Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Formalex: Mejorando la herramienta para la detección de documentos normativos

Tesis de Licenciatura en Ciencias de la Computación

Nicolás Matías Gleichgerrcht
L.U. 160/08

Director: Fernando Schapachnik

Buenos Aires, 2017

ABSTRACT

La detección automática de inconsistencias en normas legales es un área específica dentro del campo denominado Automated Legislative Drafting que trata de facilitar la creación de documentos legales correctos. Este campo fue estudiado desde varios puntos pero la mayoría de los intentos para la construcción de herramientas formales de análisis de normas se basaron en la creación de herramientas nuevas.

Por otro lado, dentro del área de la Ingeniería del Software se estudia la verificación formal de programas cuyo objetivo es validar o encontrar errores en especificaciones de programas mediante la utilización de mecanismos lógico-matemáticos. Esta área cuenta con un enorme campo de investigación que a lo largo de los años ha desarrollado varias herramientas de verificación formal.

FormaLex es una herramienta creada con el fin de detectar inconsistencias en normas legales escribiendo a estos documentos como especificaciones formales utilizando un lenguaje creado para tal fin llamado FL. Una vez especificada la norma, se aplican técnicas de la verificación formal de programas para encontrar posibles inconsistencias en la misma.

La siguiente tesis consiste en extender la herramienta FormaLex, haciendo principal foco en el lenguaje FL, agregando soporte a nuevas expresiones del lenguaje que permitan especificar un conjunto más amplio de situaciones expresadas en textos normativos, manteniendo la simplicidad actual del lenguaje.

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.

Índice general

1. INTRODUCCIÓN

1.1. Verificación de software como método para el análisis de normas legales

La redacción de normas legales es un proceso complicado donde pueden escribirse incoherencias o inconsistencias difíciles de detectar. Existe un campo de estudio llamado `Automated Legislative Drafting` [5, 15, 3] cuyo principal objetivo es brindar asistencia informática a quienes escriben estas normas. Un objetivo importante de este campo consiste en encontrar de forma automática estas inconsistencias legales de manera tal de facilitar la creación de documentos legales correctos por parte de los legisladores.

La `verificación formal de programas` es el área de la Ingeniería del Software cuyo objetivo es validar o hallar errores en especificaciones de programas mediante la utilización de propiedades lógico-matemáticas. Se destacan en ellas herramientas como los model checkers SPIN o NuSMV [12, 4].

Se puede trazar un paralelismo entre el análisis de normas legales y la especificación de un programa: ambos son una serie de reglas y cláusulas que determinan qué es posible o no de hacer, y ciertas obligaciones para algunos momentos determinados [13]. Al igual que al verificar formalmente un lenguaje, a una norma legal se le pide que sea coherente, es decir que no contenga contradicciones y que contemple todos los casos.

Tradicionalmente, los intentos para la construcción de herramientas formales de análisis de normas se basaron en la creación de herramientas nuevas, desde cero [7, 10]. Por otro lado, la verificación formal tiene un enorme campo de investigación activo que ya data de varios años y en el cuál se han creado varias herramientas que podrían ayudar al estudio de análisis legal.

A lo largo de esta tesis mostraremos los avances y mejoras que agregamos en `FormaLex`, una herramienta que brinda mediante un lenguaje lógico, la posibilidad de analizar y encontrar incoherencias en normas legales.

1.2. Lógica deóntica y lógica temporal lineal

La `lógica deóntica` es una familia de lógicas modales creada con la intención de hablar de permisos, obligaciones y prohibiciones. Su exponente más común es la Standard Deontic Logic y se basa en una extensión de la lógica proposicional. Este tipo de lógica permite la expresión de normas o proposiciones legales mediante el uso de ciertos operadores modales. Debido a su definición, resulta natural partir de está lógica para el estudio de las normas legales y es parte de la base de herramientas desarrolladas para ese fin, como es el caso de FormaLex, la herramienta que esta tesis extiende.

En la lógica deóntica tradicionalmente se define un operador básico llamado **obligation**, expresado como O(ρ), que expresa que la fórmula ρ debe cumplirse.

A partir de este operador y de la negación lógica, se puede definir el operador de la prohibición. La lógica deóntica define el operador `forbidden`, expresado como F(ρ), como que está obligado a que se cumpla la negación de ρ, es decir $F(\rho) \equiv O(\neg\rho)$.

Por último se puede definir los permisos, que la lógica deóntica lo hace diciendo que una fórmula no está prohibida. Es decir, se define el operador **permission** como $P(\rho) \equiv \neg F(\rho) \equiv \neg O(\neg\rho)$.

Por otro lado, la `lógica temporal lineal` (LTL) [2] es un tipo de lógica modal que permite expresar proposiciones cuyo valor de verdad puede modificarse a lo largo del tiempo. La misma se basa de ciertos operadores modales, descriptos a continuación, que agregan expresividad a la lógica proposicional permitiendo hacer referencias a comportamientos variantes en función del tiempo. Los modelos son interpretados en estructuras de Kripke y sus recorridos son secuencias lienales dentro de esos modelos.

Definimos los siguientes operadores para LTL:

- **Next** ($\bigcirc\phi$): ϕ se tiene que cumplir en el siguiente estado
- **Globally** ($\Box\phi$): ϕ se tiene que cumplir siempre
- **Finally** ($\Diamond\phi$): ϕ eventualmente se tiene que cumplir
- **Until** ($\Phi\mathfrak{U}\phi$): Φ si tiene que cumplir al menos hasta que se cumpla ϕ

Más adelante en la tesis cuando se entre en mayor detalle en la herramienta FormaLex, se verá como se relacionan estás dos lógicas y la utilidad que se le da para especificar normas legales y examinar inconsistencias.

1.3. El lenguaje FL

Teniendo en cuenta el fuerte paralelismo que se puede establecer entre un documento normativo y una especificación de un sistema informático [8], se desarrolló FL, un lenguaje deóntico que fue pensado para especificar una norma legal. FL es el lenguaje de entrada que utiliza la herramienta FormaLex cuyo principal objetivo es brindar un lenguaje suficientemente expresivo como para poder escribir normales legales en forma sencilla, intentando ser lo más completo posible y permitiendo a la vez la posibilidad de traducir la norma deseada a un lenguaje que sea entendido por model checkers standard para poder analizar inconsistencias.

Para poder realizar esto se divide el lenguaje en dos partes que describimos a continuación: la parte que habla del contexto de la norma, y el conjunto de reglas deónticas de la misma.

Cabe aclarar que al momento de hacer la tesis no existe ninguna herramienta que permita hacer el pasaje de un texto de una norma legal del castellano al lenguaje de FL.

1.3.1. Componentes de la background theory

Los componentes de la teoría marco o background theory tienen por objetivo brindar mecanismos sencillos para poder describir cuestiones del mundo real sobre el cual las leyes van a regir y hacer referencia, como puede ser la precedencia de eventos (e.g. el día ocurre antes que la noche), unicidad (e.g. las personas nacen una sola vez), etc.

Este conjunto de definiciones se traducen en forma automática a autómatas de Büchi, para llevarlo al lenguaje de especificación del model checker.

En las siguientes secciones describimos los componentes que existen en la teoría marco, que son los que se pueden utilizar para definir el contexto de una norma legal.

Agentes y roles

El **agente** es una entidad que representa a las personas físicas o jurídicas y cuya finalidad es llevar a cabo las acciones personales del sistema. No cuentan con una declaración explícita en la teoría marco, sino que son generados por el sistema a partir de los roles que definimos.

Los **roles** son formas de clasificar a los agentes en nuestro modelo para determinar que ciertas acciones pueden o necesitan ser realizadas por algunos y no por otros. Por ejemplo podemos definir los roles de los ciudadanos entre civil y militar.

```
roles civil, militar
```

Existen modificadores para los roles que permiten adecuar mejor los agentes que deben crease. Tenemos el modificador **disjoint** que indica que los roles son excluyentes, es decir que un agente no puede tener ambos roles. En caso de no definir este modificador se creará tantos agentes como combinaciones de roles posibles. Por otro lado, existe el modificador **cover** cuyo fin es indicar que estos roles representan todo el espectro de roles posibles para el caso que se está especificando. En caso de no definirse este modificador, se creará un agente extra sin ningún rol.

Veamos con el mismo ejemplo de antes los agentes que se crean según los modificadores.

`roles civil, militar` se crean los agentes `civil, militar, civil-militar, sin-rol`.

`roles civil, militar disjoint` se crean los agentes `civil, militar, sin-rol`.

`roles civil, militar cover` se crean los agentes `civil, militar, civil-militar`.

`roles civil, militar disjoint cover` se crean los agentes `civil, militar`.

A partir de [11], se permite definir una jerarquía de roles que da mucho más poder expresivo y detalle a la norma que se está definiendo. Esta jerarquía es en forma arbolada y por cada nivel de la jerarquía se puede definir un conjunto de roles, como los recién mostrados, junto con los posibles modificadores. Estos modificadores van a aplicarse exclusivamente al nivel de la jerarquía para los cuales se están definiendo. Con esta jerarquía de roles, el programa va a crear agentes para cada combinación de la estructura definida, y un agente puede tener varios roles. Veamos un ejemplo.

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

Luego, para esta jerarquía de roles el programa va a definir los siguientes agentes

- `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

Acciones

Las **acciones** son las construcciones del lenguaje que permiten modelar la actividad de los agentes. Las mismas pueden estar ocurriendo, haber ocurrido recientemente, o no estar ocurriendo. Los cambios de los estados de las acciones son lo que van dando sentido a las diferentes trazas del modelo, lo que hace fundamental contar con este componente.

