENING` eliminando así el indeterminismo.

2.2. Roles y Subroles

2.2.1. Motivación

Cuando recibimos este trabajo lo permitido en cuanto a roles era la posibilidad de tener conjuntos de roles con ciertas características: disjuntos o no, y que representaran una cobertura total o parcial. Pero no había un concepto de especialización de los roles en subroles.

Si bien un agente podía tener al mismo tiempo más de un rol, no se podía establecer una jerarquía de roles, es decir, contábamos con un modelo plano.

Los individuos normalmente cumplimos más de un rol dentro de la sociedad, estos roles pueden estar claramente agrupados en ámbitos independientes o bien tener una cierta jerarquía en su relacionamiento. De esta manera, podemos ser alumnos o docentes en un ámbito académico y a su vez de manera completamente independiente ser comerciantes o clientes en un ámbito comercial. Ahora bien, ¿qué pasa si queremos diferenciar a comerciantes o clientes minoristas y mayoristas?

Hasta ahora, nuestro lenguaje permite hacer esta distinción a un único nivel en el cual tendríamos los roles comercianteMayorista y comercianteMinorista pudiendo ser estos disjuntos o no, pero si existe una regla que deba ser aplicada a todos los comerciantes debe ser escrita una vez por cada uno de estos roles. ¿Y si además los comerciantes mayoristas pueden ser nacionales o internacionales?

Empieza a quedar claro que esta estructura lineal de clasificación no parece ser suficiente. A raíz de esto, surge la necesidad de poder agrupar los roles en una estructura jerárquica, de manera de que las reglas aplicadas a los roles superiores en la jerarquía se apliquen a su vez a los subroles sin necesidad de repetirlas para cada caso en particular.

2.2.2. Implementación

Se agregó entonces el concepto de una estructura de árbol donde la forma de expresar los roles y subroles ahora es la siguiente:

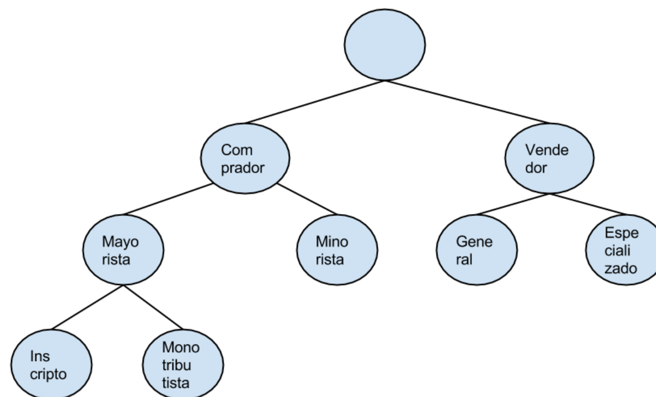
```
roles ROL1, ROL2, ..., ROLN disjoint cover
```

Donde cada ROLi puede tener asociada la misma estructura de manera recursiva expresada entre llaves y así sucesivamente logrando la jerarquía deseada:

```
roles ROL1 {SUBROL1.1, SUBROL1.2 disjoint cover},
       ROL2,..., ROLN disjoint cover
```

Es importante entender que las propiedades de *disjoint* y *cover* se siguen manteniendo en cada nivel del grupo de rol que se escribe. Es decir, si un conjunto de roles son disjuntos entre sí, no implica que lo sean con los roles que están por encima de ellos. Veámoslo con un ejemplo:

```
roles comprador
        {mayorista
          {inscripto, monotributista disjoint},
         minorista disjoint cover},
      vendedor
        {general, especializado disjoint cover}
```



- El rol *comprador* no es disjunto ni cover con el rol que está a su mismo nivel que es *vendedor*. Esto quiere decir que un mismo agente puede ser comprador, vendedor, ambos o ninguno.
- El rol *mayorista* sí es disjunto y cover con su par *minorista.*
- Lo mismo sucede con el par: *general, especializado.*
- El par *inscripto, monotributista* es disjunto pero no es cover.

Siguiendo el ejemplo, nunca tendremos un agente con los roles *mayorista* y *minorista* al mismo tiempo, porque son disjuntos, pero sí podemos combinar *vendedor, general* con la otra rama del árbol porque esos no son disjuntos. Entonces se combinan para el mismo agente los roles: *vendedor, general, comprador, minorista*

Los agentes con sus respectivos roles son:

- *agent_1*: vendedor, comprador, general, mayorista

- *agent_2*: vendedor, comprador, monotributista, general, mayorista
- *agent_3*: vendedor, especializado, comprador, inscripto, mayorista
- *agent_4*: vendedor, especializado
- *agent_5*: comprador, minorista
- *agent_6*: comprador, inscripto, mayorista
- *agent_7*: comprador, mayorista
- *agent_8*: vendedor, especializado, comprador, monotributista, mayorista
- *agent_9*: vendedor, especializado, comprador, mayorista
- *agent_10*: comprador, monotributista, mayorista
- *agent_11*: vendedor, especializado, minorista, comprador
- *agent_12*: vendedor, general
- *agent_13*: vendedor, comprador, inscripto, general, mayorista
- *agent_14*: vendedor, minorista, comprador, general
- *agent_without_role*: no_assigned_role

Tómese un tiempo para ver todas las combinaciones de roles que tienen cada uno de los agentes de este ejemplo (jugando entre los nombres, los covers y disjoint) y verá que llevan el comportamiento que justamente queremos que tengan.

Tal como mencionamos antes, los roles que son disjuntos no se "mezclan", respetando la semántica que le queremos dar.

Nótese que es correcto que aparezcan agentes con los roles *comprador, mayorista* sin ninguna de las especializaciones que le siguen porque al no ser cover, esto puede suceder sin ningún problema. Si *inscripto, monotributista* fuesen cover además de disjoint, entonces sí debería figurar alguno de éstos en los agentes que son compradores mayoristas.

3. MEJORAS REALIZADAS EN LAS CLÁUSULAS

En esta sección se describen las incorporaciones realizadas al lenguaje con respecto a la escritura de las cláusulas.

3.1. Permisos Condicionales

3.1.1. Motivación

Hay ciertos casos en los que es necesario imponer una condición para la aplicación de un permiso en particular. Por ejemplo, deberíamos poder expresar cosas como: "Si el votante se encuentra a más de 500 km de su domicilio, está eximido de votar", para hacer referencia a la excepción de la obligación que tenemos los ciudadanos de votar cuando nos encontramos a más de 500 km de distancia del lugar de votación designado.

Detrás de esta motivación existe todo un marco teórico donde se pueden distinguir distintos tipos de permisos condicionales, nos focalizamos en los primeros tres tipos de los presentados en [6] :

1. Tipo 1: la condición es suficiente pero no necesaria. Consideremos el reglamento de una universidad, que permite que los alumnos mayores de 21 años de edad puedan votar para elegir al Decano de la Facultad. En este caso el significado pretendido no es prohibir la votación si la condición no se cumple (alguna Facultad puede ser que desee permitir la votación a partir de los 18 o incluso sin tener en cuenta la edad), sino para asegurarse de que, al menos, se permita en el caso dado. Es importante entender esto último porque es lo que justamente nos indica que la condición es suficiente pero no es necesaria.

2. Tipo 2: la condición es suficiente y necesaria. En este caso, cuando no se cumple la condición, el permiso se transforma en una prohibición: "Los pasajeros pueden desabrocharse los cinturones cuando la luz roja esté apagada". Sólamente cuando la luz roja esté apagada se puede optar por desabrochar el cinturón, siendo ése el único requisito para ejercer el permiso.

3. Tipo 3: la condición es necesaria pero no suficiente y deben cumplirse condiciones adicionales. Consideremos la siguiente situación: "Los tickets pueden ser comprados siempre y cuando haya asientos disponibles." y "Los tickets deben ser comprados con 48 hs de anticipación." Ambos requerimientos deben cumplirse para que el permiso se haga efectivo, cumplir alguna de las condiciones por separado no es suficiente. Nótese que este permiso es como el tipo 1, sólo que es un poco más complejo ya que deben cumplirse un conjunto de condiciones para habilitarlo.

