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

# Mejorando FormaLex - Análisis y detección automática de defectos normativos

*Tesis de Licenciatura en Ciencias de la Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*

Juan Pablo Benedetti

*Director: Dr. Fernando Schapachnik*
*Co-director: Dr. Sergio Mera*
*Buenos Aires, mayo de 2014*

# Mejorando FormaLex - Análisis y detección automática de defectos normativos

**Resumen**

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, 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.

# Índice

# 1 Introducción

## 1.1 El problema

Una de las primeras etapas del desarrollo de un programa informático es la especificación, donde se define qué tiene que hacer el programa. Estas especificaciones suelen tener una inclinación deóntica, ya que indican los comportamientos permitidos y los prohibidos. Como estas especificaciones pueden tener errores que son difíciles de detectar por seres humanos, la Ingeniería del Software ha trabajado desde hace tiempo en un área llamada *verificación formal*, que consiste en el uso de técnicas lógico-matemáticas para encontrar dichos errores. Actualmente es una disciplina madura, que ha generado métodos, técnicas y herramientas que analizan con éxito casos de gran tamaño (e.g., [S98, GCM09]), permitiendo así encontrar tempranamente casos no cubiertos, contradicciones, etc.

Por otro lado, existe un área llamada *Automated Legislative Drafting* que busca la automatización del apoyo a la tarea legislativa. Usamos la palabra legislar en un sentido amplio, que incluye la producción de leyes, normativas y reglamentaciones.

Los intentos realizados en el área de *Automated Legislative Drafting* se han basado en la creación de nuevas herramientas totalmente desde cero sin aprovechar la amplia experiencia del mundo del software. El conjunto de herramientas que conforman FormaLex busca paliar esta situación, usando, para el campo de la legislación, herramientas de verificación formal ya maduras por su amplio uso.

## 1.2 Estado del arte

Los intentos de relacionar Lógica y Derecho son de larga data (eg, [Per68, VW63]), destacándose la posición de Kelsen, que en su “Teoría Pura del Derecho” [Kel34], presenta una visión de la ley cercana a un cuerpo de axiomas.

Muchos de los trabajos más recientes en el área están relacionados con la posibilidad de emitir veredicto automáticamente (eg, [vdVHB+08]), analizar textos en lenguaje natural para codificarlos automáticamente (eg, [LMPV09, VFLS08]), encontrar formalismos para representar el conocimiento legal (e.g., [HBBB07, ABF+07]), realizar análisis de argumentaciones (eg, [WBCA08]), etc.

El uso de herramientas informáticas para modelar y analizar sistemas legales, y de esta forma realizar sobre ellos verificación y validación automática de propiedades que se consideran deseables, es una temática que se conoce como *Automated Legislative Drafting.* Su principal objetivo es brindar asistencia informática al legislador a la hora de elaborar leyes o reglamentaciones.

La temática es actualmente abordada por una activa comunidad en la que se destacan:

- JURIX, la *International Conference on Legal Knowledge and Information Systems*, que celebrará su edición número 27 en 2014 (http://jurix.nl/announcing-jurix-2014/).
- ICAIL, la *International Conference on Artificial Intelligence and Law*, que va por su edición
  número 15 (http://sites.sandiego.edu/icail/call-for-papers/).
- DEON, la *Conference on Deontic Logic in Computer Science*, conferencia bianual que tendrá su décimo segunda edición en 2014 (http://www.deon2014.ugent.be/).
- Centros internacionales dedicados exclusivamente al tema, como el *Istituto di Teoria e Tecniche dell'Informazione Giuridica* (http://www.ittig.cnr.it/), de Italia y el *Leibniz Center for Law* (http://www.leibnizcenter.org/) de la Universidad de Amsterdan, en Holanda.

Esta comunidad acuerda que el análisis de consistencia normativa no puede reducirse a detectar la mera inconsistencia lógica [HPvdT07], por lo que se generaron muchos trabajos que tratan de abordar el problema desde distintas perspectivas:

- Una serie de autores (eg, [AvdHRA+09, MWD98, GRS05, BBF06, FMDR10, BDDM04, Pio10]) plantearon alguna combinación de lógica deóntica y lógica temporal. Sin embargo, no proveen ni herramientas ni formas de traducir sus formalismos a lógicas estándar, por lo que se confinan al ámbito teórico, a diferencia del enfoque de FormaLex, que busca ser también práctico.
- Otros autores han provisto herramientas, pero con un enfoque muy limitado: [SMjS03] se basa en la codificación manual de un contrato bajo la forma de FSM y no aborda de manera suficientemente amplia el problema de la inconsistencia normativa. [HT06] trata la detección de conflictos normativos haciendo que las reglas se codifiquen en una variación de Prolog, lo que le permite soportar ontologías, pero limita su análisis a la contradicción lógica.
- Más próximos al punto de vista de FormaLex se encuentran BCL [GP09] y CL [PS09]. BCL es un lenguaje de especificación de contratos que permite realizar monitoreo en tiempo de ejecución (para contratos electrónicos de software) y permite detectar conflictos entre reglas, entre otras características. Sin embargo, no provee soporte para razonamiento temporal, está basado en una herramienta ad-hoc llamada Dr. Contract, cuya performance no se analiza y no provee soporte para incorporar información factual al modelo deóntico.
- CL es un lenguaje basado en lógica dinámica que provee también una herramienta ad-hoc llamada CLAN [FPS09]. No permite incluir información factual y sufre de limitaciones prácticas de expresividad.

## 1.3 La propuesta de FormaLex

FormaLex tiene como objetivo modelar y analizar sistemas legales usando herramientas informáticas, para realizar sobre ellos verificación y validación automática de

propiedades que se consideran deseables. Apunta a normas de carácter administrativo, que hablen principalmente de permisos, prohibiciones, obligaciones y procedimientos.

Considerando el fuerte paralelismo que se puede establecer entre un documento normativo y una especificación de un sistema informático, se desarrolló FL, un lenguaje deóntico temporal con un fuerte componente práctico que permite reutilizar herramientas de inferencia ya existentes y que cuentan con amplio respaldo de la comunidad informática. Con FL se busca poder generar una traducción eficiente de un conjunto de normas a *Lógica Temporal Lineal* (LTL). Para ver con más detalle un análisis comparativo entre normativas legales y especificaciones de software, se puede consultar [GMS11].

FL intenta capturar las estructuras de razonamiento y representación más frecuentes en el ámbito legal. Es un lenguaje de uso específico, en el sentido que los conceptos legales que usualmente se manipulan en documentos normativos son entidades de primer orden. Por mencionar algunos ejemplos, la descripción de una falta a la regulación junto a su forma de repararla, la estipulación de plazos para cumplir una acción determinada o la descripción de intervalos temporales precisos en donde solo determinadas acciones son posibles, son algunos de los conceptos que poseen en FL construcciones sintácticas especialmente diseñadas para modelarlos.

A la hora de encontrar automáticamente problemas de coherencia, FormaLex utiliza como motor de inferencia *model checkers* estándar para LTL, que han sido ampliamente probados y su desempeño se ha ido incrementando en sucesivas versiones. Esto permite experimentar con varios *model checkers* simultáneamente (por ejemplo SPIN [Hol03], DiVinE [BB+06] y NuSMV [CCGR00]) y analizar los casos en donde uno se comporta mejor que otro. Si bien el estado de la herramienta es aún experimental, el análisis de coherencia demostró poder detectar varios problemas en algunos casos de estudio [GMS11].

## 1.4 Sus limitaciones y nuestra propuesta

FormaLex, al ser aún un prototipo, requiere la adecuada implementación de algunas de las funcionalidades especificadas para el lenguaje FL. Por otro lado, a medida que se fue usando la herramienta, han aparecido nuevas situaciones que no se podían modelar con FL. Más adelante se verá con más detalle cómo es actualmente FL y se mostrará alguna de sus limitaciones.

Con nuestra tesis nos proponemos ampliar el lenguaje FL, enriqueciéndolo para poder dar soporte a todas las nuevas situaciones que se fueron descubriendo durante su uso y poder describir nuevos conceptos útiles para el modelado de normas. Uno de los puntos principales que se incorporan, es la noción explícita de roles y un mecanismo de instanciación de los mismos. Además, adaptaremos la herramienta para dar soporte a todas las nuevas funcionalidades y presentaremos un caso de estudio para mostrar los resultados.

## 1.5 Estructura del documento

En el capítulo 2 presentaremos la vieja versión del lenguaje FL. Para ello haremos una muy breve introducción a la lógica modal, la lógica deóntica y la lógica temporal lineal y repasaremos el funcionamiento de los *model checkers*. Luego contaremos el objetivo principal del lenguaje FL, mencionaremos brevemente sus componentes más importantes y explicaremos cómo se usan. Presentaremos la sintaxis y la semántica de las reglas en FL.

El capítulo 3 está enfocado en la nueva versión de FL. Allí mostraremos en detalle todos los componentes y sus características. Contaremos cómo se traducen estos componentes a un autómata. Explicaremos cómo son las cláusulas que se pueden generar y cómo se traducen a lógica temporal lineal. Por último, hablaremos de nuestro programa, de cómo se usa y de cuáles son los pasos que ejecuta nuestra herramienta.

En el capítulo 4, presentaremos un caso de estudio, donde se mostrará cómo se usa nuestra herramienta y analizaremos el resultado que genera.

El capítulo 5 contiene las conclusiones del trabajo, las limitaciones de la herramienta y presenta posibles trabajos futuros para mejorarla y ampliarla.

Finalmente, en el capítulo 6 se encontrarán las referencias a los trabajos citados en el documento.

# 2 El viejo FL

## 2.1 Un poco de lógica modal, deóntica y temporal lineal

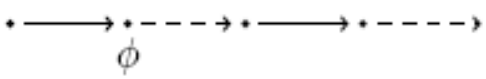
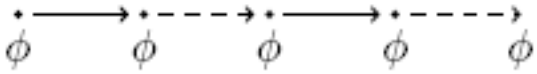
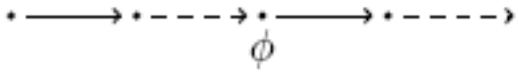
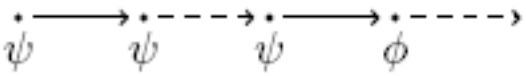
Existen varios lenguajes lógicos, cada uno de ellos con distinto poder expresivo, que describen distintos tipos de objetos matemáticos. Las **lógicas modales** son una familia de lógicas que, en su mayoría, predican sobre modelos representados por estados, valuaciones proposicionales y relaciones entre los estados. Su característica distintiva es contar con un conjunto de operadores modales que exploran el modelo de forma local y con cuantificaciones acotadas. Esto resulta en una familia de lógicas que suelen tener un muy buen comportamiento computacional, lo que permite desarrollar herramientas de aplicación práctica. Para ampliar este tema, ver [BRV02].

La **lógica deóntica** es un tipo de lógica modal que se usa para analizar formalmente normas o, con más precisión, proposiciones que tratan acerca de normas. Es el fruto de antiguos intentos por formalizar la idea de prohibición y obligación [VW63]. Mediante operadores modales se intenta describir qué es obligatorio, prohibido o permitido en determinado contexto. Estas lógicas, de las que se han desarrollado muchas variaciones, son muy adecuadas para modelar cuestiones relacionadas con el ámbito legal y han sido la base de todo el campo conocido como *Automated Legislative Drafting.*

A continuación daremos una definición informal de una lógica deóntica. Los operadores deónticos se usan de la siguiente manera, $O(\varphi)$ (de "obligation") significa que la fórmula $\varphi$ se cumple siempre. A partir de este operador y de la negación lógica se pueden definir los operadores de prohibición y permiso. $F(\varphi)$ (de "forbidden") significa que la fórmula $\varphi$ está prohibida. Una prohibición se puede ver como la obligatoriedad de lo contrario y por ello $F(\varphi)$ es equivalente a $O(\neg\varphi)$. Para representar que algo está permitido, usamos $P(\varphi)$ (de "permission"). Decimos que algo está permitido si no está prohibido, y por ello la fórmula $P(\varphi)$ es equivalente a $\neg F(\varphi)$ (o, lo que es lo mismo, equivalente a $\neg O(\neg\varphi)$). Por último, tenemos las obligaciones CTD. Una obligación CTD, por *Contrary-To-Duty Obligation*, se da en situaciones donde hay una obligación primaria y una segunda que llamamos reparación, que debe cumplirse si no se cumple la primera. La denotamos como $O(\varphi)$ repaired by $\rho$. Significa que $\varphi$ es obligatorio, pero si no se cumple, entonces es obligatorio $\rho$. Se dice que $\rho$ es la *reparación*, es decir, aquello que debe hacer el agente para seguir cumpliendo con la norma una vez que no se cumplió la primera obligación. Desde el punto de vista legal, $O(\varphi)$ repaired by $\rho$ significa que un comportamiento que no satisface $\varphi$, pero sí $\rho$, es legal, mientras que el mismo comportamiento es ilegal para la fórmula $O(\varphi)$, que no contempla una reparación. Volveremos a hablar de estos operadores más adelante.

La **lógica temporal lineal** (LTL) es un tipo de lógica modal usado para explicar cómo los valores de verdad de las proposiciones cambian a lo largo del tiempo. Sus modelos son siempre lineales, con una única relación de accesibilidad e incluye cuatro

operadores, que se explican con la siguiente tabla:

| Textual | Simbólico | Explicación | Diagrama |
|---|---|---|---|
| Operadores unarios: | | | |
| X $\phi$ | $\bigcirc\phi$ | neXt: $\phi$ se tiene que cumplir en el siguiente estado. |  |
| G $\phi$ | $\Box\phi$ | Globally: $\phi$ se tiene que cumplir siempre. |  |
| F $\phi$ | $\Diamond\phi$ | Finally: $\phi$ eventualmente se tiene que cumplir. |  |
| Operador binario: | | | |
| $\psi$ U $\phi$ | $\psi \mathcal{U} \phi$ | Until: $\psi$ se tiene que cumplir al menos hasta que se cumpla $\phi$, |  |

## 2.2 El lenguaje FL

El objetivo central del lenguaje FL es encontrar problemas de coherencia en documentos normativos. Aquí la noción de coherencia se usa de manera muy pragmática: un determinado comportamiento no puede ser obligatorio o estar permitido y, a la vez, estar prohibido para un mismo individuo. Tampoco puede haber una obligación lisa y llana y esa misma obligación con una reparación en una CTD ni esta reparación estar prohibida por otra regla. Una lista completa de los casos que configuran una incoherencia se puede consultar en [GMS11].

