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

Optimizando el rendimiento de la herramienta FormaLex de análisis de documentos normativos

Tesis de Licenciatura en Ciencias de la Computación

Carlos Augusto Faciano

Director: Fernando Schapachnik
Buenos Aires, mayo 2016

Abstract

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. Dado que existen fuertes similitudes entre la especificación de software y la de las normas legales se decidió aprovechar la experiencia en el terreno informático en esta área. Sobre esas bases se desarrollaron FL, un lenguaje basado en la lógica temporal lineal LTL y FormaLex, una herramienta centrada en el análisis de coherencia de documentos deónticos escritos en FL. El objetivo de FormaLex es poder modelar y analizar sistemas legales usando herramientas informáticas.

Hasta el momento el desarrollo de la herramienta se había concentrado en expandir su expresividad, y mostraba problemas de rendimiento en muchas áreas. En particular, no era capaz de analizar en un tiempo razonable varios casos de estudio derivados de la Ley de Defensa del Consumidor de la República Argentina. En esta tesis se implementaron tres optimizaciones, algunas de las cuales reducen fuertemente los tiempos de procesamiento logrando que de 30 casos de prueba que no lograban terminar, 19 pasen a lograrlo. Éstas se basan en cambios en la representación basada en autómatas utilizada, cambios en los operadores utilizados en las fórmulas, y eliminación de componentes inactivos.

Palabras claves: optimización, FL, textos normativos, filtrado, reductor, FormaLex

Agradecimientos

Como no podía ser de otra manera el primer agradecimiento es para mi director de Tesis, Fernando. Aparte de su capacidad técnica, valoro su aspecto humano. Me ha dedicado el tiempo justo y necesario, siempre tratando de responder lo antes posible y sobre todo con mucha paciencia. También me ha buscado y guiado cuando estaba un poco perdido en la tesis. La experiencia de hacer esta tesis no podría haber sido mejor.

También agradecer a la facultad y sobretodo a los docentes que la componen que son el activo más grande que puede tener la facultad.

A Melisa y Celeste, mis predecesoras que me ayudaron sobre todo al comienzo a familiarizarme con Formalex.

Como han sido muchas las personas que me han ayudado durante la carrera quiero agradecer en líneas generales a todos ellos, principalmente por brindarme su apoyo y palabras de aliento.

A mi familia, a mis hermanos Roxana y Leonardo por ser mi principal apoyo en la carrera y en todo lo demás. A mi sobrina Ximena porque es la fuente de amor más grande que pude haber conocido, además aprendo mucho de ella y espero que este título la inspire a seguir sus sueños.

A mis padres Zulema y Carlos que ya no están físicamente pero que siempre insistieron para que estudiara. Gracias a la computadora que compró mi papá cuando estaba en el último año de la secundaria fue lo que me terminó por decidir a estudiar esta carrera. A quién considero mi segunda madre, Juana quién cuando falleció mi mamá nos cobijó y nos dió mucho amor, y sobretodo fue la que más creyó en mí.

A mis tíos María Cristina y Marino, también unos soportes importantes cuando falleció mi madre. Me han dado mucho amor y transmitido valores.

A mis primos Marcelo, Roxana y Mabel, espejos en los que me miré durante mi infancia y adolescencia.
A los hijos de ellos, la segunda generación de primos: Matías, Yamila, Hernán, Pablo y Sol. El presente y el futuro de la familia. Matías ya abogado y los demás están cursando carreras, incluso Pablo que es mi ahijado, que por mi “culpa” está estudiando una carrera de sistemas.

A mi mejor amigo Daniel, lo conozco de los 6 años, se podría decir un hermano más. A pesar de que somos diferentes tenemos historias de vida muy parecidas. Alguien a quien admiro y quiero mucho.

A mis amigos de la facultad: Sergio, Ariel, Diego y Damián. Con todos durante estos años de carrera nos hemos dado ánimo para continuar en momentos difíciles que siempre tiene la carrera, hemos compartido mucho tiempo haciendo trabajos prácticos, preparando parciales y finales, pero por suerte también hemos compartido cosas de nuestras vidas como viajes, alegrías porque algunos ya son padres, muchas charlas, etc. Damián tiene dos particularidades, una que es hincha de Velez como yo y la otra es que va ser el que tome la posta con respecto a los temas de optimización que aún restan resolver. Exitos Damián en esta tarea!

A Victoria y a mi amigo Leonel, que próximamente serán ingenieros en informática. Personas con quienes nos dimos ánimo en los últimos años de carrera.

A Sandra, mi cábala antes de mis últimos finales, una persona que me dio mucho aliento para que no aflojara y que me pronosticaba buenas notas antes de los finales que después se terminaban cumpliendo.

A mis compañeros de trabajo Julio y Lisandro que con sus consejos y aportes han hecho que la tesis quedara mucho mejor. Y a Sergio mi compañero de trabajo más cercano y que en el último tiempo pasó por un momento difícil.

Por último un agradecimiento especial a Graciela, mi psicóloga. Un apoyo importante en las últimas materias de la carrera y en la tesis. Alguien que me ha enseñado a creer en mis posibilidades.

Índice general

1 Introducción

1.1 La herramienta FormaLex

Al desarrollar un sistema de software crítico lo que se busca es que el mismo sea correcto y que se pueda verificar que tiene el comportamiento deseado ante determinadas situaciones. Para ello primero se debe describir el comportamiento esperado para el sistema, usando una especificación que describe lo que el sistema debe hacer y lo que tiene prohibido.

Con los textos normativos se puede hacer una analogía, dado que contienen una serie de reglas y especificaciones que determinan lo que se puede o no se puede hacer y lo que estamos obligados a hacer en determinadas situaciones. Verificar la correctitud de estos textos normativos se puede entender como análogo a verificar formalmente un sistema de software crítico y de hecho presenta las mismas complejidades y desafíos: corroborar que efectivamente, el mismo es coherente, no presenta grises y que contempla todos los casos.

¿Cómo hacemos para asegurar que una norma es coherente? ¿Cómo garantizamos que ciertas reglas no se contradigan con otras que están escritas en la misma norma? A partir de estos interrogantes han surgido un conjunto de herramientas (por ejemplo, [2, 7, 8]) que sirven como asistentes al momento de redactar una norma con el fin de facilitar la verificación. *FormaLex* es una de ellas, la cual es el resultado de un amplio trabajo que viene atacando este problema, recostándose en el marco de las lógicas ya conocidas y en la utilización de herramientas que fueron creadas para dicho propósito.

A continuación una introducción a la herramienta y su lenguaje basada en [6].

1.2 La lógica como base de la verificación de textos normativos

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

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

Este lenguaje fue construido de forma tal de poder generar una traducción eficiente a una de las lógicas temporales más difundidas: Linear Temporal Logic (LTL) [1]. LTL es una lógica modal temporal cuyos modelos son lineales. Es por esta razón que FL cuenta con varias de las características de dicha lógica como ser: temporalidad, noción de localidad, etc. Este lenguaje deóntico, presentado originalmente en [2, 3], se basa en las siguientes premisas:

- Tiene como objetivo principal encontrar problemas de coherencia en documentos normativos. Donde coherencia para la herramienta significa: que no puede haber comportamientos que estén a la vez permitidos y prohibidos para los mismos individuos: “Prohibido matar” y “Permitido matar en defensa propia”, como tampoco estar prohibidos y ser obligatorios. Tampoco puede haber obligaciones lisas y llanas y al mismo tiempo obligaciones con reparaciones[1] (CTDs, del inglés Contrary-To-Duty Obligations), ni puede haber CTDs prohibidas por otras reglas: “Prohibido estacionar en parada de colectivo” y “Si estacionás en una parada, tenés que pagar una multa”, ¿está prohibido o puede hacerlo y pagar?, etc. La lista completa de casos que constituyen un problema de coherencia se encuentra en [2].

[1] Las CTDs u obligaciones con reparaciones son aquellas que se activan cuando se viola una obligación o prohibición. Por ejemplo, "prohibido cruzar el semáforo en rojo" y "si se cruza el semáforo en rojo se deberá pagar una multa".

- Otra de sus premisas de diseño es utilizar como motor de razonamiento a model checkers existentes. Esto se debe principalmente a dos motivos. El primero es que los model checkers son herramientas probadas en el tiempo. Han pasado varias décadas desde que surgieron como prototipos, y al día de hoy son herramientas capaces de empezar a manejar especificaciones de tamaño realista. El segundo motivo es que si se hubiera empezado a construir una herramienta para análisis de normas desde cero, lo más probable es que hubiera pasado por un período similar de tiempo al que pasaron los model checkers para establecerse y obtener resultados.

El lenguaje está compuesto de dos partes:

- un conjunto de reglas, que son fórmulas con operadores deónticos para expresar permisos (P), prohibiciones (F) y obligaciones (O) siguiendo la noción clásica para la lógica deóntica de que $F(\varphi) \equiv O(\neg\varphi)$ y que $P(\varphi) \equiv \neg F(\varphi)$ (ver [2,3] para una definición formal), por ejemplo:
  "Prohibido hablar dentro de la biblioteca"
  "Permitido hablar dentro de la sala parlante"
  "Es obligatorio devolver los libros antes de retirarse"

- y una background theory, o teoría marco, que provee mecanismos sencillos para describir la clase de modelos sobre los que predican las reglas. Por ejemplo si se está modelando el reglamento de una facultad, algunos conceptos que se deberían incluir en la teoría marco son: estudiante, docente y las acciones que pueden realizar estos en el ámbito de la facultad. Además en la teoría marco se pueden expresar cuestiones como precedencia de eventos (e.g., el día ocurre antes de la noche), unicidad (e.g., las personas nacen una sola vez), etc. mediante construcciones para declarar acciones, roles (que luego dan origen a agentes que ejecutan las acciones), intervalos de tiempo, entre otras, que serán explicadas en detalle en la sección 1.3.1.

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

Cada camino 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. Cada una de estas trazas describe un posible comportamiento legalmente válido de los agentes involucrados, es decir, los comportamientos que no cumplen con las reglas son descartados. Si se desea ver el tema con mayor profundidad remitirse a [13]

Este lenguaje, más la utilización de model checkers como motores de inferencia para analizar y encontrar automáticamente problemas de coherencia, dan como resultado *FormaLex*.

1.3 Describiendo FL

Como mencionamos anteriormente, lo que se desea hacer es tomar un documento normativo y analizarlo para encontrar incoherencias de forma automática. Dicho documento debe ser traducido al lenguaje FL y es el input de todo el proceso. Es importante aclarar que no existe hasta el momento ninguna herramienta de automatización o pseudo automatización que realice esta tarea, sino que se requiere de una o varias personas con cierto conocimiento de lenguajes formales que tomen el texto que se quiere tratar, lo interpreten y codifiquen en FL.

A continuación se describen los componentes que ofrece FL para escribir estas especificaciones.

1.3.1 Componentes de la teoría marco

En esta sección se presentan los componentes de la teoría marco, la cual contiene las especificaciones de las características que dan origen a la clase de modelos que utilizaremos para verificar las reglas, es decir, el conjunto de trazas posibles.

1.3.1.1 Acciones

Un modelo lineal, o traza, consiste en una serie de estados relacionados secuencialmente mediante transiciones. En cada uno de estos estados pueden estar ocurriendo o no un conjunto de acciones. Ésta es una de las razones por las que las acciones son el componente principal de la teoría marco.

Las acciones pueden ser realizadas de forma impersonal (no es necesario que alguien específico esté realizando dicha acción, por ejemplo: llover). Este tipo de acciones son útiles para representar eventos.

```
impersonal action llover
```

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

```
action vender
```

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

Además, a las acciones es posible especificar la cantidad de veces que se puede ejecutar y también quienes tienen la posibilidad de hacerlo. Por ejemplo, la acción `extraerDinero` sólo puede ser realizada por el `titularCuenta`, y solamente se pueden realizar 3 extracciones.

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

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

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

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

Por último, existen casos en que 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
```

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.

Nótese que todas estas opciones mencionadas arriba se van potenciando al combinarse entre sí, así como se combinan con los elementos que se desarrollan a continuación, con el fin de poder modelar fielmente las situaciones más frecuentes que se pueden presentar en el ámbito legal.

1.3.1.2 Roles

Como se mencionó brevemente en la sección anterior, en FL el rol nos da la posibilidad de agrupar a agentes con ciertas características en común, lo que nos permite indicar cuáles de ellos pueden ejecutar ciertas acciones.