De los tres tipos de permisos condicionales mencionados, implementaremos el tipo 1, pues la semántica buscada es la de poder expresar que se permite cierta acción bajo determinada condición pero no deseamos hacer que dicha condición sea necesaria para poder habilitar el permiso.

3.1.2. Implementación

Como dijimos antes, implementaremos el permiso condicional de tipo 1 y lo expresaremos como: `P(`α `given that C)` el cual se interpreta como "está permitido que suceda α dada la condición C".

Para el ejemplo visto en la motivación, entonces:

`P(!votar given that INSIDE`[1]`(en_viaje)).`

Conceptualmente, este nuevo tipo de permiso se traduce lógicamente como: "not forbidden (condición **and** fórmula)". Para mayor detalle sobre la definición y traducciones tanto de este tipo de permiso como de los anteriores mencionados en la motivación, ver[6]

A diferencia del permiso convencional, el permiso no es de la fórmula original sino de la conjunción de la fórmula y la condición:

```
<> (agent_1.en_viaje = ACTIVE  & !(agent_1.votar = JUST_HAPPENED))
```

Recordemos que, a diferencia de las reglas, los permisos no necesitan ser válidos en todos los modelos de nuestro sistema sino que alcanza con que exista un modelo en el que se satisfaga la fórmula. En consecuencia lo que se chequea es que

```
exista una traza T tal que
  T |= <> (agent_1.en_viaje = ACTIVE  & !(agent_1.votar = JUST_HAPPENED))
```

Además de darle más expresividad a nuestro lenguaje, este tipo de permisos nos servirá para las excepciones, que veremos en detalle en la sección 3.4.

3.2. Permisos Permanentes

3.2.1. Motivación

Actualmente la construcción del lenguaje P sirve para expresar permisos puntuales, es decir, permisos que se pueden ejercer al menos una vez. Por ejemplo, cuando escribimos: `P(hablar)` estamos indicando que un agente determinado puede realizar la acción *hablar* al menos una vez.

Sin embargo, a veces queremos expresar que una acción tenga la posibilidad de suceder de forma (potencialmente) permanente. Por ejemplo, si en el caso anterior quisiéramos expresar que se puede ejercer el permiso a hablar tantas veces como uno quiera en lugar de poder hacerlo una sóla vez.

Este tipo de permiso se acerca a la noción de los derechos universales que tienen los seres humanos, casos como, por ejemplo, tener derecho a la libertad, a estudiar, etc., que pueden ser traducidos a permisos que deberían estar habilitados todo el tiempo. En estos casos no alcanza con decir que está permitido que suceda, sino que queremos detectar como una violación si en algún momento no está permitido.

Nótese que la noción de estar permitido permanentemente nosotros lo interpretamos como que que una vez que se ejerce el permiso, vuelva a estar disponible. Hay que recordar que estos chequeos no dan una garantía al 100 %, pero si fallan, claramente el permiso no estaba siempre disponible. La garantía al 100 % no se puede dar con nuestro modelo porque no modelamos intención. Entonces, si en algún momento el permiso no está sucediendo no podemos distinguir el hecho de que alguien lo haya impedido del hecho de que el agente simplemente no quisiera ejercerlo en ese momento.

Presentada la necesidad, hemos incorporado un nuevo tipo de permiso, que denominamos "permiso permanente", y que tiene justamente la lógica descripta.

[1] INSIDE(< *nombre_intervalo* >)

indica que dicho intervalo se encuentra activo, es decir, sucedió la acción que da inicio al intervalo y aún no ha sucedido la acción que lo finaliza.

3.2.2. Implementación

La forma de escribir este tipo de permiso es muy similar a la escritura de un permiso convencional. A fin de poder identificarlo, se escribe como `PP(sentencia)`, donde `sentencia` es cualquier sentencia del lenguaje (ver la gramática completa en 4.1).

Veamos un ejemplo concreto: queremos representar el derecho que tienen los ciudadanos a la libertad de circulación, entonces decimos: "Todos los ciudadanos tienen derecho a circular libremente"

Veamos cómo se escribe:

```
#Background
roles ciudadano
action circular_libremente

#Clauses
FORALL(h:ciudadano; PP(h.circular_libremente))
```

Traducido a la lógica, lo escrito anteriormente queda:

```
[] (<> (circular_libremente = JUST_HAPPENED))
```

Vamos a interpretar la fórmula presentada: [] indica que debe cumplirse *siempre* lo que esté a continuación, es decir: <> que se traduce como *es posible*, con lo cual estamos pidiendo que la acción *circular_libremente* se ejecute de manera completa, es decir, llegue al estado JUST_HAPPENED en algún momento futuro.

Si el resultado nos dice que se encontró un comportamiento legal válido entonces nos estará indicando que hay una traza donde el *agent_1* (que es para este ejemplo el único con el rol *ciudadano*) siempre puede circular libremente.

Tengamos en cuenta que esto sigue teniendo la característica de un permiso, porque lo que queremos que se evalúe es que exista una traza donde siempre valga el permiso en cuestión:

```
Existe una traza T tal que
          T |= [] (<> (circular_libremente = JUST_HAPPENED))
```

Aquí notamos claramente la diferencia con las obligaciones, en las cuales pedimos que la sentencia se cumpla en absolutamente todas las trazas.

3.3. Permisos Permanentes Condicionales

3.3.1. Motivación

Bien, ahora contamos con permisos habilitados en forma permanente, pero... ¿qué pasa cuando ese permiso permanente se tiene bajo determinada condición? Claramente en este caso ya no queremos que siempre valga el permiso. Entonces precisamos expresar que a veces el permiso permanente sucede bajo determinadas condiciones. Por suerte ya contamos con una estructura sintáctica que nos permite declarar este tipo de cosas, simplemente lo que vamos a hacer es aplicarla a este nuevo tipo de permisos.

3.3.1.1. Implementación

Tal como mencionamos, ya contamos con una implementación para permisos condicionales, por lo tanto trasladaremos dicha lógica a los permisos permanentes para hacerlos condicionales. Estos se escriben entonces como: `PP(`α` given that C)`, lo cual se interpreta como "está permanentemente permitido que suceda α dada la condición C".

Veamos con un ejemplo concreto cómo se expresa este tipo de permiso: vamos a modelar el permiso para estacionar en el microcentro durante los días feriados. En este caso hay que notar que fuera de los días feriados no necesariamente está prohibido estacionar dado que existen varios otros permisos que también lo autorizan.

```
# Background
roles ciudadano
actions comienzoFeriado, finFeriado
action estacionarEnMicrocentro only perfomable by ciudadano
interval feriado defined by actions comienzoFeriado - finFeriado

# Clauses
FORALL(i:ciudadano; PP(i.estacionarEnMicrocentro given that INSIDE(feriado)))
```

Con este ejemplo estamos diciendo que todos los ciudadanos pueden estacionar en el microcentro en un feriado. Que sea feriado es condición suficiente, pero no necesaria (por ejemplo, pueden existir otros permisos especiales).

Hecha la traducción, la fórmula que se evalúa en este caso es:

```
[] ( <> ( agent_1.estacionarEnMicrocentro = JUST_HAPPENED
            & feriado = ACTIVE) )
```

Esto se lee como"Siempre sucede que en algún momento el *agent_1* estacionó en el microcentro durante un feriado". El objetivo del "siempre sucede" es asegurar que una vez que se ejerce el permiso, éste vuelve a estar disponible para ser ejercido en un futuro.

Y se verifica, dado que es un permiso, que

```
exista una traza T tal que T |=
[] ( <> ( agent_1.estacionarEnMicrocentro = JUST_HAPPENED
            & feriado = ACTIVE) )
```

Y el resultado de esta ejecución es: "Se ha encontrado un comportamiento legal para el permiso", que es precisamente el resultado esperado.