Las acciones pueden definirse de forma impersonal, es decir de forma tal que no necesiten de alguien para ser ejecutadas, por ejemplo

```
impersonal action llover
```

O también pueden ser definidas de forma personal, es decir que necesiten de alguien para ser ejecutadas. Este alguien que va a ejecutar la acción va a ser algún agente. Por ejemplo

```
action vender
```

Existen ciertos modificadores sobre las acciones que son de utilidad a la hora de definir una norma. Por ejemplo, muchas veces nos vemos en la necesidad de que al ejecutar una acción haya otra acción que deba ejecutarse en simultáneo. Por ejemplo cada vez que alguien compra un objeto otra persona lo debería estar vendiendo. Esto se puede definir en el lenguaje de la siguiente forma

```
action vender synchronizes with comprar
```

Otro modificador importante es poder definir la cantidad de veces que puede ocurrir una acción, por ejemplo podemos decir que la acción `extraerDinero` sólo puede darse tres veces. Para esto podemos definirlo como

```
action extraerDinero ocurrences 3
```

También podemos escribir acciones que generen 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}
```

El esquema de agentes y roles permite agregar un nuevo modificador a las acciones que permite definir para qué rol puede ejecutar la acción y sólo los agentes que tengan ese rol podrán hacerlo. Por ejemplo si queremos que sólo los mayoristas puedan comprar con descuento podemos definir

```
action extraerDinero only performable by mayorista
```

Todos los modificadores sobre las acciones pueden combinarse entre sí potenciando las posibilidades a la hora de definir una norma legal lo más fielmente posible.

Contadores

Los **contadores** sirven para contabilizar la ocurrencia de acciones y poder establecer límites para las mismas. Al momento de desarrollar la tesis existían dos tipos de contadores, los de alcance local y los de alcance global. Los contadores locales permitían que los contadores se apliquen a nivel de agente, es decir la ejecución de una acción iba a incrementar el contador únicamente del agente que lo ejecutó. Por otro lado, los contadores globales son contadores compartidos entre todos los agentes y se modifica el mismo indistintamente de que agente ejecutó la acción asociada al mismo.

Todos los contadores tienen varias propiedades y modificadores que se pueden definir como las que se listan a continuación

- Valor inicial del contador
- Conjunto de acciones que lo incrementan y en qué magnitud
- Conjunto de acciones que lo decrementan y en qué magnitud
- Conjunto de acciones que lo regresan a su valor inicial
- Conjunto de acciones que lo llevan a un valor específico y en qué valor
- Condición por la cual el contador es reiniciado
- Valor máximo y mínimo del contador
- Impedir la ejecución de las acciones que modifican al contador una vez alcanzado el límite

Por ejemplo podemos definir el siguiente contador

```
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 esta tesis agregamos un nuevo tipo de contador, llamado contador a nivel de roles y entraremos en detalle en los siguientes capítulos.

Intervalos

Los `intervalos` nos permiten definir periodos de tiempo entre un conjunto de acciones que dan inicio y fin a los mismos. Al igual que con los contadores, al momento de desarrollar la tesis existían dos tipos de intervalos, los de alcance local y los de alcance global.

Al definir un intervalo local, lo que se hace es definir un intervalo para cada agente del sistema y por ende la ejecución de una acción que da inicio o fin a un intervalo solo modificara al intervalo para el agente que ejecutó la acción en cuestión. Por otro lado, los intervalos globales son intervalos compartidos entre todos los agentes y por ende no importa que agente ejecutó una acción, si la misma da inicio a fin a un intervalo, todos los agentes verán el cambio.

Los intervalos pueden estar activos o inactivos, y esta propiedad se basa exclusivamente en los conjuntos de acciones que da inicio o fin al mismo y, por supuesto, estos conjuntos deben ser disjuntos. Un intervalo puede cambiar entre estado activo e inactivo todas las veces que sea necesario, salvo que se indique lo contrario limitando la cantidad de ocurrencias.

Los intervalos tienen ciertas propiedades y modificadores que permiten definir diferentes características de los mismos. Las mismas se listan a continuación

- Conjunto de acciones que dan inicio al intervalo
- Conjunto de acciones que dan fin al intervalo
- Cantidad máxima de veces que puede activarse el intervalo
- Definir si el intervalo inicia activo o no
- Definir si el intervalo una vez activo queda activo para siempre o no
- Definir si un intervalo sólo puede ocurrir dentro de otro

Por ejemplo podemos definir los siguiente intervalo

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

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

Junto a la definición de intervalos, también se agrega un nuevo modificador a las acciones que permite definir que cierta acción solo se puede ejecutar cuando un intervalo está activo. Por ejemplo si la acción de realizar la revisación médica sólo tiene sentido cuando es la temporada de pileta, podemos definir este ejemplo de la siguiente forma

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

En esta tesis agregamos un nuevo tipo de intervalo, llamado intervalo a nivel de roles y entraremos en detalle sobre el mismo en los siguientes capítulos.

1.3.2. Lenguaje de las reglas

El otro punto importante de FL es el lenguaje de las reglas, que permite a través de fórmulas deónticas expresar cuales deben cumplirse dentro de la norma definida. En estas fórmulas se permite hacer referencia a los componentes de la teoría marco definidos previamente.

Existen diferentes tipos de reglas que el programa permite definir y las describimos a continuación.

Obligación

El operador de obligación indica que una regla debe cumplirse siempre, es decir en todos los estados de todos los modelos. Por ejemplo, si queremos expresar que es obligatorio rendir un examen final, podemos hacerlo definiendo

```
O(rendir_examen)
```

Prohibición

El operador de prohibición indica que una regla nunca puede cumplirse, es decir que en todos los estados de todos los modelos la condición debe ser falsa, o lo que realmente se prueba, que la negación de la condición debe ser verdadera. Por ejemplo si queremos expresar que está prohibido fijar carteles podemos hacerlo como

```
F(fijar_carteles)
```

Permitido

El operador del permiso indica que una regla no está prohibida, es decir que eventualmente existe algún estado donde la condición es válida. Por ejemplo si queremos expresar que está permitido hablar podemos decir que

```
P(hablar)
```

Cuantificadores

Para otorgar un mayor poder expresivo al lenguaje, el mismo cuenta con cuantificadores que permiten definir reglas sobre roles y subroles. Existen dos tipos de cuantificadores, el cuantificador universal y el cuantificador existencial.

El cuantificador universal permite indicar que para todo agente que tenga el rol indicado, debe cumplirse la regla. Por ejemplo si queremos que cualquier tipo de alumno

esté obligado a rendir un examen podemos definir que

```
FORALL(i:alumno; O(i.rendir_examen))
```

Luego todo agente que tenga a alumno como alguno de sus roles está obligado a rendir el examen.

El cuantificador existencial indica que debe existir al menos un agente con el rol indicado para el cuál se cumpla la regla. Por ejemplo si queremos indicar que debe existir algún profesor que tenga permiso para ingresar a una sala, podemos decir que

```
EXISTS(i:profesor; P(i.entrar_sala))
```

Cada una de las estas reglas definidas en la norma, se va a traducir a fórmulas de la lógica temporal lineal, que expresarán la validez de una fórmula en función a su comportamiento en el tiempo. La traducción base que se hace partiendo de una norma obligatoria. Entonces si definimos a $O(\phi)$, lo que queremos expresar es que ϕ debe valer en todo modelo legalmente válido luego se puede traducir a $\Box\phi$, es decir que ϕ debe valer para todo estado del sistema. Partiendo de esta traducción, y usando las equivalencias de la lógica deóntica definidas previamente, podemos traducir todas las reglas que definamos a fórmulas de la LTL que se necesitan para hacer el análisis de la norma legal [9].

1.4. FormaLex: herramienta para la detección automática de defectos normativos

La idea detrás de `FormaLex` es poder modelar y analizar normas legales mediante la utilización herramientas informáticas con el fin de poder encontrar en forma automática inconsistencias en las mismas.

Partiendo como entrada de la especificación en FL de una norma legal mostramos como podemos en forma automática encontrar las incoherencias de la misma.

Una vez traducida la norma legal definida en FL a una fórmula LTL, FormaLex utiliza como motor de inferencia **model checkers** para LTL, que ya han sido ampliamente probados en diferentes ámbitos académicos y profesionales y tienen un gran respaldo de la comunidad de desarrolladores [4].

El fin de un model checker es verificar si dentro de una clase de modelos descripta por un autómata, existe alguno para el cual una fórmula no es válida. Más en detalle, recibe como entrada un autómata A y una fórmula φ, y su objetivo 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.

Para poder utilizar los model checkers FormaLex lo que hace es generar el autómata A traduciendo lo que se conoce como la teoría marco del lenguaje FL a un autómata de Büchi. Por otro lado, las fórmulas a probar que recibe el model checker se obtiene mediante

la traducción de las reglas definidas en FL a una LTL y ciertas operaciones lógicas, como la conjunción, para poder generar las diferentes fórmulas a probar. Llamemos φ a una de estas fórmulas, luego queremos ver si existe al menos una corrida de A que satisfaga φ. Entonces le proporcionamos al model checker la fórmula $\neg\varphi$ junto con el autómata A, y de esta forma sabemos que si encuentra un contraejemplo, el mismo sería una violación de $\neg\varphi$ y por ende se satisface φ. Es decir, podemos entenderlo como que existe un testigo de que existe al menos una forma de cumplir con todas las reglas para la teoría marco descripta y que por ende la norma se encuentra libre de contradicciones.

A continuación se muestra en forma detallada la manera en que FormaLex prueba si una norma legal tiene inconsistencias.