```
roles hombre, mujer
```

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

```
roles hombre, mujer disjoint
```

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

Por último, a veces se necesita que un conjunto de roles cubran todas las posibilidades. Si nos referimos a la población de un país podemos dividir a la población en dos, por un lado los menores de edad y por otra lado los mayores de edad, de manera que con los roles `menorDeEdad` y `mayorDeEdad` ya estamos cubriendo todas las posibilidades. En FL lo escribimos como:

```
roles menorDeEdad, mayorDeEdad cover
```

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

En el trabajo realizado en [5] se incorporó esta noción de rol a FL y en [6] se profundizó aún más en este tema, incorporando la posibilidad de especializar los roles en subroles (sin límites de niveles de profundidad).

1.3.1.3 Agentes

Los agentes no cuentan con una declaración explícita en FL[2] sino que son generados a partir de la información que le damos al sistema sobre los roles. Su finalidad es ser las entidades del modelo que realizan las acciones. Cada uno de ellos representa en general a una persona física, jurídica o incluso a un conjunto de estas (por ejemplo, un agente podría ser “la sociedad”).

Tal como lo indicamos en la sección anterior, se crearán tantos agentes como combinaciones posibles de roles se hayan definido. Para que quede plasmada la idea general de cómo funciona esto lo haremos con dos ejemplos sencillos (si se desea ver el tema con más detalle remitirse a [5]).

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

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

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

Aquí se generan 4 agentes ya que podemos combinar los dos roles porque no son disjuntos:

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

Por último, la condición de cover en un conjunto de roles implica que alguno de esos roles debe estar en todos los agentes creados ya que como cubren todos los posibles valores, el agente debe ser de alguno de ellos. En ese caso no se crea el agente `agent_without_role`.

1.3.1.4 Contadores

Los contadores, como su nombre lo indica, son variables numéricas que se utilizan para contabilizar. El valor de cada contador únicamente puede ser modificado por las acciones especificadas al momento de definirlo.

Cuando definimos un contador podemos especificarle:

[2] Si bien no se pueden declarar explícitamente, pueden referenciarse de manera indirecta mediante EXIST y FORALL (ver sección “1.3.2 Lenguaje de fórmulas”).

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

Veamos con un ejemplo las opciones mencionadas:

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

En [6] se agregó la posibilidad de definir valores máximos y mínimos para cada contador y especificar un comportamiento determinado cuándo se alcanzan esas cotas máximas y/o mínimas. Un ejemplo de comportamiento es si se quiere impedir/permitir la ocurrencia de las acciones que modifican el valor del contador. Por ejemplo, al contador `librosPrestados` se le puede poner un límite de 5 e indicar si al llegar a ese valor no se permiten nuevas ejecuciones de la acción `retirarLibro` o se permiten pero sin alterar el valor del contador.

1.3.1.5 Intervalos

Los intervalos sirven para incorporar la noción de un periodo de tiempo, lo cual es de suma utilidad por ejemplo para expresar que una acción debe realizarse en un plazo de tiempo determinado. Los intervalos se inician con una acción, y a partir de ahí el intervalo está activo, y finalizan con otra acción: desde ese momento el intervalo permanece inactivo. A un intervalo se le pueden especificar varias acciones de inicio y varias de fin, pero se debe cumplir que las acciones que dan inicio y fin al intervalo no pueden ser ejecutadas en forma simultánea y además: si se ejecuta una acción de inicio del intervalo, no se puede ejecutar cualquier otra de inicio (incluso ella misma) hasta que no se ejecuta una acción que da fin al intervalo y lo mismo sucede con las acciones de fin.

A un intervalo se le pueden especificar los siguientes valores:
- Alcance: local o global. Con el mismo sentido que en los contadores
- Conjunto de acciones que lo pueden iniciar
- Conjunto de acciones que lo pueden finalizar
- Cantidad máxima de veces que puede ocurrir
- Ocurrencia dentro de otro intervalo

Adicionalmente a un intervalo se le puede indicar que su estado inicial sea activo, es decir que no necesite de una acción para iniciar. También se puede indicar que cuando un intervalo pase a estar activo, conserve de ahí en más ese estado. Estos tipos de intervalos los denominamos infinitos.

Veamos algunos ejemplos de intervalos especificados en FL:

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

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

- intervalo dentro de otro intervalo

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

- intervalo infinito y con cantidad máxima de ocurrencias

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

Por último vamos a dar un ejemplo más de la utilidad que se le puede dar a los intervalos, en este caso combinado con acciones. El uso que se le puede dar es que a una acción se le puede definir que ocurra únicamente dentro de un intervalo, por ejemplo, la acción de realizarse la revisación médica para ser admitido en la pileta solo se podrá realizar mientras la temporada de pileta esté activa:

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

1.3.1.6 Timers

Tal como se explica en [5], para modelar el paso del tiempo de forma discreta se utilizan timers. El timer está compuesto por una serie de estados que se suceden en un orden predeterminado, y que son los que se considera relevante representar. Por ejemplo, un timer puede ser: Primavera, Verano, Otoño, Invierno.

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

Su sintaxis es la siguiente:

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

1.3.2 Lenguaje de fórmulas

La otra parte que compone el lenguaje FL es la que se utiliza para escribir las fórmulas, que como ya mencionamos es un lenguaje deóntico, por lo que precisamos expresar los operadores propios de este tipo de lenguajes. Comencemos por el operador de obligación (en inglés Obligation) O(φ) significa que la fórmula φ se cumple siempre, es decir, en todos los estados de todos los modelos. Si queremos expresar que es obligatorio ser amable con los demás, usamos este operador

```
O(ser_amable)
```

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

```
F(fijar_carteles)
```

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

```
P(hablar)
```

Se puede considerar a la prohibición como la obligación del contrario, por lo cual podemos decir que F(φ) es equivalente a O(¬φ). De la misma forma decimos que algo está permitido si no está prohibido, aunque los operadores P en lugar de modificar el modelo se transforman en chequeos que se realizan luego (ver [3]).

Las acciones pueden asociarse a roles en su definición (ver sección “2.1.2 Expansión de componentes”), lo que luego permite el uso de los cuantificadores universal y existencial, cuya forma de uso es la siguiente:

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

Para un mayor detalle sobre las características del lenguaje, se puede consultar [2].

1.4 La herramienta en ejecución

En las secciones anteriores describimos el marco teórico de la herramienta pero no detallamos nada con respecto a los aspectos de implementación de FormaLex.

1.4.1 Tecnologías utilizadas

FormaLex está escrito en el lenguaje de programación Java y contamos con dos herramientas más de soporte:

- *JavaCC*[3], que es un generador de parsers que utilizamos para generar código Java a partir de la gramática definida para el lenguaje FL. Ese código generado se encarga de validar el archivo de input y poblar las entidades a partir de éste.

- *Velocity*[4]*,* lo utilizamos para escribir templates especıficos de input para cada model checker que querramos utilizar. Para este trabajo se usó *NuSMV*[5]*,* pero si se quisiera usar otro model checker lo que se debería hacer es escribir un nuevo template con Velocity para el nuevo model checker.

1.4.2 El model checker

El model checker NuSMV, en lugar de tomar autómatas definidos como tales, utiliza variables y un lenguaje que permite operar sobre ellas, de manera tal que el autómata se construye implícitamente a partir de un archivo de configuración. La entrada es un conjunto de reglas ecuacionales que describen, incluso de manera no determinística, los posibles valores que puede tomar cada variable en el siguiente estado en base a los valores actuales de todas las variables. De esta forma, el model checker elige para cada traza los valores en el siguiente estado como alguna de las combinaciones que son válidas de acuerdo a todas las ecuaciones provistas. Eso permite en algunos casos agregar ecuaciones para brindar o restringir comportamiento, sin necesidad de modificar otras.

Como parte de este trabajo consiste en modificar el comportamiento de algunas de estas variables (principalmente las que representan a las acciones), precisamos conocerlas un poco más en detalle. Por lo dicho anteriormente vamos a dar una introducción de cómo se representan las acciones en el model checker. Si se quiere conocer más en detalle cómo se representan las otras entidades de la teoría marco, éstas se encuentran descriptas en [5].

1.4.2.1 Acciones

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

1. Puede tener sólo alguno de los siguientes valores: NOT HAPPENING, HAPPENING o JUST HAPPENED.
2. Su valor inicial es NOT HAPPENING.
3. Puede cambiar de valor respetando el siguiente orden: NOT HAPPENING → HAPPENING → JUST HAPPENED → NOT HAPPENING → . . . y repite el ciclo.
4. Si su valor es JUST HAPPENED en el siguiente estado tiene que cambiar de valor, mientras que en los demás casos puede conservar su valor indefinidamente.
5. Para poder pasar de un estado a otro utiliza el operador NEXT().

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

1.4.3 El proceso de verificación por dentro

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

Su input consiste en un autómata A y una fórmula φ, y su misión consiste en verificar si existe algún modelo τ dentro del lenguaje generado por A, tal que τ |= ¬φ, es decir, si hay alguna corrida de A que viole φ. Si existe, el model checker la devolverá como un contraejemplo a la validez de φ.

En nuestro caso el autómata A se genera traduciendo la teoría marco. Las reglas deónticas se traducen a LTL y se unen mediante conjunciones en una fórmula que llamaremos φ . El objetivo es ver si existe al menos una corrida de A que satisfaga φ. Por lo tanto, le proporcionamos al model checker la fórmula ¬φ . Si encuentra un contraejemplo, ese contraejemplo es una violación de ¬φ y por ende satisface φ . Es decir, se trata de un “testigo” de que existe al menos una forma de cumplir todas las reglas bajo la teoría marco y por ende de que no hay contradicciones.

A continuación se describen cada uno de los pasos y los elementos necesarios para que esto funcione:

a) Archivo de configuración. Actualmente se llama config.ini y contiene:
- Ruta del model checker
- Ruta donde la herramienta deja los archivos de salida.
- Ruta del Template de *Velocity*.
- Ruta del archivo de entrada escrito en FL.

b) Punto de entrada del proceso. Es mediante el archivo Main.java que básicamente se encarga de ejecutar los siguientes pasos:

1. Parsea el archivo de entrada.
2. Valida su sintaxis.
3. De ser correcto, puebla todas las entidades de la teoría marco con los datos contenidos en dicho archivo. Caso contrario detiene la ejecución y devuelve los errores encontrados.
4. Se generan los agentes a partir de los roles definidos.
5. Se invoca a *Velocity* para que genere el autómata para el model checker correspondiente. (El autómata se genera en un archivo plano y se guarda en el directorio indicado como directorio de salida en el punto a).
6. Las fórmulas correspondientes a las normas escritas en FL se expanden y se instancian con los agentes generados en el paso 4 según corresponda.
7. Las fórmulas se traducen de FL a LTL.
8. Se genera una nueva fórmula que es la conjunción de todas las reglas menos los permisos (para entender por qué se dejan de lado remitimos al lector a [5]).
9. Se crea otra fórmula que es la negación de la fórmula del paso anterior (esto es importante: el model checker recibe la fórmula negada).
10. En este punto ya tenemos el autómata por un lado y la conjunción negada de todas las fórmulas por el otro. Ambos archivos se le envían al model checker.
11. El model checker devuelve el resultado de la validación, el cual queda también guardado en el directorio de salida.
12. Nuestra herramienta analiza el resultado arrojado por el model checker que puede ser que se encontró un comportamiento válido, es decir, no hay incoherencias en el archivo de entrada, o bien que no hay un comportamiento válido.
13. Sólo en caso de que la herramienta encuentre un comportamiento válido se procede a validar cada uno de los permisos (en caso de que los hubiere). Para ello se hacen tantas validaciones como permisos hubiere, agregando en cada validación un permiso a la fórmula generada en el paso 8 y negando esa fórmula. Luego, por cada permiso, se repiten los pasos 10, 11 y 12.

2 Incorporaciones a la nueva versión

2.1 Filtrado de la teoría marco

2.1.1 Motivación

Hasta el momento en la herramienta no había ningún proceso de optimización, ya que surgió como un prototipo académico y sólo se habían hecho unas pruebas de concepto bastante pequeñas en el tamaño de la teoría marco. Debido a que posteriormente se comenzaron a hacer pruebas con un caso real -en nuestro caso la Ley de Defensa del Consumidor de la República Argentina[6]- se comenzaron a detectar características a mejorar. Por ejemplo, el tamaño del autómata que sirve de entrada para el model checker NuSMV era grandísimo. Para darnos una idea del problema, si la codificación de la ley eran cientos de líneas de código, el autómata generado tenía millones de líneas de código. Entonces, por el problema expuesto, se decidió buscar la forma de reducir el tamaño del autómata.

2.1.2 Expansión de componentes

Dado que a lo largo de la descripción de la solución se van a utilizar de manera intercambiable las expresiones *"componentes expandidas"* o *"expansión de componentes"*, es necesario dar una explicación de dichas expresiones. La expansión de componentes, empieza a ser parte de la herramienta cuando se decide incorporar al lenguaje FL los conceptos de roles y agentes para brindarle más expresividad al lenguaje. Los roles tienen como objetivo poder determinar qué agentes pueden realizar determinadas acciones. Para ello es necesario definir roles y asignarlos a las acciones.

Por otra parte, los agentes son entidades que cumplen roles y ejecutan acciones. Son componentes usados internamente por nuestra herramienta para la generación del autómata. La herramienta crea un agente por cada combinación válida de roles. Nuestra herramienta brinda la posibilidad de que una acción se defina sin un rol, en ese caso dicha acción puede ejecutarse por cualquier agente, independientemente de los roles que tenga asignado. A su vez se puede especificar que una acción es impersonal, esto significa que no es necesario que un agente la ejecute. Dicha construcción del lenguaje es útil para representar eventos, por ejemplo `inicioDeNuevoDia`.

A grandes rasgos la expansión de componentes de la teoría marco, consta de los siguientes pasos:

1. Generación de los agentes a partir de los roles definidos
2. Expansión de las acciones
3. Expansión de los intervalos
4. Expansión de los contadores

Dado que los pasos 1 y 2 están muy relacionados, se va a utilizar el mismo ejemplo para poder comprenderlos mejor.