3.4. Permisos Excepcionales

3.4.1. Motivación

Escribir texto con contradicciones es moneda corriente. En realidad, obedecen a un patrón que no sólo es muy común en la escritura sino también en el pensamiento coloquial y del Derecho que consiste en expresar una regla general y luego casos particulares que constituyen sus excepciones.

Un típico ejemplo de esto es la regla: "Prohibido matar" y luego, el permiso: "Se permite matar en defensa propia".

Escribir de forma aislada estas dos sentencias, toma un efecto de contradicción en lo que respecta a la lógica porque estamos diciendo por un lado que no está permitido matar y por otro que sí lo está.

Si bien es una forma natural de concebir la idea, en cuanto a la intención de lo que se desea expresar, muchas veces las cláusulas expresadas de esta manera no hacen más que responder a un uso y costumbre sobre cómo expresar excepciones.

Tal vez es necesario reformular estas sentencias para que no se contradigan. Por ejemplo, si se combinan ambas de la siguiente forma:

- Prohibido matar "excepto que" sea en defensa propia.

Queda claramente expresado que el actuar en defensa propia toma un carácter de excepción sobre la prohibición y esto se describe de manera explícita.

Dada esta situación surge la necesidad de extender el lenguaje dando la posibilidad de escribir reglas y luego permisos que derivan en restricciones a dichas reglas.

Con esta ampliación expresiva, las contradicciones que obtendremos serán las que realmente nos interesan y no aquellas que surjen de la falta de poder expresivo de nuestro lenguaje.

3.4.2. Implementación

La forma de escribir estos permisos excepcionales entonces es:

```
tag: ruleTag RULE
PERMISSION is exception of ruleTag
```

donde ruleTag es un nombre arbitrario que se le asigna a la regla. Luego, el ejemplo mencionado antes queda expresado ahora como:

```
actions matar, inicio_ataque, fin_ataque

interval bajo_ataque defined by actions inicio_ataque - fin_ataque

tag: noMatar F(matar)
P(matar given that INSIDE(bajo_ataque)) is exception of noMatar
```

Dado que la forma de averiguar si un modelo es legal o no consiste en, primero tomar aquellos que cumplen todas las prohibiciones y obligaciones y luego evaluar si se cumplen los permisos, ante la introducción de las excepciones a la reglas nos vemos en la necesidad de reemplazar la regla original (ya sea prohibición u obligación) por una nueva fórmula según el caso:

- Caso obligación: `O(A)`, `P(B) is exception of O(A)` se reemplaza por: `O(A or B), P(B)`. Es decir, es obligatorio que se cumpla A o bien alguna de las excepciones. Veamos un ejemplo para que quede más claro:

  "Es obligatorio votar excepto que el ciudadano se encuentre a más de 500 km"

  ```
  actions votar, iniciar_viaje, finalizar_viaje
  interval en_viaje defined by actions iniciar_viaje - finalizar_viaje

  tag: votoOblig O(votar)
  P(!(votar) given that INSIDE(en_viaje)) is exception of votoOblig
  ```

  donde `en_viaje` representa que el ciudadano está atravezando el intervalo en viaje a un lugar situado a más de 500 km de su lugar de residencia. Luego, la reescritura de la regla incluyendo la excepción queda:

  ```
  O(votar | INSIDE(en_viaje))
  ```

Es decir, es obligatorio que, o bien se vote, o bien el ciudadano no se encuentre en el sitio de votación, lo cual queda expresado por estar dentro del intervalo `en_viaje`. Cabe aclarar que este ejemplo está muy simplificado y está armado de este modo para mostrar la sintaxis nueva.

- Caso prohibición: `F(A), P(B) is exception of F(A)` se reemplaza por: `F(A and not B), P(B)`. Es decir, está prohibido que se cumpla A y a su vez no se cumpla también alguna de sus excepciones. Dicho de otra forma, si ocurre A también debe ocurrir al mismo tiempo alguna de las excepciones. Para este caso, veamos el siguiente ejemplo:

  "Prohibido matar, excepto que sea en defensa propia", queda expresado como

```
actions matar, inicio_ataque, fin_ataque

interval bajo_ataque defined by actions inicio_ataque - fin_ataque

tag: noMatar F(matar)
P(matar given that INSIDE(bajo_ataque)) is exception of noMatar
```

  y al reescribir la regla incluyendo la excepción se obtiene

```
F(matar & !INSIDE(bajo_ataque))
```

  Es decir, no puede darse el caso de matar y al mismo tiempo que no haya sido en defensa propia.

Como mencionamos anteriormente, si la regla original no se reformula, al evaluar el permiso que genera la excepción, éste será detectado como una contradicción y es el comportamiento que deseamos evitar.

Permisos excepcionales cuantificados

En los ejemplos vistos arriba se puede notar que las reglas están escritas en forma directa, pero normalmente estas reglas no son "universales" sino que se aplican a los que denominamos *agentes*, que son los que ejecutan las acciones que están siendo reguladas por estas sentencias. Por esta razón, lo más usual es escribir estas reglas utilizando cuantificadores sobre los roles que pueden tomar los agentes.

Para los ejemplos vistos antes, se tiene entonces:

- "Todos los ciudadanos deben votar, excepto aquellos que se encuentren a más de 500 km"

```
actions votar, iniciar_viaje, finalizar_viaje

local interval en_viaje defined by actions
  iniciar_viaje - finalizar_viaje

tag: votoOblig FORALL(i:ciudadano; O(i.votar))
FORALL(h:ciudadano;
  P(!(h.votar) given that INSIDE(h.en_viaje)) is exception of votoOblig)
```

  Supongamos que los agentes *agente_1* y *agente_2* tienen el rol *ciudadano*, al reescribir la regla para incluir la excepción se obtiene:

```
O(agente_1.votar | agente_1.en_viaje) &
   O(agente_2.votar | agente_2.en_viaje)
```

Y respecto a los permisos:

```
P(!(agente_1.votar) given that INSIDE(agente_1.en_viaje)) &
P(!(agente_2.votar) given that INSIDE(agente_2.en_viaje))
```

- "Está prohibido para todos los ciudadanos matar, excepto que sea en defensa propia", queda expresado como

```
tag: noMatar FORALL(i:ciudadano; F(i.matar))
FORALL(i:ciudadano;
  P(i.matar given that INSIDE(i.bajo_ataque)) is exception of noMatar)
```

Nuevamente supongamos que los agentes *agente_1* y *agente_2* tienen el rol *ciudadano*, al reescribir la regla para incluir la excepción se obtiene:

```
F(agente_1.matar & !INSIDE(agente_1.bajo_ataque) ) &
F(agente_2.matar & !INSIDE(agente_2.bajo_ataque) )
```

Y respecto a los permisos:

```
P(!(agente_1.matar) given that INSIDE(agente_1.bajo_ataque)) &
P(!(agente_2.matar) given that INSIDE(agente_2.bajo_ataque))
```

Nótese que la traducción es igual a la que vimos antes cuando no estaban involucrados los cuantificadores porque en ambos ejemplos estamos hablando del cuantificador universal. Veamos ahora qué pasa cuando los cuantificadores están combinados.

- Cuantificador universal en la regla y existencial en la excepción:

```
tag: rule FORALL (i:rol; O(i.A))
EXIST (j:rol; P(j.B) is exception of rule)
```

En este caso la obligación sigue manteniendo la misma traducción que antes, pero sólamente uno de los agentes tiene permitida la excepción, por lo tanto:

```
O(agente_1.A | agente_1.B) &
O(agente_2.A | agente_2.B)
```

Y respecto a los permisos:

```
P(agente_1.B) | P(agente_2.B)
```

Aquí es donde se nota la combinación de los cuantificadores, el permiso no es para todos los agentes sino para alguno/s de ellos.