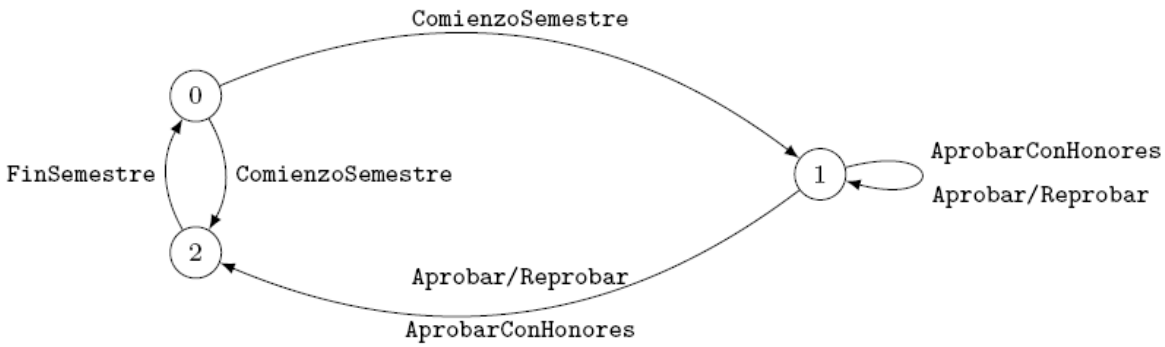
El lenguaje se divide en una primera parte con la teoría marco (*background theory*) y una segunda formada por un conjunto de reglas. En la primera se provee mecanismos sencillos con los cuales se puede describir la clase de modelos sobre los que predican las reglas, donde se expresan asuntos como la precedencia de eventos (e.g., el día ocurre antes de la noche), unicidad (e.g., las personas nacen una única vez), cantidad de veces que

puede ocurrir un suceso, etc. Esta teoría se traduce, con nuestra herramienta, a un autómata de Büchi.

El conjunto de reglas son fórmulas LTL con operadores deónticos que permiten expresar permisos, obligaciones y prohibiciones (las últimas dos pueden incluir reparaciones).

Teniendo en cuenta que FormaLex se basa en el uso de *model checkers,* repasemos su funcionamiento. Toman como entrada una descripción del sistema a analizar y una fórmula. La descripción del sistema se realiza mediante un autómata que representa los estados posibles del mismo y sus transiciones, que se definen como cambios posibles entre un estado y otro. Entonces, la fórmula de entrada se analiza en términos de los posibles recorridos definidos por el autómata, donde se verifica si siempre es posible satisfacerla sin importar el camino elegido. Dicho de otra manera, el autómata define la clase de modelos sobre los que luego se analiza la validez de la fórmula.

Vayamos a un ejemplo para clarificar estos conceptos. Se quiere modelar un examen con tres posibles resultados (`Reprobar`, `Aprobar` y `AprobarConHonores`) teniendo en cuenta que el examen ocurre solamente durante el transcurso de un semestre, que se delimitan con los eventos `ComienzoSemestre` y `FinSemestre`. Para representar este ejemplo se puede usar el siguiente autómata:




Cada camino distinto distinto dentro del autómata genera una traza lineal, que representa un modelo. El conjunto de todas las trazas posibles conforma la clase de modelos sobre la que se evalúan las fórmulas. Algunas de las trazas que se pueden generar con este autómata son:

```
ComienzoSemestre → FinSemestre → …
ComienzoSemestre → Reprobar → FinSemestre → …
ComienzoSemestre → Aprobar → FinSemestre → …
ComienzoSemestre → AprobarConHonores → FinSemestre → …
ComienzoSemestre → Reprobar → Aprobar → FinSemestre → …
```

Cada una de estas trazas describe un posible comportamiento legalmente válido de los agentes involucrados. Los comportamientos que no cumplen con las reglas son descartados. Si una regla es obligatoria, entonces se tiene que cumplir en todo modelo

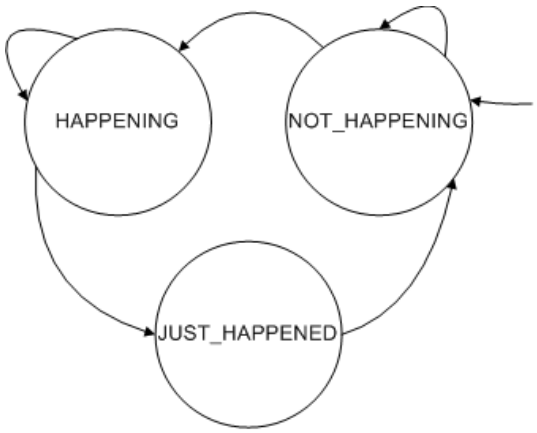
legalmente válido. Por lo tanto, si una regla dice $O(\varphi)$ debe interpretarse como la fórmula LTL $\Box\varphi$, indicando de esta manera que en todo estado del sistema la fórmula $\varphi$ es válida.

En FL una prohibición de una acción es equivalente a la obligación de su contrario (i.e., $F(\varphi) \equiv O(\neg\varphi)$ ). Las obligaciones con reparación se escriben como $O(\varphi)$ *repaired by* $\rho$ y $F(\varphi)$ *repaired by* $\rho$ es similar a $O(\neg\varphi)$ *repaired by* $\rho$.

Un permiso se interpreta como ausencia de prohibición. Se trata no como un operador que modifica el conjunto de comportamientos legalmente válidos, sino como un predicado que el resto de los modelos legalmente válidos deben cumplir. Si eso no ocurre, se considera que el sistema normativo bajo análisis (NSUA, por sus siglas en inglés) tiene un problema de coherencia, ya que indica que existe algo que está permitido cuando en realidad no lo está.

El componente más importante de la teoría marco es la acción, que puede estar sucediendo o no en cada instante y conservar su situación durante una cantidad arbitraria de estados consecutivos. Las acciones pueden representar acciones del agente implícito propiamente dichas (por ejemplo, `conducir_auto`) o eventos externos, no controlables (como la acción `choque`). Nótese que en esta versión no hay una noción implícita de roles y, por lo tanto, si se necesita el sujeto de la acción, debe representarse dentro de la misma, esto es, `conducir_auto_juan` y `conducir_auto_pablo`. Esta versión tampoco soporta cuantificadores en las reglas que permitan fórmulas donde se indique que todos los conductores deben cumplir una determinada norma. Las mejoras realizadas en la nueva versión incluyen la solución de estas dos limitaciones.

Las acciones se representan con una variable enumerada, que puede tener los valores *NOT_HAPPENING*, *HAPPENING* o *JUST_HAPPENED*. Se inicializan con *NOT_HAPPENING*, que indica que la acción no está pasando, y puede conservar ese valor durante un número arbitrario de estados consecutivos. Luego, la variable solo puede pasar al estado *HAPPENING*, que representa que la acción ha empezado a suceder. Este valor también lo puede conservar durante varios estados hasta que pasa a *JUST_HAPPENED*, que señala que la acción ha terminado de suceder. En la siguiente transición del autómata debe cambiar necesariamente de valor y pasar nuevamente al estado *NOT_HAPPENING*. Salvo casos particulares (que se explicarán más adelante, en la sección 3.1 Componentes de la teoría marco) este ciclo se puede repetir indefinidamente, siempre respetando el orden en que van cambiando de estado.




**Transiciones de una acción.**

Algunos requerimientos, como por ejemplo indicar que sea obligatorio obtener un registro para poder conducir un auto, parecen fáciles de formalizar, al prohibir `conducir_auto` si antes no ocurre la acción `obtener_registro`, pero esta facilidad es solo aparente ya que un individuo podría obtener el registro, luego perderlo por cometer alguna infracción y, sin embargo, podría conducir ya que la norma anterior se estaría cumpliendo. Con la acción `revocar_registro`, escribir una fórmula que señale si una persona puede o no conducir es complejo. Por ello, en FL existe la noción de intervalo, delimitado por una acción de comienzo y una de fin. Siguiendo nuestro ejemplo, se puede definir el intervalo:

```
interval autorizado_a_conducir
delimited by actions obtener_registro - revocar_registro
```

y la fórmula se escribiría:

$F(\neg \texttt{INSIDE(autorizado\_a\_conducir)} \wedge \texttt{conducir\_auto})$

donde `INSIDE(intervalo)` es una función que se hace verdadera cuando se está dentro del intervalo.

Para los intervalos, se agrega al autómata una variable enumerada, con el nombre del intervalo, que puede tener los valores *ACTIVE* o *INACTIVE*. En este ejemplo, la variable *autorizado_a_conducir* se inicia con el valor *INACTIVE* y conserva ese valor hasta que suceda la acción *obtener_registro*. En ese momento el intervalo pasa a tener el valor *ACTIVE*. Este valor continuará hasta que ocurra la acción *revocar_registro*. También se agregan al autómata restricciones que obligan a que *obtener_registro* no pueda ocurrir si el intervalo ya está activo (o sea, si ya está autorizado a conducir) y a que *revocar_registro* no pueda ocurrir si el intervalo no está activo.

Los intervalos se pueden utilizar también para acotar la ocurrencia de otras acciones, por ejemplo:

```
interval periodo_escolar
delimited by actions comienzo_cursada - fin_cursada
action rendir_examen only occurs in scope periodo_escolar
```

Una obligación se podría considerar como algo que debe ocurrir indefectiblemente, sin necesidad de que esté cumpliendo todo el tiempo, sino solo una vez, normalmente dentro de un periodo de tiempo. Esta obligación se suele denominar obligación no persistente y se representa en FL de esta manera: $O^{\diamond}(\varphi)$, estableciendo la obligación de que en algún momento tiene que valer $\varphi$, dejando de ser obligatorio una vez que se cumple. Si se desea considerar cotas temporales, para, por ejemplo, poder señalar que la matrícula debe pagarse durante el periodo de inscripción, se puede definir: $O^{\diamond}_{\texttt{periodo\_inscripción}}(\texttt{pagar\_matrícula})$.

Otra característica de FL son los contadores, que se incrementan o disminuyen cuando ocurren determinadas acciones. Permiten utilizar expresiones como $O(\texttt{clp} > 0 \rightarrow \diamond\ \texttt{clp} = 0)$, donde clp, cantidad de libros prestados, se incrementa en uno cada vez que

se presta un libro y se disminuye en uno con cada libro devuelto. La fórmula se lee “es obligatorio que cada vez que clp sea mayor que cero, en algún momento futuro vuelva a ser cero”. Sin los contadores se podría intentar expresar (de manera incorrecta) la misma fórmula con $O$(`retirar_libro -> ◇devolver_libro`), sin embargo esta fórmula se satisface al retirar varios libros y devolver uno solo, que claramente no es el comportamiento deseado.

Para ver un caso de estudio completo usando esta versión de FL y para ver una formalización de los operadores deónticos, se puede consultar [GMS11]. Más adelante, en la sección 4 Caso de estudio, presentaremos un ejemplo con la nueva versión de FL.

## 2.3 Sintaxis de reglas en FL

Sea `PROPS` un conjunto infinito contable de símbolos, `INTERVALS` ⊆ `(PROPS x PROPS)` un conjunto de intervalos y `FORMS` el conjunto de fórmulas FL con ⟨`PROPS, INTERVALS`⟩ definido como

`INNER_FORMS ::=` ⊤ | ⊥ | $p$ | ¬$\varphi$ | $\varphi_1$ ∧ $\varphi_2$ | ◇ $\varphi$ | $◇_i$ $\varphi$ | `INSIDE(i)`
`FORMS ::=` $O(\varphi)$ | $F(\varphi)$ | $F(\varphi)$ `repaired by` $\rho$ | $O(\varphi)$ `repaired by` $\rho$ | $O(◇\ \varphi)$ | $P(\varphi)$

donde $p \in$ `PROPS`, $\varphi, \rho, \varphi_1, \varphi_2 \in$ `INNER_FORMS` e `i` ∈ `INTERVALS`. Trabajamos con un conjunto de `FORMS` al especificar un reglamento, por lo que la conjunción entre fórmulas `FORMS` no necesita ser definido formalmente. En la especificación se suele escribir una fórmula debajo de la otra, lo que implícitamente significa la conjunción entre todas ellas.

## 2.4 Semántica de reglas en FL

Presentamos la semántica de las reglas en FL mediante la función *Tr*, que traduce de FL a LTL clásico[1]. Sea *F* un conjunto de fórmulas FL que definen el conjunto de modelos legales para el NSUA. Recordemos que los modelos LTL representan posibles caminos del autómata definido por la teoría marco. Primero se separan los permisos del resto y se define el dominio de *Tr* como $F_{np}$, el conjunto de todos los términos de FL excluyendo a los permisos. *Tr* actúa como identidad para las construcciones INNER_FORMS no especificadas explícitamente.

| | | |
|---|---|---|
| $Tr(◇_i \varphi)$ | = | i = ACTIVE → (i = ACTIVE $U$ $Tr(\varphi)$) |
| $Tr(\ F(\varphi)\ )$ | = | □¬$Tr(\varphi)$ |
| $Tr(\ F(\varphi)$ repaired by $\rho\ )$ | = | □($Tr(\varphi)$ → $Tr(\rho)$ ) |

[1] Esto es, la versión básica de LTL con los conectivos booleanos estándar más el operador *until*.

| | | |
|---|---|---|
| $Tr(\ O(\varphi)\ )$ | = | $\Box\, Tr(\varphi)$ |
| $Tr(\ O(\varphi)$ repaired by $\rho\ )$ | = | $\Box(\neg Tr(\varphi) \rightarrow Tr(\rho)\ )$ |
| $Tr(\ O^{\Diamond}(\ \varphi))$ | = | $^{\Diamond} Tr(\ \varphi)$ |

Tomemos al autómata $A$ definido por la teoría marco y a $C_A$ como la clase de modelos que representa todas las corridas del autómata $A$. Sea $F$ el conjunto de fórmulas FL que representan el NSUA. La clase de modelos válidos definido por $F$ sobre $C_A$ se define como:

$$C_A^F = \{ M \in C_A \mid M \vDash Tr(F_{np}) \}\ .$$

Esto es, cada modelo legal debe satisfacer las obligaciones y prohibiciones especificadas en $F$.

Los permisos son una verificación que se debe hacer en $C_A^F$ para garantizar la coherencia. La condición que $C_A^F$ debe satisfacer es: para cada $\varphi$ de la forma $P(\psi)$ en $F$ debe haber al menos un modelo $M$ en $C_A^F$ tal que $M \vDash Tr(\psi)$. O sea, si algo está permitido, entonces el resto de el NSUA no impide que suceda.

# 3 La nueva versión

Nuestra tesis consiste en extender el lenguaje FL, agregando funcionalidades para dar soporte a un conjunto más amplio de situaciones expresadas en los documentos normativos y adaptar la herramienta para que funcionen estas nuevas características. Además, se realizaron varias mejoras a los componentes existentes, que le agregan expresividad. Muchos de estos requerimientos surgen del trabajo realizado por el grupo de investigación de Informática y Derecho de la Universidad FASTA de Mar del Plata que usó FL para analizar la Ley 24240 de Defensa al Consumidor de la República Argentina y encontró una serie de limitaciones en la versión existente.