1. Recibe un archivo de entrada del lenguaje FL
2. Parsea el archivo y valida su sintaxis. De no ser correcto detiene la ejecución indicando el problema
3. Crea las diferentes entidades de la teoría marco especificada
4. Generá el autómata correspondiente a la teoría marco
5. Expande las reglas definidas en la entrada para que sea usado por las diferentes entidades del programa
6. Traduce las reglas expandidas a LTL
7. Crea una fórmula con la conjunción de todos las obligaciones y prohibiciones (ya traducidas) y se niega dicha fórmula
8. Ya con el autómata y la fórmula generada, se invoca por primera vez al model checker para ver probar si existe incoherencias en la norma legal
9. Por último, para cada permiso descripto en el lenguaje se prueba uno a uno agregándolos a la conjunción negada de reglas y vuelve a probar la nueva fórmula en el model checker
10. El programa dará que la norma legal es válida si analizando los resultados de las diferentes corridas del model checker, en ningún caso se encontró incoherencias

Algunas observaciones de este proceso. La herramienta FormaLex utiliza el lenguaje de programación `Java` para todo el proceso de parseo y traducciones. En (2) para hacer el parseo de la entrada en FL, se utilizó la herramienta `JavaCC`, que permite dado un lenguaje definido, generar en Java todas las clases para poder interactuar. Para hacer las traducciones a LTL se utilizó `Velocity` que permite generar templates de traducciones, permitiendo que sea fácil utilizar diferentes model checkers. En el caso particular de esta tesis se utilizó el model checker `NuSMV`.

1.5. El alcance de esta tesis

Siguiendo el trabajo que presentado en [1] y en [11], en esta tesis nos vamos a enfocar en expandir el poder expresivo del lenguaje FL tanto en la teoría marco como en las reglas posibles a definir. Para esto se van a encarar los siguientes problemas.

- Contadores a nivel de roles: se da la posibilidad de definir contadores para cierto rol y sus subroles con nuevos modificadores y propiedades
- Intervalos a nivel de roles: se da la posibilidad de definir intervalos para cierto rol y sus subroles
- Permisos condicionales necesarios: se extienden los permisos condicionales [14] para poder indicar que un conjunto de permisos son necesarios para que se cumplan cierta cláusula
- Permiso condicionales necesarios y suficientes: se extienden los permisos condicionales para indicar que un permiso particular tiene una condición necesaria y suficiente para que se cumpla

En los próximos capítulos entraremos en detalle sobre las diferentes mejoras que agregamos al lenguaje, que problemas quieren solucionar, las decisiones tomadas en cada caso, la sintaxis y ejemplos de uso.

2. CONTADORES A NIVEL DE ROLES

2.1. Motivación

A partir del uso de la aplicación en diferentes ejemplos, surge la necesidad de utilizar contadores compartidos a nivel de roles. Los contadores son un componente del marco teórico, proporcionados por el lenguaje, que, como su nombre lo indica, permiten llevar cuenta de la ocurrencia de algún suceso.

Actualmente, se pueden utilizar contadores a nivel de agente o a nivel global. En el primer caso, las acciones que modifican al contador por un agente sólo tendrán efecto dentro del scope de ese agente y ningún otro agente verá este cambio. En el caso de los contadores globales, todo agente puede modificar un contador compartido por todos los agentes. Al modificarse un contador global mediante una acción, el contador es afectado para todos los agentes del sistema.

Ahora bien, tomemos como ejemplo querer modelar la venta de entradas a un recital. Este recital cuenta con dos tipos de entradas para vender, las regulares y las VIP. Se ponen la venta 500 entradas regulares y 100 entradas VIP.

Con los tipos actuales de contadores, puede resultar muy engorroso modelar este ejemplo, ya que un contador local, sólo nos permitiría saber la cantidad compradas por una sola persona, pero no el total de entradas, mientras que un contador global nos permitiría saber la cantidad total de entradas compradas pero para distinguir y poner límites entre los diferentes tipos de entrada deberíamos crear varios contadores y acciones diferentes.

Se ve la necesidad de agregar contadores a nivel de roles, dando esto la posibilidad de utilizar contadores que se vean afectados por las acciones de los agentes propios de un tipo de un rol y todos sus subroles, pero sin que lo puedan afectar las acciones de otros roles.

Siguiendo con el ejemplo de ventas de entradas, supongamos que además se quiere ofrecer una promoción exclusiva para los clientes de cierto banco, que podrán sacar entradas regulares a 2x1.

Los modificadores actuales de los contadores, permiten indicar en cuánto se ve afectado un contador para cierta acción, sin embargo este modificador no permite indicar diferente variaciones según el agente que ejecuta la acción. Luego, por ejemplo en un contador global, todos los agentes que realicen la misma acción modificarán al contador siempre bajo la misma variación.

En el caso recién descripto de la venta de entradas, no sería posible permitir la compra de entradas regulares con diferente variación para el rol de independientes y clientes del banco, teniendo como única alternativa la separación de los casos en dos acciones diferentes, agregando así ruido al sistema.

Luego, junto con la posibilidad de utilizar contadores a nivel de roles, también se agrega la posibilidad de manejar diferente variaciones a los contadores para una misma acción según el tipo de agente que la ejecute. A lo largo de las siguientes secciones, se entrará en detalle en estas extensiones, mostrando su utilización, los problemas que surgen a partir de la jerarquía de roles, cómo los solucionamos y la implementación a nivel técnico.

2.2. Contadores a nivel de roles

Partiendo de las motivaciones recién indicadas, se va a extender el lenguaje provisto por el marco teórico para agregar la posibilidad de utilizar contadores a nivel de roles.

2.2.1. Descripción

Hasta el momento se podían definir los contadores a nivel de agente (local) o nivel del sistema (global). Ahora se pueden definir los contadores a nivel de roles.

Decimos que un `contador a nivel de roles` es un contador que puede ser modificado por todos los agentes que tengan como rol al rol para el cual está definido o alguno de sus subroles. Al ser esta una extensión sobre el alcance de los contadores ya existentes, la modificación del contador sigue siendo realizada a través de las diferentes acciones de los agentes que tienen alcance sobre este contador.

También cabe destacar que los modificadores existentes para los contadores actuales siguen siendo válidos para el nuevo tipo de contador, como la posibilidad de definir límites y la capacidad de impedir la ejecución de una acción al llegar a uno de los limites definidos.

2.2.2. Extensión

Para poder utilizar los contadores a nivel de roles extendemos el lenguaje del marco teórico agregando la siguiente sintaxis.

```
shared counter counterName for roleName
```

De esta manera, agregamos *shared* como un nuevo modificador de alcance de los contadores, acompañado con la expresión *for roleName* para indicar el rol al que el alcance del contador hace referencia.

Por ejemplo para el caso descripto en la motivación, podemos definir teniendo los roles `compradorRegular` y `compradorVIP` podemos definir los siguientes contadores.

```
 shared counter entradasRegulares for compradorRegular max value 500
increases with action comprarEntrada
reaching max impedes actions

 shared counter entradasVIP for compradorVIP max value 100
```

```
increases with action comprarEntrada
reaching max impedes actions
```

De esta forma, todos los agentes que tengan como rol `compradoresRegular` y ejecuten la acción `comprarEntrada` incrementaran en 1 (valor por defecto) al contador `entradasRegulares`, mientras que todos los agentes con rol `compradorVIP`, al ejecutar la misma acción, modificarán el contador `entradasVIP`.

Acá surgen dos grandes problemas a resolver, en la jerarquía de roles, para un conjunto de roles de igual jerarquía pueden existir dos modificadores que hay que tener en cuenta. Por un lado un conjunto de roles puede ser disjunto o no serlo, por otro lado puede ser cover o no. Cada una de estas variaciones puede llegar a generar un conflicto a la hora de tomar la decisión de modificar un contador compartido. En la siguiente sección se va atacar este problema, agregando restricciones y nuevas variaciones al momento de definir las acciones que incrementan y decrementan un contador.

2.3. Variaciones de contadores por agente

La otra extensión al lenguaje que surge en las motivaciones es la posibilidad de definir diferentes valores para el incremento o decremento de un contador a partir de una misma acción, según el agente que la ejecute. Es decir, dados por ejemplo dos agentes con roles diferentes, que uno pueda incrementar el contador compartido entre ambos en 1 unidad mientras que el otro agente lo pueda incrementar en 5 unidades, realizando ambos la misma acción.

2.3.1. Descripción

Hasta el momento, a la hora de definir un contador era posible indicar para una acción a ejecutar por un agente, cuál era la variación que debía sufrir el contador. Sin embargo, no era posible indicar diferentes variaciones para una acción según el tipo de agente para una misma acción.

A partir de esta tesis, se permite definir diferentes variaciones según los roles de los agentes que ejecutan una misma acción. Sin embargo para poder permitir esto se tuvieron que definir diferentes condiciones y restricciones que surgieron de las características propias de la jerarquía de roles del lenguaje.

La jerarquía de roles consiste en una estructura de árbol donde los hijos del árbol son considerados subroles de sus padres. Cada conjunto de roles hermanos en la jerarquía, es decir aquellos roles en el mismo nivel del árbol que compartan un mismo padre, pueden tener a lo sumo dos propiedades

- **Disjoint** un agente no puede tener dos roles del conjunto a la vez
- **Cover** el agente debe tener al menos un rol de conjunto

Luego, al momento de tener que definir en cuánto se ve afectado un contador según una acción ejecutada por un agente, es importante tener en cuenta los posibles roles que puede tener este agente. Por un lado para no tener ambigüedades a la hora de saber qué variación ejecutar. Por el otro, para contemplar todos los roles posibles para que no quede indefinido en cuánto debe modificar una contador una acción de un agente dado.