- El caso inverso a lo visto recién sería:

```
tag: rule EXIST(i:rol; O(i.A))
FORALL(i:rol; P(i.B) is exception of rule)
```

Aquí la regla está habilitada para algún agente pero la excepción para todos los de dicho rol, y se traduce como:

```
O(agente_1.A | agente_1.B) | O(agente_2.A | agente_2.B)
```

Y respecto a los permisos:

```
P(agente_1.B) & P(agente_2.B)
```

Tipos de excepciones

Ya vimos que las excepciones son permisos. Esos permisos pueden estar relacionados con permitir una determinada acción aún cuando existe otra regla que la prohíba, como el ejemplo de "Prohibido matar" que se mencionó mas arriba.

Las otras excepciones tiene que ver con los roles que puede tomar un agente, por ejemplo: "Todos los clientes tienen prohibido ingresar con alimentos" y "Todos los clientes que son celíacos tienen permitido ingresar con alimentos". Ser cliente celíaco, en este caso, es la combinación de roles que genera la excepción a la regla.

Para estos casos, precisamos escribir en una misma sentencia que un agente tiene ambos roles en simultáneo. Con la sintaxis anterior esto no era posible, ya que, por ejemplo para el caso de arriba podíamos escribir estas dos sentencias:

```
FORALL(i:cliente; F(i.ingresarConAlimentos))
FORALL(i:celiaco; P(i.ingresarConAlimentos))
```

pero no expresan lo que queremos decir, la primer sentencia sí expresa la prohibición, pero la segunda no es clara para la excepción, ya que lo que estamos expresando es "Todos los celíacos tienen permitido ingresar con alimentos" pero lo que realmente necesitamos es que solamente los *clientes* que además son *celíacos* cuenten con ese permiso.

Surje la necesidad entonces de tener esta posibilidad y para ello se agrega la expresión: `belongsTo`, cuya sintaxis es la siguiente:

```
variableName.belongsTo:role
```

Nótese que esto es una sentencia más, por lo que nos aumenta notablemente el poder expresivo, podemos usarlo para expresar el permiso del ejemplo mencionado de la siguiente manera:

```
tag: noIngresarAlimentos FORALL(i:cliente; F(i.ingresarConAlimentos) )
FORALL(i:cliente; P(i.belongsTo:celiaco & i.ingresarConAlimentos)
  is exception of noIngresarAlimentos)
```

O por ejemplo, como parte de un permiso condicional:

```
tag: noIngresarAlimentos FORALL(i:cliente; F(i.ingresarConAlimentos) )
FORALL(i:cliente; P(i.ingresarConAlimentos given that i.belongsTo:celiaco)
  is exception of noIngresarAlimentos)
```

4. SINTAXIS DE FL

Finalmente, luego de todos los cambios y extensiones mencionadas, detallamos a continuación toda la sintaxis posible de nuestro sistema. Aquellas partes que figuran con signo de interrogación son opcionales y las que partes que están marcadas en color rojo son las que se han agregado y/o modificado como consecuencia del presente trabajo.

4.1. Teoría Marco

De la teoría marco sólo expondremos lo que ha sido modificado, la parte de intervalos, timers y acciones funciona exactamente igual que lo hacía antes de extender el trabajo.

roles	:=	roles (estructuraRol)$^{+}$
estructuraRol	:=	nombre_rol_1 ({estructuraRol})*
		(, estructuraRol)* (disjoint)? (cover)?
contadores	:=	(local | global) counter nombre_contador
		(init value valor_inicio)?
		(min value valor_minimo)?
		(max value valor_maximo)?
		((increases with action nombre_accion_incremental)
		((by valor_a_incrementar,) | ,))*
		((decreases with action nombre_accion_decremental)
		((by valor_a_decrementar,) | ,))*
		(reaching min impedes actions,)?
		(reaching max impedes actions)?
		(resets with action nombre_accion_reset)?
		(sets with action nombre_accion_set to value valor_a_setear)?

4.2. Cláusulas

cláusula := P(sentencia) |
PP(sentencia) |
(identificador)? O(sentencia) (reparación)? |
(identificador)? F(sentencia) (reparación)? |
(identificador)? EXISTS (var (: rol)? ; O(sentencia) (reparación)? |
F(sentencia) (reparación)?) |
EXISTS (var (: rol)? ; P(sentencia (given that sentencia_1)?)
(is exception of label_de_la_regla)?) |
EXISTS (var (: rol)? ; PP(sentencia (given that sentencia_1)?))) |
(identificador)? FORALL (var (: rol)? ; (O(sentencia) (reparación)? |
F(sentencia) (reparación)?)) |
FORALL (var (: rol)? ; P(sentencia_1 (given that sentencia_2)?)
(is exception of label_de_la_regla)?)
FORALL (var (: rol)? ; PP(sentencia_1 (given that sentencia_2)?)))

sentencia := terminal |
! (sentencia) |
<>(_nombre_del_intervalo)? sentencia |
[ ] (_nombre_del_intervalo)? sentencia |
sentencia and sentencia |
sentencia or sentencia |
sentencia − > sentencia |
EXISTS (var (: rol)? ; sentencia) |
FORALL (var (: rol)? ; sentencia)

terminal := (var.)?acción |
(var.)?acción results in un_valor |
evento_de_un_timer |
INSIDE ((var.)?nombre_de_intervalo) |
(var.)?nombre_de_contador (> | >= | = | < | <=)
(un_número | otro_contador)

reparación := repaired by sentencia

identificador := tag: label_de_la_regla

pertenencia := var.belongsTo (: rol)

and := &

or := |

4.3. Traducción a LTL

Estas nuevas fórmulas generadas deben ser traducidas desde el formato FL original al de LTL. Daremos una reducida definición formal y algunos ejemplos para clarificar.

Definimos la función Tr que traduce de FL a LTL considerando a $\mathcal{M}$ como la clase de modelos lineales, φ y ρ como fórmulas FL, i como nombre de un intervalo, a x como

variable libre y a R como el nombre de un rol. La traducción anterior era:

$$
\begin{array}{rcl}
Tr(\Diamond i \varphi) & = & i = \text{ ACTIVE } \rightarrow (i = \text{ACTIVE U } Tr(\varphi)) \\
Tr(F(\varphi)) & = & \Box \neg Tr(\varphi) \\
Tr(F(\varphi) \text{ repaired by } \rho) & = & \Box (Tr(\varphi) \rightarrow Tr(\rho)) \\
Tr(O(\varphi)) & = & \Box Tr(\varphi) \\
Tr(O(\varphi) \text{ repaired by } \rho) & = & \Box (\neg Tr(\varphi) \rightarrow Tr(\rho)) \\
Tr(O(\Diamond \varphi)) & = & \Diamond Tr(\varphi) \\
Tr(P(\varphi)) & = & \exists\, \tau \in \mathcal{M} \text{ tal que } \tau \models \Diamond Tr(\varphi) \\
Tr(\varphi \wedge \rho) & = & Tr(\varphi) \wedge Tr(\rho) \\
Tr(\varphi \vee \rho) & = & Tr(\varphi) \vee Tr(\rho) \\
Tr(\varphi \rightarrow \rho) & = & Tr(\varphi) \rightarrow Tr(\rho) \\
Tr(\text{una_accion}) & = & \text{una_accion} = \text{JUST_HAPPENED} \\
Tr(\text{INSIDE}(i)) & = & i = \text{ACTIVE} \\
Tr(\text{FORALL}(x : R; \varphi)) & = & \bigwedge_{a \in R} Tr(\varphi[x \leftarrow a]) \\
Tr(\text{EXISTS}(x : R; \varphi)) & = & \bigvee_{a \in R} Tr(\varphi[x \leftarrow a])
\end{array}
$$