Entre las mejoras realizadas cabe destacar:

- La principal mejora, es la incorporación de roles y agentes, y el uso de los cuantificadores existenciales y universales en las fórmulas.
- Se modificó la codificación del autómata, para eliminar restricciones que limitaban la expresividad del lenguaje (antes, en cada transición, una acción tenía que cambiar de estado, y solo una acción podía estar en el estado que representa que acaba de terminar de suceder).
- Se agregó sincronización entre acciones (se puede indicar que una acción ocurre simultáneamente con otra, por ejemplo si alguien compra entonces otra persona vende).
- Se modificaron los intervalos para que puedan ser más de una las acciones que provocan que el intervalo comience o finalice.
- Se agregó un modificador a los intervalos para que estos pueden iniciarse activos y otro para permitir que una vez que el intervalo esté activo, conserve de ahí en adelante ese valor.
- Se agregaron a los contadores la posibilidad que ante determinados sucesos se le asigne al contador un nuevo valor.
- Se modificaron los contadores permitiendo que sean varias las acciones que pueden reiniciar su valor, incrementarlo o disminuirlo.
- Se agregó el concepto de *local* y *global* para contadores, permitiendo en el primer caso que haya un contador por cada agente, o, si es global, que todos los agentes determinen el valor de la misma instancia de contador.
- Se agregó el mismo concepto de *local* y *global* para intervalos.
- Se agregaron las *impersonal action*, que son acciones como por ejemplo llover, que no las ejecuta ningún agente.
- Se permite que haya más de un *Timer* (llamado en la versión anterior *Acciones Temporales*) en la especificación.
- Se hicieron pequeñas modificaciones sintácticas para facilitar la claridad del lenguaje.

## 3.1 Componentes de la teoría marco

A continuación se detallarán todos los componentes de FL junto con las características de cada uno de ellos y más adelante, en la sección 3.2 Generación del autómata se mostrará su comportamiento y cómo se genera el autómata.

### 3.1.1 Acciones

Son el principal componente de la teoría marco. Pueden representar eventos, como *llover* o *comienzo_del_año*, y acciones como *comprar* o *vender*. Los eventos, en general, no tienen a alguien que los ejecute. En el caso de las acciones propiamente dichas, es común que sea algo o alguien el que las ejecute, por ejemplo *Pedro compra* o *Juan vende*. FL provee la noción de roles que se usan para definir quiénes pueden ejecutar una acción (el tema de los roles se verá en detalle más adelante en la sección 3.1.5 Roles). Las acciones se definen así:

```
impersonal action llover
action vender only performable by vendedor[2]
action comprar only performable by comprador
```

Se pueden definir varias acciones a la vez:

```
actions comprar, vender
```

Se debe tener en cuenta que si se definen las acciones de esta última manera, todos los modificadores que se agreguen se aplican a todas las acciones especificadas.

Se puede generar acciones que tengan como resultado un valor de salida. Para ello, cuando se define una acción hay que especificar cuáles son los posibles valores de retorno. Por ejemplo:

```
action rendir_examen output values {aprobado,desaprobado}
```

En algunos casos, una acción ocurre de manera simultánea con otra. Por ejemplo, siempre que alguien compra hay otro que vende. En este caso se puede definir que las acciones *vender* y *comprar* están sincronizadas.

```
action comprar synchronizes with vender only performable by
comprador
```

[2] En este ejemplo se indica que la acción vender la puede realizar un solo rol, pero se puede definir varios roles (separados por coma) en caso de que se necesite especificar de esa manera. Más adelante, al hablar de roles se verá un ejemplo de acción con más de un rol.

Con esta definición, ambas acciones ocurren en simultáneo y no puede ocurrir una sin que ocurra la otra. Al definir que una acción está sincronizada con otra acción, no es necesario definir que esta segunda está sincronizada con la primera, eso se asume a partir de la primera definición.

Si se quiere que una acción ocurra, a lo sumo, una determinada cantidad de veces, se puede indicar dicha cantidad al momento de definir la acción con el modificador *occurrences*:

```
action vender occurrences 5 only performable by vendedor
```

En este caso, la acción *vender* puede ocurrir, como máximo, cinco veces.

De estas características, en esta tesis se incorporaron los conceptos de acciones impersonales, acciones sincronizadas y roles (*only performable by*).

### 3.1.2 Intervalos

Un intervalo es un espacio temporal que comienza cuando ocurre una determinada acción y finaliza cuando ocurre otra. Una vez que ocurre la acción inicial, y hasta que ocurra la final, se dice que el intervalo está activo. Los intervalos se pueden usar para definir cuándo una acción puede ocurrir. Por ejemplo, se puede definir el intervalo *periodo_escolar* delimitado por las acciones *comienzo_cursada* y *fin_cursada.*

```
interval periodo_escolar defined by actions comienzo_cursada
- fin_cursada
```

Luego se pueden definir acciones que ocurren sólo en un determinado periodo. Por ejemplo, la acción *rendir_examen* sólo ocurre en el intervalo *periodo_escolar*.

```
action rendir_examen only occurs in scope periodo_escolar
```

Además, cuando se crea un intervalo, se puede indicar que ocurre dentro de otro intervalo, por ejemplo:

```
interval periodo_exámenes defined by actions
comienzo_exámenes - fin_exámenes only occurs in scope
periodo_escolar
```

Para un intervalo se puede definir una o varias acciones que lo inicializan y una o varias acciones que lo hacen finalizar. A continuación, un ejemplo de un intervalo con dos acciones iniciales y dos finales:

```
interval estudiante defined by actions inscripcion_on_line,
inscripcion_ventanilla - graduarse, abandonar_carrera
```

Si se quiere que un intervalo ocurra no más de una determinada cantidad de veces, se puede indicar dicha cantidad al momento de definir el intervalo con el modificador *occurrences*:

```
interval periodo_escolar defined by actions comienzo_cursada
- fin_cursada occurrences 4
```

En este caso, el intervalo periodo_escolar puede ocurrir, como máximo, cuatro veces.

Se pueden definir intervalos cuyo estado inicial sea activo. También se puede indicar que cuando un intervalo pase a estar activo, conserve de ahí en más ese estado. A continuación, un ejemplo del primer caso y luego otro del segundo:

```
interval antes_de_cristo defined by actions infinite -
nace_cristo

interval después_de_cristo defined by actions nace_cristo -
infinite
```

Las acciones que delimitan al intervalo suceden en orden, esto es, una acción inicial solo puede ocurrir si el intervalo está inactivo y una acción que finaliza el intervalo solo puede suceder si el intervalo está activo, garantizando así que la acción final ocurra luego de una acción inicial y no antes. Siguiendo el ejemplo del intervalo *estudiante*, que tiene como acciones iniciales a *inscripcion_on_line* y a *inscripcion_ventanilla*, y como acciones finales a *graduarse* y a *abandonar_carrera*, se garantiza que estas dos últimas acciones no pueden suceder si antes no ocurrió la *inscripcion_on_line* o la *inscripcion_ventanilla*. Tampoco pueden suceder dos acciones iniciales seguidas, sin que antes suceda una acción final, ni dos acciones finales sin que entre ellas suceda una acción inicial. Siguiendo el ejemplo, si ocurre *inscripcion_on_line* el intervalo pasa a estar activo y, mientras esté en ese estado, no puede volver a suceder esa misma acción ni la acción *inscripcion_ventanilla*. A su vez, si el intervalo está activo y ocurre una acción final, no puede volver a ocurrir ni esa ni otra acción final hasta que el intervalo esté activo de nuevo.

A partir de esta tesis, pueden ser más de una las acciones que inician o finalizan a un intervalo. Se agregó la posibilidad de que un intervalo empiece con estado activo (usando *infinite*) y de que una vez que esté activo conserve para siempre ese valor. Además, se incorporó el concepto de intervalo local y global, que será explicado más adelante, al hablar de los agentes.

### 3.1.3 Contadores

Un contador es una variable entera que puede ser modificada cada vez que ocurren algunas acciones. Al definirlo, se puede indicar su valor inicial, una o varias acciones que lo

incrementan (se puede indicar, por cada acción, en cuánto se quiere incrementar), que lo disminuyen, que lo vuelven a su valor inicial o que le asignan un determinado valor.

Veamos como ejemplo un resumen del sistema de puntos de la licencia de conducir en la CABA. Éste es un sistema de evaluación de conductores que consiste en la asignación de un puntaje inicial a cada conductor y el descuento de puntos en función de las infracciones cometidas:

```
local counter puntaje init value 20
decreases with action conducir_sin_cinturón by 2,
decreases with action conducir_sin_licencia by 4,
decreases with action conducir_en_contramano by 5,
decreases with action conducir_ebrio by 10,
resets with action cumple_inhabilitación,
sets with action pasan_2_años_sin_infracción to value 20
```

Se puede definir a cualquiera de los modificadores del contador, que solo lo modifiquen si se cumple un determinado predicado, agregando la expresión: *provided that (*condición*).*

A partir de esta tesis se agregó la posibilidad de usar el modificador *sets with action* (que permite asignarle al contador el valor que se defina, a diferencia de *resets* que le asigna el valor inicial). Además, se permite que sean más de una las acciones que pueden reiniciarlo, incrementarlo o disminuirlo. Por último, se incorporó el concepto de contador local y global, que será explicado más adelante, al hablar de los agentes.

### 3.1.4 Timers

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. La idea de los *timers* es permitir definir cuándo puede ocurrir una acción.

Los *timers* son similares, en algunos aspectos, a los intervalos, pero tienen un comportamiento diferente. Los eventos de un timer ocurren en orden. Cuando una etapa del timer termina, recién ahí comienza la siguiente y lo hace de manera automática. Un *timer* es independiente de cualquier otro *timer* que se defina. Los intervalos pueden definirse de manera tal que uno esté dentro del otro, pero no se pueden encadenar como sí lo hacen los *timers*.

Se definen así:

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

Con esta tesis hemos eliminado la restricción anterior que permitía un solo *timer* por especificación.

### 3.1.5 Roles

FL incorpora, a partir de esta tesis, la noción de rol. La idea es poder definir roles y asignarlos a las acciones en el momento de especificarlas. De esta manera se fija qué roles son los que pueden realizar una determinada acción. Por ejemplo, se pueden definir los roles *vendedor* y *comprador* y luego definir la acción *vender* indicando que la puede realizar el rol *vendedor*, y la acción *comprar* realizable por el rol *comprador*.

```
roles vendedor, comprador
action vender only performable by vendedor
action comprar only performable by comprador
```

Se puede definir acciones que las pueden realizar más de un rol, por ejemplo:

```
roles vendedor_minorista, vendedor_mayorista
action vender only performable by vendedor_minorista,
vendedor_mayorista
```

Al momento de definir un conjunto de roles, se puede indicar que estos son disjuntos entre sí (utilizando la palabra clave *disjoint*) para señalar que una persona que cumple un rol no puede cumplir otro de ese grupo. Por ejemplo, se pueden definir los roles *perro* y *gato* indicando que son disjuntos. Luego, ningún agente podría cumplir los dos roles a la vez.

```
roles perro, gato disjoint
```

Además, se puede indicar que un conjunto de roles cubre todo el universo posible (utilizando la palabra clave *cover*) . En este caso, todos los agentes que ejecutan acciones tienen que tener alguno de esos roles. Un ejemplo sería definir los roles *civil* y *militar* con esta propiedad, para que todas las personas sean civiles o militares, sin la posibilidad de una tercera opción.

```
roles civil, militar cover
```

### 3.1.6 Agentes

Los agentes, que se incorporan a partir de esta tesis, son entidades que cumplen roles y ejecutan acciones. Son componentes usados internamente por nuestra herramienta para la generación del autómata. La idea es definir un agente por cada combinación válida de roles. En principio, todas las combinaciones son válidas, salvo cuando se usan los modificadores de roles mencionados más arriba. Por ejemplo, si tenemos los roles *perro* y *gato* definidos como *disjoint*, no puede haber un agente que tenga ambos roles. Por otro lado, si se definen, por ejemplo, los roles *civil* y *militar* como *cover*, todos los agentes tienen que tener por lo menos uno de esos roles. En el caso de que una acción se defina sin un rol, entonces esa acción puede ejecutarse por cualquier agente, independientemente de los roles que tenga asignado.

En el momento en que se genera el autómata, estos agentes se combinan con las acciones para generar nuevas acciones que permiten representar todos los estados posibles del sistema y las transiciones posibles entre un estado y el siguiente. Veamos con un ejemplo cómo sería la generación de agentes a partir de algunos roles y acciones.

```
roles vendedor, comprador
impersonal action llover
action publicar only performable by vendedor
action vender only performable by vendedor
action comprar only performable by comprador
action pedir_factura only performable by comprador
action pagar_impuesto
```

Tenemos 2 roles y 6 acciones, una de ellas *impersonal*, otra sin roles definidos, dos con el rol *vendedor* y dos con el rol *comprador*. Hacemos todas las combinaciones posibles de roles para generar los agentes. A continuación están los agentes generados con los roles asignados:

```
agent_1: vendedor
agent_2: comprador
agent_3: vendedor, comprador
```

Luego, se combinan estos agentes con las acciones que correspondan. Por ejemplo, el agente *agent_1*, que es *vendedor*, se combina con las acciones que puede ejecutar ese rol, en este caso *publicar* y *vender*. De esta manera se generan las siguientes acciones:

```
agent_1.publicar
agent_1.vender
```

El agente *agent_2*, que es *comprador*, se combina con las acciones que puede ejecutar ese rol, en este caso *comprar* y *pedir_factura*. Se generan las acciones:

```
agent_2.comprar
agent_2.pedir_factura
```

El último agente es *agent_3*, que es *comprador* y *vendedor* a la vez, se combina con las acciones que pueden ejecutar ambos roles. Las acciones que se generan son:

```
agent_3.vender
agent_3.publicar
agent_3.comprar
agent_3.pedir_factura
```

La acción sin rol definido, se agrega para todos los agentes:

```
agent_1.pagar_impuesto
agent_2.pagar_impuesto
agent_3.pagar_impuesto
```

Además, al autómata, se agrega la acción *impersonal*, sin agregarle agente:

```
llover
```

Si al definir los roles *vendedor* y *comprador* se indica que estos son disjuntos, entonces, al momento de generar los agentes, no se generaría el *agent_3*, ya que ese agente está cumpliendo dos roles que son disjuntos y, por lo tanto, no se generarían tampoco las acciones con ese agente.