```
roles proveedor, consumidor cover
action Consumir only performable by Consumidor
action SolicitarInformacion only performable by Consumidor
action ComienzaOferta only performable by Proveedor
action TerminaOferta only performable by Proveedor
action PagarImpuesto
impersonal action FuerzaMayor
```

El paso 1 consiste en generar los agentes, a partir de los roles definidos. En el ejemplo se definieron dos roles. Haciendo todas las combinaciones posibles de roles, se generan los siguientes agentes con los roles asignados:

```
agent_1: Consumidor
agent_2: Proveedor
agent_3: Consumidor, Proveedor
```

[6] Se trata de la Ley 24240 y sus modificatorias. La codificación original fue realizada por la Universidad FASTA de Mar del Plata (ver [9]).

Como se puede observar, debido a que está el modificador `cover` (ver sección "1.3.1.3 Agentes") no se genera el agente sin roles.

El paso 2, consiste en expandir las acciones, de acuerdo a los roles asignados a cada agente y a cada acción.
Por ejemplo, el agente `agent_1`, que es `Consumidor`, se combina con las acciones que puede ejecutar ese rol, en este caso `Consumir` y `SolicitarInformacion`. Por lo tanto se generan las siguientes acciones:

```
agent_1.Consumir
agent_1.SolicitarInformacion
```

El agente `agent_2`, que es `Proveedor`, se combina con las acciones que puede ejecutar ese rol, en este caso `ComienzaOferta` y `TerminaOferta`. Se generan las acciones:

```
agent_2.ComienzaOferta
agent_2.TerminaOferta
```

El `agent_3`, que es `Proveedor` y `Consumidor` a la vez, se combina con las acciones que pueden ejecutar ambos roles. Las acciones generadas son:

```
agent_3.Consumir
agent_3.SolicitarInformacion
agent_3.ComienzaOferta
agent_3.TerminaOferta
```

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

```
agent_1.PagarImpuesto
agent_2.PagarImpuesto
agent_3.PagarImpuesto
```

Por último, como la acción `FuerzaMayor` es impersonal, se genera tal cual está especificada, es decir sin ningún agente:

```
FuerzaMayor
```

Los pasos 3 y 4 consisten en expandir los intervalos y contadores respectivamente, a partir de los agentes y las acciones con agentes generados en los dos pasos previos.

Los intervalos y los contadores pueden ser de dos tipos: local y global. En el caso de ser local sólo afecta a un agente, en cambio, uno global es compartido por todos los agentes.

Si un intervalo se define como local, la herramienta genera un intervalo por cada agente que realice al menos una de las acciones iniciales y otra de las acciones finales del intervalo.
Por ejemplo, a partir del siguiente intervalo local :

```
local interval enContratoVigente defined by actions firmarContrato - rescindirContrato,
finDelContrato
```

se generan intervalos de la forma:

```
interval agent_X.enContratoVigente defined by actions
agent_X.firmarContrato - agent_X.rescindirContrato, agent_X.finDelContrato
```

siendo X el número de agente. Si por la asignación de roles, el agente para el cual se define el intervalo no puede realizar alguna de las acciones, entonces estas no formarán parte el nuevo intervalo. Supongamos que en el ejemplo anterior, el `agent_X` no puede realizar la acción `rescindirContrato` pero sí las demás, entonces el nuevo intervalo quedaría así:

```
interval agent_X.enContratoVigente defined by actions
agent_X.firmarContrato - agent_X.finDelContrato
```

En el caso de que el intervalo sea global, se genera un único intervalo para todos los agentes, y las acciones que lo delimitan se reemplazan por las nuevas con los agentes(las generadas en el paso 2). Por ejemplo si un intervalo se define así:

```
global interval abierto defined by actions abrir - cerrar
```

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

```
interval abierto defined by actions agent_1.abrir,
agent_2.abrir, ... - agent_1.cerrar, agent_2.cerrar, ...
```

De la misma forma que ocurre con los intervalos, si se especifica un contador local sólo afecta a un agente, en cambio, uno global es compartido por todos los agentes.

Por ejemplo, dada la siguiente especificación en FL:

```
roles titularCuenta, noTitularCuenta cover disjoint
actions depositarEnCuenta100, depositarEnCuenta500
actions retirarDeCuenta100, retirarDeCuenta500 only performable by titularCuenta

local counter saldoEnCuenta init value 0
increases with action depositarEnCuenta100 by 100,
increases with action depositarEnCuenta500 by 500,
decreases with action retirarDeCuenta100 by 100,
decreases with action retirarDeCuenta500 by 500.
```

a partir de la definición de un contador, se generan varios contadores, uno por cada agente que realice al menos una de las acciones que modifican al contador:

```
local counter agent_X.saldoEnCuenta init value 0
increases with action agent_X.depositarEnCuenta100 by 100,
increases with action agent_X.depositarEnCuenta500 by 500,
decreases with action agent_X.retirarDeCuenta100 by 100,
decreases with action agent_X.retirarDeCuenta500 by 500.
```

siendo X el número de agente. Si por la asignación de roles, el agente para el cual se define el contador no realiza alguna de las acciones, entonces estas no aparecerán como modificadoras del nuevo contador, es decir únicamente aparecerán las acciones que el agente puede realizar. Supongamos que en el ejemplo anterior, el `agent_X` sólo tiene asignado el rol `noTitularCuenta`, por lo tanto no puede realizar las acciones `retirarDeCuenta100` ni `retirarDeCuenta500` , pero sí las demás, entonces el nuevo contador quedaría así:

```
local counter agent_X.saldoEnCuenta init value 0
increases with action agent_X.depositarEnCuenta100 by 100,
increases with action agent_X.depositarEnCuenta500 by 500.
```

En el caso que se especifique un contador global, se crea un solo contador para todos los agentes, y las acciones que lo modifican se reemplazan por las nuevas con los agentes (las generadas en el paso 2). Por ejemplo, dada la siguiente situación:

Supongamos que se quiere calcular cuál es la ganancia de una empresa de celulares, por subconsumo, es decir cada usuario gasta menos de lo que paga. Para simplificar el ejemplo, vamos a asumir que los usuarios únicamente pueden cargar saldo mediante el pago de un abono mensual fijo o mediantes tarjetas prepagas.

```
roles cliente {clientePagaAbonoMensual, clienteConTarjetaPrepaga cover disjoint}
actions cargarSaldoConTarjeta30, cargarSaldoConTarjeta50, cargarSaldoConTarjeta100,
relizarLLamadaPor5, relizarLLamadaPor10, mandarSms
actions cargarAbonoMensua150, cargarAbonoMensua250 only performable by clientePagaAbono

global counter gananciaPorSubconsumo init value 0
increases with action cargarAbonoMensua150 by 150,
increases with action cargarAbonoMensua250 by 250,
increases with action cargarSaldoConTarjeta30 by 30,
increases with action cargarSaldoConTarjeta50 by 50,
increases with action cargarSaldoConTarjeta100 by 100,
decreases with action relizarLLamadaPor5 by 5,
decreases with action relizarLLamadaPor10 by 10,
decreases with action mandarSms by 1
```

Se genera un único contador “expandido” reemplazando cada una de las acciones involucradas por las nuevas creadas con los agentes que realicen esas acciones:

```
counter gananciaPorSubconsumo init value 0
increases with action agent_1.cargarAbonoMensua150 by 150,
increases with action agent_1cargarAbonoMensua250 by 250,
increases with action agent_1.cargarSaldoConTarjeta30 by 30,
increases with action agent_1.cargarSaldoConTarjeta50 by 50,
increases with action agent_1.cargarSaldoConTarjeta100 by 100,
increases with action agent_2.cargarSaldoConTarjeta30 by 30,
increases with action agent_2.cargarSaldoConTarjeta50 by 50,
increases with action agent_2.cargarSaldoConTarjeta100 by 100,
decreases with action agent_1.relizarLLamadaPor5 by 5,
decreases with action agent_1.relizarLLamadaPor10 by 10,
decreases with action agent_1.mandarSms by 1,
decreases with action agent_2.relizarLLamadaPor5 by 5,
decreases with action agent_2.relizarLLamadaPor10 by 10,
decreases with action agent_2.mandarSms by 1
```

2.1.3 Solución - Marco teórico

La solución consiste en eliminar de la teoría marco todo lo que no se usa, con el objetivo de que el autómata generado sea lo más chico posible.

¿Qué quiere decir que no se “usa”?
Son todas las componentes (agentes, acciones, intervalos y contadores) de la teoría marco a las cuales no se hace referencia (directa o indirectamente) en las fórmulas (reglas y permisos) a validar. Además en las acciones que se definan con output values, se van a eliminar los valores de salida que **no** son referenciados en las fórmulas.

La idea a grandes rasgos es la siguiente:

Paso 1: Armar un grafo dirigido de dependencias de la teoría marco. El grafo se construye por única vez, al finalizar la expansión de los componentes de la teoría marco.

Los siguientes pasos se realizan previamente a cada llamada al model checker.
Paso 2: Recorrer las fórmulas e ir marcando en el grafo de dependencia los componentes que se usan en cada una de las fórmulas.
Paso 3: Eliminar de la teoría marco todas las componentes que no están marcadas en el grafo.
Paso 4: Recorrer cada una de las acciones de la teoría marco y en caso de que la acción tenga definidos valores de salida, eliminar todos los valores de salida que no se referencian en las fórmulas

A continuación se incluye una descripción más detallada de los pasos.

2.1.3.1 Paso 1 - Construcción del grafo

Lo primero que uno podría preguntarse es por qué se necesita construir un grafo de dependencia y, posteriormente, cómo se establecen dichas dependencias. La razón por la cual es necesario el grafo de dependencia es porque en una fórmula puede haber una o varias referencias “indirectas” a un componente de la teoría marco, entonces el grafo sirve para registrar esas dependencias indirectas. Veámoslo mejor con un ejemplo

Dada la siguiente codificación en FL:

```
# Background

roles estudiante cover
action censarse only performable by estudiante
impersonal action empieza_censo
impersonal action termina_censo
impersonal action inicio_ciclo_lectivo
impersonal action fin_ciclo_lectivo
global interval ciclo_lectivo defined by actions inicio_ciclo_lectivo - fin_ciclo_lectivo
global interval en_censo defined by actions empieza_censo - termina_censo

# Clauses

FORALL(i:estudiante; O(<>_{en_censo} i.censarse))
FORALL(i:estudiante; F(i.censarse U inicio_ciclo_lectivo))
```

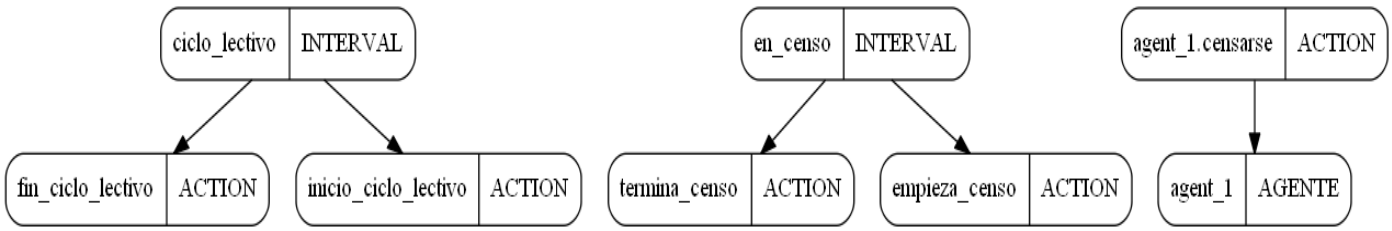
Al expandir la teoría marco queda lo siguiente:

```
Agentes creados:
agent_1: estudiante

Acciones con agentes:
action agent_1.censarse performable by estudiante

Acciones impersonales:
impersonal action empieza_censo
impersonal action termina_censo
impersonal action inicio_ciclo_lectivo
impersonal action fin_ciclo_lectivo

Intervalos:
global interval en_censo defined by actions empieza_censo - termina_censo
global interval ciclo_lectivo defined by actions inicio_ciclo_lectivo - fin_ciclo_lectivo
```

Las fórmulas quedan así:

```
G ( en_censo = ACTIVE  -> (en_censo = ACTIVE  U agent_1.censarse = JUST_HAPPENED) )
G ( !( agent_1.censarse = JUST_HAPPENED U inicio_ciclo_lectivo = JUST_HAPPENED ) )
```

El grafo resultante es el siguiente:



En las fórmulas las componentes referenciadas directamente son: `[inicio_ciclo_lectivo, en_censo, agent_1.censarse]`. Pero debido a que el intervalo **`en_censo`** tiene dependencias con las acciones **`empieza_censo y termina_censo`** y la acción **`agent_1.censarse`** con el agente **`agent_1`** podemos detectar mediante el grafo estas referencias indirectas. Otra referencia indirecta es la que existe entre el intervalo `ciclo_lectivo` con las acciones `inicio_ciclo_lectivo` y `fin_ciclo_lectivo`, pero en este último caso a diferencia de los anteriores el intervalo `ciclo_lectivo` no es referenciado directamente en las fórmulas.

La construcción del grafo de dependencias permitirá detectar referencias indirectas para luego calcular qué elementos no son utilizados incluso de manera transitiva (elementos que parecen ser usados pero lo son por otros que a su vez no están siendo referenciados). Veamos cómo se construye este grafo de dependencias.

Los nodos del grafo representan a las componentes expandidas de la teoría marco. Las aristas del grafo van a modelar las dependencias entre las componentes.

- Si una **acción a** ocurre en un **intervalo i,** entonces se agrega una arista (a,i). Un ejemplo en la codificación de la teoría marco podría ser este:
  `action` **`rendir_examen`** `only occurs in scope` **`periodo_escolar.`** En este ejemplo se agregaría la arista (**`rendir_examen, periodo_escolar`**)

- Si una **acción a** tiene una **acción sincronizada b,** entonces se agrega una arista (a,b) y otra arista (b, a). Un ejemplo en la codificación de la teoría marco podría ser este:
  `action` **`vender`** `synchronizes with` **`comprar.`** En este ejemplo se agregarían dos aristas: (**`vender, comprar`**) y (**`comprar`** , **`vender`**)

- Si una **acción a** es ejecutada por un **agente g,** entonces se agrega una arista (a,g). Veámoslo con un ejemplo:

  ```
  roles estudiante, docente cover
  action censar only performable by estudiante
  actions empieza_censo, termina_censo
  ```

  A partir de los roles definidos se generan los siguientes agentes:

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

  Una vez generado los agentes se expanden las acciones:

  ```
  action agent_3.termina_censo
  action agent_1.empieza_censo
  action agent_3.censar performable by estudiante
  action agent_2.empieza_censo
  action agent_2.termina_censo
  action agent_1.censar performable by estudiante
  action agent_3.empieza_censo
  action agent_1.termina_censo
  ```

  Como se puede ver arriba, las acciones expandidas tienen al agente que los ejecuta, por lo tanto en este ejemplo se estarían agregando 8 aristas:

```
(agent_3.termina_censo , agent_3)
(agent_1.empieza_censo , agent_1)
(agent_3.censar     , agent_3)
(agent_2.empieza_censo , agent_2)
(agent_2.termina_censo , agent_2)
(agent_1.censar     , agent_1)
(agent_3.empieza_censo , agent_3)
(agent_1.termina_censo , agent_1)
```

- Si un **intervalo i** tiene **acciones $a_1..a_n$** que lo inician entonces se agrega una arista (i,a_i) por cada acción de inicio. Un ejemplo en la codificación de la teoría marco podría ser este:
  `local interval estudiante defined by actions inscripcion_online,`
  `inscripcion_ventanilla` `- graduarse, abandonar_carrera .` En este ejemplo se agregarían dos aristas: `(estudiante, inscripcion_online)` y `(estudiante, inscripcion_ventanilla)`

- Si un **intervalo i** tiene **acciones $a_1..a_n$** que lo finalizan entonces se agrega una arista (i,a_i) por cada acción de fin. Usando el mismo ejemplo de definición de intervalo de arriba, también se agregarían dos aristas:
  `(estudiante, graduarse)` y `(estudiante, abandonar_carrera)`

- Si un **intervalo i** ocurre dentro de un **intervalo j,** entonces se agrega una arista (i,j). Un ejemplo en la codificación de la teoría marco podría ser este:
  `interval` **`periodo_exámenes`** `defined by actions comienzo_exámenes`
  `fin_exámenes only occurs in scope` **`periodo_escolar`**

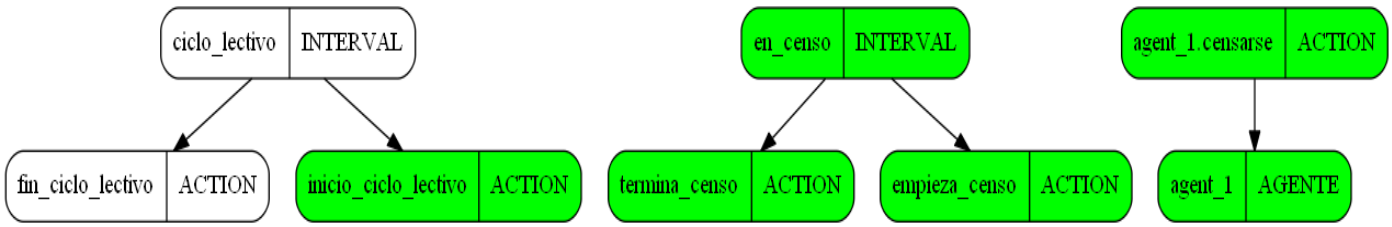
- Si un **counter c** tiene **acciones $a_1..a_n$** que lo incrementan/decrementan entonces se agrega una arista (c,a_i) por cada acción de incremento/decremento. Un ejemplo en la codificación de la teoría marco podría ser este:
  `local counter` **`libros_retirados`** `init value 0, increases with`
  `action` **`retirar_libro`**`, decreases with action` **`devolver_libro.`** En este ejemplo se agregarían dos aristas: `(libros_retirados, retirar_libro)` y `(libros_retirados, devolver_libro)`

- Si un **counter c** tiene **acciones $a_1..a_n$** que le establecen valores entonces se agrega una arista (c,a_i) por cada acción que le setea el valor.
  ```
  local counter puntaje init value 20
  decreases with action conducir_sin_cinturón by 2,
  decreases with action conducir_sin_licencia by 4,
  resets with action cumple_inhabilitación,
  sets with action pasan_2_años_sin_infracción to value 20.
  ```
  En este ejemplo se agregarían dos aristas: `(puntaje, cumple_inhabilitación)` y `(puntaje, pasan_2_años_sin_infracción)`

2.1.3.2 Paso 2 - Marcado del grafo

En este paso se recorren las fórmulas, para obtener las componentes de la teoría marco referenciadas directamente y por cada una de las componentes se va marcando en cascada en el grafo. Por lo tanto al finalizar este proceso tenemos marcadas en el grafo, todas las componentes referenciadas directa o indirectamente en las fórmulas que se le van a pasar al model checker, es decir en el grafo quedan marcadas todas las componentes utilizadas en las fórmulas, por lo tanto NO se pueden eliminar de la teoría marco.

Basado en el ejemplo del paso anterior, el grafo resultante en este paso, es el siguiente (en fondo verde aparecen las componentes marcadas en el grafo)



En el ejemplo se puede ver gráficamente cómo es el marcado del grafo. Las componentes `en_censo` y `agent_1.censarse` referenciadas directamente hacen que se marque en cascada a las componentes `termina_censo, empieza_censo y agent_1.` En cambio la acción `inicio_ciclo_lectivo` que se referencia directamente en la segunda fórmula es un ejemplo de componente sin dependencias.

2.1.3.3 Paso 3 - Filtrado de la teoría marco

Con el grafo de dependencias armado y marcado según las componentes que se utilizan en las fórmulas, lo que resta es eliminar o filtrar de la teoría marco todo lo que no se necesita, es decir todas las componentes que NO están marcadas en el grafo. Para ello se recorren todos los nodos del grafo, y por cada nodo NO marcado se lo elimina de la teoría marco

2.1.3.4 Paso 4 - Filtrado de output values

Mientras se trabajaba en la solución del filtrado de la teoría marco y a medida que se iban generando los casos de estudios, se detectaron dos tipos de situaciones referentes al uso de las acciones con output values, en los cuales se podía aplicar una optimización. Las dos situaciones son las siguientes:

1- Acciones que se definen con valores de salida, pero en las fórmulas **no se hace referencia a ninguno de ellos**. Por ejemplo, dada la siguiente acción con valores de salida:

```
action darServicioTecnico output values {adecuado, inadecuado} only performable by Proveedor
```

y supongamos que sólo queremos expresar que todo proveedor tiene la obligación de dar un servicio técnico:

```
FORALL(p:Proveedor; O(p.darServicioTecnico))
```

Claramente éste es un caso donde hay una acción que se define con valores de salida, pero no se referencia ninguno. Por lo tanto esto genera un desperdicio de espacio de estados en el autómata, ya que para representar una acción con valores de salida, se utiliza una variable extra que mantiene los valores de salida.

La acción después del filtrado debería quedar así:

```
action darServicioTecnico
```

es decir, sin ningún valor de salida.

2- Acciones que se definen con valores de salida, pero en las fórmulas **se hace referencia a alguno de ellos, pero no a todos**. Es similar al caso anterior pero menos extremo. Por ejemplo una acción con valores de salida se la define así:

```
action elegir_ganadores output values {valor1, valor2, valor3, valor4}
```

y supongamos que en las fórmulas sólo se hace referencia a valor1. Entonces lo correcto sería que en el filtrado la acción quede así:

```
action elegir_ganadores output values {valor1, ningunoDeLosOtrosValores}
```

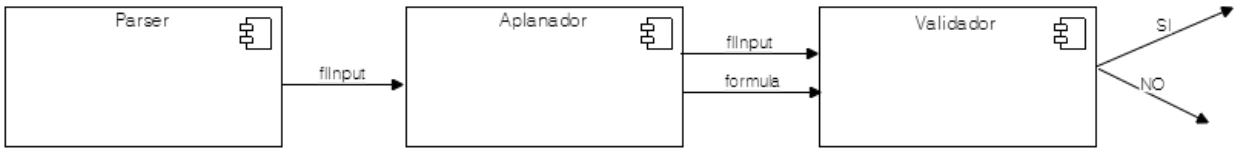
es decir, los valores de salida que no se usan (`valor2, valor3 y valor4 en este caso)` se unifican en un solo valor de salida por defecto al que llamamos `ningunoDeLosOtrosValores.`

Es importante aclarar que queda pendiente hacer una demostración rigurosa de que la optimización no modifica la semántica de la herramienta. A grandes rasgos la demostración consistiría en mostrar que una traza que satisface φ (nuestra fórmula) existe en el modelo filtrado M' sii existe en el modelo original M. Eso se probaría por inducción en la construcción de la fórmula y viendo que el filtrado no cambia la validez por la manera en la que está construido.

2.1.4 Solución - Marco práctico

2.1.4.1 Arquitectura

La arquitectura de la herramienta es la siguiente:



Luego de que el Parser analiza los archivos de entrada y construye los objetos Java correspondientes, los agrupa en un objeto llamado flInput que es pasado al Aplanador. La construcción del grafo se realiza en el Aplanador, que es el componente de la herramienta que realiza la expansión de los elementos de la teoría marco (por ejemplo, a partir de los roles especificados, crea los agentes) y de las fórmulas. Por último, ambas son pasadas al Validador, que es el módulo encargado de comunicarse con el model checker.

En el caso de las fórmulas, se realizan las siguientes dos tareas en el Aplanador:

- Instanciación
- Traducción

La primera tarea consiste en que las fórmulas se instancian con los agentes que correspondan, en base a la especificación de los roles y de las acciones. Además, en el caso de las fórmulas que contengan a los cuantificadores FORALL o EXISTS, se debe realizar la expansión de dichos cuantificadores.

La segunda tarea es la traducción de las fórmulas FL en fórmulas LTL.

Los demás pasos de la solución se realizan en el Validador. El Validador, es el componente que se encarga de validar las reglas y permisos especificadas en FL. Para cumplir con dicha meta, el Validador debe crear el autómata a partir de la teoría marco con todas las componentes expandidas. Otra función del Validador es invocar al model checker, pasándole como parámetro el autómata y la fórmula a validar y por último analizar el resultado devuelto por este.

Decisiones de Implementación

Para implementar el marcado del grafo explicado en 2.1.3.2 se debía hacer una propagación por todos los nodos alcanzados a partir de este. Actualmente se conocen dos algoritmos para realizar dicha propagación: BFS y DFS.
En lo único en lo que difieren es en el orden en que son visitados los nodos. Nuestra implementación utiliza el algoritmo BFS.

2.1.4.2 Casos de estudio

En este apartado se incluyen las pruebas realizadas para cuantificar la mejora introducida.

Las pruebas consistieron en medir y comparar la cantidad de componentes de la teoría marco, el tamaño del autómata y el tiempo que tarda el model checker en dar una respuesta, con o sin filtrado. Además, debido a que el filtrado disminuye la reducción a medida que crece la cantidad de cláusulas que contiene la especificación se van a incluir diferentes pruebas variando la cantidad de cláusulas[7]. El criterio que se adoptó para elegir el subconjunto de cláusulas que se incluyen en las pruebas, es el orden en que aparecen en la ley original, otros criterios válidos que se podrían haber tomado son: por longitud de cláusula o tomarlas aleatoriamente.

Por otra parte, dado que una cláusula en FL, se traduce en varias fórmulas LTL (de acuerdo a la cantidad de agentes que puedan realizar las acciones de dicha cláusula), se decidió incluir como métrica el tamaño de la fórmula LTL. Por último, para tener una medida de cuán compleja es la fórmula LTL se contabilizan la cantidad de operadores modales y signos '=', que es una forma fácil de estimar la cantidad de términos de la fórmula, ya que se trata en general de combinaciones de 'variable = valor'.

Para poder referenciar a las pruebas de manera simple se decidió darles un nombre. La convención es la siguiente:
LDCn_m: Para el caso de estudio n de la Ley de Defensa del Consumidor, con cantidad de cláusulas m. Por ejemplo LDC1_5: Para el caso de estudio 1 de la Ley de Defensa del Consumidor, con cantidad de cláusulas 5

Una última aclaración sobre las pruebas, más precisamente sobre el *"Tiempo que tarda el model checker",* en algunas ocasiones se pondrá "NO" como resultado de la medición, esto quiere decir que el model checker no llegó a arrojar un resultado para la prueba, algunos motivos pueden ser:

- Time out. Por un tema de practicidad dado que había muchas pruebas para hacer, se decidió tomar un timeout de 90 minutos.
- La fórmula LTL excede el límite que puede procesar.

Estas son las especificaciones de la computadora en que se corrieron las pruebas:
Procesador: Intel Xeon E5-2690 v2 de 3.00GHz
Memoria: 16 Gb
Sistema operativo: Debian 3.16.7 de 64 bits

2.1.4.2.1 Caso de estudio 1: Ley de Defensa Del Consumidor

Para realizar las pruebas se ha tomado como ejemplo la codificación de la Ley de Defensa del Consumidor de la República Argentina, realizada por el grupo de investigación de Informática y Derecho de la Universidad FASTA de Mar del Plata. En este caso de estudio se incluye la teoría marco completa.

[7] Las pruebas se encuentran en https://github.com/formalex/FL/tree/branchCarlos/resources/CasosDeEstudioFiltrado

Nombre: LDC1_1
 # Cláusulas: 1
Tamaño de la fórmula LTL: 4.1 Kb
Cantidad de operadores modales (F y G): 23
Cantidad de signos = : 92

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	23	66.66
# acciones	2870	92	96.79
# contadores	34	0	100
# intervalos	183	0	100
Tamaño del autómata (en Kb)	2333	39.2	98.32
Tiempo que tarda el model checker (en segundos)	NO	413	

Nombre: LDC1_5
 # Cláusulas: 5
Tamaño de la fórmula LTL: 9.6 Kb
Cantidad de operadores modales (F y G): 95
Cantidad de signos = : 190

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	2870	189	93.41
# contadores	34	0	100
# intervalos	183	0	100
Tamaño del autómata (en Kb)	2333	84.9	96.36
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_15
 # Cláusulas: 15
Tamaño de la fórmula LTL: 34.2 Kb
Cantidad de operadores modales (F y G): 327
Cantidad de signos = : 701

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	2870	491	82.89
# contadores	34	0	100
# intervalos	183	23	87.43
Tamaño del autómata (en Kb)	2333	269.8	88.44
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_25
 # Cláusulas: 25
Tamaño de la fórmula LTL: 65.4 Kb
Cantidad de operadores modales (F y G): 557
Cantidad de signos = : 1277

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	2870	815	71.60
# contadores	34	0	100
# intervalos	183	23	87.43
Tamaño del autómata (en Kb)	2333	456.7	80.42
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_60
 # Cláusulas: 60
Tamaño de la fórmula LTL: 116.3 Kb
Cantidad de operadores modales (F y G): 970
Cantidad de signos = : 2165

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	55	20.29
# acciones	2870	1155	59.76
# contadores	34	4	88.24
# intervalos	183	24	86.89
Tamaño del autómata (en Kb)	2333	664.3	71.53
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.1.4.2.1.1 Análisis de las pruebas del caso de estudio

Como se dijo al comienzo de la tesis, ésta es la especificación más grande con la cual se ha probado la herramienta, por lo tanto no son menos grandes los desafíos para lograr mejoras en la performance.

Una mejora que se puede observar en las pruebas es una importante reducción en el tamaño del autómata, la cual es muy notoria cuando la especificación tiene muy pocas cláusulas, por consiguiente es más probable que se puedan obtener resultados en ejemplos con pocas cláusulas, a diferencia de la versión anterior de la herramienta, que a nivel tamaño de autómata le da lo mismo correr una especificación con 1 o con 60 cláusulas, ya que el tamaño del autómata siempre es el mismo (2333 Kb). A raíz de esa importante reducción en el tamaño del autómata se logra que en la primer prueba con filtrado termine a los 453 segundos, es decir en menos de 7 minutos.

En resumen con la mejora introducida se podrían hacer pruebas incrementales en la cantidad de cláusulas de la especificación. Esta característica puede ser aprovechada, cuando se están empezando a escribir las primeras cláusulas de una especificación.

Mientras se realizaban las pruebas se detectó una limitación del model checker. Dicha limitación consiste en que el model checker sólo puede procesar fórmulas LTL de hasta 65.536 bytes de tamaño. Ese límite se alcanzó en el ejemplo que tiene 25 cláusulas que traducidas a LTL dan un total de 65.425 bytes. Es decir está en el límite de lo que puede procesar el model checker que estamos usando. En la sección *"Conclusiones y trabajo futuro"* se incluyen algunas mejoras que se le podrían hacer a la herramienta para poder mitigar los efectos de esta limitación del model checker.

2.1.4.2.2 Caso de estudio 2: Ley de Defensa del Consumidor - primera parte

La Ley de Defensa del Consumidor es un ejemplo demasiado grande, que además tiene la particularidad de estar dividida en capítulos, los cuales poseen la característica de que algunos de ellos son independientes entre sí. Además, como se vio en las pruebas anteriores, la mejora introducida en la herramienta fomenta el trabajo incremental. Se decidió incluir dos casos de estudio más, que surgen de dividir la teoría marco original en dos partes "casi" independientes (sólo comparten dos o tres acciones). Sin embargo es importante aclarar que esta división no deja de ser problemática porque efectivamente hay comportamientos producto de la interacción de las partes que podrían estar perdiéndose. Es decir, esta división no reemplaza por el momento a un análisis completo.

Para armar este caso de estudio se tuvieron en cuenta los siguientes capítulos de la ley: del 1 al 4, y del 7 al 8.

Dichos capítulos hacen referencia a la noción de Consumidor, Proveedor, relación de Consumo, Condiciones sobre la oferta y la venta, Venta fuera del establecimiento del Proveedor, y operaciones de venta realizadas mediante crédito financiero. Es decir, para este caso de estudio se incluye sólo la porción de la teoría marco original que modela los capítulos seleccionados. De la misma forma las cláusulas que se incluyen en las pruebas de este caso de estudio son las que únicamente hacen referencia a la porción de teoría marco seleccionada.

Nombre: LDC2_1
 # Cláusulas: 1
Tamaño de la fórmula LTL: 4.1 Kb
Cantidad de operadores modales (F y G): 23
Cantidad de signos = : 92

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	23	66.67
# acciones	1629	92	94.35
# contadores	0	0	-
# intervalos	92	0	100
Tamaño del autómata (en Kb)	1269	39.2	96.91
Tiempo que tarda el model checker (en segundos)	NO	346	

Nombre: LDC2_5
 # Cláusulas: 5
Tamaño de la fórmula LTL: 9.6 Kb
Cantidad de operadores modales (F y G): 95
Cantidad de signos = : 190

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	1629	189	88.40
# contadores	0	0	-
# intervalos	92	0	100
Tamaño del autómata (en Kb)	1269	84.9	93.31
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_15
 # Cláusulas: 15
Tamaño de la fórmula LTL: 34.2 Kb
Cantidad de operadores modales (F y G): 327
Cantidad de signos = : 701

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	1629	491	69.86
# contadores	0	0	-
# intervalos	92	23	75.00
Tamaño del autómata (en Kb)	1269	269.8	78.74
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_25
 # Cláusulas: 25
Tamaño de la fórmula LTL: 61.8 Kb
Cantidad de operadores modales (F y G): 557
Cantidad de signos = : 1254

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	1629	814	50.03
# contadores	0	0	-
# intervalos	92	23	75.00
Tamaño del autómata (en Kb)	1269	438.8	65.42
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_28
 # Cláusulas: 28.
Tamaño de la fórmula LTL: 66.4 Kb
Cantidad de operadores modales (F y G): 583
Cantidad de signos = : 1331

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	47	31.88
# acciones	1629	842	48.94
# contadores	0	0	-
# intervalos	92	23	75.00
Tamaño del autómata (en Kb)	1269	454.4	64.19
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.1.4.2.2.1 Análisis de las pruebas del caso de estudio

Empecemos comentando algunas particularidades. El tamaño del autómata sin filtro es cercano a la mitad del autómata del caso de estudio 1, 1269 Kb vs 2333 Kb. Esto es una casualidad, ya que nuestro objetivo no era dividir en la mitad la codificación del caso de estudio 1, sino era hacer una partición donde las especificaciones resultantes fueran los más independientes posibles.

Otra particularidad se da en que la segunda columna (mediciones con filtro) de las primeras tres pruebas (con # Cláusulas: 1, 5 y 15) de este caso de estudio es igual a la del caso de estudio 1. Esto se debe a que las primeras 15 cláusulas de este caso de estudio, corresponden a los primeros cuatro capítulos de la ley. Es decir los dos casos de estudio coinciden en las primeras 15 cláusulas. Recién se empiezan a notar diferencias en la segunda columna en las pruebas con 25 y 28 cláusulas.

La última particularidad que tiene este caso de estudio, es que al igual que en el caso de estudio 1 en el ejemplo con 25 cláusulas se alcanza el límite de fórmulas que puede procesar el model checker. En este último caso el largo de la fórmula LTL es de 61.8 Kb.

La buena noticia es que al igual que en el caso de estudio anterior se obtienen resultados en la primera prueba, en este caso la herramienta tarda 346 segundos, es decir menos de 6 minutos por lo cual es un poco mas rapida que la prueba del caso de estudio anterior.

2.1.4.2.3 Caso de estudio 3: Ley de Defensa del Consumidor - segunda parte

En este caso de estudio se incluye la segunda parte en la que se dividió la codificación de la Ley de Defensa del Consumidor.
Para este caso de estudio se tuvieron en cuenta los siguientes capítulos de la ley: Del 5 al 6 y del 9 al 13.
Estos capítulos se refieren a la prestación de servicios, a los servicios públicos, a cláusulas abusivas, a Responsabilidad por daños y por último a la autoridad de aplicación de la ley, como así también a los procedimientos, sanciones y acciones judiciales en caso de que un proveedor no cumpla con lo que dicta la Ley de Defensa del Consumidor. Es decir, para este caso de estudio se incluye sólo la porción de la teoría marco original que modela los capítulos seleccionados. De la misma forma las cláusulas que se incluyen en las pruebas de este caso de estudio son las que únicamente hacen referencia a la porción de teoría marco seleccionada.

Nombre: LDC3_1
 # Cláusulas: 1
Tamaño de la fórmula LTL: 2.6 Kb
Cantidad de operadores modales (F y G): 23
Cantidad de signos = : 46

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	23	66.67
# acciones	1367	46	96.63
# contadores	34	0	100
# intervalos	91	0	100
Tamaño del autómata (en Kb)	1143	23.8	97.92
Tiempo que tarda el model checker (en segundos)	NO	1	

Nombre: LDC3_5
 # Cláusulas: 5
Tamaño de la fórmula LTL: 15.9 Kb
Cantidad de operadores modales (F y G): 115
Cantidad de signos = : 253

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	23	66.67
# acciones	1367	162	88.15
# contadores	34	0	100
# intervalos	91	0	100
Tamaño del autómata (en Kb)	1143	101.1	91.15
Tiempo que tarda el model checker (en segundos)	NO	4145(1h 9m 5s)	

Nombre: LDC3_15
 # Cláusulas: 15
Tamaño de la fórmula LTL: 34.6 Kb
Cantidad de operadores modales (F y G): 280
Cantidad de signos = : 577

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	29	57.97
# acciones	1367	200	85.37
# contadores	34	0	100
# intervalos	91	0	100
Tamaño del autómata (en Kb)	1143	124.9	89.07
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_25
 # Cláusulas: 25
Tamaño de la fórmula LTL: 47 Kb
Cantidad de operadores modales (F y G): 348
Cantidad de signos = : 779

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	33	52.17
# acciones	1367	278	79.66
# contadores	34	0	100
# intervalos	91	0	100
Tamaño del autómata (en Kb)	1143	185.7	83.75
Tiempo que tarda el model checker (en segundos)	NO	1134	

Nombre: LDC3_32
 # Cláusulas: 32.
Tamaño de la fórmula LTL: 50.1 Kb
Cantidad de operadores modales (F y G): 387
Cantidad de signos = : 834

	Sin filtrado	Con filtrado	Porcentaje de reducción
# agentes	69	40	42.03
# acciones	1367	336	75.42
# contadores	34	4	88.24
# intervalos	91	1	98.90
Tamaño del autómata (en Kb)	1143	222.3	80.55
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.1.4.2.3.1 Análisis de las pruebas del caso de estudio

Lo más destacable de estas pruebas, es que hay tres ejemplos en que la herramienta devuelve resultados.

Por lo tanto en el caso de trabajar con un conjunto de normas grande como es el de la Ley de Defensa del Consumidor, se podría comenzar a realizar la tarea de codificación en grupos independientes, es decir la codificación de un conjunto de normas se podría hacer mediante al menos dos grupos independientes de personas, obviamente eso depende del conjunto de normas a codificar, sobre todo en cuántos subconjuntos independientes se puede dividir. En el caso de la Ley de Defensa del Consumidor, se decidió dividirla en dos partes, pero se podría haber llegado a dividir hasta en tres partes.

Otro aspecto a remarcar, es que la reducción en el tamaño del autómata es bastante notoria, tanto en porcentaje (80.55%), como así también en el tamaño final. En la última prueba llega a tener un tamaño de 222.3 Kb contra 454.4 Kb y 664.3 Kb, de los casos de estudio anteriores. Incluso en la primera prueba, llega a tener un tamaño de 23.8 Kb, siendo el menor de todas las pruebas.

Haciendo un análisis más detallado de los ejemplos en los que se obtuvieron buenos resultados, podemos decir que en el primer ejemplo se encontró un comportamiento legal válido, es decir que no hay inconsistencias del tipo que impide la existencia de alguna forma de cumplir con todas las normas en simultáneo en esa porción de la ley que se está verificando. Otro aspecto a remarcar del primer ejemplo es que la fórmula LTL es la más chica de todas las pruebas realizadas, con un tamaño de 2.6 Kb. Probablemente esta característica haya favorecido a la obtención de buenos resultados, dado que la herramienta tarda sólo un segundo.

En los otros dos ejemplos la herramienta no puede encontrar un comportamiento legal válido. Queda pendiente para cuando la herramienta permita determinar cuáles son las fórmulas en conflicto analizar más en detalle estos casos

Con respecto al resultado de las pruebas uno se podría realizar las siguientes preguntas:

1. Por qué la LDC3_25 devuelve resultados y la LDC3_15 que es un subconjunto de la LDC3_25 no?
2. Por qué la LDC3_25 tarda mucho menos que la LDC3_5 que es un subconjunto de la LDC3_25?

Estas dos preguntas tiene una misma respuesta y es que el agregado de una fórmula puede hacer que:

a. Se limite el árbol de backtracking de la exploración o
b. que no se pueda limitar el árbol.

Volviendo a las dos preguntas originales una explicación plausible[8] de los resultados obtenidos es la siguiente:

1. Alguna de las fórmulas de la 6 a la 15 hace que no se pueda limitar el árbol. Por eso la LDC3_15 no devuelve resultados.
2. Alguna de las fórmulas de la 16 a la 25 hace que SÍ se pueda limitar el árbol. Por eso la LDC3_25 devuelve resultados.

Por último, algo para resaltar de la métrica “tamaño de la fórmula LTL”. Si nos ponemos a analizar las diferentes pruebas con número de cláusulas 25, en el último caso de estudio, el tamaño de la fórmula LTL es 47 Kb contra 65.4 Kb y 61.8 Kb del primer y segundo caso de estudio respectivamente. Es decir en estos ejemplos se justifica la inclusión de la métrica “tamaño de la fórmula LTL”, ya que comparando tres ejemplos con igual número de cláusulas, difieren en el tamaño de la fórmula LTL significativamente.

2.2 Reducción de la representación de una acción a dos estados

2.2.1 Motivación

Con el objetivo de reducir el tamaño y la complejidad del autómata se propone en los casos en que sea factible la reducción de la forma en que se representan las acciones en el autómata. Actualmente las acciones se representan con tres estados, y en algunos casos podrían llevarse a sólo dos estados, prescindiendo del estado **`JUST_HAPPENED`**. Además, con el fin de darle más expresividad al lenguaje de forma de poder detectar en qué casos esta optimización puede realizarse, se propone el agregado de dos nuevos operadores. El primero de ellos, **`isHappening()`**, servirá para hacer referencia al momento en que una acción comienza a ocurrir, diferenciándose de la semántica actual en donde toda referencia a una acción es al momento en el que se completa. El segundo, **`justHappened()`**, sirve justamente para hacer referencia a este comportamiento.

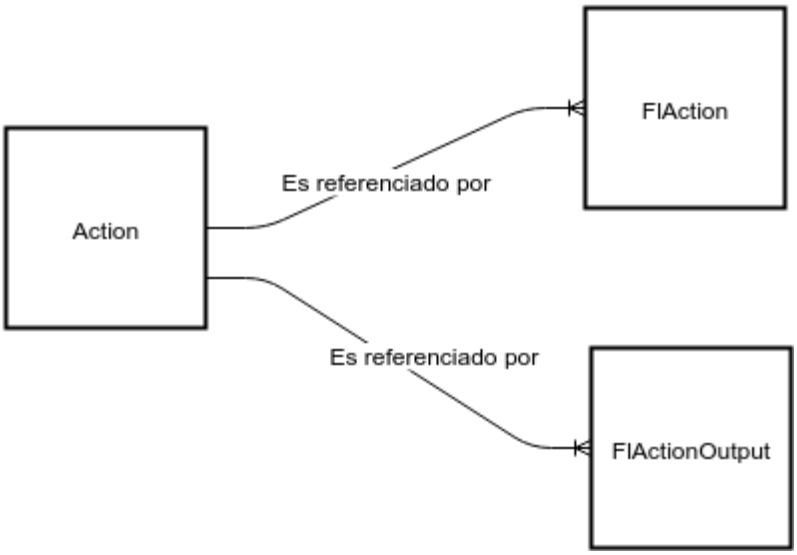
Antes de meternos en el detalle de la solución propuesta vamos a explicar algunos aspectos de las acciones que van a ser necesarios al momento de describir la solución.

2.2.2 Acciones y sus referencias

En la teoría marco se definen acciones y éstas a su vez son referenciadas una o más veces en las diferentes cláusulas. A cada acción se la representa internamente con un objeto de tipo **`Action y`** por otra parte a cada una de las referencias a una acción en una cláusula se la puede representar con dos objetos diferentes, según el contexto de uso: **`FlAction`** O **`FlActionOutput.`**

El primer caso se da cuando el objeto `Action` no tiene output values o cuando tiene output values pero en las cláusulas no se hace referencia a ninguno de ellos. El segundo caso se da cuando el objeto `Action` tiene output values y en las cláusulas se hace referencia a alguno de ellos (mediante **`results in`**).

[8] Sobre todo en la pregunta 2 también puede estar influyendo el ordenamiento de las variables mencionado en la sección “2.3.3.5.1.2 Análisis de las pruebas de los casos de estudio”.



Para que quede más claro vamos a verlo con dos ejemplos. En el primero de ellos vamos a mostrar únicamente referencias a **`FlAction`**, mientras que en el segundo vamos a mostrar alguna referencias a **`FlActionOutput`**