Incorporamos:

$$
\begin{array}{rcl}
Tr(P(\varphi \text{ given that } \rho)) & = & Tr(P(\rho \wedge \varphi)) \\
Tr(PP(\varphi)) & = & \exists\, \tau \in \mathcal{M} \text{ tal que } \tau \models \Box \Diamond Tr(\varphi) \\
Tr(PP(\varphi \text{ given that } \rho)) & = & \exists\, \tau \in \mathcal{M} \text{ tal que } \tau \models \Box \Diamond Tr(\rho \wedge \varphi)
\end{array}
$$

Y modificamos la traducción de Obligation y Forbidden para que en el caso de encontrar construcciones con excepciones:

$$
\begin{array}{r}
P(\varphi_1) \text{ is exception of } O(\rho) \\
\ldots \\
P(\varphi_n) \text{ is exception of } O(\rho) \\
P(\psi_1) \text{ is exception of } F(\rho) \\
\ldots \\
P(\psi_m) \text{ is exception of } F(\rho)
\end{array}
$$

las traducciones queden expresadas respectivamente como:

$$
\begin{array}{rcl}
Tr(O(\rho)) & = & Tr(O(\rho \vee \varphi_1 \vee \ldots \vee \varphi_n)) \\
Tr(F(\rho)) & = & Tr(F(\rho \wedge \neg \psi_1 \wedge \ldots \wedge \neg \psi_m))
\end{array}
$$

5. INTEGRACIÓN: ROLES, SUBROLES, EXCEPCIONES, CÓMO INTERPRETARLOS Y NO MORIR EN EL INTENTO

Para mostrar la integración de los roles con sus subroles correspondientes vamos a utilizar un ejemplo con bastantes datos y aunque el ejemplo no sea muy realista, nos sirve para dejar en claro las interacciones cuando tenemos una estructura con muchas relaciones. Además, para hacerlo jugar con lo que ya hemos explicado en pasos previos, vamos a ver cómo se comportan con las excepciones y permisos condicionales y qué pasa cuando ahora hablamos de un rol de primer nivel y un subrol cualquiera.

Tenemos entonces el siguiente ejemplo:

- Respecto a los roles, vamos a trabajar con el concepto de ciudadano, dividiéndolos entre *civiles* y *militares* y sabiendo que además los ciudadanos civiles pueden ser *menores* o *mayores* de edad.
- Respecto a las acciones que pueden realizar los agentes de este ejemplo, las mismas son: votar, iniciar_viaje y finalizar_viaje.
- Nótese que las acciones iniciar y finalizar viaje están siendo utilizadas en el intervalo *en_viaje*.
- Respecto a las cláusulas que queremos evaluar, tienen que ver con cumplir nuestra obligación ciudadana de votar, y la única regla que tenemos en este ejemplo es justamente: *Todo ciudadano está obligado a votar.*
  Pero además, contamos con dos excepciones, la primera tiene que ver con el permiso de no votar si uno se encuentra de viaje a más de 500 km de distancia de su domicilio (que en este ejemplo está representado con el intervalo *en_viaje*).
  La segunda excepción que se nos presenta en el ejemplo no está relacionada con ejecutar una acción sino con tener la característica de ser menor de edad, representado con el rol *menor*. Nótese que cuando hablamos de dicha característica no nos interesan sus especializaciones (si las tuviese), sino que la excepción se cumple para cualquier *menor*.

Presentado el caso, veremos a continuación qué combinación de roles genera el sistema para los agentes y quiénes pueden ser exceptuados de su obligación como ciudadanos. [1]

```
#Background

roles ciudadano { civil { menor,
                          mayor disjoint cover },
                  militar disjoint cover },
      noCiudadano disjoint cover

actions votar, iniciar_viaje, finalizar_viaje

local interval en_viaje defined by actions iniciar_viaje - finalizar_viaje
```

[1] Este ejemplo sería más realista si la obligación de votar fuese durante un periodo de elecciones, pero eso haría que las fórmulas se volviesen aún más complejas sin aportar a lo que queremos mostrar en este caso particular que es la excepción.

```
#Clauses
tag: votoOblig FORALL(i:ciudadano; O(i.votar))
FORALL(h:ciudadano; P(!(h.votar) given that INSIDE(h.en_viaje))
  is exception of votoOblig)
FORALL(j:ciudadano; P(!(j.votar) given that j.belongsTo:menor)
  is exception of votoOblig)
```

Los agentes que la herramienta crea para este ejemplo son:

- *agent_1*: ciudadano, militar
- *agent_2*: ciudadano, menor, civil
- *agent_3*: noCiudadano
- *agent_4*: ciudadano, mayor, civil

Nótese que la especialización *menor, mayor* es cover, esto quiere decir que en nuestro contexto siempre que un agente sea *civil* va a ser especializado en algunos de los dos subroles porque ellos representan todas las posibles especializaciones para dicho rol. No puede existir un *ciudadano civil* que no sea *mayor* o *menor* (reiteramos, para nuestro ejemplo particular). Si no fuese cover, entonces sí tendríamos agentes que tengan dicho rol sin ninguna especialización.

Por otro lado, el conjunto de especializaciones *menor, mayor* son disjuntos con lo cual dichos roles no son combinables. Esto mismo sucede con los subroles de primer nivel de *ciudadano*: *civil* y *militar*, no se combinan porque son disjuntos.

Ahora veamos qué pasa con las fórmulas que tiene nuestro ejemplo: nosotros estamos expresando la obligación de cumplir con el voto que tienen todos los ciudadanos, pero por otro lado, estamos diciendo que si hay un ciudadano que no debería votar es porque ocurrió alguna de las dos excepciones que declaramos a la regla: o bien el ciudadano viajó o bien el ciudadano es menor de edad y no tiene entonces la obligación de votar.

Todo esto, traducido se escribe como:

```
#Reglas
( [] ( agent_2.votar = JUST_HAPPENED ) |
<> !( agent_2.votar = JUST_HAPPENED )  |
<> ( agent_2.en_viaje = ACTIVE  & !( agent_2.votar = JUST_HAPPENED ) ) ) &

( [] ( agent_4.votar = JUST_HAPPENED ) |
<> ( agent_4.en_viaje = ACTIVE  & !( agent_4.votar = JUST_HAPPENED ) ) ) &

( [] ( agent_1.votar = JUST_HAPPENED ) |
<> ( agent_1.en_viaje = ACTIVE  & !( agent_1.votar = JUST_HAPPENED ) ) )

#Permisos
<> ( agent_2.en_viaje = ACTIVE  & !( agent_2.votar = JUST_HAPPENED ) ) &
<> ( agent_4.en_viaje = ACTIVE  & !( agent_4.votar = JUST_HAPPENED ) ) &
<> ( agent_1.en_viaje = ACTIVE  & !( agent_1.votar = JUST_HAPPENED ) )
```

```
<> ( !( agent_2.votar = JUST_HAPPENED ) )
```

¡No permita que las fórmulas lo mareen! Vamos a darle el significado y entender qué pasa con cada una de ellas. En primer lugar vemos que el agente 3 no figura en ninguna de las fórmulas, si vemos la regla dice: que para todo "ciudadano", detengamosnos aquí y vayamos a ver qué rol tiene el agente 3:

- *agent_3*: noCiudadano

¡Bien! Esto explica por qué este agente no forma parte de las fórmulas: no es ciudadano. Observemos ahora las fórmulas, está claro que en cada una de ellas varía el agente sobre el que se evalúa, veamos entonces más en detalle su significado teniendo en cuenta que todos los agentes involucrados son ciudadanos:

O el agente_X votó, o bien el agente_X estaba de viaje, entonces tiene permitido no votar o bien si el agente_X es menor (rol) entonces tiene permitido no votar.

Recordemos que las acciones pueden ser realizadas, a priori, por todos los agentes, por lo tanto las primeras dos partes de las fórmulas son comunes a todos ellos, pero ¿qué pasa con la parte que hace referencia al rol, la que pregunta si el agente_X es menor? Allí nos encontramos con que los agentes: 1 y 4 tienen una fórmula menos mientras que el agente 2 cuenta con una fórmula que indica "permitido no votar", dicha fórmula está condicionada a la pregunta de si *menor* es parte de sus roles. Nótese que en este momento ya contamos con la respuesta a dicha pregunta. La sentencia "belongsTo" es la que justamente resuelve si el agente cumple o no con cierto rol.