En este ejemplo tiene sentido que siempre que un agente venda otro compre y viceversa. Para ello se puede definir esas acciones como sincronizadas. Cuando se sincronizan acciones, se pueden usar los modificadores `allow autosync` y `disallow autosync`, que indican, respectivamente, si un mismo agente puede realizar las dos acciones sincronizadas, o no. Ejemplo:

```
action vender synchronizes with comprar disallow autosync
only performable by vendedor
```

En este caso, luego de combinar los agentes con las acciones, como se mostró más arriba, se combinan las acciones sincronizadas generando nuevas acciones que las reemplazan. Siguiendo el ejemplo, se toma cada uno de los agentes que pueden realizar la acción de *comprar* y se lo combina con cada uno de los agentes que pueden *vender*. Las nuevas acciones que se generan son:

```
agent_1.vender-agent_3.comprar
agent_1.vender-agent_2.comprar
agent_3.vender-agent_2.comprar
```

Si al definir las acciones sincronizadas se reemplaza el modificador '*disallow autosync*' por el de '*allow autosync*', entonces se agregaría la acción:

```
agent_3.vender-agent_3.comprar
```

### 3.1.7 Actualizaciones de acciones con agentes para intervalos

Al modificarse las acciones y crear nuevas con agentes, se deben actualizar las que delimitan los intervalos. Hay dos tipos de intervalos: local y global (para definirlos se debe usar los modificadores *local* y *global* al crear el intervalo). La idea es que uno local sólo

afecta a un agente, en cambio, uno global es compartido por todos los agentes. Por ejemplo, si el intervalo es local, se define así:

```
local interval estudiante defined by actions
inscripcion_online, inscripcion_ventanilla - graduarse,
abandonar_carrera
```

Una vez que se definen los agentes, a partir de un intervalo local, se generan varios intervalos, uno por cada agente que realice por lo menos una de las acciones iniciales y una de las acciones finales del intervalo:

```
interval agent_X.estudiante defined by actions
agent_X.inscripcion_online, agent_X.inscripcion_ventanilla -
agent_X.graduarse, agent_X.abandonar_carrera
```

siendo X el número del agente. Si por la asignación de roles, el agente para el cual se define el intervalo no realiza alguna de las acciones, entonces estas últimas no delimitarán el nuevo intervalo. Si en el ejemplo anterior, el *agent_X* no puede realizar la acción *abandonar_carrera*, pero sí las otras, entonces el nuevo intervalo quedaría así:

```
interval agent_X.estudiante defined by actions
agent_X.inscripcion_online, agent_X.inscripcion_ventanilla -
agent_X.graduarse
```

En el caso de que el intervalo sea global, las acciones que lo delimitan se reemplazan por las nuevas con los agentes. Por ejemplo si un intervalo se define así:

```
global interval luz_prendida defined by actions prender_luz -
apagar_luz
```

Cada una de las acciones iniciales y finales se reemplazan por las nuevas creadas con los agentes que realicen esas acciones:

```
interval luz_prendida defined by actions agent_1.prender_luz,
agent_2.prender_luz, ...- agent_1.apagar_luz,
agent_2.apagar_luz, ...
```

### 3.1.8 Actualizaciones de acciones con agentes para contadores

De manera similar a lo que ocurre con los intervalos, al modificarse las acciones y crear nuevas con agentes, se deben actualizar las que modifican a los contadores. Hay dos tipos de contadores: local y global. La idea es la misma que con los intervalos, uno local sólo afecta a un agente, en cambio, uno global es compartido por todos los agentes. Por ejemplo, si el contador es local, se define así:

```
local counter puntaje init value 20
decreases with action conducir_sin_cinturón by 2,
decreases with action conducir_sin_licencia by 4,
decreases with action conducir_en_contramano by 5,
decreases with action conducir_ebrio by 10,
resets with action cumple_inhabilitación,
sets with action pasan_2_años_sin_infracción to value 20
```

Una vez que se definen los agentes, a partir de un contador local, se generan varios contadores, uno por cada agente que realice por lo menos una de las acciones que modifican al contador:

```
local counter agent_X.puntaje init value 20
decreases with action agent_X.conducir_sin_cinturón by 2,
decreases with action agent_X.conducir_sin_licencia by 4,
decreases with action agent_X.conducir_en_contramano by 5,
decreases with action agent_X.conducir_ebrio by 10,
resets with action agent_X.cumple_inhabilitación,
sets with action agent_X.pasan_2_años_sin_infracción to value
20
```

siendo X el número del agente. Si por la asignación de roles, el agente para el cual se define el contador no realiza alguna de las acciones, entonces estas últimas no aparecerán como modificadoras del nuevo contador, pero sí aparecerán las que sí puede realizar. En este ejemplo no tiene mucho sentido, pero si el *agent_X* no puede realizar la acción `conducir_sin_cinturón`, pero sí las otras, entonces el nuevo contador quedaría así:

```
local counter agent_X.puntaje init value 20
decreases with action agent_X.conducir_sin_licencia by 4,
decreases with action agent_X.conducir_en_contramano by 5,
decreases with action agent_X.conducir_ebrio by 10,
resets with action agent_X.cumple_inhabilitación,
sets with action agent_X.pasan_2_años_sin_infracción to value
20
```

En el caso de que el contador sea global, las acciones que lo modifican se reemplazan por las nuevas con los agentes. Por ejemplo, si un contador se define así:

```
global contador plata_acumulada init value 0 increases with
action pone_5 by 5, decreases with action retira_5 by 5
```

Cada una de las acciones involucradas se reemplazan por las nuevas creadas con los agentes que realicen esas acciones:

```
contador plata_acumulada init value 0 increases with action
agent_1.pone_5 by 5, increases with action agent_2.pone_5 by
5,..., increases with action agent_N.pone_5 by 5, decreases
with action agent_1.retira5 by 5, decreases with action
agent_2.retira5 by 5,..., decreases with action
agent_N.retira5 by 5
```

## 3.2 Generación del autómata

Antes de mostrar cómo se genera el autómata que contiene la teoría marco, mencionaremos algunos detalles de nuestra herramienta. El programa está hecho con Java[3] . Comienza con la lectura de la especificación en FL. Esta lectura se hace a partir de fuentes Java generados previamente con la herramienta JavaCC[4], que es un generador de *parsers* y de analizadores léxicos. Dicha generación se hace a partir de una especificación de entrada donde se indica la sintaxis válida de FL. Entonces, a partir de una especificación en FL, nuestra herramienta genera automáticamente un autómata que se usa como una de las entradas al *model checker* y contiene la definición de las distintas instancias de los componentes de la teoría marco, sus comportamientos y sus relaciones con otros componentes. Está representado por un archivo de texto plano que debe respetar el formato esperado por el *model checker*. Nosotros lo generamos usando la herramienta *Velocity*[5] que es un motor de *templates* basado en Java. Usando esta herramienta, con la definición de un *template*, generamos el autómata con el formato del *model checker*. De esta manera logramos, a la hora de generar el autómata, desacoplar lo máximo posible nuestra herramienta del *model checker* elegido. En particular para este trabajo usamos el *model checker NuSMV*[6]. Para poder usar otros *model checker* alcanza con definir un nuevo *template*. La otra entrada al *model checker* es la fórmula a validar. Esta surge de la especificación en FL, a partir de la cual se traduce a LTL. Dependiendo de la especificación puede haber más de una fórmula a validar. Más detalles de los pasos que sigue nuestra herramienta se verán más adelante en 3.4 Corriendo el programa.

Veremos ahora en detalle cómo se genera el autómata y cómo se define su comportamiento.

El archivo que representa al autómata se forma de la siguiente manera:

- comienza con: `MODULE main`
- a continuación se encuentra la sección donde se definen las variables: `VAR`
- luego se inicializan las variables en la sección: `INIT`
- continúa con la sección donde se definen las transiciones: `TRANS`

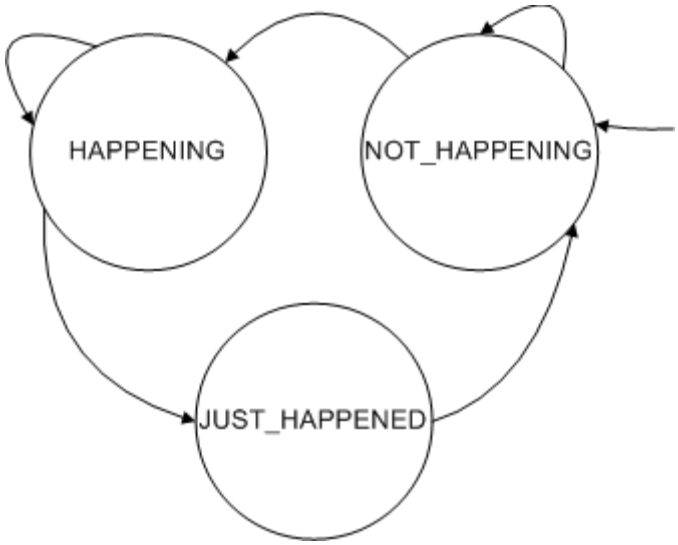
[3] http://www.java.com/
[4] https://javacc.java.net/
[5] http://velocity.apache.org/
[6] http://nusmv.fbk.eu/

- por último, se puede definir una lista de fórmulas LTL para verificar la correcta definición del autómata. Estas comienzan con la palabra clave: `LTLSPEC` y continúan con la fórmula que se desea validar.

### 3.2.1 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.




Transiciones de una acción.

Para lograr esto, en la implementación del autómata se define, por cada acción:
(1) una variable en la sección '`VAR`':

- `NOMBRE_DE_LA_ACCIÓN: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};`

(2) se inicializa en la sección '`INIT`':

- `NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING`

(3 y 4) se definen las transiciones en la sección '`TRANS`', mediante fórmulas:

- `NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED -> next(NOMBRE_DE_LA_ACCIÓN) = NOT_HAPPENING`
- `NOMBRE_DE_LA_ACCIÓN = HAPPENING -> next(NOMBRE_DE_LA_ACCIÓN) != NOT_HAPPENING`
- `NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING -> next(NOMBRE_DE_LA_ACCIÓN) != JUST_HAPPENED`

Para verificar el comportamiento, se puede agregar algunas fórmulas LTL que validan lo especificado anteriormente:

La acción debe poder alcanzar cualquiera de sus valores posibles (por eso estas fórmulas deben ser falsas)

- `LTLSPEC !F (NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED)`
- `LTLSPEC !F (NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING)`
- `LTLSPEC !F (NOMBRE_DE_LA_ACCIÓN = HAPPENING)`

La acción no necesariamente debe, a partir de un punto, quedarse con el mismo valor (por eso estas fórmulas deben ser falsas)

- `LTLSPEC !F (G NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING)`
- `LTLSPEC !F (G NOMBRE_DE_LA_ACCIÓN = HAPPENING)`
- `LTLSPEC !F (G NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED)`

Si la acción está en estado ‘JUST_HAPPENED’ no puede luego de una transición conservar ese estado (esta fórmula debe ser verdadera)

- `LTLSPEC !F (NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED & X(NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED))`

La acción no puede hacer ciertas transiciones (estas fórmulas deben ser verdaderas)

- `LTLSPEC G (NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED -> (!X(NOMBRE_DE_LA_ACCIÓN = HAPPENING)& !X(NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED))`
- `LTLSPEC G (NOMBRE_DE_LA_ACCIÓN = HAPPENING -> !X(NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING))`
- `LTLSPEC G (NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING -> !X(NOMBRE_DE_LA_ACCIÓN = JUST_HAPPENED))`

### 3.2.2 Acciones con valores de salida

Las acciones pueden definirse con resultados. En este caso, se agrega una nueva variable enumerada, que, una vez ocurrida la acción, tiene el valor de salida. Debe respetar las siguientes condiciones:

1. La variable sólo puede tomar alguno de los valores definidos.

2. La variable conserva su valor hasta que vuelve a ocurrir la acción.
3. Cuando ocurre la acción, la variable toma un nuevo valor (que puede ser igual al anterior).

Para lograr esto, en la implementación del autómata se define, por cada acción con resultado:
(1) una variable enumerada en la sección 'VAR':

- `ACCION_OUTPUT: {VALOR_1, VALOR_2, ..., VALOR_N};`

(2) se definen las transiciones en la sección 'TRANS', mediante fórmulas:

- `next(ACCION) != JUST_HAPPENED -> ACCION_OUTPUT= next(ACCION_OUTPUT)`

(3)

- `next(ACCION) = JUST_HAPPENED -> next(ACCION_OUTPUT)= {VALOR_1, VALOR_2, ..., VALOR_N}`

Esta última transición se podría no agregar, ya que ese es el comportamiento que tiene si no se define otro, o sea, toma cualquier valor (del enumerado) de manera no determinística.

Para verificar el comportamiento, se puede agregar algunas fórmulas LTL que validan lo especificado anteriormente:

Si la acción no pasa en el próximo estado a *JUST_HAPPENED*, el resultado se conserva (esta fórmula debe ser verdadera)

- `LTLSPEC G ( (X(ACCION) = NOT_HAPPENING | X(ACCION) = HAPPENING )-> ( X(ACCION_OUTPUT) = ACCION_OUTPUT ))`

Si en el próximo estado el resultado cambia de valor, entonces la acción estará en *JUST_HAPPENED* (esta fórmula debe ser verdadera)

- `LTLSPEC G ( X(ACCION_OUTPUT) != ACCION_OUTPUT -> X(ACCION_OUTPUT) = JUST_HAPPENED )`

### 3.2.3 Acciones con occurrences

Las acciones pueden definirse con el modificador *occurrences* para lograr que no ocurran más veces que las especificadas. Para que se cumpla esta condición, se agrega en la sección VAR una nueva variable por cada acción de este tipo, indicando que puede tomar valores en el rango numérico que va desde cero hasta el número máximo de ocurrencias posible:

- `NOMBRE_DE_LA_ACCIÓN_OCCURRENCE: 0 .. VALOR_ESPECIFICADO;`

En la sección INIT la inicializamos en cero:

- `NOMBRE_DE_LA_ACCIÓN_OCCURRENCE = 0`

Queremos que si la acción hace la transición de NOT_HAPPENING a HAPPENING, entonces el contador se incremente en uno, y si no se hace esa transición, el contador conserva su valor. Para lograrlo, en la sección TRANS agregamos la siguiente fórmula:

- `(NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING & next(NOMBRE_DE_LA_ACCIÓN) = HAPPENING <-> NOMBRE_DE_LA_ACCIÓN_OCCURRENCE + 1 = next(NOMBRE_DE_LA_ACCIÓN_OCCURRENCE) )&(!(NOMBRE_DE_LA_ACCIÓN = NOT_HAPPENING & next(NOMBRE_DE_LA_ACCIÓN) = HAPPENING) <-> NOMBRE_DE_LA_ACCIÓN_OCCURRENCE = next(NOMBRE_DE_LA_ACCIÓN_OCCURRENCE) )`

Para verificar el comportamiento, se puede agregar la siguiente fórmula LTL, que controla que la variable que representa el contador no supere el número permitido:

- `LTLSPEC G (NOMBRE_DE_LA_ACCIÓN_OCCURRENCE <= VALOR_ESPECIFICADO)`

### 3.2.4 Acciones con "only occurs in scope"

Las acciones pueden definirse indicando que sólo pueden ocurrir en determinado intervalo. Por ejemplo, se puede definir que la acción *RENDIR_EXAMEN* sólo pueda ocurrir durante el intervalo *AÑO_LECTIVO*. Para lograr esto, en la implementación del autómata, se agrega por cada acción con esta característica, en la sección '`TRANS`', la fórmula:

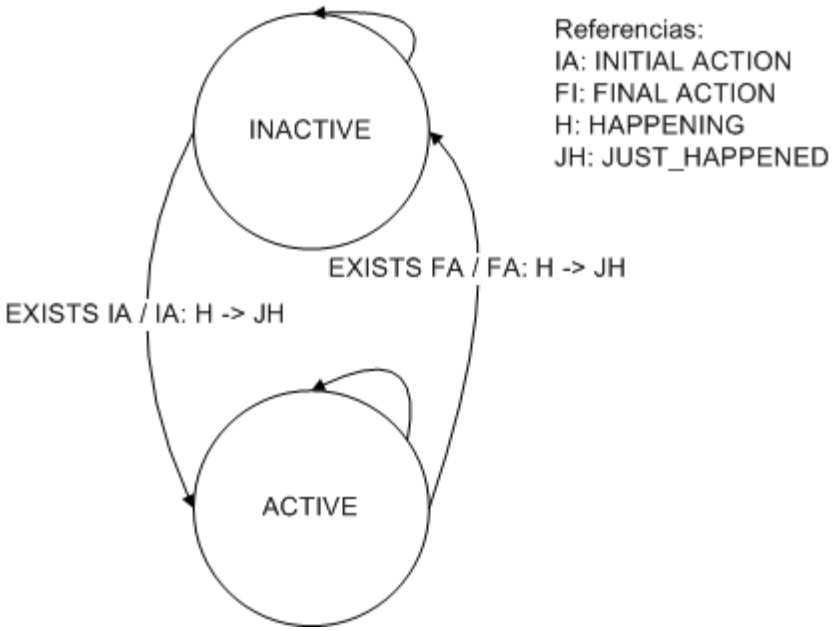
- `next(NOMBRE_DE_LA_ACCION) = HAPPENING -> next(NOMBRE_DEL_INTERVALO) = ACTIVE`

### 3.2.5 Intervalos

Un intervalo se representa en el autómata con una variable. En cada transición puede cambiar de valor, dependiendo de si las acciones que lo delimitan cambian a determinados estados. Debe respetar las siguientes condiciones:

1. puede tener sólo alguno de los siguientes valores: *INACTIVE* y *ACTIVE.*
2. si al definir el intervalo en vez de acciones iniciales se usa el modificador *infinite*, su valor inicial es *ACTIVE, en caso contrario es INACTIVE.*
3. puede conservar o cambiar de valor indefinidamente, salvo que se limite la cantidad de veces que puede estar activo, o que se haya usado, en vez de acciones finales, el modificador *infinite*, en cuyo caso una vez que el intervalo está activo, conservará siempre ese valor.
4. si su estado es *INACTIVE* y una de las acciones iniciales pasa del estado *HAPPENING* al de *JUST_HAPPENED*, el intervalo pasa al de *ACTIVE*.
5. si su estado es *ACTIVE* y una de las acciones finales pasa del estado *HAPPENING* al de *JUST_HAPPENED*, el intervalo pasa al de *INACTIVE*.
6. en cualquier otro caso el valor del intervalo se conserva durante la transición.

7. una acción inicial solo puede empezar a suceder (i.e., pasar de NOT_HAPPENING a HAPPENING) si el intervalo está *INACTIVE.*
8. una acción final solo puede empezar a suceder (i.e., pasar de NOT_HAPPENING a HAPPENING) si el intervalo está *ACTIVE.*
9. una acción inicial no puede pasar del estado NOT_HAPPENING a HAPPENING si existe otra acción inicial en estado HAPPENING
10. una acción final no puede pasar del estado NOT_HAPPENING a HAPPENING si existe otra acción final en estado HAPPENING




Transiciones de un intervalo

Para lograr esto, en la implementación del autómata se define, por cada intervalo:

(1) una variable enumerada en la sección '`VAR`':

- `NOMBRE_DEL_INTERVALO: {ACTIVE, INACTIVE};`

(2) se inicializa en la sección '`INIT`':

- `NOMBRE_DEL_INTERVALO = INACTIVE`

Si se usó, en vez de acciones iniciales, el modificador *infinite*, el estado arranca activo:

- `NOMBRE_DEL_INTERVALO = ACTIVE`

(4)

- ```
  NOMBRE_DEL_INTERVALO = INACTIVE & (
  (ACC_INICIAL_1 = HAPPENING & next(ACC_INICIAL_1) =
  JUST_HAPPENED)|
  (ACC_INICIAL_2 = HAPPENING & next(ACC_INICIAL_2) =
  JUST_HAPPENED)|
  ... |
  (ACC_INICIAL_N = HAPPENING & next(ACC_INICIAL_N) =
  JUST_HAPPENED)
  ```
  ```

```
    ) -> next(NOMBRE_DEL_INTERVALO) = ACTIVE
```

(5)

- ```
  NOMBRE_DEL_INTERVALO = ACTIVE & (
    (ACC_FINAL_1 = HAPPENING & next(ACC_FINAL_1) =
  JUST_HAPPENED)|
    (ACC_FINAL_2 = HAPPENING & next(ACC_FINAL_2) =
  JUST_HAPPENED)|
    ...  |
    (ACC_FINAL_N = HAPPENING & next(ACC_FINAL_N) = JUST_HAPPENED)
    ) -> next(NOMBRE_DEL_INTERVALO) = INACTIVE
  ```

(6)

- ```
  NOMBRE_DEL_INTERVALO = INACTIVE & next(NOMBRE_DEL_INTERVALO)
  = ACTIVE ->
  (ACC_INICIAL_1 = HAPPENING & next(ACC_INICIAL_1) =
  JUST_HAPPENED) |
  (ACC_INICIAL_2 = HAPPENING & next(ACC_INICIAL_2) =
  JUST_HAPPENED) |
  ...|
  (ACC_INICIAL_N = HAPPENING & next(ACC_INICIAL_N) =
  JUST_HAPPENED)
  ```
- ```
  NOMBRE_DEL_INTERVALO = ACTIVE  & next(NOMBRE_DEL_INTERVALO) =
  INACTIVE ->
  (ACC_FINAL_1 = HAPPENING & next(ACC_FINAL_1) = JUST_HAPPENED)
  |
  (ACC_FINAL_2 = HAPPENING & next(ACC_FINAL_2) = JUST_HAPPENED)
  |
  ...|
  (ACC_FINAL_N = HAPPENING & next(ACC_FINAL_N) = JUST_HAPPENED)
  ```

(7)

- ```
  ((ACC_INICIAL_1 = NOT_HAPPENING & next(ACC_INICIAL_1) =
  HAPPENING) |
  (ACC_INICIAL_2 = NOT_HAPPENING & next(ACC_INICIAL_2) =
  HAPPENING) |
  ...|
  (ACC_INICIAL_N = NOT_HAPPENING & next(ACC_INICIAL_N) =
  HAPPENING)) ->
  next(NOMBRE_DEL_INTERVALO) = INACTIVE
  ```

(8)

- ```
  ((ACC_FINAL_1 = NOT_HAPPENING & next(ACC_FINAL_1) =
  HAPPENING) |
  (ACC_FINAL_2 = NOT_HAPPENING & next(ACC_FINAL_2) = HAPPENING)
  |
  ...|
  (ACC_FINAL_N = NOT_HAPPENING & next(ACC_FINAL_N) =
  HAPPENING)) ->
  ```
  ```

```
next(NOMBRE_DEL_INTERVALO) = ACTIVE
```

(9)

- ```
  (next(ACC_INICIAL_1) = HAPPENING -> next(ACC_INICIAL_2) != HAPPENING & …& next(ACC_INICIAL_N) != HAPPENING ) &
  (next(ACC_INICIAL_2) = HAPPENING -> next(ACC_INICIAL_1) != HAPPENING & …& next(ACC_INICIAL_N) != HAPPENING ) &
  … &
  (next(ACC_INICIAL_N) = HAPPENING -> next(ACC_INICIAL_1) != HAPPENING & …& next(ACC_INICIAL_N-1) != HAPPENING)
  ```

(10)

- ```
  (next(ACC_FINAL_1) = HAPPENING -> next(ACC_FINAL_2) != HAPPENING & …& next(ACC_FINAL_N) != HAPPENING ) &
  (next(ACC_FINAL_2) = HAPPENING -> next(ACC_FINAL_1) != HAPPENING & …& next(ACC_FINAL_N) != HAPPENING ) &
  … &
  (next(ACC_FINAL_N) = HAPPENING -> next(ACC_FINAL_1) != HAPPENING & …& next(ACC_FINAL_N-1) != HAPPENING)
  ```

Para verificar el comportamiento, se puede agregar algunas fórmulas LTL:

Si una acción inicial pasa de HAPPENING a JUST_HAPPENED y el intervalo está INACTIVE, entonces debe pasar a ACTIVE.

- ```
  LTLSPEC G (((ACC_INICIAL_1 = HAPPENING & X(ACC_INICIAL_1) = JUST_HAPPENED) | (ACC_INICIAL_2 = HAPPENING & X(ACC_INICIAL_2) = JUST_HAPPENED) | ... |(ACC_INICIAL_N = HAPPENING & X(ACC_INICIAL_N) = JUST_HAPPENED) )-> (NOMBRE_DEL_INTERVALO = INACTIVE -> X(NOMBRE_DEL_INTERVALO) = ACTIVE))
  ```

Si una acción final pasa de HAPPENING a JUST_HAPPENED, entonces, si el intervalo está activo pasa a estar INACTIVE.

- ```
  LTLSPEC G (((ACC_FINAL_1 = HAPPENING & X(ACC_FINAL_1) = JUST_HAPPENED)| (ACC_FINAL_2 = HAPPENING & X(ACC_FINAL_2) = JUST_HAPPENED)|...| (ACC_FINAL_N = HAPPENING & X(ACC_FINAL_N) = JUST_HAPPENED)      )-> (NOMBRE_DEL_INTERVALO = ACTIVE  -> X(NOMBRE_DEL_INTERVALO) = INACTIVE ))
  ```

### 3.2.6 Intervalos con occurrences

Los intervalos pueden definirse con el modificador *occurrences* para lograr que ocurran, a lo sumo, la cantidad de veces especificadas. Para ello, se agrega en la sección

VAR una nueva variable por cada intervalo con esta característica, indicando que puede tomar valores en el rango numérico que va desde cero hasta el número máximo de ocurrencias posible:

- `NOMBRE_DEL_INTERVALO_OCCURRENCE: 0 .. VALOR_ESPECIFICADO;`

En la sección INIT la inicializamos en cero:

- `NOMBRE_DEL_INTERVALO_OCCURRENCE = 0`

Queremos que si el intervalo pasa del estado INACTIVE al de ACTIVE, entonces el contador se incremente en uno, y si no se hace esa transición, el contador conserve su valor. Para lograrlo, en la sección TRANS agregamos la siguiente fórmula:

- ```
  ((ACC_INICIAL_1 = HAPPENING & next(ACC_INICIAL_1) =
  JUST_HAPPENED | (ACC_INICIAL_2  = HAPPENING &
  next(ACC_INICIAL_2) = JUST_HAPPENED | … | (ACC_INICIAL_N  =
  HAPPENING & next(ACC_INICIAL_N) = JUST_HAPPENED )<->
  NOMBRE_DEL_INTERVALO_OCCURRENCE + 1 =
  next(NOMBRE_DEL_INTERVALO_OCCURRENCE) ) &
  (!(ACC_INICIAL_1  = HAPPENING & next(ACC_INICIAL_1) =
  JUST_HAPPENED | (ACC_INICIAL_2  = HAPPENING &
  next(ACC_INICIAL_2) = JUST_HAPPENED | … | (ACC_INICIAL_N  =
  HAPPENING & next(ACC_INICIAL_N) = JUST_HAPPENED )<->
  NOMBRE_DEL_INTERVALO_OCCURRENCE =
  next(NOMBRE_DEL_INTERVALO_OCCURRENCE) )
  ```

Para verificar el comportamiento, se puede agregar la siguiente fórmula LTL, que controla que la variable que representa el contador no supere el número permitido:

```
LTLSPEC G (NOMBRE_DEL_INTERVALO_OCCURRENCE <= VALOR_ESPECIFICADO)
```

### 3.2.7 Intervalos con “only occurs in scope”

Los intervalos pueden definirse indicando que sólo pueden ocurrir dentro de otro intervalo. Por ejemplo, se puede definir que el intervalo *PERIODO_EXÁMENES* sólo puede ocurrir durante el intervalo *AÑO_LECTIVO*. Para lograr esto, suponiendo que se quiere que *INTERVALO_MENOR* ocurra durante *INTERVALO_MAYOR,* en la implementación del autómata se agrega en la sección ‘`TRANS`’, la fórmula:

- `next(`*`INTERVALO_MENOR`*`) = ACTIVE -> next(`*`INTERVALO_MAYOR`*`) = ACTIVE`

### 3.2.8 Contadores

Un contador se representa con una variable numérica. El valor inicial de la variable es el indicado en su especificación. En caso de que no se indique este valor, arranca en cero. 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 la vez. Para lograr esto, en la implementación del autómata se define, por cada contador:

Una variable enumerada en la sección '`VAR`':

- `contador: MIN_VALUE`[7] `.. MAX_VALUE;`

Se inicializa en la sección '`INIT`' la variable con el valor inicial.

- `contador = VALOR_INICIAL`

En la sección *TRANS*, se agregan las condiciones para que el contador funcione según la definición.
Por cada acción que incremente el contador, se agrega

- `next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED -> next(CONTADOR) = CONTADOR + x`

siendo x igual al valor definido con el modificar *by* (o con el valor 1 en caso de que no se hubiera especificado).

Para las acciones que disminuyen al contador, se hace lo mismo que con las que incrementan pero en vez de sumar se resta:

- `next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED -> next(CONTADOR) = CONTADOR - x`

Por cada acción que reinicia el contador, se agrega

- `next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED -> next(CONTADOR) = VALOR_INICIAL`

Por cada acción que asigna un nuevo valor al contador (usando *sets with action*), se agrega

- `next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED -> next(CONTADOR) = VALOR_DEFINIDO`

En los cuatro casos anteriores, si junto al "increases with action", "decreases with action", "resets with action" o "sets with action", luego de la acción, se agrega "provided that(*condición*)", entonces la primera parte de la fórmula:

`next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED →`

se reemplaza por:

`next(NOMBRE_DE_LA_ACCIÓN) = JUST_HAPPENED & (`*condición*`) →`

Para lograr que si no ocurre ninguna de las acciones que modifican al contador, este no cambie de valor, agregamos lo siguiente:

[7] `MIN_VALUE` y `MAX_VALUE` son valores enteros constantes definidos en el mismo template de Velocity que aplican a todos los contadores. Estos valores actualmente se ajustan de manera manual teniendo en cuenta que si el rango entre los dos valores es muy amplio, la *performance* se reduce bastante, y si el rango es muy chico puede ocurrir que haya modelos que queden sin contemplar. En una futura versión de FL se permitirá definir estos dos valores en la especificación de cada uno de los contadores, señalando en cada caso qué debería suceder si intenta salirse de ese rango.