```
roles Consumidor, Proveedor disjoint

action SolicitarInformacion only performable by Consumidor
action SuministrarInformacion only performable by Proveedor output values
{CiertaClaraDetalladaGratuita, NocumpleCiertaClaraDetalladaGratuita, EsRiesgosa}
action ProveerBySRiesgoso only performable by Proveedor
action EntregarManual only performable by Proveedor output values
{EnEspCInstdeUsoInstalyMant,SinEspCInstdeUsoInstalyMant}


FORALL(i:Consumidor; O(i.SolicitarInformacion -> EXISTS(j:Proveedor; j.SuministrarInformacion)))
FORALL(i:Proveedor;  F(i.ProveerBySRiesgoso & !i.SuministrarInformacion))
FORALL(i:Proveedor; O(i.ProveerBySRiesgoso -> i.EntregarManual))
```

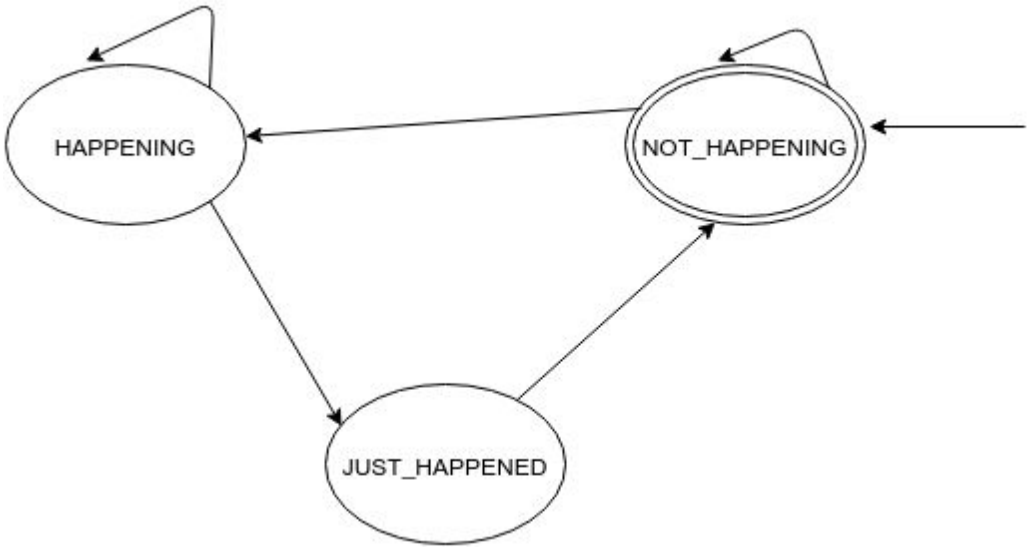
En el ejemplo tenemos a las acciones `SuministrarInformacion` y `ProveerBySRiesgoso`, las cuales son referenciadas en dos cláusulas diferentes, mientras que las acciones `SolicitarInformacion` y `EntregarManual` son referenciadas en una sola cláusula, es decir tenemos un total de seis referencias. Por otra parte las acciones `SuministrarInformacion` y `EntregarManual` están definidas con output values pero en ninguna de las fórmulas son referenciadas mediante la construcción gramatical **`results in.`** Por lo tanto las 6 referencias a las 4 acciones definidas son a **`FlAction.`**

```
action SuministrarInformacion only performable by Proveedor output values
{CiertaClaraDetalladaGratuita, NocumpleCiertaClaraDetalladaGratuita, EsRiesgosa}
action ProveerBySRiesgoso only performable by Proveedor
action EntregarManual only performable by Proveedor output values
{EnEspCInstdeUsoInstalyMant,SinEspCInstdeUsoInstalyMant}


FORALL(i:Proveedor;  F(i.ProveerBySRiesgoso & i.SuministrarInformacion results in
NocumpleCiertaClaraDetalladaGratuita ))
FORALL(i:Proveedor; O(i.ProveerBySRiesgoso -> i.EntregarManual results in
EnEspCInstdeUsoInstalyMant ))
```

En este segundo ejemplo las acciones `SuministrarInformacion` y `EntregarManual` están definidas con output values y en cada una de las cláusulas donde se las referencia se hace mediante **`results in.`**Por lo tanto las dos referencias a estas acciones son a **`FlActionOutput`** a diferencia de las dos referencias de la acción `ProveerBySRiesgoso.`

2.2.3 Representación de una acción en el autómata

Una acción se representa en el autómata con una variable, la cual puede tomar alguno de los siguientes tres valores: `NOT_HAPPENING, HAPPENING ó JUST_HAPPENED`. El valor inicial de la variable es `NOT_HAPPENING`. Por último, los cambio de estados posibles son los siguientes:



2.2.4 Solución - Marco teórico

Como se expresó más arriba, nuestro principal objetivo es reducir el tamaño y la complejidad del autómata que se le envía al model checker, pero para poder lograr dicho objetivo necesitamos un paso intermedio, que es el agregado de dos nuevos operadores, es por esto que vamos a comenzar explicando la solución de este paso intermedio.

2.2.4.1 Agregado de dos nuevos operadores

Se propone el agregado de dos nuevos operadores para poder ser utilizados en las cláusulas con el fin de darle más expresividad al lenguaje. Dichos operadores son:
`isHappening`: El cual toma una acción y devuelve `true` o `false` **si la acción está ocurriendo o no**. Viéndolo a más bajo nivel esto quiere decir que la variable que representa a la acción está en el estado `HAPPENING`.
`justHappened`: El cual toma una acción y devuelve `true` o `false` **si la acción terminó de ocurrir o no**. Este operador representa el comportamiento de la herramienta, antes de la modificación. Viéndolo a más bajo nivel esto quiere decir que la variable que representa a la acción está en el estado `JUST_HAPPENED`. Si bien al momento de realizar la modificación el default ya era `justHappened` (con lo cual no tendría mucho sentido agregar este operador), la idea era evaluar si ese default podía cambiarse y además eliminar ambigüedad. Es más en cada uno de los casos de estudio se agregó una prueba con el default en `isHappening` (ver Sección "2.2.5.3 Casos de estudio")

Antes de pasar a los ejemplos, vamos a hacer una aclaración con respecto al modelado de las acciones, que va a ser útil para comprender lo que viene. En nuestro modelo, cuando una acción comienza a suceder nada puede evitar que termine de completarse, por lo tanto en muchas situaciones (aunque no en todas, ver más adelante) va a dar lo mismo expresar que una acción comenzó a suceder o que una acción se completó.

A continuación se incluyen algunas situaciones en las que se podrían utilizar los nuevos operadores.

Si se quiere expresar **prohibiciones**, en la gran mayoría de los casos alcanza con decir que la acción está ocurriendo, independientemente de si termina de ocurrir o no, esto vale por lo explicado anteriormente. Por ejemplo:
"Prohibido pisar el césped ", se podría escribir así:

```
FORALL(p; F(isHappening(p.pisarElCesped)))
```

Para los **permisos**, **también** alcanza con decir que la acción está ocurriendo, independientemente de si termina de ocurrir o no. Por ejemplo:
“Los estudiantes tienen derecho a participar en actividades de investigación”, se podría escribir así:
`FORALL(i:estudiante; P(isHappening(i.hacerInvestigacion)))`

En cambio lo que se suele buscar modelar usando **obligaciones** presenta un escenario un poco más complejo. Parecería que en los casos en que la obligación debe cumplirse dentro de un intervalo hace falta que la acción termine de ocurrir, ya que podría darse el caso de que la acción comience a ocurrir dentro del intervalo, pero finalice después, por lo cual la obligación no estaría satisfecha. Tomemos como ejemplo el censo y rematriculación de alumnos de la UBA, que deben ser completados obligatoriamente por los estudiantes cada año dentro de un periodo determinado. En el caso de que el alumno comience el censo, pero no lo termine a tiempo, la obligación aún no está cumplida. Por lo tanto estaría mal modelar la obligación de la siguiente manera:

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

La forma correcta es:
```
FORALL(i:estudiante; O(<>_{en_censo} justHappened(i.censar)))
```

Donde `en_censo` es el intervalo que representa el periodo de tiempo en que los estudiantes están obligados a censarse.

Por otro lado, si hablamos de **obligaciones** de este estilo `O(accionA -> accionB)` el sentido común nos puede jugar una mala pasada. Uno se vería tentado a pensar que lo que se quiere expresar es que para que comience a ocurrir la `accionB`, la `acciónA` debería haber finalizado. Es decir, cuando una acción es antecedente de una implicación parecería que siempre tiene que referenciarse con el operador `justHappened`. Por lo tanto nos parecería correcto reescribir la fórmula de la siguiente manera : `O(justHappened(accionA) -> accionB)`. Sin embargo esto no es así para todos los casos. Mediante un ejemplo vamos a mostrar la excepción a esta última regla.

Un artículo de la Ley de Defensa del Consumidor expresa que *al momento de la venta* es obligatorio redactar un documento. En FL esto se modela así

```
FORALL(i:Proveedor; O(isHappening(i.Vender) -> isHappening(i.RedactarDocVenta)))
```

como se puede ver la acción `Vender`, que figura como antecedente de una implicación es referenciada con el operador `isHappening` contradiciendo la regla general de que el antecedente de una implicación debe referenciarse con `justHappened`.

2.2.4.2 Reducción de la representación de una acción a dos estados

La solución consiste en recorrer cada una de las acciones de la teoría marco y decidir para cada una de ellas si se puede reducir su representación a dos estados y en ese caso reducirla. Este proceso hay que hacerlo después del filtrado, así tenemos acciones que son referenciadas al menos una vez en las cláusulas.

A continuación se describen las condiciones que debe cumplir una acción para que se pueda reducir su representación a dos estados.

1. **Todas** las referencias en las cláusulas**,** deben ser mediante el operador `isHappening`, ya que una acción tiene una sola representación por más que tenga muchas referencias, y por lo tanto en el hipotético caso de que una acción fuera referenciada al menos una vez con el operador `justHappened` es motivo suficiente para no poder reducir su representación a dos estados.

2. **No puede ser acción de inicio ni de fin de algún intervalo**, ni tampoco ser acción modificadora de algún contador, ya que eso implica la necesidad de contar con el estado `JUST_HAPPENED`. Esto se debe a que en el autómata los contadores cambian su valor y los intervalos cambian de estado cuando alguna de las acciones pasa al estado `JUST_HAPPENED`. Supongamos que existe un intervalo `i` con acciones `a` y `b` de inicio y fin respectivamente, para esas acciones es necesario saber cuándo terminan de ejecutarse (lo que se traduce en que las acciones alcancen el estado `JUST_HAPPENED`) porque recién ahí el intervalo comienza o termina efectivamente. Por lo tanto en este caso las acciones `a` y `b` no podrían ser reducidas ya que tendrían una referencia implícita a `JUST_HAPPENED`.
3. **No tener referencias** en las cláusulas mediante **results in,** porque en caso de tener al menos una, eso implica la necesidad de contar con el estado `JUST_HAPPENED`. Esto se debe a que en el autómata una acción produce sus valores de salida recién cuando pasa al estado `JUST_HAPPENED`. Supongamos que existe una acción `a` con valores de salida `out1, out2, out3`. Para que la acción `a` tome uno de esos valores es necesario saber cuando `a` termina de ejecutarse (lo que se traduce en que `a` alcance el estado `JUST_HAPPENED`) porque es en ese momento en que la acción puede tomar un nuevo valor de salida. Por lo tanto en este caso la acción `a` no podría ser reducida ya que tendría una referencia implícita a `JUST_HAPPENED`.

Por último el proceso de reducción sería así:

```
Para cada acción a de la teoría marco
       Si se puede reducir la representación de a
              a.setRepresentation(DOS_ESTADOS)
       Fin Si
FinPara
```

2.2.5 Solución - Marco práctico

2.2.5.1 Agregado de dos nuevos operadores

2.2.5.1.1 Modificaciones al Modelo

A la clase **`FlAction`,** la cual representa una referencia en particular a una acción en una cláusula, se le agrega un nuevo campo de tipo Enum, el cual indica si a la acción se la está referenciando en una cláusula con el operador `isHappening` o `justHappened`. Además se modifica el método `toString()`[9], el cual se usa para generar el término de la fórmula LTL que se le pasa al model checker. De esta forma se calcula el valor que debe tener la variable de acuerdo al valor del nuevo campo.