Recordemos qué roles tienen los agentes 1 y 4:

- *agent_1*: ciudadano, militar
- *agent_4*: ciudadano, mayor, civil

Ninguno de estos dos agentes son menores, por lo tanto es correcto que no tengan la fórmula instanciada dado que no puede estar exceptuado a su obligación porque no tiene el rol que lo habilita a no votar. El agente 2 sí tiene el rol menor, por lo tanto tiene la fórmula y accede al permiso de no votar.

Lo que hemos visto con este ejemplo es, por un lado, la conformación de los roles de un agente dada la nueva posibilidad de escribir ya no un conjunto de roles sin relación, sino una estructura completa de roles y relaciones con subroles. Por otro lado, vemos que si queremos hablar de un subrol en particular, no tenemos que mencionar ninguna relación con sus roles padres, simplemente podemos expresar fórmulas haciendo mención a dicho rol (no precisamos hablar de todas las combinaciones en el ejemplo anterior, simplemente dijimos: para todo ciudadano menor sucede X).

Esto nos da un amplio poder de expresión, ya que si bien a nivel agentes tenemos todas las relaciones de los roles, subroles si son disjuntos, cover, etc, cuando tenemos que hacer mención de esos roles lo hacemos libremente expresando lo que realmente precisamos.

6. CASO DE ESTUDIO

A continuación presentaremos un caso de estudio para dejar en evidencia cómo se aplica todo lo que hemos mencionado hasta aquí, con un caso concreto.

6.1. Términos y Condiciones del ingreso a la sala VIP de los Aeropuertos con la Tarjeta VIP

1. Sobre la admisión en las salas y costos
    - *a*) La Tarjeta VIP es intransferible y sólo es válida hasta la fecha de caducidad. Ninguna otra persona, aparte del titular designado, podrá usar la Tarjeta VIP.
    - *b*) Presentando la Tarjeta VIP el titular tendrá acceso a todas las salas de todos los aeropuertos alrededor del mundo. Las salas ubicadas en el Aeropuerto de Dallas, USA y de Sorrento, Italia sólo podrán ser accedidas por Tarjetas VIP Select.
    - *c*) Los miembros invitados que no tengan Tarjeta VIP podrán ingresar a las salas abonando la tarifa vigente al momento de ingresar a la misma. Si es acompañante de un titular de Tarjeta VIP Select abonará el 75 % de la tarifa.
    - *d*) Los acompañantes menores abonarán el 50 % de la tarifa vigente de la sala correspondiente.
2. Sobre el uso de las instalaciones
    - *a*) Cada sala ofrece bebidas y comidas que están totalmente incluidas en el pase y su consumo es ilimitado.
    - *b*) No se podrá ingresar a las salas con ningún tipo de alimentos y/o bebidas. Si uds. o alguno de sus invitados posee alguna afección de salud relacionada con la alimentación tal como diabetes o celiaquía se permitirá su acceso.
3. Sobre la cantidad de ingresos permitidos
    - *a*) Cada titular de la Tarjeta VIP podrá ingresar hasta 5 (cinco) veces por año. Tomándose el inicio del año el primer uso de la tarjeta y no el año calendario.
4. Sobre el Programa de Recompensas VIP Member Rewards
    - *a*) Cada vez que utilice una sala del programa sumará 1 punto.
    - *b*) Cada vez que un nuevo socio lo mencione como referente sumará 2 puntos. Si el socio nuevo es de la membresía Select, sumará 5 puntos.
    - *c*) El tope máximo de puntos de la membresía es de 20 por año.
    - *d*) Los puntos se vencen en su totalidad al año desde el inicio de la membresía, si los mismos no son canjeados, se perderán.

6.2. Pasando los Términos y Condiciones a FL

A continuación vamos a codificar algunos de los términos enumerados anteriormente en el lenguaje de FL a fin de ejemplificar las mejoras que se agregaron. Para mayor claridad en los ejemplos, dividiremos los términos en dos inputs de acuerdo al tema que se esté tratando: admisión, cantidad de ingresos en las salas y programa de recompensas por un lado, y uso de las instalaciones por el otro.

Comencemos por los términos de admisión, cantidad de ingresos a las salas y programa de recompensas:

Para las reglas $1c$ y $1d$ necesitamos considerar agentes que tengan roles de socio y no socio o acompañante, y a su vez gracias a la incorporación de subroles podemos identificar los distintos tipos de socios y no socios de acuerdo a su membresía o relación con un miembro:

```
roles ciudadano {
cliente {
    socioTitular {premium, select disjoint cover},
    relacionadoASocio {
            relacionadoASocioPremium {menor, mayor disjoint cover},
            relacionadoASocioSelect {menor, mayor disjoint cover}
            disjoint cover}
disjoint cover},
noCliente disjoint cover}
```

Definimos las acciones que podrán ser realizadas por estos agentes y algunos intervalos definidos por las mismas:

```
impersonal action pasaUnAnio
impersonal action nuevoSocioRecomendadoPremium
impersonal action nuevoSocioRecomendadoSelect
actions activaPase,primerUsoTarjeta only performable by socioTitular

local interval tarjetaVigente defined by actions activaPase - pasaUnAnio

actions ingresarASala, ingresarASalaDallas, ingresarASalaSorrento
action abonarTarifaCompleta only performable by relacionadoASocioPremium
action abonarTarifaAl75 only performable by relacionadoASocioSelect
action abonarTarifaAl50 only performable by menor
```

Luego generamos un contador para validar la condición $3a$, específicamente para controlar la cantidad de ingresos realizados por un socio:

```
local counter pasesRestantes init value 5
min value 0
decreases with action ingresarASala by 1,
decreases with action ingresarASalaDallas by 1,
decreases with action ingresarASalaSorrento by 1,
reaching min impedes actions
resets with action pasaUnAnio
```

Y tendremos otro contador para cuantificar la cantidad de puntos del programa de recompensas:

```
local counter puntosMembresia init value 0
```

```
min value 0 max value 20
increases with action nuevoSocioRecomendadoPremium by 2,
increases with action nuevoSocioRecomendadoSelect by 5,
increases with action ingresarASala by 1,
resets with action pasaUnAnio
```

Ahora bien, pasemos a los términos que vamos a analizar en esta parte:

- Los *noCliente* nunca podrá ingresar a las salas aún queriendo pagar (implícito en el punto 1*b*):

  ```
  FORALL(i:noCliente; F(i.ingresarASala))
  ```

- Solamente los *Socios Select* podrán ingresar a las salas de los aeropuertos de Dallas y Sorrento, para ellos diremos que todos tienen prohibido el ingreso a dichas salas, excepto los *Socios Select* (también del punto 1*b*):

  ```
  tag: prohibidoIngresoSalaDallas FORALL(i:ciudadano;
    F(i.ingresarASalaDallas))

  FORALL(i:select; PP(i.ingresarASalaDallas)) is exception of
    prohibidoIngresoSalaDallas

  tag: prohibidoIngresoSalaSorrento FORALL(i:ciudadano;
    F(i.ingresarASalaSorrento))

  FORALL(i:select; PP(i.ingresarASalaSorrento)) is exception of
    prohibidoIngresoSalaSorrento
  ```

  Nótese que aquí contamos con dos acciones diferentes para el ingreso a las salas *Select*, la realidad es que queríamos escribir esto de otra forma, porque tuvimos que escribir dos reglas particulares para poder expresarlo cuando en realidad debería poder decirse con la regla *ingresarASala* que ya teníamos. Más adelante, en la sección de Trabajos Futuros detallaremos más este punto.