- `(next(ACTION_1) != JUST_HAPPENED | !(`*`condición_1`*`) ) &`
  `(next(ACTION_2) != JUST_HAPPENED | !(`*`condición_2`*`) ) & …&`
  `(next(ACTION_N) != JUST_HAPPENED | !(`*`condición_N`*`) )->`
  `next(CONTADOR) = CONTADOR`
-

siendo ACTION_1, ACTION_2, …, ACTION_N todas las acciones que modifican al contador (ya sea incrementándolo, disminuyéndolo, reiniciándolo o asignándole un nuevo valor). En este ejemplo se asumió que la ACTION_1 se definió con un "provided that (condición_1)", etc. Si no fuera así, simplemente no se agrega la parte "`| !(`*`condición_1`*`)`", etc.

Por último, tenemos que indicar que si ocurre una de las acciones que modifican al contador, entonces no puede ocurrir otra de ellas. Para ello agregamos:

- `next(ACTION_1) = JUST_HAPPENED -> next(ACTION_2) != JUST_HAPPENED & …& next(ACTION_N) != JUST_HAPPENED`
- `next(ACTION_2) = JUST_HAPPENED -> next(ACTION_1) != JUST_HAPPENED & …& next(ACTION_N) != JUST_HAPPENED`
- `next(ACTION_N) = JUST_HAPPENED -> next(ACTION_1) != JUST_HAPPENED & …& next(ACTION_N-1) != JUST_HAPPENED`

### 3.2.9 Timers

Un *timer* es una herramienta que permite modelar el paso del tiempo mediante una serie de eventos. Un ejemplo de *timer* puede ser: *primer_tiempo*, *entretiempo*, *segundo_tiempo*. Los *timers* se pueden usar para definir cuándo puede ocurrir una acción.

El comportamiento es similar al de las acciones, pero debe respetar las siguientes condiciones:

1. Ocurren en orden.
2. Cuando un evento termina comienza el siguiente.
3. Los eventos del *timer* solo pueden ocurrir una vez, no se repiten.

Para lograr esto, en la implementación del autómata se define, por cada *timer*:
Una variable enumerada en la sección '`VAR`' por cada evento:

```
EVENTO: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
```

(3) Una variable booleana para evitar que los eventos del timer se repitan.

```
PRIMER_EVENTO_FINALIZED: boolean;
```

Se inicializan en la sección '`INIT`' las nuevas variables. Los eventos todos en *NOT_HAPPENING*

```
EVENTO = NOT_HAPPENING
```

El flag booleano en false:

```
PRIMER_EVENTO_FINALIZED = FALSE
```

(3) El primer evento del *timer* sólo puede pasar de *NOT_HAPPENING* a *HAPPENING* si no ocurrió previamente. Para ello, en la sección *TRANS*, se agrega:

```
PRIMER_EVENTO = NOT_HAPPENING & next(PRIMER_EVENTO) = HAPPENING ->
!PRIMER_EVENTO_FINALIZED
```

(3) Cuando el primer evento del timer pasa del estado *HAPPENING* a *JUST_HAPPENED* se marca el flag indicando que ya ocurrió.

```
PRIMER_EVENTO = HAPPENING & next(PRIMER_EVENTO) = JUST_HAPPENED <->
!PRIMER_EVENTO_FINALIZED & next(PRIMER_EVENTO_FINALIZED)
```

(3) Una vez que el flag está en verdadero, conserva su valor

```
PRIMER_EVENTO_FINALIZED -> next(PRIMER_EVENTO_FINALIZED)
```

(1 y 3) Los eventos, si cambian de valor, sólo pueden ir a un determinado nuevo valor y se tienen que cumplir algunas condiciones.

```
EVENTO_1 != next(EVENTO_1) -> (
      (EVENTO_1 = NOT_HAPPENING & next(EVENTO_1) = HAPPENING)|
      (EVENTO_1 = HAPPENING & next(EVENTO_1) = JUST_HAPPENED &
      EVENTO_2 = HAPPENING)|
      (EVENTO_1 = JUST_HAPPENED & next(EVENTO_1) =
      NOT_HAPPENING)) &
EVENTO_2 != next(EVENTO_2) -> (
      (EVENTO_2 = NOT_HAPPENING & next(EVENTO_2) = HAPPENING &
      next(EVENTO_1) = JUST_HAPPENED)|
      (EVENTO_2 = HAPPENING & next(EVENTO_2) = JUST_HAPPENED &
      next(EVENTO_3) = HAPPENING)| (EVENTO_2=JUST_HAPPENED &
      next(EVENTO_2) = NOT_HAPPENING)) &
... &
EVENTO_N != next(EVENTO_N) -> (
      (EVENTO_N = NOT_HAPPENING & next(EVENTO_N) = HAPPENING &
      next(EVENTO_N-1) = JUST_HAPPENED)|
      (EVENTO_N = HAPPENING & next(EVENTO_N) = JUST_HAPPENED)|
      (EVENTO_N = JUST_HAPPENED & next(EVENTO_N) =
      NOT_HAPPENING))
```

Para verificar el comportamiento, se puede agregar algunas fórmulas LTL:

Valido que si un evento está en JUST_HAPPENED en el próximo estado estará en NOT_HAPPENING.

```
LTLSPEC G (EVENTO = JUST_HAPPENED -> X EVENTO = NOT_HAPPENING
)
```

Valido que se respete el orden de valores del intervalo. Si un evento está en JUST_HAPPENED, todos los anteriores ya deben haber estado en HAPPENING y en

JUST_HAPPENED y los eventos posteriores posteriores no estuvieron nunca en JUST_HAPPENED.

```
LTLSPEC G (EVENTO_X = JUST_HAPPENED ->
      O[8] EVENTO_1 = HAPPENING & O EVENTO_1 = JUST_HAPPENED &
      O EVENTO_2 = HAPPENING & O EVENTO_2 = JUST_HAPPENED &
      ... &
      O EVENTO_X-1 = HAPPENING & O EVENTO_X-1 = JUST_HAPPENED
&
      !O EVENTO_X+1 = JUST_HAPPENED &
      !O EVENTO_X+2 = JUST_HAPPENED &
      ... &
      !O EVENTO_N = JUST_HAPPENED &)
```

## 3.3 Cláusulas

Dentro de la especificación en FL, donde se define la teoría marco que genera al autómata, se definen todas las fórmulas que definen qué comportamiento está prohibido, es obligatorio o está permitido. Esto se hace usando los operadores deónticos *F, O* y *P* (de *Forbidden*, *Obligation* y *Permission* respectivamente).

Una de las características que se incorporaron a FL a partir de esta tesis es el uso de cuantificadores. Esto junto a los roles (otra de las nuevas funcionalidades) permite definir fórmulas muy ricas. Por ejemplo:

```
FORALL (i: estudiante; O( i.rendir_examen → EXISTS
(j:docente; j.tomar_examen)))
```

que dice que "es obligatorio para todo estudiante que cuando rinde un examen, debe haber por lo menos un docente que lo tome".

### 3.3.1 Sintaxis

A continuación presentamos un resumen de la sintaxis para que quede claro qué tipo de fórmulas se pueden generar en FL.

*cláusula* := P(*sentencia*)
O(*sentencia*) *(reparación)?*
F(*sentencia*) *(reparación)?*
EXISTS (var *(*: rol*)*? ; (O(*sentencia*) *(reparación)?* | F(*sentencia*) *(reparación)?*
| P(*sentencia*)) ) |
FORALL (var *(*: rol*)*? ; (O(*sentencia*) *(reparación)?* | F(*sentencia*) *(reparación)?*
| P(*sentencia*)) )

[8] Este es un operador temporal. *O p* (del inglés, *once p*) declara que una condición *p* se cumple en uno de los instantes anteriores.

*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*
*and* := &
*or* := |

### 3.3.2 Intervalos

Dentro de las fórmulas, se pueden usar intervalos de tres maneras. Una de ellas es con 'INSIDE (nomble_del_intervalo)'. Este es un predicado que es verdadero si se está dentro del intervalo (o sea, si el intervalo está activo) y falso en otro caso. Un ejemplo:

```
FORALL(i: estudiante; O(i.censar -> INSIDE(en_censo)))
```

indica que es obligatorio que si un estudiante se censa, entonces se esté dentro del intervalo 'en_censo'.

Una segunda alternativa para el uso de intervalos es con '<>_nombre_del_intervalo'. Aquí, luego de '<>_nombre_del_intervalo' debe ir una fórmula que, si el intervalo comienza, tiene que ser verdadera antes de que ese intervalo termine. Por ejemplo:

```
FORALL(i: estudiante; O( <>_{en_censo} i.censar))
```

dice que es obligatorio que todo estudiante se cense durante el periodo de censo.

La tercera forma en que se pueden usar los intervalos es con '[]_nombre_del_intervalo'. Aquí, luego de '[]_nombre_del_intervalo' debe ir una fórmula que tiene que ser verdadera durante todo ese intervalo. Por ejemplo:

```
FORALL(i: comerciante; O([]_{jornada_electoral}
!i.vender_alcohol))
```

dice que es obligatorio, para todo comerciante, no vender alcohol durante la jornada electoral.

### 3.3.3 Expansión de cuantificadores

A continuación, mediante ejemplos, veremos cómo se comportan las fórmulas con cuantificadores al expandirse con agentes. Supongamos que se definen los roles *estudiante* y *docente*, el periodo *en_censo* y algunas acciones:

```
roles estudiante, docente
action censar only performable by estudiante
action empieza_censo, termina_censo
global interval en_censo defined by actions empieza_censo-
termina_censo
```

Dada la especificación anterior, la herramienta genera los agentes que cumplen los distintos roles:

```
agent_1: estudiante
agent_2: docente
agent_3: estudiante, docente
```

Cuando se usan los cuantificadores, estos se expanden de la manera habitual, por ejemplo dado el rol estudiante y la fórmula:

```
FORALL(i: estudiante; O( <>_{en_censo} i.censar))
```

se expande usando los agentes antes definidos:

```
O(<>_{en_censo} agent_1.censar) & O(<>_{en_censo}
agent_3.censar)
```

Si el censo fuera para todos y no solo para estudiantes, debería expresarse en la fórmula sin identificar el rol, como en la siguiente fórmula:

```
FORALL (i; O( <>_{en_censo} i.censar))
```

y la expansión sería:

```
O(<>_{en_censo} agent_1.censar) &
O(<>_{en_censo}agent_2.censar) & O(agent_3.censar)
```

Si en vez del cuantificador universal se usara el existencial, se reemplazaría la conjunción por una disyunción:

```
EXISTS (i: estudiante; O( <>_{en_censo} i.censar))
```

se expande:

```
O(<>_{en_censo} agent_1.censar) | O(<>_{en_censo}
agent_3.censar)
```

En los ejemplos anteriores, se usó el periodo `en_censo`, asumiendo que se definió como un periodo global, o sea, es el mismo periodo para todos los agentes. Si se lo hubiera definido como local, entonces la fórmula se debería escribir de otra manera y su expansión sería distinta (notar el "`i.en_censo`" en lugar de "`en_censo`" a secas):

```
FORALL(i: estudiante; O( <>_{i.en_censo} i.censar))
```

se expande, asumiendo los mismos agentes que antes:

```
O(<>_{agent_1.en_censo} agent_1.censar) &
O(<>_{agent_3.en_censo} agent_2.censar)
```

Algo similar ocurre con los contadores si se los define como locales. Al expandir las fórmulas que los usan, estos se reemplazan por los contadores nuevos que incluyen al agente correspondiente.

No se permiten fórmulas con el cuantificador FORALL sobre un rol y que ejecuten acciones que no son de ese rol, por ejemplo, no está permitido:

```
O( FORALL(i:comprador; i.comprar | i.vender)).
```

Si bien puede haber agentes que tengan ambos roles (en este ejemplo 'comprador' y 'vendedor') la herramienta, en estos casos, reporta un error al intentar hacer un análisis.

Tampoco se permiten fórmulas con el cuantificador EXISTS sobre un rol y que ejecuten acciones que no son de ese rol y tengan el operador *F* u *O.* El único caso válido es con el operador *P.* Un ejemplo válido sería:

```
P( EXISTS(i:comprador; i.comprar & i.vender)).
```

Veamos un ejemplo de fórmulas que pueden parecer similares, pero que en realidad son distintas:

1. `FORALL(j:estudiante; F(j.cometer_falta)`
2. `F(FORALL(j:estudiante; j.cometer_falta)`

En la primera se dice que todos los estudiantes tienen prohibido cometer una falta, en cualquier momento. En la segunda, lo que está prohibido es que todos los estudiantes cometan la falta a la vez. Imaginemos una traza donde los estudiantes comenten faltas pero no a la vez. En ese caso se viola la primera fórmula pero no la segunda. Esto se ve claramente si se comparan ambas fórmulas expandidas. Veamos cómo quedan si los agentes que son estudiantes son dos, agent_1 y agent_2:

1. `G(!agent_1.commit_fault = JUST_HAPPENED) &`
   `G(!agent_2.commit_fault = JUST_HAPPENED )`

```
2. G(!( agent_1.commit_fault = JUST_HAPPENED &
   agent_2.commit_fault = JUST_HAPPENED ))
```

En el primer punto decimos que nunca sucede que agent_1 cometa una falta y nunca sucede que agent_2 cometa una falta. En la segunda fórmula estamos diciendo que nunca se llega a un estado en el que los dos agentes hayan cometido una falta.

### 3.3.4 Generación de fórmulas LTL

Antes de invocar al *model checker* se debe traducir las fórmulas 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 $\varphi$ y $\rho$ como fórmulas FL, a *i* como nombre de un intervalo, a *x* como variable libre y a *R* como el nombre de un rol. Sea *F* un conjunto de fórmulas FL que definen el conjunto de modelos legales para el NSUA. Primero se separan los permisos del resto y se define el dominio de *Tr como* $F_{p'} = \{\varphi \mid \varphi \in F$ tal que $\varphi$ no sea de la forma $P(\psi)\}$:

| | | |
|---|---|---|
| $Tr(\diamond_i \varphi)$ | = | i = ACTIVE $\rightarrow$ (i = ACTIVE $U$ $Tr(\varphi)$) |
| $Tr(F(\varphi))$ | = | $\square\neg Tr(\varphi)$ |
| $Tr(F(\varphi)$ repaired by $\rho)$ | = | $\square(Tr(\varphi) \rightarrow Tr(\rho))$ |
| $Tr(O(\varphi))$ | = | $\square\, Tr(\varphi)$ |
| $Tr(O(\varphi)$ repaired by $\rho)$ | = | $\square(\neg Tr(\varphi) \rightarrow Tr(\rho))$ |
| $Tr(O(\diamond \varphi))$ | = | $\diamond Tr(\varphi)$ |
| $Tr(\varphi \,\&\, \rho)$ | = | $Tr(\varphi)$ & $Tr(\rho)$ |
| $Tr(!\varphi)$ | = | $!Tr(\varphi)$ |
| $Tr(\varphi \mid \rho)$ | = | $Tr(\varphi)$ \| $Tr(\rho)$ |
| $Tr(\varphi \rightarrow \rho)$ | = | $Tr(\varphi) \rightarrow Tr(\rho)$ |
| $Tr$(una_acción) | = | una_acción = JUST_HAPPENED |
| $Tr$(INSIDE (i)) | = | i = ACTIVE |
| $Tr$(FORALL(x:R; $\varphi$)) | = | $\wedge_{a \in R}\ Tr(\varphi[x \leftarrow a])$[9] |

[9] Es decir, la conjunción sobre todos los agentes que tienen el rol R de la traducción de la fórmula $\varphi$ con la variable x instanciada en cada rol, tal como se describe en la sección 3.3.3 Expansión de cuantificadores.