Para tratar el problema de la indefinición, al momento de correr el programa con las definiciones, se probará si todos los agentes que puedan ejecutar determinada acción tienen definido un valor para la modificación del contador, y de no tenerlo, el programa emitirá un error indicando del problema.

Para el problema de la ambigüedad, se divide en dos casos. Por un lado, el caso en que estén definidos dos roles tal que ninguno sea ancestro del otro. En este caso debe existir al menos un ancestro en común que tenga un conjunto disjunto de roles. De no existir esta propiedad, el programa emitirá error al intentar correrse.

Por otro lado está el caso en que uno de los dos roles sea el ancestro otro. En este caso se permite indicar al momento de definir el contador si la prioridad la tiene el rol de mayor refinamiento (es decir el rol del que es descendiente), o si la variación con la cuál se modifica el contador debe ser la suma de las variaciones. Al analizar el caso de permitir que el rol ancestro sea el prioritario, vimos que no aportaba valor a la especificación ya que no valdría la pena especificar el valor a modificar en los roles descendientes.

2.3.2. Extensión

Para poder indicar a qué roles afecta la variación particular de una acción sobre un contador, extendemos el lenguaje para agregar los siguientes modificadores.

```
 shared counter counterName for roleName
increases with action actionName by X for roleName1
increases with action actionName by Y for roleName2, roleName3
```

La idea es poder indicar para cada valor con el que se incrementa/decrementa cada contador dada una acción y pudiendo diferenciar por los distintos subroles del rol al que aplica el contador. Se agrega la expresión `for roleName` para indicar al rol que debe tener el agente que ejecuta la acción. Se permite poner varios roles en una misma sentencia para facilitar la escritura.

Junto a esto, se agrega la siguiente sentencia que tiene varios propósitos.

```
 shared counter counterName for roleName
increases with action actionName by X for roleName2
increases with action actionName by Y for roleName without further refinement
```

La idea es que `roleName1` sea ancestro de `rolName2`. La expresión `without further refinement` da la posibilidad de desambiguar qué hacer cuando un rol padre tiene definido la variación para una acción al igual que alguno de sus hijos. En caso de tener la expresión,

el contador sólo se modifica en la cantidad que dice el rol de mayor refinamiento (es decir el más abajo en el árbol) del agente que está ejecutando la acción. En caso de no tener esta sentenciá, se modificará el contador en lo que indica cada rol.

2.4. Implementación

Transformando las nuevas expresiones a expresiones más básicas ya provistas por el lenguaje, es posible implementar los contadores a nivel de roles. En el proceso, si se detecta algún posible error por ambigüedad o por incompletitud se arrojara un error y finalizará el programa. Llamamos `rol raíz` al rol para el cual se define el contador.

Primero partiendo del subárbol de roles para el rol raíz del contador, se calculan todos los agentes que pueden modificarlo, es decir los agentes que tienen algún rol, subrol del rol raíz o el mismo rol raíz. Luego, para cada acción que se define como modificadora del contador, se calculan todos los agentes que pueden ejecutar dicha acción. A partir de ahí, para cada acción definida se computa la intersección con los agentes anteriormente calculados, y se comprueba si existen reglas para cada agente posible. De no existir se produce un error, finalizando el programa.

Por otro lado, para que no haya ambigüedad entre dos roles, en la construcción de la estructura se comprobará que dado dos pares de roles con diferentes variación del contador, uno sea ancestro del otro, o exista un ancestro común que tenga sus ancestros directos disjuntos. De no cumplir alguna de estas propiedades, el programa producirá un error y se finalizará.

En caso de poder generar todas las estructuras correctamente, se procede a transformar la expresión a una más específica.

Partiendo del ejemplo dado en la motivación, se muestra como se transforma la expresión del contador a una más básica ya soportada programa. Tenemos los roles
`roles regular { independiente, clienteDeBanco disjoint cover }, vip disjoint cover`

Luego, se desprenden de esta definición los siguientes agentes

```
agent1: independiente, regular
agent2: socioDeBanco, regular
agent3: vip
```

Entonces según lo visto en la sección anterior, definimos el contador de la siguiente manera.

```
shared counter entradas for regular
increases with action comprarEntrada by 1 for independiente
increases with action comprarEntrada by 2 for socioDeBanco
```

Esta expresión se transformará a una más básica en donde las acciones generadas a partir de los agentes que tengan a `regular` como rol, sean descriptas en forma explícita. En el ejemplo, la sentencia queda como

```
counter regular
increases with action agent1.comprarEntrada by 1
increases with action agent2.comprarEntrada by 2
```

Para el resto de los modificadores de los contadores (decreases, reset, set) la transformación funciona de la misma manera a la descrita.

Partiendo del ejemplo, podemos generalizar la transformación que se genera, en caso de pasar todas las validaciones, definiéndola a partir de todas las acciones de todos los agentes que contengan al rol para el cuál se define y que modifiquen al contador según la variación que correspondan. Para saber la variación se calcula según las reglas descritas en los capítulos anteriores.

2.5. Ejemplo

Vamos a modelar el ejemplo de la motivación. Empezamos por definir bien los requerimientos del mismo.

Hay un nuevo recital para el cuál se quiere modelar las condiciones para la venta de entradas. Se tiene dos tipos de entradas, las regulares y las VIP. Hay 100 entradas VIP para repartir y 500 entradas regulares. Para las entradas regulares, los clientes de un banco tienen la posibilidad de comprar las entradas al 2x1. A este banco se le pone la restricción que no pueden vender más de 10 promociones por día. Por último, se quiere llevar un control sobre los ingresos por las entradas vendidas, las entradas VIP se venden a $300 y las entradas regulares se venden a $100.

Basados en todo el comportamiento descripto, podemos modelar las condiciones de la siguiente manera.

```
roles comprador {
   regular { independiente, socio_de_banco disjoint cover },
   vip disjoint cover
}
```

```
impersonal action pasa_un_dia
actions comprar_entrada
```

```
shared counter entradas_para_vip for vip max value 100,
increases with action comprar_entrada,
```

```
reaching max impedes actions


shared counter entradas_para_regular for regular max value 500,
increases with action comprar_entrada by 1 for independiente,
increases with action comprar_entrada by 2 for socio_de_banco,
reaching max impedes actions


shared counter entradas_del_banco for socio_de_banco max value 10,
increases with action comprar_entrada by 1 for socio_de_banco,
reaching max impedes actions,
resets with action pasa_un_dia


shared counter ingresos_entradas for comprador,
increases with action comprar_entrada by 300 for vip,
increases with action comprar_entrada by 100 for comprador without further refinement
```

3. INTERVALOS A NIVEL DE ROLES

3.1. Motivación

Continuando con el ejemplo del recital explicado en la sección de contadores, supongamos que ahora se plantea la necesidad de vender bebidas y comidas durante el recital. Para fomentar el consumo, la organización decide implementar momentos llamados happy hour, donde las personas pueden comprar ciertos productos al 2x1.

Habiendo hecho un análisis de la situación se decide limitar a tres happy hour en la sección VIP y sólo uno en la sección regular. Durante la duración del happy hour, los espectadores podrán comprar dos bebidas alcohólicas al precio de una, promoción no válida fuera de estos momentos.

Partiendo de este ejemplo, se ve la necesidad de tener intervalos a nivel de roles, donde acciones de agentes que tengan algún rol o subrol del rol raíz para el cuál se define el intervalo den comienzo y fin al mismo, y éste sea compartido por todos los agentes del rol raíz.

Al momento de comenzar la tesis sólo había dos tipos de intervalos, los globales y los locales. Se define a un intervalo global como un intervalo compartido entre todos los agentes de la especificación actual, haciendo que cualquiera de estos pueda dar comienzo y fin al mismo. Por otro lado, se define a un intervalo local, como un intervalo propio de cada agente donde el único que puede dar comienzo y fin al intervalo es el mismo agente para el cuál está definido.

Basados en estos tipos de intervalos definir problema planteado en forma correcta puede resultar muy engorroso utilizando los tipos de intervalos disponibles hasta el momento. De utilizar intervalos locales no es posible permitir que dos agentes diferentes den comienzo y fin a un mismo intervalo, haciendo que el happy hour tenga que ser basado en una sola persona. En el caso de intervalos globales, no sería posible utilizar las mismas acciones de los espectadores de diferente rol para interactuar con diferentes intervalos, siendo necesario dividir las acciones de los vendedores regulares y los vendedores VIP, agregando tamaño y ruido al sistema.

En las siguientes secciones mostraremos cómo podemos extender el lenguaje para permitir la utilización de los intervalos a nivel de roles, la implementación de los mismos y cómo se ve reflejado este ejemplo con la nuevas propiedades del sistema.

3.2. Intervalos a nivel de roles

Partiendo de la motivación descripta en la sección anterior, se ve la necesidad de extender el lenguaje del marco teórico dando la posibilidad de utilizar intervalos a nivel

de roles.

3.2.1. Descripción

Definimos a los `intervalos a nivel de roles` como un nuevo tipo de intervalo, en donde el alcance del mismo son todos los agentes que tienen entre alguno de sus roles al rol para el cuál fue definido el intervalo o alguno de los subroles del mismo. Siendo ésta una extensión sobre intervalos, la misma tendrá todas las propiedades ya existentes para los mismos, como la cantidad máxima de ocurrencias, o la posibilidad de definir un intervalo padre del cual depende.

Al definir el alcance del intervalo en base a un rol, lo que se indica es que las acciones que dan inicio y fin al mismo sólo puede ser ejecutadas por alguno de los agentes pertenecientes al rol o subrol para el cuál fue definido. En caso de haber un agente que no sea parte del scope del intervalo, pero que ejecute una acción con el mismo nombre que las que dan inicio y fin al mismo, el intervalo no se verá afectado y la acción seguirá su curso normal.