- Para los puntos 1*b* y 3*a* escribimos una regla y un permiso permanente para expresar la posibilidad de ingresar a las salas de forma permanente mientras su tarjeta esté vigente.

  ```
  FORALL(i:socioTitular; PP(i.ingresarASala given that
  INSIDE(i.tarjetaVigente)))
  FORALL(i:socioTitular; F(i.pasesRestantes = 0 & i.ingresarASala))
  ```

- Los miembros invitados que no tengan Tarjeta VIP podrán ingresar a las salas abonando la tarifa vigente al momento de ingresar a la misma. Si es acompañante de un titular de Tarjeta VIP Select abonará el 75 % de la tarifa (Punto 1*c*).

  Si bien el texto no lo dice explícitamente, los que abonan estas tarifas son los mayores, porque como sí sabemos, los menores abonan el 50 %, regla que veremos luego de éstas.

```
FORALL(i:relacionadoASocioPremium;
  O((i.belongsTo:mayor & i.ingresarASala) -> i.abonarTarifaCompleta))

FORALL(i:relacionadoASocioSelect;
  O((i.belongsTo:mayor & i.ingresarASala) -> i.abonarTarifaAl75))
```

- Los acompañantes menores abonarán el 50 % de la tarifa vigente de la sala correspondiente (Punto 1*d*).

```
FORALL(i:relacionadoASocio;
  O((i.belongsTo:menor & i.ingresarASala) -> i.abonarTarifaAl50))
```

- Lo que quisimos expresar con las tres reglas descriptas arriba es que si los acompañantes de los socios, según la calidad de acompañante que sea, ha ingresado a la sala, es porque ha abonado su tarifa correspondiente. Ahora lo que queremos decir es que si un acompañante abona la tarifa de ingreso correspondiente tenga el permiso de ingresar a la misma:

```
FORALL(i:relacionadoASocioPremium; P(i.ingresarASala given that
(i.belongsTo:mayor & i.abonarTarifaCompleta)))

FORALL(i:relacionadoASocioSelect; P(i.ingresarASala given that
(i.belongsTo:mayor & i.abonarTarifaAl75)))

FORALL(i:menor; P(i.ingresarASala given that i.abonarTarifaAl50))
```

- Sobre el programa de recompensas: El manejo de los puntos lo está controlando el contador, diciendo que se inicializa en 0, que su máximo es de 20 puntos, que pasado el año resetea su valor volviendo a 0 puntos.

  Además, nos indica que si un socio nuevo del tipo Premium es referencia suya suma 1 punto, si es Select suma 5 puntos y por cada ingreso a las salas suma un punto.

  Por último, nos indica que llegado al tope máximo no se impide la acción de seguir sumando socios o bien ingresando a las salas, lo que sí se hace es dejar de sumar los puntos.

  Resta entonces tener una regla que nos indique que ningún socio podrá tener más de 20 puntos en su programa de recompensas:

```
FORALL(i:socioTitular; F(i.puntosMembresia > 20))
```

- Sobre el uso de las instalaciones: este es el segundo archivos de input y acá nos enfocamos puntualmente en los clientes que son los que harán uso de nuestras salas. Definimos nuevos roles y subroles que están directamente relacionados a las reglas que luego queremos expresar:

```
roles cliente{conInconvenientesAlimenticios,
  sinInconvenientesAlimenticios disjoint cover}
```

- Para la regla 2*b* debemos establecer en `FL` una regla que indique que se prohibe el ingreso a la sala con alimentos:

```
tag: prohibidoIngresarAlimentos FORALL(i:cliente;
  F(i.ingresarASala & i.ingresarAlimentos))
```

- Pero no perdamos de vista que aquellos clientes que tienen ciertos inconvenientes con algunos tipos de alimentos tienen permitido ingresar a las salas con sus propios alimentos. Para ello agregaremos una excepción a la regla escrita anteriormente:

```
FORALL(i:cliente; P(i.ingresarAlimentos given that
i.belongsTo:conInconvenientesAlimenticios)
is exception of prohibidoIngresarAlimentos)
```

6.3. Verificando nuestro Caso de Estudio

Ahora que ya hemos explicado en detalle la manera en que especificamos nuestro caso de estudio llegó el momento de ingresar esta especificación a nuestra herramienta para validarla.

- Primer input: en este caso la herramienta genera los agentes:

```
agent_1: cliente, menor, relacionadoASocio,
   relacionadoASocioSelect, ciudadano
agent_2: relacionadoASocioPremium, cliente, mayor,
   relacionadoASocio, ciudadano
agent_3: cliente, mayor, relacionadoASocio,
   relacionadoASocioSelect, ciudadano
agent_4: relacionadoASocioPremium, cliente, menor,
   relacionadoASocio, ciudadano
agent_5: noCliente, ciudadano
agent_6: cliente, select, socioTitular, ciudadano
agent_7: cliente, premium, socioTitular, ciudadano
```

  Aquí podemos ver la distribución de los agentes para los subroles especificados

- Segundo input: para este caso solamente se crearán dos agentes ya que solamente precisamos separar los clientes en aquellos con restricciones sobre los alimentos y aquellos que no las tienen:

```
agent_1: cliente, sinInconvenientesAlimenticios
agent_2: cliente, conInconvenientesAlimenticios
```

  La salida del checker indica que se ha encontrado un comportamiento legal, tal como se esperaba. Esto es gracias a que se agregó la posibilidad de expresar las excepciones como tales y también la posibilidad de identificar subroles para agentes con características diferentes, algo que hasta ahora no era posible.

  Es importante notar que es con la combinación de ambas extensiones que esto puede ser logrado.

7. CONCLUSIONES Y TRABAJO FUTURO

7.1. Conclusiones

Cuando comenzamos este trabajo, la herramienta tenía ciertas limitaciones, naturales por cierto, dada la etapa prematura en la que se encuentra. Pero cosas tan necesarias como poder declarar valores máximos y mínimos individuales a los que puede llegar un contador no se podía hacer dado que esos valores eran establecidos de manera global para todos los contadores creados.

De allí en adelante, todas las mejoras y novedades que hemos podido volcar aquí nos han permitido ganar, ya sea en flexibilidad, o bien en incrementar el poder expresivo que queremos lograr para esta herramienta.

Incluir los permisos excepcionales fue otro trabajo que nos orientó mucho en el camino que queremos darle a esta herramienta, porque el tipo de incoherencia que antes nos entregaba el model checker por casos donde en realidad lo que queríamos expresar eran excepciones a reglas existentes, no es el tipo de incoherencia que nos interesa analizar, sino, justamente, eliminar. A partir de esta mejora, hemos podido escribir reglas con sus respectivas excepciones sin que la herramienta nos dijera que hay una incoherencia y centrarnos en las que efectivamente sí lo son.

Otro punto interesante en el que se abre todo un abanico de posibilidades a partir de este trabajo es con la incorporación de los permisos condicionales, porque, aunque se haya implementado un solo tipo de los existentes, la posibilidad de abordar un mundo de permisos a los cuales se accede bajo determinadas condiciones se acerca mucho más a la realidad con la que nos encontramos a la hora de convertir textos normativos a `FL`. Antes no teníamos la posibilidad de indicar que el permiso se podía ejecutar ante el cumplimiento de las condiciones (amén de si la condición es necesaria, suficiente o ambas).

No podemos darle menor valor a la importancia de contar ahora con permisos permanentes, en sus dos versiones: simples o condicionales. Es otra incorporación que nos permite modelar la realidad, como ya hemos dicho, hay muchos permisos que están constantemente latentes para poder ser ejecutados y antes solamente contábamos con aquellos cuya ejecución estaba latente para ser realizada una sola vez.