$Tr(\text{EXISTS(x:R; } \varphi))$ = $\vee_{a \in R} Tr(\varphi[x \leftarrow a])$[10]

Tomemos al autómata *A* definido por la teoría marco y a $C_A$ como la clase de modelos que representa todas las corridas del autómata *A*. Sea *F* el conjunto de fórmulas FL que representan el NSUA. La clase de modelos legales definido por *F* sobre $C_A$ se define como:

$$C_A^F = \{M \in C_A \mid M \vDash Tr(F_{p'})\}.$$

Esto es, cada modelo legal debe satisfacer las obligaciones y prohibiciones especificadas en *F*.

Los permisos son una verificación que se debe hacer en $C_A^F$ para garantizar la coherencia. La condición que $C_A^F$ debe satisfacer es: para cada $\varphi$ de la forma $P(\psi)$ en *F* debe haber un modelo *M* en $C_A^F$ tal que $M \vDash Tr(\psi)$. O sea, si algo está permitido, entonces el resto de el NSUA no impide que suceda.

Veamos un ejemplo donde se usan varias de las traducciones. Queremos expresar que "está prohibido para todos los estudiantes cometer una falta, y si la cometen tienen prohibido entrar al edificio durante el periodo que dure el castigo":

```
FORALL(j:student; F(j.commit_fault) repaired by
(<>_{j.penalty_period} !INSIDE (j.inside_building)))
```

se traduce a[11]:

```
G( agent_1.commit_fault = JUST_HAPPENED ->
(agent_1.penalty_period = ACTIVE -> (agent_1.penalty_period =
ACTIVE U !agent_1.inside_building = ACTIVE )) )
```

En este ejemplo hay un solo agente que es estudiante. Si hubiera otro agente, a la fórmula anterior se le agregaría una conjunción seguida de la misma fórmula pero con el otro agente, y lo mismo se agregaría si hubiera otros agentes que tuvieran ese rol.

## 3.4 Corriendo el programa

Para correr nuestro programa hay que pasarle como único parámetro el path y nombre de un archivo de configuración donde se definen todos los parámetros necesarios. Los más importantes de ellos son el archivo de entrada (con la especificación en FL), la

[10] Esto es, la disyunción sobre todos los agentes que tienen el rol R de la traducción de la fórmula $\varphi$ con la variable x instancia en cada rol, tal como se describe en la sección 3.3.3 Expansión de cuantificadores.

[11] Nótese que anteriormente se usaron los operadores simbólicos □ y ◇, que corresponden a los operadores textuales aquí utilizados *G* (de *Globally*) y *F* (de *Finally*) respectivamente

dirección del binario que ejecuta al *model checker*, el *template* de Velocity a usar y un directorio de salida.

Esta herramienta al ejecutarse, básicamente, sigue los siguientes pasos:

1. Parsea el archivo de entrada.
2. Valida la consistencia y la sintaxis del mismo.
3. A partir de los roles definidos se generan los agentes y con estos se definen las nuevas acciones, intervalos y contadores, y se actualizan todas las referencias entre ellos.
4. Con todo lo generado en el punto anterior se invoca al motor de Velocity para que genere, en un archivo, al autómata.
5. Se expanden las fórmulas instanciándose con los agentes que correspondan.
6. Se traducen las fórmulas de FL a LTL.
7. Se genera una nueva fórmula que contiene la conjunción de todas las fórmulas LTL que sean Obligaciones y Prohibiciones (excluyendo por ahora los Permisos). Luego se crea otra nueva fórmula que es la negación de la fórmula recién creada.
8. Se genera un archivo con los comandos que se le pasarán al *model checker*, donde se indica cuál es el archivo que contiene al autómata y cuál es la fórmula que se va a validar (la última generada en el punto anterior).
9. Se invoca al *model checker* con el archivo de comandos generados en el punto anterior, el resultado de la validación queda en un archivo.
10. Se analiza el resultado que indica si se encontró un comportamiento válido (lo cual reflejaría que las normas son consistentes). Si no lo encontró, se informa al usuario y se termina el programa.
11. Si el conjunto de reglas formado por las Obligaciones y Prohibiciones son válidas, a continuación se validan los permisos. Se hacen tantas validaciones como permisos hubiere, agregando en cada validación un permiso al conjunto de reglas usados en el punto 7 y negando esa fórmula. Luego, por cada permiso, se repiten los pasos 8, 9 y 10.

El hecho de dejar en archivos al autómata, los comandos, la salida del *model checker* y el log generado por el programa (que contiene todos los detalles de cómo se generó el autómata, la creación de agentes, las nuevas acciones, contadores e intervalos y la expansión de las fórmulas, entre otra información), nos facilita luego la búsqueda de problemas (en caso de que los hubiere) y nos permite modificar a mano estos archivos para hacer distintos tipos de nuevas pruebas (por ejemplo, para analizar la *performance* del programa ante distintas situaciones o, si las normas no son válidas, buscar cuáles de ellas son las responsables).

En el punto 7 se construye la fórmula final a validar, cuyo último paso es hacer una negación de toda la fórmula anterior. Esto es así, ya que el *model checker*, en caso de no encontrar una traza válida devuelve un contraejemplo. Con la negación, nosotros logramos que si encuentra un contraejemplo ese sea en realidad una traza válida de nuestro modelo, lo que nos garantiza que sea válido.

Hasta la versión anterior de esta herramienta, toda la generación del autómata, su estructura y su traducción al formato esperado por el *model checker* se hacía directamente

desde Java. Con la adopción de la herramienta *Velocity*, buscamos desacoplar la decisión del *model checker* elegido y simplificar toda la generación del autómata. Esta simplificación, junto a otros cambios, nos permitieron reducir la cantidad de fuentes en más de un 50% y la cantidad de líneas de código en un 5% aproximadamente, a pesar de todas las funcionalidades incorporadas. Esta gran disparidad entre la reducción de fuentes y líneas de código, se debe a que el 78% de estas últimas pertenecen a los fuentes generados con JavaCC y el gran crecimiento de estas clases es el resultado de la ampliación del lenguaje FL.

# 4 Caso de estudio

Presentamos ahora un ejemplo de un reglamento, donde se verá cómo se especifica en FL y mostraremos el resultado del análisis con FormaLex. El ejemplo que utilizaremos es un resumen de un hipotético reglamento universitario que, si bien es breve, mostrará el uso de varias de las características de FL.

Extracto del reglamento:

1. Capítulo 1, Estudiantes.
    a. Todo individuo que se haya inscripto a una carrera y que aún no se haya graduado en la misma es considerado un estudiante de esta Universidad.
    b. Los estudiantes deben mostrar respeto mutuo. Las faltas disciplinarias graves se castigarán impidiendo el acceso a las instalaciones universitarias durante el año posterior a la falta.
    c. Los estudiantes tienen derecho a: ..., participar en actividades de investigación, ...
2. Capítulo 2, Docentes.
    a. Existen tres categorías docentes: c1) ayudantes-alumnos, c2) auxiliares, c3) profesores.
    b. Los docentes se eligen por concurso, y los aspirantes deben anotarse en el mismo a partir de la fecha de apertura. La selección se realizará de acuerdo a los siguientes criterios: [omitidos por no ser relevantes para el caso de estudio].
    c. Los aspirantes a ayudantes-alumnos deberán ser estudiantes al momento del concurso.
    d. Los docentes deben cumplir sus funciones a partir de los 30 días siguientes a la finalización del concurso en el que hayan resultado seleccionados.
    e. Si bien se permite el trabajo desde el hogar, se requiere que los docentes pasen al menos un día a la semana en las instalaciones de la Casa de Estudios.
3. Capítulo 3, Investigación.
    a. Las actividades científicas solo pueden ser realizadas por miembros de grupos de investigación.
    b. Los grupos de investigación están conformados por profesores y auxiliares docentes.
4. Capítulo 4, Biblioteca de la Universidad.
    a. Los alumnos podrán retirar hasta un máximo de tres libros.
    b. Los docentes podrán retirar más de tres libros.

Si bien lo ideal sería modelar todo el reglamento en una sola especificación, en este caso, para simplificar y para poder ver distintas características de FL, haremos dos especificaciones. Parece natural, para este reglamento, definir los roles *estudiante* y *docente*. En este caso, la herramienta definirá tres agentes, uno con el rol de *estudiante*, otro con el de *docente* y el último con los dos roles a la vez. El problema aquí es que este último

agente complica el modelado de los primeros capítulos, donde se necesita que un estudiante, para además ser docente (es el caso de la categoría *ayudante-alumno*), tiene que cumplir ciertos pasos (presentarse a concurso y ganarlo) y con el tiempo agrega un rol al que ya tenía. Modelarlo de esta manera sería demasiado complejo y nos alejaría del objetivo de este capítulo que es mostrar con un ejemplo simple las características de FL. Para poder ver un ejemplo del uso de roles, haremos una segunda especificación donde modelaremos solamente el capítulo cuatro y, aprovechando que son pocas normas, mostraremos más detalles de la generación del autómata y de los resultados obtenidos.

En la primera especificación, modelaremos los capítulos 1, 2 y 3. Abreviamos aquí, para simplificar, las declaraciones de algunas de las acciones, asumiendo que ya fueron escritas. Empecemos con el intervalo que define qué es un estudiante:

```
local interval estudiante defined by actions inscribirse -
graduarse
```

Los alumnos, no deben cometer faltas graves o se les prohibirá el acceso durante un año. Definiremos dos intervalos para modelar esto.

```
local interval prohibición defined by actions cometer_falta -
un_año_después
local interval dentro_del_edificio defined by actions entrar
- salir
```

La prohibición se define así:

(1) `FORALL(i; F(i.cometer_falta & INSIDE(i.estudiante)) repaired by (<>_{i.prohibición} !INSIDE (i.dentro_del_edificio) ) )`

e indica que la falta no se debe cometer, pero si ocurre, durante el periodo de prohibición, el estudiante no puede entrar al edificio.

El artículo 1c permite a los estudiantes hacer investigación. Definir este permiso sirve para invalidar futuras prohibiciones:

(2) `FORALL(i; P( INSIDE(i.estudiante) -> i.hacer_investigación ))`

Para los concursos, debemos modelar el plazo del concurso y sus posibles resultados, que son ser seleccionado para alguna de las categorías o no ser elegido:

```
action elegir_ganadores output values {doc_c1, doc_c2,
doc_c3, no_elegido}
global interval concurso defined by actions abrir_concurso -
elegir_ganadores
action postularse only occurs in scope concurso
```

Modelaremos el requisito para la categoría c1 (ayudantes-alumnos). Para abreviar, modelaremos solo esta categoría:

(3) `FORALL(i; O(<>_{concurso} (postularse -> INSIDE(i.estudiante))))`

Los docentes deben empezar a cumplir su cargo a los 30 días de resultar electos:

```
global interval periodo_de_gracia defined by actions
elegir_ganadores - 30_días_después
local interval en_el_puesto defined by actions
30_días_después - infinite
```

La obligatoriedad de la visita semanal se escribe así:

(4)
```
global interval semana defined by actions comienza_semana -
termina_semana

FORALL(i; O(i.postularse & (elegir_ganadores.doc_c1 |
elegir_ganadores.doc_c2 | elegir_ganadores.doc_c3) ->
<>_{semana} i.entrar))
```

y básicamente dice que si es docente, entonces tiene que entrar al edificio por lo menos una vez a la semana.

En el capítulo 3, la restricción de las actividades científicas a miembros de los grupos de investigación, se especifica:

(5)
```
FORALL(i; F(i.hacer_investigación &
!i.unirse_a_grupo_de_investigación))
```

La otra restricción del capítulo, que señala que se debe ser profesor o auxiliar docente para estar en un grupo de investigación, se escribe:

(6)
```
FORALL(i; F(i.unirse_a_grupo_de_investigación &
!(elegir_ganadores.doc_c2 | elegir_ganadores.doc_c3)))
```

El capítulo 4 lo dejamos para la próxima especificación. Al correr esta especificación con nuestra herramienta, luego de casi 600 milisegundos[12], se informa por consola el siguiente mensaje:

```
No se ha encontrado un comportamiento legal para las normas.
```

A partir del análisis con FormaLex, surgen dos problemas de coherencia. El primero está relacionado con la norma que pide un mínimo de una visita semanal (art. 2e). Se detecta que la reparación de la regla que prohíbe las faltas graves (1) contradice la obligación de la visita semanal (4). El problema de coherencia radica en que existe la posibilidad de que un

[12] Las pruebas se hicieron en un equipo con procesador Intel i5-2520M 2.5 GHz, 4 GB de RAM y sistema operativo Windows 7.

estudiante cometa una falta grave y se le prohíba entrar al edificio, se postule para ser docente, sea elegido y luego no pueda cumplir con su visita semanal.

El segundo problema de coherencia aparece por permitir la investigación científica solo a auxiliares y profesores (reglas 5 y 6) y permitir a los alumnos participar en actividades de investigación (en la regla 2).

A continuación, modelaremos el capítulo 4, sobre la biblioteca, en una nueva especificación. Aquí cambiaremos el enfoque en cuanto a los roles y tomaremos a los alumnos y a los docentes como roles, sin preocuparnos por el momento en que un alumno se puede convertir también en docente (con la categoría ayudante-alumno).

Primero definimos los roles y algunas acciones:

```
roles alumno, docente
actions retirar_libro, devolver_libro only performable by
alumno, docente
```

Definimos el contador que indica la cantidad de libros prestados:

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

Escribimos las reglas 1 y 2 que indican, respectivamente, que los alumnos no pueden sacar más de tres libros y los docentes sí pueden:

```
FORALL(i:alumno; F(i.libros_retirados > 3))
FORALL(i:docente; P(i.libros_retirados > 3))
```

Con esta especificación, invocamos a nuestro programa. Para el análisis, la herramienta genera tres agentes, con los roles asignados:

```
agent_1: alumno, docente
agent_2: alumno
agent_3: docente
```

y genera el autómata con toda la teoría marco. A continuación mostramos un extracto del autómata generado.