Es importante destacar que si bien el intervalo se define para cierto rol, pueden definirse acciones para agentes por fuera de los que el intervalo alcanza y que su ejecución dependa de que el mismo esté activo. En el caso del ejemplo de la motivación, se puede pedir que haya un agente controlando al momento de estar activo el happy hour separar algún incidente que puede existir. En este caso, el agente con rol `seguridad`, podría definir la acción `separar espectadores` que tenga validez sólo cuando el intervalo `happy hour regular` esté activo, si bien el agente seguridad no pertenece al alcance del intervalo definido para el rol `vendedor regular`.

Por otro lado, supongamos que tenemos el rol A, que tiene como hijos al rol B y C. Supongamos que los tres agentes de la jerarquía (agenteA, agenteB, agenteC) pueden ejecutar las acciones `Inicio` y `Fin`. Queremos definir un intervalo compartido para estos roles, en donde sólo el agenteA pueda dar comienzo al intervalo ejecutando la acción Inicio pero cualquiera de los agentes pueda finalizarlo ejecutando la acción Fin. Junto a lo descripto previamente se agrega al intervalo compartido la posibilidad de definir para cada acción si se aplica a todos los agentes del rol raíz para los que la acción está definida, o simplemente para el agente asociado al rol del intervalo.

3.2.2. Extensión

Al igual que como hicimos en los contadores a nivel de roles, extendemos el lenguaje agregando la siguiente sintaxis posible al momento de definir un intervalo.

shared `interval intervalName` *for roleName* `defined by actions init - end`

De esta forma, se agrega `shared` como modificador de alcance del intervalo que junto con la expresión `for roleName`, definen un intervalo a nivel de roles. Luego, todos los agentes que que tengan como rol a `roleName` o alguno de sus subroles, y puedan ejecutar las acciones de `init` o `end` darán inicio o fin al intervalo `intervalName`.

Por otro lado, análogo a como se hizo con los contadores compartidos damos la posibilidad de especificar que cada acción pueda ser ejecutada sólo para el rol que define al intervalo. Esto lo logramos extendiendo la sintaxis con:

```
 shared interval intervalName for roleName
defined by actions init1, init2 without further refinment -
end1 without further refinment, end2
```

De esta forma, agregando la frase `without further refinment` a cualquiera de las acciones que define al intervalo, se puede especificar que la acción no se expanda para todos los hijos del rol definido.

3.3. Implementación

La implementación de los nuevos tipos de intervalos se basa en transformar la expresión de la definición del mismo a una expresión más básica la cuál el programa ya sabe cómo resolver. Llamamos `rol raíz` al rol para el cuál se define el intervalo y procedemos de la siguiente manera.

Primero se calculan todos los agentes que pueden interactuar con el intervalo. Para esto se parte del rol raíz y se calculan todos los agentes que tengan alguno de sus roles, a este rol o alguno de sus subroles. Luego, para cada una de las acciones que dan inicio y fin al intervalo, se calculan cuáles de estos agentes pueden ejecutarlas, y ya uniendo todos los datos, se define al nuevo intervalo basado exclusivamente en las acciones ejecutables por los agentes, ya sin importar para el rol que fue definido.

Partiendo del ejemplo de la motivación, podemos definir a los roles simplificandos para el ejemplo como

```
roles empleado {
   seguridad, vendedor_regular, {
      vendedor_regular_bebidas, vendedor_regular_comida disjoint cover
   } vendedor_vip disjont cover
} disjoint cover
```

Luego, se desprenden de esta definición los siguientes agentes

```
agent1: vendedor_regular_bebidas, vendedor_regular, empleado
agent2: vendedor_regular_comida, vendedor_regular, empleado
agent3: vendedor_vip, empleado
agent4: seguridad, empleado
```

Y tenemos definidas las acciones

```
action iniciar_happy_hour, finalizar_happy_hour
```

Definimos al intervalo de roles de los happy hour regulares como

```
shared interval happy_hour_regular for vendedor_regular
defined by actions iniciar_happy_hour - finalizar_happy_hour
occurrences 1
```

Entonces, para transformar el intervalo a una expresión más básica, el programa analiza los agentes que tienen a `vendedor_regular` o alguno de sus subroles como rol y que ejecuten las acciones definidas para el intervalo. De esa forma, para este ejemplo la transformación queda de siguiente forma.

```
interval happy_hour_regular
defined by actions agent1.iniciar_happy_hour, agent2.iniciar_happy_hour
- agent1.finalizar_happy_hour, agent2.finalizar_happy_hour
occurrences 1
```

Todas las propiedades de cantidad de ocurrencias, la capacidad de ocurrir sólo en el lapso de otro intervalo, la posibilidad de iniciar activo o de nunca finalizar se mantienen vigentes para este nuevo tipo de intervalo como para los anteriores.

Para resolver el modificador `without further refinment`, simplemente se omite a todos los agentes con roles hijos para el cuál se define el intervalo, dejando como única acción la del agente del rol raíz. Al ser el modificador propio de cada acción, este cálculo se realiza para cada una pudiendo una acción estar definida para un sólo agente y otras acciones definidas para varios.

3.4. Ejemplo

Vamos a modelar un ejemplo donde se muestra el uso de los intervalos compartidos. Extendemos el ejemplo de la planteado en la sección de contadores con los siguientes requerimientos.

Al recital para el cuál se vendieron las entradas (regulares y VIP), ahora desea vender bebidas y comidas. Tras un análisis del mercado, la organización decide implementar happy hours, intervalos de tiempo en donde se podrá vender bebidas alcohólicas y hamburguesas a dos por el precio de una. Se decide que haya tres de estos momentos en la sección vip, pero solamente uno en la sección regular. Para enviar los disturbios durante estos períodos, se decide poner empleados de seguridad a separar problemas y evitar hechos de violencia.

Basados en todo el comportamiento descripto, podemos modelar las condiciones de la siguiente manera.

```
roles compradores, empleado {
   seguridad, vendedor_regular, {
      vendedor_regular_bebidas, vendedor_regular_comida disjoint cover
   } vendedor_vip {
      vendedor_vip_bebidas, vendedor_vip_comida disjoint cover
   } disjont cover
} disjoint cover}


actions comprar_entrada only performable by compradores


impersonal actions iniciar_recital, finalizar_recital


global interval recital defined by actions iniciar_recital - finalizar_recital



actions iniciar_happy_hour, finalizar_happy_hour for empleado


shared interval happy_hour_regular for vendedor_regular
defined by actions iniciar_happy_hour - finalizar_happy_hour
occurrences 1



shared interval happy_hour_vip for vendedor_vip
defined by actions iniciar_happy_hour - finalizar_happy_hour
occurrences 3



actions vender2x1 only occurs in scope happy_hour_regular
only performable by vendedor_regular
action sacar_espectador only occurs in scope happy_hour_regular only performable
by seguridad
```

4. PERMISOS CONDICIONALES

4.1. Motivación

Al momento de definir las cláusulas que definen a una norma legal puede, surgir la necesidad de definir condiciones sobre las mismas, es decir, indicar que las clausulas no son de cumplimiento irrestricto sino que sólo se activan bajo ciertas condiciones. Cuando se trata de permisos llamamos a estas clausulas `permisos condicionales` y los podemos clasificar en diferentes variaciones cómo se explican a continuación [14].

Consideremos el reglamento de una universidad, en donde un estatuto indica que permite a todos los alumnos mayores de 21 años de edad votar en las elecciones para elegir al Decano de su Facultad. Lo que esta regla quiere expresar no es prohibir a los estudiantes menores de 21 años votar para Decano, sino que con tener al menos de 21 años se tiene garantizado la posibilidad de participar en las votaciones. Entonces si alguna Facultad lo desea, por ejemplo, le podría dar a sus alumnos la posibilidad de votar a partir de los 18 años, sin que esto contradiga el permiso reglamentado por la universidad. En este caso, estamos definiendo un permiso donde la condición es `suficiente` para que el permiso se cumpla, pero no es necesaria cumplir la condición para que el permiso sea otorgado. Siguiendo la nomenclatura definida en [14] llamaremos a este tipo de permiso como `permiso condicional de tipo 1`.

Ahora consideremos una regla para pasajeros de un avión que indica que los pasajeros podrán desabrocharse los cinturones cuando la luz roja esté apagada. En este caso, lo que la regla quiere expresar no es simplemente que el pasajero puede desabrocharse el cinturón cuando la luz roja esté apagada, sino que además debe tener abrochado el mismo si la luz roja está encendida. Luego la regla no sólo indica que de cumplirse la condición se le otorga el permiso al pasajero, sino que también indica que de no cumplirse la misma, el permiso se transforma en una prohibición. Decimos entonces que este es un permiso donde la condición es `necesaria y suficiente` y llamamos a este tipo de permiso `permiso condicional de tipo 2`.

Por último, consideremos el caso de ventas de entradas para un recital. Podemos tener varias condiciones para la adquisición de una entrada. Por ejemplo, se puede requerir que los tickets deben ser comprados con al menos 48 horas de anticipación y también que los tickets pueden ser comprados siempre y cuando haya asientos disponibles. En este caso, ambas condiciones deben cumplirse para que el permiso sea otorgado y no alcanza simplemente con cumplir alguna de las condiciones por separado. En este caso, estamos definiendo condiciones `necesarias` para que el permiso se cumpla y llamamos a este caso `permiso condicional de tipo 3`.

Sobre estos tipos de permisos condicionales, al momento de comenzar la tesis estaba implementado solamente el de tipo 1. En las siguientes secciones se mostrará la forma de

uso elegida y la implementación de los tipo 2 y 3 recientemente explicados.