Por último, la posibilidad (en la medida que el model checker lo permita) de especialización en subroles para cada uno de los que definimos, abre el juego de una manera más que interesante: ya podemos hablar de todo lo que puede llegar a ser un agente y, mejor aún, podemos escribir normas para roles específicos o genéricos sin necesidad de tener que repetirlas por cada especialización de dicho rol. Y todo esto, sin perder las combinaciones que ya tenía la herramienta de jugar con los roles disjuntos y con cobertura total o parcial. Antes no podíamos ni siquiera conocer qué roles tenía un determinado agente, ahora podemos preguntarlo y hasta usarlo como condiciones o excepciones en las reglas.

Sentimos que hemos colaborado con la herramienta de tal manera de facilitarle el trabajo a las personas que se dedican a traducir los textos normativos a `FL`, en prime-

ra instancia, ubicándonos en su lugar a través de ejemplos concretos o de laboratorio donde uno se pregunta: ¿Esto puede ser escrito de una manera más sencilla o más entendible? O, más simple aún: ¿Esto cómo se expresa, cómo se traduce? Y, en segundo lugar, atendiendo la lista de necesidades concretas que estaban pendientes y tratando de priorizarlas en la facilidad que le puede generar a las personas que diariamente la utilizan para lograr el gran objetivo: contar con textos normativos sin incoherencias.

7.2. Trabajo futuro

Este trabajo tiene aún mucho camino por recorrer. Proyectando hacia el futuro, los aspectos que se pueden mejorar los podríamos dividir en tres grandes grupos según el aporte que cada uno realiza:

1. Performance en el análisis

   Hoy en día cuando generamos archivos de input con muchas variantes de agentes (más aún desde este trabajo donde se agregó la posibilidad de especializar los roles), o con contadores que tienen máximos de medianos a altos se traducen en problemas de performance a la hora de ser evaluados.

   A nosotras mismas nos sucedió que hemos tenido que dividir el caso de estudio o bien otros casos de prueba que hemos realizado en diferentes archivos para que el *model checker* lo pueda procesar sin problemas y en tiempos razonables.

   Trabajar en este campo ayudaría mucho sobre todo cuando se abordan casos más cercanos a los reales.

   De hecho (¡y por suerte!) ya contamos con un compañero que está investigando y trabajando en esta área y nos ha ayudado con la corrida de ciertos casos que en nuestro entorno de trabajo tardaba horas de procesamiento (el caso de estudio aquí presentado lo hemos dejado corriendo una noche entera sin obtener resultados) y en el de él, como ya había realizado algunas mejoras, se pudieron correr sin problemas en menos de una hora.

2. Soporte con la traducción de documentos normativos a `FL` Como mencionamos al principio de este trabajo, no contamos con mecanismos automáticos ni pseudo automáticos para traducir los documentos normativos tal y como los utilizamos a `FL` sino que se realiza de forma totalmente manual y precisando de personas que tengan conocimientos en el tema como para poder llevar esta tarea a cabo. Obviamente, contar con una automatización que colabore con este proceso sería de muchísima utilidad y permitiría que más personas utilicen estas herramientas ya que no precisaríamos depender del conocimiento de `FL` para realizar esta etapa del proceso.

3. Información arrojada por el *model checker* cuando no encuentra un comportamiento legal

   Otra área donde se pueden hacer buenos aportes está relacionada con la información que entrega el *model checker* cuando aborta la evaluación porque no encuentra un comportamiento legal para el archivo que se está verificando.

   Como mencionamos al principio de este trabajo, nuestra herramienta le envía al *model checker* la fórmula negada para tener una traza válida cuando se encuentra un comportamiento legal. El tema es que cuando no se encuentra un comportamiento legal, no tenemos información que nos brinde una orientación para entender qué está fallando ya que únicamente se indica que la fórmula ingresada es válida.

Poder agregar cierta información a lo que retorna el *model checker* y a lo que finalmente muestra *FormaLex* le proveería a los usuarios de nuestra herramienta más información sobre por qué no se está encontrando el comportamiento legal que buscamos.

4. Ampliación y flexibilización del lenguaje `FL` para potenciar el poder expresivo

   Ésta es el área donde nosotras estuvimos haciendo crecer *FormaLex* durante este trabajo. Si bien estamos muy satisfechas con el trabajo realizado, entendemos que hay mucho por hacer para continuar mejorando.

   Existe una lista donde se detalla punto por punto que surge de los propios usuarios de *FormaLex*, pero a nosotras mismas nos surgieron ciertas necesidades cuando realizamos nuestras propias pruebas. Algunas de ellas las describimos a continuación:

   - Contadores para determinados roles: a veces precisamos expresar el mismo contador pero con características particulares para determinados roles, por ejemplo, si quisiéramos decir que los socios Select acumulan 10 puntos por ingresar a las salas y los socios Premium sólo 5, entonces no podríamos expresarlo ya que todos los agentes tendrán siempre asociado el mismo contador. Nótese que la necesidad puntual es poder contabilizar bajo la misma acción, en este caso *ingresarASala*, valores diferentes según el rol.

     Hoy esto puede de alguna forma establecerse mediante el uso de dos acciones diferentes, pero esa incomodidad es justamente la que queremos evitar.
   - Contadores que utilizan los valores de otros contadores: hoy en día existen varias posibilidades de establecer valores para los contadores: valor inicial, mínimo y máximo, se puede resetear el valor del contador ante determinadas acciones e incluso establecer un valor determinado distinto del inicial ante determinados predicados, pero siempre debemos utilizar un valor fijo, un número concreto. No tenemos la posibilidad de utilizar un valor existente de otro contador, es decir, poder decir algo del estilo: "El valor actual del contador X + 1".
   - Otro punto interesante lo comenzamos a mencionar en el caso de estudio cuando hablamos de la regla que le permitía sólo a los socios Select el ingreso a determinadas salas. Como se puede observar precisamos crear dos acciones nuevas: *ingresarASalaDallas* e *ingresarASalaSorrento* cuando en realidad podríamos haber utilizado la acción *ingresarASala* que ya teníamos. El problema se presenta porque actualmente en `FL` no podemos hablar de objetos directos, es decir, no podemos hacer referencia a la *Sala Dallas* o *Sala Sorrento* para decir con una regla que a esas salas sólo pueden ingresar los *Socios Select*. Esta es otra buena extensión que se podría incorporar al lenguaje.
   - Tipos de permisos condicionales: parte de nuestro trabajo fue introducir los permisos condicionales, hasta hoy inexistentes, pero sólo hemos implementado el de tipo 1. Sin embargo, poder contar con la implementación de los otros tipos de permisos condicionales, tanto para los permisos convencionales como para los permanentes, nos permitiría expresar situaciones que hoy no podemos.

Bibliografía

[1] Patrick (ed.) Blackburn, Johan (ed.) van Benthem, and Frank (ed.) Wolter. *Handbook of modal logic*. Studies in Logic and Practical Reasoning 3. Amsterdam: Elsevier., 2007.

[2] Daniel Gorín, Sergio Mera, and Fernando Schapachnik. A Software Tool for Legal Drafting. In *FLACOS 2011: Fifth Workshop on Formal Languages an Analysis of Contract-Oriented Software*, pages 1-15. Elsevier, 2011.

[3] Daniel Gorín, Sergio Mera, and Fernando Schapachnik. Model Checking Legal Documents. In *Proceedings of the 2010 conference on Legal Knowledge and Information Systems: JURIX 2010*, pages 111–115, December 2010.

[4] Daniel Gorín, Sergio Mera, and Fernando Schapachnik. Verificación automática de documentos normativos: Ficción o realidad? In *SID 2010, Simposio Argentino de Informática y Derecho, 39JAIIO*, pages 2215–2229, September 2010.

[5] Juan Pablo Benedetti. FormaLex - Análisis y detección automática de defectos normativos. *Tesis de Licenciatura en Ciencias de la Computación*, Mayo de 2014.

[6] Gordon J. Pace, Fernando Schapachnik and Gerardo Schneider. Conditional Permissions in Contracts. In *Legal Knowledge and Information Systems - JURIX 2015*, pages 61-70, December 2015.