Comienza con las palabras reservadas `MODULE main` y `VAR`, para luego definir todas las variables con sus posibles valores. En este caso, como todos los roles pueden ejecutar las dos acciones, se generan variables para cada combinación de agente-acción. Además, como el contador `libros_retirados` es local, se genera un contador por cada agente. El rango de valores posibles de los contadores, sale de una constante definida en el *template* que genera al autómata (en una futura versión de FL se permitirá definir este rango en el mismo lugar donde se define el contador).

```
MODULE main
VAR
```

```
     agent_1.retirar_libro: {HAPPENING, NOT_HAPPENING,
JUST_HAPPENED};
     agent_1.devolver_libro: {HAPPENING,NOT_HAPPENING,
JUST_HAPPENED};
     agent_1.libros_retirados: 0 .. 20;
     agent_2.retirar_libro: {HAPPENING, NOT_HAPPENING,
JUST_HAPPENED};
     agent_2.devolver_libro: {HAPPENING,NOT_HAPPENING,
JUST_HAPPENED};
     agent_2.libros_retirados: 0 .. 20;
     agent_3.retirar_libro: {HAPPENING, NOT_HAPPENING,
JUST_HAPPENED};
     agent_3.devolver_libro: {HAPPENING,NOT_HAPPENING,
JUST_HAPPENED};
     agent_3.libros_retirados: 0 .. 20;
```

A continuación se inicializan las variables, luego de agregar la palabra reservada `INIT` que marca el comienzo de la sección:

```
     INIT
     agent_1.devolver_libro = NOT_HAPPENING &
     agent_1.retirar_libro = NOT_HAPPENING &
     agent_1.libros_retirados =  0 &
     ...
```

y se repite esa asignación de valores para las variables de los otros dos agentes. Por último, se agrega la palabra reservada `TRANS` y a continuación se definen las fórmulas proposicionales que definen el comportamiento del autómata. Veremos algunas de estas fórmulas. Si una acción está en estado `JUST_HAPPENED` en el siguiente estado debe pasar a `NOT_HAPPENING`:

```
     agent_1.devolver_libro = JUST_HAPPENED ->
     next(agent_1.devolver_libro) = NOT_HAPPENING
```

Si la acción está en estado `HAPPENING` en el siguiente estado puede conservar ese valor o cambiar al de `JUST_HAPPENED`:

```
     agent_1.devolver_libro = HAPPENING ->(
     next(agent_1.devolver_libro) = HAPPENING |
     next(agent_1.devolver_libro) = JUST_HAPPENED )
```

Se definen las fórmulas que definen el comportamiento del contador:

```
     (next(agent_1.devolver_libro) = JUST_HAPPENED   ->
     next(agent_1.libros_retirados) = agent_1.libros_retirados -1)
     &
```

```
(next(agent_1.retirar_libro) = JUST_HAPPENED    ->
next(agent_1.libros_retirados) = agent_1.libros_retirados +
1)
```

Todas estas fórmulas que mostramos acá, se repiten para la otra acción (`retirar_libro`) y para los demás agentes.

Una vez generado el autómata, el programa, expande la primera fórmula:

```
FORALL(i:alumno; F(i.libros_retirados > 3))
```

generando:

```
F(agent_1.libros_retirados > 3) & F(agent_2.libros_retirados
> 3)
```

Luego traduce esta fórmula a LTL:

```
G(!(agent_1.libros_retirados > 3)) &
G(!(agent_2.libros_retirados > 3))
```

Una vez generado el autómata y traducido las fórmulas, se invoca al *model checker* para buscar una *traza.* En este caso el *model checker* en menos de 100 milisegundos informa que ha encontrado una traza, verificando así que existe un comportamiento legal. El sistema informa en qué archivo quedó la traza encontrada. Mostramos a continuación la traza, donde se ve los valores de las variables en cada transición. Para simplificar, no mostramos las variables si no cambian de valor entre un estado y el siguiente. Para que se vea mejor el ejemplo, le indicamos al *model checker* que evaluará un mínimo de 15 transiciones antes de terminar. El archivo generado tiene lo siguiente:

```
-- specification !( G !(agent_1.libros_retirados > 3) &
G !(agent_2.libros_retirados > 3))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  agent_3.devolver_libro = NOT_HAPPENING
  agent_1.devolver_libro = NOT_HAPPENING
  agent_1.retirar_libro = NOT_HAPPENING
  agent_2.devolver_libro = NOT_HAPPENING
  agent_3.retirar_libro = NOT_HAPPENING
  agent_2.retirar_libro = NOT_HAPPENING
  agent_1.libros_retirados = 0
  agent_2.libros_retirados = 0
  agent_3.libros_retirados = 0
-> State: 1.2 <-
```

```
  agent_1.retirar_libro = HAPPENING
-> State: 1.3 <-
  agent_1.retirar_libro = JUST_HAPPENED
  agent_1.libros_retirados = 1
-> State: 1.4 <-
  agent_1.retirar_libro = NOT_HAPPENING
  agent_2.retirar_libro = HAPPENING
-> State: 1.5 <-
  agent_1.retirar_libro = HAPPENING
  agent_2.retirar_libro = JUST_HAPPENED
  agent_2.libros_retirados = 1
-> State: 1.6 <-
  agent_1.retirar_libro = JUST_HAPPENED
  agent_2.retirar_libro = NOT_HAPPENING
  agent_1.libros_retirados = 2
-> State: 1.7 <-
  agent_1.retirar_libro = NOT_HAPPENING
  agent_2.retirar_libro = HAPPENING
-> State: 1.8 <-
  agent_1.retirar_libro = HAPPENING
  agent_2.retirar_libro = JUST_HAPPENED
  agent_2.libros_retirados = 2
-> State: 1.9 <-
  agent_1.retirar_libro = JUST_HAPPENED
  agent_2.retirar_libro = NOT_HAPPENING
  agent_1.libros_retirados = 3
-> State: 1.10 <-
  agent_1.retirar_libro = NOT_HAPPENING
  agent_3.retirar_libro = HAPPENING
-> State: 1.11 <-
  agent_3.retirar_libro = JUST_HAPPENED
  agent_2.retirar_libro = HAPPENING
  agent_3.libros_retirados = 1
-> State: 1.12 <-
  agent_3.retirar_libro = NOT_HAPPENING
  agent_2.retirar_libro = JUST_HAPPENED
  agent_2.libros_retirados = 3
-> State: 1.13 <-
  agent_1.devolver_libro = HAPPENING
  agent_2.retirar_libro = NOT_HAPPENING
-> State: 1.14 <-
  agent_1.devolver_libro = JUST_HAPPENED
  agent_2.devolver_libro = HAPPENING
  agent_1.libros_retirados = 2
-> State: 1.15 <-
  agent_2.devolver_libro = JUST_HAPPENED
```

```
  agent_2.libros_retirados = 2
-- Loop starts here
-> State: 1.16 <-
  agent_2.devolver_libro = NOT_HAPPENING
```

Este *trace* muestra un comportamiento válido de nuestro reglamento.

A continuación, se expande la segunda fórmula con el permiso y se hace la validación con el *model checker*. En este caso, en menos de 30 milisegundos, el sistema informa:

```
No se ha encontrado un comportamiento legal para el permiso.
```

y termina el programa. Este problema de coherencia se debe a que el agente *agent_1*, que es alumno y docente a la vez, por su carácter de alumno no puede retirar más de tres libros, pero por su condición de docente sí tiene ese permiso. Esta inconsistencia es la que no permite encontrar una *traza* con un comportamiento legal.

# 5 Conclusiones y trabajo futuro

Hemos presentado una extensión del lenguaje FL, incorporando nuevas características para enriquecer nuestra herramienta permitiendo así poder modelar normas que antes no se podían modelar o que era bastante complejo hacerlo. El enfoque de FormaLex, que parte de la base de que los sistemas normativos son muy similares a las especificaciones de *software*, busca aprovechar la experiencia en el campo de la informática con las herramientas desarrolladas para analizar las especificaciones, y utilizar estas últimas para hacer un análisis de coherencia de los documentos normativos. Hemos presentado en detalle todos los componentes del lenguaje y sus características, y mostramos cómo funciona la herramienta. Se ha analizado un caso de estudio que muestra la utilidad de la herramienta a la hora de encontrar inconsistencias en un reglamento. Creemos haber logrado una mejora de la herramienta dando así un paso más hacia un objetivo más ambicioso, que es proveer herramientas prácticas para evitar generar normas y reglamentos con inconsistencias.

Una de las limitaciones de la herramienta se relaciona con cuestiones de *performance*. Este es uno de los puntos en los cuales hay que seguir investigando, utilizando distintos enfoques, para lograr poder trabajar con un conjunto de normas más extenso. Otra de las cuestiones en las hay que seguir trabajando es en la detección automática de las normas que generan la inconsistencia, en el caso de que no hubiera un modelo que satisfaga la conjunción de fórmulas.

Además, a medida que se avanzaba con este trabajo, se detectaron varias mejoras que se le puede hacer al lenguaje (no listadas acá porque sería engorroso explicarlas todas), que no se incorporaron a esta tesis por cuestiones de tiempo, pero con las cuales se confeccionó una lista de tareas pendientes, que pueden ser abordadas en futuros trabajos sobre la herramienta.

# 6 Referencias


[ABF+07] T. Agnoloni, L. Bacci, E. Francesconi, P. Spinosa, D. Tiscornia, S. Montemagni, and G. Venturi. Building an ontological support for multilingual legislative drafting. In Proceedings of the 2007 conference or Legal Knowledge and Information Systems, pages 9-18, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[AvdHRA+09] Thomas Agotnes, Wiebe van der Hoek, Juan A. Rodríguez-Aguilar, Carles Sierra, and Michael Wooldridge. A Temporal Logic of Normative Systems. Towards Mathematical Philosophy, pages 69-106, 2009.

[BBF06] Julien Brunel, Jean-Paul Bodeveix, and Mamoun Filali. A state/event temporal deontic logic. In Lou Goble and John-Jules Ch. Meyer, editors, DEON, volume 4048 of Lecture Notes in Computer Science, pages 85-100. Springer, 2006.

[BB+06] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. DiVinE - A Tool for Distributed Verification (Tool Paper). In Computer Aided Verification, volume 4144/2006 of LNCS, pages 278-281. Springer Berlin / Heidelberg, 2006.

[BDDM04] Jan Broersen, Frank Dignum, Virginia Dignum, and John-Jules Ch. Meyer. Designing a deontic logic of deadlines. In Alessio Lomuscio and Donald Nute, editors, DEON, volume 3065 of Lecture Notes in Computer Science, pages 43-56. Springer, 2004.

[BRV02] Patrick Blackburn, Maarten de Rijke, Yde Venema. Modal Logic (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press, 2002.

[CCGR00] 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.

[FMDR10] Tim French, John Christopher McCabe-Dansted, and Mark Reynolds. Axioms for obligation and robustness with temporal logic. In Guido Governatori and Giovanni Sartor, editors, DEON, volume 6181 of Lecture Notes in Computer Science, pages 66-83. Springer, 2010.

[FPS09] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Clan: A tool for contract analysis and conflict discovery. In Zhiming Liu and Anders P. Ravn, editors, ATVA, volume 5799 of Lecture Notes in Computer Science, pages 90-96. Springer, 2009.

[GCM09] Goga, Nicolae, Stefania Costache, and Florica Moldoveanu. "A formal analysis of ISO/IEEE P11073-20601 standard of medical device communication." Systems Conference, 2009 3rd Annual IEEE. IEEE, 2009.

[GMS11] Daniel Gorín, Sergio Mera, and Fernando 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.

[GP09] G. Governatori and D.H. Pham. Dr-contract: An architecture for e-contracts in defeasible logic. International Journal of Business Process Integration and Management, 4(3):187-199, 2009.

[GRS05] Guido Governatori, Antonino Rotolo, and Giovanni Sartor. Temporalised normative positions in defeasible logic. In ICAIL, pages 25-34. ACM, 200

[HBBB07] Rinke Hoekstra, Joost Breuker, Marcello Di Bello, and Alexander Boer. The lkif core ontology of basic legal concepts. In Pompeu Casanovas, Maria Angela Biasiotti, Enrico Francesconi, and Maria-Teresa Sagri, editors, LOAIT, volume 321 of CEUR Workshop Proceedings, pages 43-63. CEUR-WS.org, 2007

[Hol03] Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.

[HPvdT07] Jorg Hansen, Gabriella Pigozzi, and Leendert W. N. van der Torre. Ten philosophical problems in deontic logic. In Guido Boella, Leendert W. N. van der Torre, and Harko Verhagen, editors, Normative Multi-agent Systems, volume 07122 of Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[HT06] S. Hagiwara and S. Tojo. Discordance Detection in Regional Ordinance: Ontologybased Validation. In Proceedings of the 2006 conference on Legal Knowledge and Information Systems: JURIX 2006: The Nineteenth Annual Conference, pages 111-120. IOS Press, 2006.

[Kel34] Hans Kelsen. Teoría pura del derecho. Universidad Nacional Autónoma de México, Dirección General de Publicaciones, Instituto de Investigaciones Jurídicas, 2 edición, 1982 (1934).

[LMPV09] Alessandro Lenci, Simonetta Montemagni, Vito Pirrelli, and Giulia Venturi Ontology learning from italian legal texts. In Proceedings of the 2009 conference on Law, Ontologies and the Semantic Web, pages 75-94, Amsterdam, The Netherlands, The Netherlands, 2009. IOS Press.

[MWD98] J.J.C. Meyer, R.J. Wieringa, and F.P.M. Dighum. The role of deontic logic in the specification of information systems. In Logics for Databases and Information Systems, pages 71-116. Kluwer, 1998.

[Per68] C. Perelman. What Is Legal Logic. Isr. L. Rev., 3:1, 1968.

[Pio10] G. Piolle. A Dyadic Operator for the Gradation of Desirability. Deontic Logic in Computer Science, pages 33-49, 2010.

[PS09] Cristian Prisacariu and Gerardo Schneider. CL: An action-based logic for reasoning about contracts. In WoLLIC '09: Proceedings of the 16th

International Workshop on Logic, Language, Information and Computation, pages 335-349, Berlin, Heidelberg, 2009. Springer-Verlag.

[S98] Schneider, Francis, et al. "Validating requirements for fault tolerant systems using model checking." Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on. IEEE, 1998.

[SMjS03] Ellis Solaiman, Carlos Molina-jimenez, and Santosh Shrivastava. Model checking correctness properties of electronic contracts. In Proceedings of the International conference on Service Oriented Computing (ICSOC03), pages 303-318. Springer-Verlag, 2003.

[vdVHB+08] Saskia van de Ven, Rinke Hoekstra, Joost Breuker, Lars Wortel, and Abdallah El-Ali. Judging amy: Automated legal assessment using owl 2. I Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, OWLED, volume 432 of CEUR Workshop Proceedings. CEUR-WS.org, 2008.

[VFLS08] Johanna Volker, Sergi Fernandez Langa, and York Sure. Supporting the construction of spanish legal ontologies with text2onto. Computable Models of the Law: Languages, Dialogues, Games, Ontologies, pages 105-112, 2008.

[VW63] G.H. Von Wright. Norm and action: a logical enquiry. Routledge & Kegan Paul,1963.

[WBCA08] Adam Z. Wyner, Trevor J. Bench-Capon, and Katie Atkinson. Three senses of “argument”. Computable Models of the Law: Languages, Dialogues, Games, Ontologies, pages 146-161, 2008.