4.2. Permiso condicional de tipo 2

4.2.1. Desarrollo

Los permisos condicionales de tipo 2 son los permisos en donde la condición es la vez necesaria y suficiente. Es decir, si la condición se cumple el permiso es otorgado y a su vez, si la condición no se cumple el permiso se transforma en una prohibición.

Partiendo del ejemplo de las motivaciones, definimos la condición `luz_roja_apagada` y el permiso `P(desabrochar_cinturon)`. Entonces, nos interesa expresar que valga `P(desabrochar_cinturon)` cuando se dé la condición `luz_roja_apagada` y `F(desabrochar_cinturon)` cuando se dé la condición $\neg$`luz_roja_apagada`.
Es decir, se busca que si se cumple la condición de tener la luz roja apagada, el pasajero puede desabrocharse el cinturón, pero de no cumplirse la condición, está prohibido desabrocharlo.

Generalizando el ejemplo, partiendo de una condición `C` y un un permiso `P(`α`)` interpretamos el permiso condicional de tipo 2 para el permiso α bajo la condición `C` como permitir α cuando se cumple la condición `C` y como prohibir α cuando no se cumple la condición `C`.

Para poder expresar esta idea en una forma aceptada por FL, vamos a dividirla en dos fórmulas del lenguaje. Podemos pensar en que cuando una condición implica un permiso, estamos ante un caso de permiso condicional de tipo 1, y tal como se ve en [11] se deriva que la fórmula se puede escribir como $P(\alpha \text{ given that } C) = P(C \wedge \alpha)$.

La fórmula de la prohibición lo que nos está diciendo es que está prohibido que se dé la condición C y se ejecute la acción, entonces esto lo podemos escribir como una regla de tipo F. Queremos que en toda traza siempre que valga $\neg C$ esté vigente la prohibición $F(\alpha)$. Es decir, queremos que valga $\Box(\neg C \implies F(\alpha))$. Traducimos $F(\alpha)$ en $\Box(\neg\alpha)$, haciendo que en toda traza deba valer siempre que $\Box(\neg C \implies \Box(\neg\alpha))$.

Usando que $p \implies q$ es equivalente a $\neg p \vee q$ nos queda que siempre debe valer $\Box(C \vee \Box(\neg\alpha))$. Podemos sacar el $\Box$ de adentro y nos queda $\Box(C \vee \neg\alpha)$ es decir, que en ninguna traza valga $\Diamond\neg(C \vee \neg\alpha)$ es decir que en ninguna traza valga $\Diamond(\neg C \wedge \alpha)$. Es decir que en toda traza valga $\Box\neg(\neg C \wedge \alpha)$. Es decir que en toda traza valga $F(\neg C \wedge \alpha)$.

Entonces las dos fórmulas planteadas las podemos reescribir para que sean fórmulas válidas lenguaje FL cómo

$$P(\alpha \text{ given that } C) = P(C \wedge \alpha)$$

$$F(\neg C \wedge \alpha)$$

La primera cláusula indica la condición suficiente del permiso, y la segunda cláusula muestra la condición necesaria del mismo.

4.2.2. Implementación

Para aceptar este tipo de permisos, extendemos el lenguaje de las clausulas del sistema agregando la expresión `P(`α `only when C)` donde α es una acción o expresión del lenguaje y C una condición del mismo, ambos aceptados dentro del marco teórico existente.

Luego, volviendo al ejemplo planteado, podemos definir la regla del avión como `P(desabrochar_cinturon only when !INSIDE(luz_roja_prendida))` donde luz_roja_prendida puede ser un intervalo global del sistema e INSIDE es un predicado para intervalos que indica si está activo o inactivo.

Basado en lo explicado en la sección anterior, este permiso se va a dividir en dos cláusulas. Por un lado, se va a agregar a las reglas del sistema actual, la prohibición `F(INSIDE(luz_roja_prendida) & desabrochar_cinturon)`. Por otro lado, se va a agregar `P(!INSIDE(luz_roja_prendida) & desabrochar_cinturon)` a los permisos del sistema.

Luego, para la prohibición agregada lo que se busca es que el modelo sea siempre valido para todas las trazas posibles. Es decir se pide que para cualquier traza T
`T |= []` $(\neg$`((luz_roja_prendida = ACTIVE)` $\wedge$ `(desabrochar_cinturon = JUST_HAPPENED)))`

Entonces, ya habiendo agregado la nueva prohibición al sistema, para comprar que el permiso sea posible se va a buscar que exista una traza T tal que
`T |= <>(`$\neg$`(luz_roja_prendida = ACTIVE)` $\wedge$ `(desabrochar_cinturon = JUST_HAPPENED))`

Por otro lado, al no tener sentido lógico definir dos permisos diferentes que sean a su vez necesarios y suficientes para una misma acción o expresión α, se hace que el programa tire un error al intentar hacerlo e indique el problema en cuestión.

4.3. Permiso condicional de tipo 3

4.3.1. Desarrollo

Los permisos condicionales de tipo 3, son los permisos en donde se plantean un conjunto de condiciones relativas a otorgar un permiso y en donde las condiciones no necesariamente ocurren todas juntas en el texto. Cada condición por separado es necesaria para otorgar el permiso, pero el conjunto de condiciones resulta suficiente.

Volviendo al ejemplo dado en la sección anterior, definimos las condiciones `mas_de_48hs` y `asientos_disponibles` y el permiso `P(comprar_entrada)` podemos entender la regla cómo:

$$(mas\_de\_48hs \wedge asientos\_disponibles) \implies P(comprar\_entradas)$$

Generalizando el ejemplo, tenemos que dando un conjunto de condiciones $C_1, C_2, ..., C_n$ y un permiso $P(\alpha)$ interpretamos el permiso condicional de tipo 3 para el permiso α bajo las condiciones $C_1, C_2, ..., C_n$ cómo

$$C_1 \wedge C_2 \wedge ... \wedge C_n \implies P(\alpha)$$

De esta forma, si definimos $C = C_1 \wedge C_2 \wedge ... \wedge C_n$ podemos ver que este tipo condicional es equivalente al de tipo 1 para la condición C y usando el C definido como la conjunción de todas las condiciones necesarias, podemos escribir el permiso como

$$P(\alpha \text{ given that } C) = P(C \wedge a) = P(C_1 \wedge C_2 \wedge ... \wedge C_n \wedge \alpha)$$

4.3.2. Implementación

Para incorporar los permisos condicionales de tipo 3, vamos a extender el lenguaje de las cláusulas dando la posibilidad de definir varios permisos condicionales de tipo 1 sobre una mimsa acción o expresión del lenguaje. Es decir, dado α es una acción o expresión del marco teórico y C_i $i \in 1..n$ condiciones del mismo, se permite escribir n permisos condicionales `P(`α `given that` C_i`)` por separado, tal que se interpreten todos como un permiso condicional de tipo 1 de la forma `P(`α `given that` C_1 `&` ... `&` C_n`)`.

Por ejemplo en el caso del ejemplo planteado podemos definir la cláusula `P(comprar_entradas given that INSIDE(48hs_antes_del_evento))` y la cláusula `P(comprar_entradas given that entradas_vendidas <100)`, donde comprar_entradas es una acción, 48hs_antes_del_evento es un intervalo y entradas_vendidas es un contador definidos para el ejemplo.
Luego, dado este permiso, se interpretará como el permiso condicional de tipo 1 `P(comprar_entradas given that INSIDE(48hs_antes_del_evento) & entradas_vendidas <100)` que terminará traduciéndose como

```
P(INSIDE(48hs_antes_del_evento) & entradas_vendidas <100 & comprar_entradas
```

Entonces, lo que se va a buscar es que exista una traza T tal que

```
T |= <>(48hs_antes_del_evento = ACTIVE ∧
        entradas_vendidas <100 ∧
        desabrochar_cinturon = JUST_HAPPENED)
```

Cabe aclarar que para el desarrollo de esta tesis los permisos condicionales de tipo 3 se van a agrupan cuando los α de los diferentes permisos condicionales coincide sintácticamente y que eso deja pendiente los casos en los que la acción se escribe de maneras semánticamente equivalentes.

Al momento de correr el programa se emite un mensaje indicando todos los permisos condicionales de tipo 3 que se combinaron para formar un permiso condicional de tipo 1. Esto se realiza para para evitar el caso en los que escribir `P(`α `given that` C_1`)` y `P(`α `given that` C_2`)` es en realidad un error más que la intención de escribir `P(`α `given that` C_1 `&` C_2`)`.

5. EJEMPLO INTEGRADOR

5.1. Introducción

A continuación se va a mostrar a través de un ejemplo integrador cómo las nuevas extensiones agregadas al lenguaje, tanto en el marco teórico como en las cláusulas, agregan expresividad al mismo permitiendo un análisis más profundo y amplio de las normas legales.

5.2. Ejemplo

Vamos a definir un reglamento para la inscripción y uso de un gimnasio.

1. `Requisitos para inscribirse al gimnasio`
   - *a*) Podrán hacerse miembros del gimnasio personas mayores de 18 años
   - *b*) Deberán tener tarjeta de crédito para ser miembros
   - *c*) Se considera miembro activo a aquel miembro que tenga la cuota mensual paga
   - *d*) Existen dos tipos de membresía, la clásica y la VIP
   - *e*) Se deberá presentar un certificado médico obligatorio para hacer uso de las instalaciones
   - *f*) Los miembros activos podrán hacer uso de las instalaciones
2. `Requisitos para acceder al gimnasio`
   - *a*) Sólo se puede acceder al gimnasio cuando el mismo esté abierto
   - *b*) Sólo pueden acceder al gimnasio miembros activos
3. `Requisitos para acceder a la pileta`
   - *a*) Sólo pueden acceder a la pileta cuando la pileta esté abierta
   - *b*) No puede haber más de 20 miembros en la pileta
   - *c*) Debe haber un guardavidas durante el tiempo en que la pileta está abierta
4. `Requisitos para las clases`
   - *a*) Hay clases de spinning de cupo limitado
   - *b*) Hay que inscribirse a las clases de spinning para asistir
   - *c*) Durante la duración de las clases, los asistentes podrán hacer uso de las bicicletas

5. `Requisitos para el bar`

   a) Los miembros que deseen pueden hacer uso del bar del gimnasio

   b) A todos los miembros regulares se les ofrecerá una bebida gratuita por día

   c) A todos los miembros VIP se les ofrecerá dos bebidas gratuitas por día

   d) Hay una cantidad limitadas de 100 bebidas gratuitas diarias

5.3. Desarrollo

Basados en los términos y condiciones para la inscripción y uso del gimnasio vamos a hacer la transcripción de la misma al lenguaje FL y analizar su consistencia.

5.3.1. Roles

Empezamos definiendo la jerarquía de roles que salen de analizar todo el documento y de la cuál se van a desprender los diferentes agentes involucrados en el mismo.

Basado en todos los puntos del documento definimos la jerarquía como

```
roles
  miembros {
     normal, VIP disjoint cover
  }, empleados disjoint cover
```

Analizando esta jerarquía en FL, se desprende los siguientes agentes

```
agent_without_role: no_assigned_role
agent_1: empleados
agent_2: miembros, VIP
agent_3: miembros, normal
```

Estos son los agentes que se utilizarán a lo largo de todo el desarrollo.

En cuanto a la posibilidad de un miembro de ser activo o inactivo lo vamos a definir en base a un intervalo local como veremos en la siguiente sección.

5.3.2. Inscripción

Ahora modelamos los requisitos para inscribirse al gimnasio. Empezamos por definir las acciones que vamos a necesitar para definir las diferentes cláusulas.

```
impersonal action pasa_un_anio
impersonal action pasa_un_mes
```

```
actions cumplir_18, pagar_cuota, activar_tarjeta,
   desactivar_tarjeta, presentar_certificado
   inscribirse, usar_instalaciones only performable by miembros
```

Luego definimos los intervalos a utilizar. En este caso queremos definir características sobre cada individuo, así que todos los intervalos serán de tipo local.

```
local interval mayor_de_edad defined by actions cumplir_18 - infinite
local interval tajeta_activa defined by actions activar_tarjeta - desactivar_tarjeta
local interval miembro_activo defined by actions pagar_cuota - pasa_un_mes
local interval certificado_medico_activo defined by actions presentar_certificado
- pasa_un_anio
```

Ahora ya estamos en condición de definir las diferentes cláusulas para la sección de inscripción al gimnasio. Las siguientes cláusulas definen quiénes pueden hacerse miembros del gimnasio.

```
FORALL(i:miembros; P(i.inscribirse given that INSIDE(i.mayor_de_edad)))
FORALL(i:miembros; P(i.inscribirse given that INSIDE(i.tajeta_activa)))
```

Estas cláusulas se interpretan como permisos condicionales necesarios para hacerse miembro, y el programa lo interpretará de tal forma que se busque una traza T tal que:

```
T |=
   <>(agent_2.mayor_de_edad = ACTIVE ∧
      agent_2.tajeta_activa = ACTIVE ∧
      agent_2.inscribirse = JUST_HAPPENED) ∧
   <>(agent_3.mayor_de_edad = ACTIVE ∧
      agent_3.tajeta_activa = ACTIVE ∧
      agent_3.inscribirse = JUST_HAPPENED) ∧
   <>(agent_4.mayor_de_edad = ACTIVE ∧
      agent_4.tajeta_activa = ACTIVE ∧
      agent_4.inscribirse = JUST_HAPPENED) ∧
```

Es interesante notar que previo a la realización de esta tesis la interpretacion de estos dos permisos iban a tratarse como dos permisos completamente diferentes o cual llevaba que se buscaran trazas que cumplieran una u otra condición, pero no ambas. Al agregar los permisos condicionales de tipo 3, agregamos esta posibilidad permitiendo escribir la especificación de la norma en una forma mucho mas parecida a la que la misma esta escrita.

Por otro lado se habla sobre el uso de las instalaciones. Dice que se deberá presentar un certificado médico obligatorio para hacer uso de las instalaciones. Esto lo podemos ver como un permiso condicional necesario y suficiente (tipo 2) siendo que lo que se intenta indicar es que sin el certificado no se pueden hacer uso de las instalaciones, y siendo éste además el único requisito que se indica, lo podemos ver como. Esto lo escribimos como `FORALL(i:miembros; P(i.usar_instalaciones only when INSIDE(i.certificado_medico_activo)))` lo cual se va a traducir como dos cláusulas.

```
FORALL(i:miembros; P(INSIDE(i.certificado_medico_activo) ∧ i.usar_instalaciones))
```

```
FORALL(i:miembros; F(¬INSIDE(i.certificado_medico_activo) ∧ i.usar_instalaciones))
```

Por otro lado hay una cláusula que dice que los miembros activos podrán hacer uso de las instalaciones. Esto lo escribimos como `FORALL(i:miembros; P(i.usar_instalaciones given that INSIDE(i.miembro_activo)))`.

Al ejecutar el programa nos dice que no se encontró un comportamiento legal válido. Analizando un poco los requisitos, nos damos cuenta de que estamos definiendo para un mismo conjunto de agentes y acción un permiso condicional de tipo 2 (necesario y suficiente) y uno de tipo 1 (suficiente). Esto no está permitido ya que no tiene un sentido lógico tener una condición suficiente teniendo ya definida una necesaria y suficiente. El programa entiende esta inconsistencia y nos lo informa.

Podemos intentar resolver este problema reescribiendo las cláusulas como un permiso condicional de tipo 3 (necesario) y una prohibición.

```
FORALL(i:miembros; P(i.usar_instalaciones given that
      INSIDE(i.certificado_medico_activo)))
FORALL(i:miembros; P(i.usar_instalaciones given that INSIDE(i.miembro_activo)))
FORALL(i:miembros; F(¬INSIDE(i.certificado_medico_activo) ∧ i.usar_instalaciones))
```

Si volvemos a correr el programa con las nuevas cláusulas nos indica ahora sí, que se puede encontrar un comportamiento válido del mismo.

5.3.3. Sobre el uso de la pileta

Ahora modelamos las condiciones para hacer uso de la pileta. La principal restricción radica en la cantidad de miembros que pueden hacer uso en simultaneo de la misma. Empezamos por modelar las acciones que los agentes puedan realizar.

```
actions abrir_area_pileta, cerrar_area_pileta only performable by empleados
global interval pileta_abierta defined by actions
   abrir_area_pileta - cerrar_area_pileta
actions entrar_pileta, salir_pileta
```

Definimos un intervalo global para indicar que la pileta esta abierta. De esta manera tanto miembros como empleados podrán acceder a la misma. Cómo los miembros sólo pueden entrar y salir de la pileta cuando la misma se encuentra abierta, definimos las siguientes prohibiciones.

```
FORALL(i:miembro; F(i.entrar_pileta given that !INSIDE(pileta_abierta)))
FORALL(i:miembro; F(i.salir_pileta given that !INSIDE(pileta_abierta)))
```

A continuación definimos los contadores compartidos a nivel de roles que nos ayuden a limitar las acciones y que nos van a servir para ser usados en las cláusulas.

```
 shared counter miembros_en_pileta for miembros init value 0,
```

```
  increases with action entrar_pileta by 1,
  decreases with action entrar_pileta by 1,
  reaching max impedes actions

shared counter miembros_en_pileta for empleados init value 0,
  increases with action entrar_pileta by 1,
  decreases with action entrar_pileta by 1,
  reaching max impedes actions
```

Definimos un contador para los miembros, que va a limitar en 20 la cantidad máxima de miembros posibles a través de la siguiente cláusula.

```
F(miembros_en_pileta > 20)
```

Hace que esté prohibido dejar entrar a más miembros que el máximo definido por este contador compartido. Sin embargo, si van a poder seguir entrado empleados del gimnasio a la pileta, una vez llegado al máximo de miembros. Vale la pena destacar que previo a esta tesis para lograr este comportamiento había que definir varias acciones y contadores adicionales, y ahora lo podemos hacer de forma sencilla gracias a la incorporación al lenguaje de los contadores compartidos a nivel de roles.

Definimos el contador para los empleados para poder usarlo en nuestra definición de las cláusulas en esta parte del reglamento. Se pide que haya un guardavidas durante el tiempo en que la pileta este abierta. Para esto definimos definimos que

```
O(INSIDE(pileta_abierta) ⟹ guardavias_en_pileta >0)
```

De esta forma, y usando el contador para de empleados podemos definir las condiciones que piden que exista al menos un empleado en la pileta a la hora de abrir la misma.

5.3.4. Requisitos para las clases

Ahora modelamos las condiciones para las clases de spinning. Lo interesante a modelar es la posibilidad de hacer uso de las bicicletas exclusivamente cuando la clase de spinning esté siendo dada. Las clases las pueden iniciar o finalizar cualquier empleado del gimnasio y el uso de las bicicletas está restringido para los miembros que se hayan inscripto a la clase.

Empezamos modelando las posibles acciones a ejecutar por los agentes del sistema.

```
actions iniciar_clase, finalizar_clase only performable by empleados
actions inscribirse_a_clase, salir_de_clase only performable by miembros
```

La inscripción a la clase se podrá hacer durante el momento en que la clase no se esté llevando a cabo, con lo cual definimos el siguiente intervalo para poder realizar la inscripción.

```
global interval inscripcion_abierta defined by actions infinite, finalizar_clase
- iniciar_clase
```

Y para cada miembro vamos a querer saber si está inscripto a la clase de spinning correspondiente. Para eso definimos un intervalo local que nos va a permitir saber si se encuentra inscripto a no.

```
local interval inscripto_a_clase defined by actions
inscribirse_a_clase - salir_de_clase, finalizar_clase
```

Luego definimos el intervalo que indica que la clase está siendo dada. La clase puede comenzarse y finalizarse por cualquier agente de tipo empleado del gimnasio, por lo cuál vamos a definirlo como un intervalo a nivel de roles de la siguiente manera.

```
shared interval clase_spinning for empleados defined by actions iniciar_clase
- finalizar_clase
```

Por último definimos la acción para poder usar las bicicletas. Las bicicletas sólo se podrán usar durante el momento en que la clase esté siendo impartida, por lo cuál restringimos la acción para estar disponible durante ese intervalo.

```
action usar_bicicleta
FORALL(i:miembros; P(i.usar_bicicleta given that !INSIDE(clase_spinning)))
```

También debemos agregar ciertas cláusulas para asegurar que las bicicletas sean usadas por miembros inscriptos a la clase.

```
FORALL(i:miembros; P(i.usar_bicicleta given that INSIDE(i.inscripto_a_clase)))
```

5.3.5. Sobre el uso del bar

Para finalizar modelamos las condiciones del uso del bar del gimnasio. En esta sección lo importante a destacar es la posibilidad de modificar un contador compartido con diferentes valores dependiendo de quién es el agente que ejecuta cierta acción.

Primero definimos la acción de los miembros para pedir bebidas en el bar.

```
action pedir_bebida_gratuita only performable by miembros action pedir_dos_bebidas_gratuita
only performable by VIP
```

Luego definimos el intervalo compartido de las bebidas gratuitas disponibles para ofrecer a los diferentes miembros.

```
shared counter bebidas_gratuitas_disponibles for miembros
  init value 100 min value 0
  decreases with action pedir_bebida_gratuita for normal by 1,
  decreases with action pedir_bebida_gratuita for VIP by 1,
```

```
  decreases with action pedir_dos_bebidas_gratuitas for VIP by 2,
  reaching min impedes actions
  resets with action pasa_un_dia
```

De esta forma definimos el intervalo que cada día permite ofrecer al gimnasio hasta 100 bebidas gratuitas a sus miembros, otorgando una bebida a los miembro regulares y una o dos bebidas a los miembros VIP que lo deseen.

Por otro lado queda limitar la cantidad de consumiciones diarias por miembro. Para esto definimos un intervalo local para limitar a cada agente.

```
 local counter bebida_por_miembro init value 0
   increases with action pedir_bebida_gratuita by 1,
   increases with action pedir_dos_bebidas_gratuitas by 2,
   resets with action pasa_un_dia
```

Y las restricciones según el tipo de miembro.

```
FORALL(i:regular; F(i.bebida_por_miembro > 1))
FORALL(i:VIP; F(i.bebida_por_miembro > 2))
```

De esta forma modelamos que no se entregan más de 100 bebidas por día y que ningún miembro puede pedir más de una bebida y ningún miembro VIP más de dos bebidas por día.

6. CONCLUSIONES Y TRABAJO FUTURO

Partiendo del trabajo realizado en [1, 11], la idea principal era extender el poder expresivo y la flexibilidad de la herramienta FormaLex para poder modelar lo más fielmente posible normas legales y analizar automáticamente sus eventuales inconsistencias.

La posibilidad de pasar a disponer con contadores a nivel de roles permite controlar los agentes que pueden modificar a un cierto contador, posibilidad que antes no existía. Junto con esto ahora también es posible especificar diferentes valores en los que se modifica el contador dependiendo del agente que ejecute la correspondiente acción. No había una forma directa de realizar esto antes, teniendo que encontrar ciertos trucos como crear acciones distintas ofuscando la especificación y perjudicando la performance.

También se agregó la posibilidad de utilizar intervalos a nivel de roles que da la posibilidad de que dos agentes con roles diferentes puedan dar inicio y fin a un intervalo, pero que otro agente con otro rol ejecutando la misma acción no tenga efecto en el intervalo. Esto da la posibilidad de definir intervalos donde intervengan sólo cierta parte de la jerarquía de roles en su accionar que suele darse seguido a la hora de definir normas legales.

Agregar los tipos de permisos condicionales permite completar todas las definiciones de permisos condicionales definidos en [14] y abre la posibilidad a describir normas legales mucho mas fieles a la realidad y en forma más clara. El permiso condicional suficiente agrega la posibilidad de encontrar inconsistencias a la hora de pedir más de una condición sobre una fórmula mientras que el permiso el permiso necesario y suficiente permite especificar las restricciones de una fórmula en diferentes cláusulas.

Creemos que las extensiones realizadas llevan a lograr una mejora de la herramienta tanto en el poder expresivo de la misma como también facilitar la escrituras de ciertas cláusulas facilitando la traducción de la norma legal al lenguaje FL, y son un paso más en el objetivo final que tiene la herramienta FormaLex para el análisis automático de normas legales.

Aún quedan varios pasos para poder cumplir con el objetivo final de la herramienta que se fueron detectando en este trabajo o en trabajos anteriores.

Un área donde es posible mejorar la herramienta es en los resultados que devuelve el model checker. De no encontrar un comportamiento legal válido hoy en día no es posible saber cuál es el motivo de la inconsistencia o dar una orientación sobre cuál es el problema de la misma. Poder contar con esta información sería de suma utilidad para los usuarios de la herramienta ya que facilitaría entender el problema de la norma.

Otro área importante para mejorar es la velocidad de la herramienta. Se hizo mucho énfasis en este punto en [6], pero aún queda mucha investigación y desarrollo posible cómo se indica es la tesis en cuestión.

Por último es importante destacar que FL puede seguir siendo extendido tanto en su poder expresivo como en la claridad para definir normas legales. Existe una lista donde se detalla punto por punto algunas necesidades que surgieron del uso de la herramienta tanto por usuarios como por los trabajos anteriores, ver [1, 11]. Agregando a esa lista, a lo largo de esta tesis surgió la necesidad de poder comparar cláusulas en forma semántica. Por ejemplo al momento de definir condiciones necesarias puede darse que `P(a given that C)` y `P(!!a given that D)` y el programa debería entender que $a = !!a$ y unir ambas cláusulas. Hoy en día esa comparación es solamente sintáctica.

Bibliografía

[1] J. P. Benedetti. Mejorando formalex, una herramienta de análisis y detección automática de defectos normativos. Master's thesis, Depto de Computación, Facultad de Ciencias Exactas y Naturales, UBA, jun 2014.

[2] P. e. Blackburn, J. e. van Benthem, and F. e. Wolter. *Handbook of modal logic*. Studies in Logic and Practical Reasoning 3. Amsterdam: Elsevier., 2007.

[3] J. Breuker, E. Petkov, and R. Winkels. Drafting and validating regulations: The inevitable use of intelligent tools. In *AIMSA '00: Proceedings of the 9th International Conference on Artificial Intelligence*, pages 21–33, London, UK, 2000. Springer-Verlag.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.

[5] N. den Haan. Tools for automated legislative drafting. In *ICAIL '97: Proceedings of the 6th international conference on Artificial intelligence and law*, page 252, New York, NY, USA, 1997. ACM.

[6] C. Faciano. Optimizando el rendimiento de la herramienta formalex de análisis de documentos normativos. Master's thesis, Depto de Computación, Facultad de Ciencias Exactas y Naturales, UBA, jul 2016.

[7] S. Fenech, G. J. Pace, and G. Schneider. Clan: A tool for contract analysis and conflict discovery. In Z. Liu and A. P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 90–96. Springer, 2009.

[8] D. Gorín, S. Mera, and F. Schapachnik. Model Checking Legal Documents. In *Proceedings of the 2010 conference on Legal Knowledge and Information Systems: JURIX 2010*, pages 111–115, Dec. 2010.

[9] D. Gorín, S. Mera, and F. Schapachnik. A Software Tool for Legal Drafting. In *FLACOS 2011: Fifth Workshop on Formal Languages and Analysis of Contract-Oriented Software*, pages 1–15. Elsevier, 2011.

[10] G. Governatori and D. Pham. Dr-contract: An architecture for e-contracts in defeasible logic. *International Journal of Business Process Integration and Management*, 4(3):187–199, 2009.

[11] M. C. Gunski and M. G. Raiczyk. Formalex: Mejorando el poder expresivo del lenguaje `FL` para la detección de defectos normativos. Master's thesis, Depto de Computación, Facultad de Ciencias Exactas y Naturales, UBA, mar 2016.

[12] Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] G. Pace and F. Schapachnik. Permissions in contracts, a logical insight. In *The 24th International Conference on Legal Knowledge and Information Systems. University of Vienna, Austria*, Frontiers in Artificial Intelligence and Applications. IOS Press, sep 2011.

[14] G. J. Pace, F. Schapachnik, and G. Schneider. Conditional permissions in contracts. In *Legal Knowledge and Information Systems - JURIX 2015: The Twenty-Eighth Annual Conference, Braga, Portual, December 10-11, 2015*, volume 279 of *Frontiers in Artificial Intelligence and Applications*, pages 61–70. IOS Press, 2015.

[15] R. Winkels and N. D. Haan. Automated legislative drafting: Generating paraphrases of legislation. In *Proceedings of the Fifth International Conference on AI and Law*, pages 112–118. ACM, 1995.