```
@Override
public String toString() {
    return getNameWithAgent() + " = " + this.getReferencedState().getValueInLtlFormula();
}
```

Veámoslo con un ejemplo:

```
roles estudiante
action hacerInvestigacion only performable by estudiante
FORALL(i:estudiante; P(isHappening(i.hacerInvestigacion)))
```

En este caso la referencia a la acción `hacerInvestigacion` se llamaría `"agent_1.hacerInvestigacion"`, entonces lo que devuelve el método `toString()` es `"agent_1.hacerInvestigacion = HAPPENING"`.

[9] Desde el punto de vista de diseño es discutible asignar esa tarea al método toString(), porque lo que devuelve es una comparación en una traducción particular, por lo cual es poco abierto al cambio. Esto es un aspecto a cambiar en próximas versiones de la herramienta.

En el caso que se hiciera referencia a la acción mediante el operador `justHappened`, lo que devolvería el método, sería lo siguiente `"agent_1.hacerInvestigacion = JUST_HAPPENED"`

2.2.5.1.2 Modificaciones al parser

El primer cambio a realizar es en la gramática de FL, específicamente en donde se define qué cláusulas son sintácticamente válidas. En este caso el cambio en la gramática consiste en agregar una nueva opción por cada operador a la definición de terminal de una fórmula. Si pensamos a una fórmula como un árbol de componentes léxicos, los terminales son las hojas del árbol.

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

El segundo cambio consiste en definir qué se hace en el parser al encontrar alguno de estos dos operadores. Lo más interesante es que a la `flAction` se le establece el valor de la propiedad **`referencedState`**, de acuerdo a si la acción se la referencia con el operador `isHappening` o `justHappened`. En caso de que la acción no esté referenciada por ninguno de los dos operadores, el valor por defecto es `justHappened`.

2.2.5.2 Reducción de la representación de una acción a dos estados

2.2.5.2.1 Arquitectura

El proceso de reducción se lleva a cabo en el Validador, más precisamente después de realizar el filtrado de la teoría marco (recordar que el filtrado se realiza momentos antes de crear el autómata de la teoría marco). La primera parte de dicho proceso consiste en recopilar la información necesaria para decidir si se puede reducir la representación de una acción o no. La segunda y última parte del proceso de reducción se realiza en el template de Velocity, que sirve como "plantilla" para generar el autómata de la teoría marco con el formato requerido por el model checker.

2.2.5.2.2 Primera parte del proceso de reducción: creación e invocación del reductor

A continuación se van a detallar las modificaciones y/o agregados que se le tuvieron que realizar al modelo, y posteriormente se va a describir de manera más detallada los pasos necesarios del proceso de reducción.

El primer agregado al modelo fue un nuevo objeto de tipo `ReductorDeRepresentacionDeAccionADosEstados` Este objeto va a a tener la responsabilidad de decidir si se puede o no reducir la representación de una acción. Para cumplir con esta tarea va a tener que recopilar y calcular información a partir de la teoría marco y de la fórmula que se le va a enviar al model checker.

En la clase que representa a las acciones, se le agrega un nuevo campo llamado `representation`, el cual va a indicar si la acción se representa en el autómata con dos o tres estados, siendo el valor por defecto el de tres estados.

En la clase que representa a la teoría marco se agregaron dos métodos que sirven para saber los nombres de las acciones que son de inicio o de fin de algún intervalo y los nombres de las acciones que modifican a algún contador.
En la clase que representa a las fórmulas, se agregaron tres métodos: uno para calcular los terminales que aparecen en una fórmula y otros dos para calcular las referencias a acciones en la fórmula (es decir a `flAction`) y las referencias a acciones con output values mediante el uso de la construcción "`results in`" (es decir a `flActionOutput`).

Al momento de crear el reductor se le deben pasar como parámetros la teoría marco y la fórmula que se le va a enviar al model checker. Con esa información el reductor realiza los siguientes pasos:

1. Obtiene los `flAction` de la fórmula.
2. Obtiene los `flActionOutput` de la fórmula.
3. Para tener un buen rendimiento arma un diccionario con los datos del paso 1, donde la clave es el nombre de la acción y el valor es una lista de `flAction`. Con esto se tiene para cada acción sus correspondientes referencias a `flAction` en la fórmula.
4. De la misma forma se arma otro diccionario pero tomando como entrada los datos del paso 2. Con esto se tiene para cada acción sus correspondientes referencias a `flActionOutput` en la fórmula.
5. Invoca al método de la teoría marco que devuelve los nombres de las acciones que son de inicio o de fin de algún intervalo.
6. Invoca al método de la teoría marco que devuelve los nombres de las acciones que modifican a algún contador.

Por último, una vez creado el reductor se llama al método `ejecutar()`, el cual se encarga de recorrer cada una de las acciones de la teoría marco verificando si se puede reducir la representación de la acción o no (es decir, si cumplen las tres condiciones o no), y en caso afirmativo se establece la propiedad `representation` a dos estados.

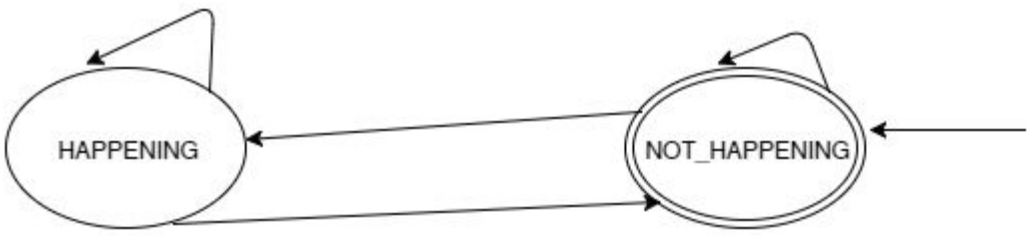
2.2.5.2.3 Segunda parte del proceso de reducción: modificación del template de Velocity

Como se dijo anteriormente el template de Velocity es la “plantilla” para generar el autómata, el cual sirve de representación de cada uno de los elementos de la teoría marco (entre ellos las acciones), y es por esto que la última parte del proceso impacta acá.

En el template se tuvieron que realizar dos modificaciones. La primera es en la definición de las variables que representan a las acciones, ya que de acuerdo a la propiedad `representation` de la acción se decide si se la representa con dos o tres estados.

```
#foreach ($action in $acciones )
	## De acuerdo a la representación, se definen con dos o tres estados
	#if ($action.hasTwotSates())
		$action: {HAPPENING, NOT_HAPPENING};
	#else
		$action: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
	#end
#end
```

La segunda modificación es en la definición de las transiciones de los estados de la acción, ya que en el caso de que una acción se represente con dos estados, obviamente impacta en las transiciones que se pueden realizar.
Las transiciones en este caso serían de la siguiente manera:



2.2.5.3 Casos de estudio

En esta sección se incluyen las pruebas realizadas para cuantificar la mejora introducida.

Las pruebas consistieron en medir y comparar la cantidad de acciones representadas con 3 estados, el tamaño del autómata y el tiempo que tarda el model checker en dar una respuesta, antes y después de llamar al reductor. Debido que la reducción varía de acuerdo a la cantidad de cláusulas que contiene la especificación se van a incluir diferentes pruebas variando la cantidad de cláusulas. El criterio que se adoptó para elegir el subconjunto de cláusulas que se incluyen en las pruebas, es el orden en que aparecen en la ley original[10].

Dado que vamos a reutilizar los casos de estudio de la Ley de Defensa del Consumidor a veces introduciendo algunos cambios, es necesario hacer algunas aclaraciones. Al momento de realizar las pruebas para el filtrado se había tomado la convención de llamar LDCn_m al caso de estudio n de la Ley de Defensa del Consumidor, con cantidad de cláusulas m.

Para reutilizar los casos de estudios basados en la Ley de Defensa del Consumidor se establece la siguiente convención: llamaremos LDCn_m', al caso de estudio n de la Ley de Defensa del Consumidor, con cantidad de cláusulas m, siendo m<=25. Donde LCDn_m y LCDn_m' tienen la misma teoría marco. La única diferencias entre LCDn_m y LCDn_m', es que este último incluye fórmulas con los operadores `isHappening` y `justHappened`.[11]

Para que quede más claro vamos a verlo con un ejemplo. Supongamos que LCD2_2 tiene las siguientes fórmulas:

```
FORALL(i:Proveedor; O(i.RespetarLeyDefensaConsumidor))
FORALL(i; O(i.ProveerBySRiesgoso -> i.EntregarManual))
```

Un conjunto de fórmulas válidas para LCD2_2' sería el siguiente:

```
FORALL(i:Proveedor; O(isHappening(i.RespetarLeyDefensaConsumidor)))
FORALL(i; O(isHappening(i.ProveerBySRiesgoso) -> isHappening(i.EntregarManual)))
```

Es decir, los dos casos de estudio comparten las mismas fórmulas, la diferencia se da que en el segundo caso para referenciar a las acciones se hace uso de los operadores `isHappening` y `justHappened`.

Para los casos de estudio de la Ley de Defensa del Consumidor, con cantidad de cláusulas mayor a 25 se usan exactamente los mismos casos de estudio sin modificación alguna. Lo único que se cambia al momento de realizar las pruebas es el valor por defecto del atributo que indica si a la acción se la está referenciando en una cláusula con el operador `isHappening` o `justHappened`. Para estas pruebas se lo va a cambiar a `isHappening`, es decir el efecto que se logra es que si la acción no es referenciada con ninguno de los dos operadores se supone que está siendo referenciada por `isHappening`. Esto se hace para poder medir fácilmente cuál es la máxima reducción que se puede lograr.

2.2.5.3.1 Casos de estudio con filtrado

Las pruebas incluidas en esta sección se corren con el filtrado activado para poder cuantificar la mejora introducida por el filtrado más el reductor, es decir partiendo de la optimización anterior cuánto puede mejorar el reductor. En la sección posterior se incluirán pruebas para medir solamente la mejora introducida por el reductor, es decir se van a correr sin filtrado.

[10] Otros criterios válidos que se podrían haber tomado son: por longitud de cláusula o tomarlas aleatoriamente.
[11] Las pruebas se encuentran en https://github.com/formalex/FL/tree/branchCarlos/resources/CasosDeEstudioReductor

2.2.5.3.1.1 Caso de estudio 1: Ley de Defensa Del Consumidor

Nombre: LDC1_1'

Tamaño de la fórmula LTL: 3.8 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	92	0	100
Tamaño del autómata (en Kb)	39.2	12.1	69.13
Tiempo que tarda el model checker (en segundos)	810.8	0.3	99.96

Nombre: LDC1_5'

Tamaño de la fórmula LTL: 8.9 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	189	0	100
Tamaño del autómata (en Kb)	84.9	25	70.55
Tiempo que tarda el model checker (en segundos)	NO	3.5	

Nombre: LDC1_15'

Tamaño de la fórmula LTL: 31.5 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	491	65	86.76
Tamaño del autómata (en Kb)	269.8	133.5	50.52
Tiempo que tarda el model checker (en segundos)	NO	623	

Nombre: LDC1_25'

Tamaño de la fórmula LTL: 61.3 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	815	204	74.96
Tamaño del autómata (en Kb)	456.7	254.6	44.25
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_60 (Con default isHappening)

Tamaño de la fórmula LTL: 110.1 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	1155	184	84.07[12]
Tamaño del autómata (en Kb)	664.3	334.9	49.59
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.2.5.3.1.2 Análisis de las pruebas del caso de estudio

Lo más importante de este conjunto de pruebas, es que producto de las dos optimizaciones hay tres ejemplos en que la herramienta devuelve resultados. Además los tiempos son muy pequeños, las dos primeras pruebas tardan menos de 4 segundos mientras que la tercera alrededor de 10 minutos. Otro aspecto relevante es que la segunda y tercera prueba debido a la optimización del reductor pueden terminar cosa que antes no se podía.
Otro dato alentador es el alto grado de reducción de acciones a 2 estados, con un piso del 74%, llegando a una efectividad ideal del 100% en las primeras dos pruebas.
Si tomamos la última prueba y la comparamos con la del filtrado, se puede ver una clara reducción en el tamaño de la fórmula LTL (pasa de 116.3 Kb a 110.1), esto se debe a que por cada acción que se representa con dos estados se produce una reducción de 4 bytes como resultado de reemplazar en la fórmula el término `JUST_HAPPENED` por `HAPPENING`. En las demás pruebas si se las compara con las del filtrado también hay reducción, pero el impacto es menor.

[12] En la sección "Análisis de las pruebas del caso de estudio" del último caso de estudio de la Ley de Defensa del Consumidor se explica por qué el porcentaje de reducción no es del 100%.

2.2.5.3.1.3 Caso de estudio 2: Ley de Defensa del Consumidor - primera parte

Nombre: LDC2_1'

Tamaño de la fórmula LTL: 3.8 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	92	0	100
Tamaño del autómata (en Kb)	39.2	12.1	69.13
Tiempo que tarda el model checker (en segundos)	466.3	0.3	99.94

Nombre: LDC2_5'

Tamaño de la fórmula LTL: 8.9 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	189	0	100
Tamaño del autómata (en Kb)	84.9	25	70.55
Tiempo que tarda el model checker (en segundos)	NO	4.2	

Nombre: LDC2_15'

Tamaño de la fórmula LTL: 31.5 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	491	65	86.76
Tamaño del autómata (en Kb)	269.8	133.5	50.52
Tiempo que tarda el model checker (en segundos)	NO	651.6	

Nombre: LDC2_25'

Tamaño de la fórmula LTL: 57.9Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	814	204	74.94
Tamaño del autómata (en Kb)	438.8	239.6	45.40
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_28 (Con default isHappening)

Tamaño de la fórmula LTL: 61.6 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	842	111	86.82
Tamaño del autómata (en Kb)	454.4	213.3	53.06
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.2.5.3.1.4 Análisis de las pruebas del caso de estudio

Al igual que en el caso de estudio anterior en las tres primeras pruebas la herramienta devuelve resultados. En la primera prueba se puede ver el alto grado de reducción del tiempo que tarda la herramienta en devolver resultados, pasa de 7 minutos 46 segundos a menos de un segundo. Nuevamente en la segunda y tercera prueba se obtienen resultados como consecuencia de la reducción cuando antes de esta no se podía. En este conjunto de pruebas se mantiene el alto grado de reducción de acciones a 2 estados, con un piso del 74%, llegando a una efectividad ideal del 100% en las primeras dos pruebas.

En la última prueba también se puede ver una reducción en el tamaño de la fórmula LTL en comparación con la misma prueba en el filtrado, pasando de 66.4 Kb a 61.6 Kb. En este caso esa reducción es muy importante ya que permite correr un caso que antes no se podía porque el tamaño de la fórmula LTL excedía el límite permitido por el model checker.

2.2.5.3.1.5 Caso de estudio 3: Ley de Defensa del Consumidor - segunda parte

Nombre: LDC3_1'

Tamaño de la fórmula LTL: 2.4 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	46	0	100
Tamaño del autómata (en Kb)	23.8	7.8	67.23
Tiempo que tarda el model checker (en segundos)	1.2	0.1	91.67

Nombre: LDC3_5'

Tamaño de la fórmula LTL: 15.1 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	162	46	71.60
Tamaño del autómata (en Kb)	101.1	58.1	42.53
Tiempo que tarda el model checker (en segundos)	NO	6	

Nombre: LDC3_15'

Tamaño de la fórmula LTL: 33.8 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	200	81	59.50
Tamaño del autómata (en Kb)	124.9	80.8	35.31
Tiempo que tarda el model checker (en segundos)	NO	14.5	

Nombre: LDC3_25'

Tamaño de la fórmula LTL: 46 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	278	120	56.83
Tamaño del autómata (en Kb)	185.7	128.5	30.80
Tiempo que tarda el model checker (en segundos)	3629.3(1h 29.3s)	93	97.44

Nombre: LDC3_32 (Con default isHappening)

Tamaño de la fórmula LTL: 48.6 Kb

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	336	73	78.27
Tamaño del autómata (en Kb)	222.3	127.1	42.83
Tiempo que tarda el model checker (en segundos)	NO	61.5	

2.2.5.3.1.6 Análisis de las pruebas del caso de estudio

Lo más importante de las pruebas de este caso de estudio es que debido a la optimización todas las pruebas terminan. Lo que tambień es relevante es que las pruebas LDC3_5', LDC3_15' y LDC3_32' previo a la reducción no terminaban y ahora sí lo hacen.

Con respecto a LDC3_1' y LDC3_25' se puede ver la importante reducción en el tiempo que el model checker tarda en devolver resultados, sobre todo en la LDC3_25' que se reduce el tiempo de una hora a un minuto y medio.
Otra cosa que se puede decir con respecto a los tiempos es que todas las pruebas después de la reducción tardan menos de 2 minutos.

Por otra parte, en este conjunto de pruebas hay una disminución del grado de reducción de acciones a 2 estados, el piso es del 56.83% un poco alejado del piso del 74% de los casos de estudio anteriores. Al igual que en los casos de estudio anteriores en la última prueba se puede ver una reducción en el tamaño de la fórmula LTL, pasando de 50.1 Kb a 48.6 Kb.

Un dato interesante es que en la primer prueba se obtiene el autómata más chico de los casos de estudio de la Ley de Defensa del Consumidor, con un tamaño de 7.8 Kb.

Por último, si analizamos el resultado de las pruebas **LDC1_60, LDC2_28, LDC3_32** en las que se puso como valor por defecto `IS_HAPPENING`, para medir fácilmente el caso ideal en que todas las acciones se referencian con `isHappening` el pronóstico es alentador ya que el grado de reducción de acciones a dos estados está por arriba del 78%, teniendo un pico de casi el 87%, es decir hay un techo alto sobre el cual es posible avanzar. Una pregunta interesante es por qué no se llegó al 100% si todas las referencias a las acciones son mediante `isHappening`, la respuesta es que hay acciones que no se pudo reducir su representación a dos estados porque están referenciadas en las fórmulas mediante `results in` o por contadores o intervalos.

2.2.5.3.2 Casos de estudio sin filtrado

En esta sección se incluyen las pruebas del reductor sin el filtrado, con el objetivo de mostrar la mejora obtenida invocando únicamente al reductor. Es decir, es una forma de ver la utilidad de la optimización por sí sola.

Nombre: LDC1_5'

	Sin reductor	**Con reductor**	**Porcentaje de reducción**
# acciones con 3 estados	2870	333	88.40
Tamaño del autómata (en Kb)	2333	1428	38.79
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_60 (Con default isHappening)

	Sin reductor	**Con reductor**	**Porcentaje de reducción**
# acciones con 3 estados	2870	468	83.69
Tamaño del autómata (en Kb)	2333	1475	36.78
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_5'

	Sin reductor	**Con reductor**	**Porcentaje de reducción**
# acciones con 3 estados	1629	130	92.02
Tamaño del autómata (en Kb)	1269	762.9	39.88
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_28 (Con default isHappening)

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	1629	199	87.78
Tamaño del autómata (en Kb)	1269	784.5	38.18
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_1’

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	1367	203	85.15
Tamaño del autómata (en Kb)	1143	698.8	38.86
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_5’

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	1367	249	81.78
Tamaño del autómata (en Kb)	1143	718.1	37.17
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_32 (con default isHappening)

	Sin reductor	Con reductor	Porcentaje de reducción
# acciones con 3 estados	1367	269	80.32
Tamaño del autómata (en Kb)	1143	723.4	36.71
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.2.5.3.2.1 Análisis de las pruebas de los casos de estudio sin filtrado

En todas las pruebas se ve un alto porcentaje de reducción de acciones a 2 estados, con un piso de 80%. Otro aspecto interesante de las pruebas es que a pesar de tener un alto porcentaje de reducción de acciones a 2 estados, no tiene el mismo impacto en el porcentaje de reducción del tamaño del autómata el cual oscila entre el 36% y el 40%. Esto se debe principalmente a que la optimización tiene efecto sobre una porción de la teoría marco que son las acciones, y no en agentes, intervalos y contadores, que son responsables por la mayoría del archivo. Debido a esto los porcentajes de reducción del tamaño del autómata son bastante inferiores a los alcanzados en las pruebas del filtrado y esto se ve reflejado en que el piso del tamaño de los autómatas está cercano a los 700 Kb, muy alejado de los 24 Kb que se logró con la optimización del filtrado.

Por lo expuesto anteriormente esta optimización por sí misma no alcanza para que la herramienta retorne resultados en tiempo y forma, un ejemplo de esto es la prueba LDC3_1' donde a diferencia de la optimización del filtrado no se obtienen resultados.

2.3 Reemplazo del estado JUST_HAPPENED

2.3.1 Motivación

Con el fin de reducir el tamaño y la complejidad del autómata se propone la eliminación del estado **`JUST_HAPPENED`** en la representación de las acciones. Hasta antes de este cambio las acciones se representaban con tres estados, y en algunos casos particulares - cuando todas las referencias a la acción eran mediante `isHappening` y no tenía referencias mediante `results in` ni era referenciada por contadores e intervalos - se podía llevar a dos estados, prescindiendo del estadO **`JUST_HAPPENED`**.

Lo que se pretende es que el estado **`JUST_HAPPENED`** desaparezca completamente de la representación de las acciones en el autómata y también de la fórmula LTL que se le pasa al model checker.

2.3.2 Solución - Marco teórico

En la introducción se expuso que el objetivo es eliminar completamente el estado `JUST_HAPPENED` de la representación de una acción. Para esto debemos comprender qué representa el estado `JUST_HAPPENED` y posteriormente encontrar la forma de reemplazarlo con los dos estados restantes manteniendo el comportamiento de la herramienta.

Cuando una acción se encuentra en el estado `JUST_HAPPENED` significa que terminó de ocurrir. Por lo tanto si queremos prescindir de ese estado y representar una acción sólo con los estados `NOT_HAPPENING y HAPPENING`, una forma de hacerlo es cambiar en las fórmulas la referencia de `acción = JUST_HAPPENED` por siguiente: `acción = HAPPENING & next(acción) = NOT_HAPPENING`, es decir una acción termina de ocurrir cuando estaba ocurriendo y en el siguiente estado dejó de ocurrir. Lo que se está haciendo es reemplazar una referencia a un estado del autómata por una transición.

Por consiguiente, la solución consiste en reemplazar en la fórmula toda referencia a `acción = JUST_HAPPENED` por `acción = HAPPENING & X(acción) = NOT_HAPPENING` y que todas las acciones que componen la teoría marco sean representadas con dos estados. X es el operador LTL que sirve para referirse al próximo estado.

2.3.3 Solución - implementación

2.3.3.1 Arquitectura

La eliminación del estado `JUST_HAPPENED` se realiza en el Validador. La primera parte ocurre al momento de crear el autómata[13], ya que en ese punto se debe decidir qué template de Velocity usar para la representación de los componentes de la teoría marco. La segunda y última parte se realiza al traducir la fórmula FL a LTL. En ese momento se hace el reemplazo de `JUST_HAPPENED` por `acción = HAPPENING & X(acción) = NOT_HAPPENING`.

2.3.3.2 Modificación del template de Velocity

En el template se tuvieron que realizar varias modificaciones. La primera es en la definición de las variables que representan a las acciones. Ahora todas las acciones se representan con sólo dos estados (se prescinde del estado `JUST_HAPPENED`).

```
#foreach ($action in $acciones)
	$action: {HAPPENING, NOT_HAPPENING};
#end
```

La segunda modificación es en la definición de las transiciones de los estados de la acción: a partir de ahora no hay restricciones para cambiar entre los dos estados.

Debido a que el estado `JUST_HAPPENED` se utilizaba para saber si un intervalo comienza o termina -de acuerdo a si alguna de las acciones de inicio o de fin alcanzaba ese estado respectivamente-, si un contador debe actualizarse o si una acción con valores de salida debe cambiar su valor, se tuvieron que realizar las siguientes modificaciones:

- *Si el intervalo está inactivo y una de las acciones iniciales finaliza -pasa del estado `HAPPENING` al de `JUST_HAPPENED`- el intervalo se activa.*

```
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
```

se debe cambiar por:

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

[13] Se explica con más detalle en la sección "Decisiones de implementación".

- *Si el intervalo está activo y una de las acciones finales finaliza -pasa del estado `HAPPENING` al de `JUST_HAPPENED`- el intervalo pasa a inactivo.*

```
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
```

se debe cambiar por:

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

- *En cualquier otro caso el valor del intervalo se conserva durante la transición.*

```
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)
```

se debe cambiar por:

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

y

```
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)
```

se debe cambiar por:

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

- *Por cada acción que incrementa un contador*

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

se debe cambiar por:

```
(NOMBRE_DE_LA_ACCIÓN = HAPPENING & next(NOMBRE_DE_LA_ACCIÓN) = NOT_HAPPENING) -> next(CONTADOR) =
CONTADOR + x
```

Este cambio debe repetirse para las acciones que:

- decrementan el valor del contador
- reinician el valor del contador
- asignan un nuevo valor al contador

- *Además, tenemos que indicar que si ocurre una de las acciones que modifican al contador, entonces no puede ocurrir otra de ellas. Veamos primero un ejemplo que muestra cómo tienen que quedar:*

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_N1) != JUST_HAPPENED

El código que lo construye, para facilitar la construcción agrega un "TRUE &" luego de la implicación.

```
#foreach ($actJH in $contador.getAllActions() )
    (next($actJH) = JUST_HAPPENED -> TRUE
 #foreach ($actNotJH in $contador.getAllActions() )
  #if (!$actJH.equals($actNotJH))
    & next($actNotJH) !=  JUST_HAPPENED
  #end
 #end
    ) &
#end
```

se debe cambiar por:

```
#foreach ($actJH in $contador.getAllActions() )
       (($actJH = HAPPENING & next($actJH) = NOT_HAPPENING) -> TRUE
  #foreach ($actNotJH in $contador.getAllActions() )
       #if (!$actJH.equals($actNotJH))
           & !($actNotJH = HAPPENING & next($actNotJH) = NOT_HAPPENING)
       #end
  #end
    ) &
#end
```

- *Cuando una acción con valores de salida termina de ocurrir, debe tomar un nuevo valor*

```
#foreach ($action in $acciones )
 #if ($action.hasOutputValues())
  ##Si la action finalizó => el output_value toma un nuevo valor
       (next($action) = JUST_HAPPENED -> (
  #foreach ($ov in $action.getOutputValues() )
##Se hace un or con cada output_value
             next(${action}$sufijoOutputValue) = $ov |
  #end
```

se debe cambiar por:

```
#foreach ($action in $acciones )
 #if ($action.hasOutputValues())
  ##Si la action finalizó => el output_value toma un nuevo valor
       (($action = HAPPENING & next($action) = NOT_HAPPENING) -> (
  #foreach ($ov in $action.getOutputValues() )
##Se hace un or con cada output_value
             next(${action}$sufijoOutputValue) = $ov |
  #end
```

2.3.3.3 Traducción de la fórmula LTL

Para saber cómo traducir la fórmula LTL, más precisamente si se va a reemplazar a `JUST_HAPPENED` o no, se agrega un enum **`LTLTranslationType`** el cual tiene dos valores posibles:

- **`WITH_JH`**, se sigue usando `JUST_HAPPENED`.
- **`WITH_NEXT_FOR_JH`**, se reemplaza a `acción = JUST_HAPPENED` por `acción = HAPPENING & X(acción) = NOT_HAPPENING`.

En la clase **`FlAction`**, la cual representa una referencia en particular a una acción en una cláusula, se utiliza dicho enum para saber cómo traducir la fórmula a LTL.

```
@Override
    public String translateToLTL(LTLTranslationType anLTLTranslationType) {

             String actionNameWithAgent = getNameWithAgent();
             if (!LTLTranslationType.WITH_NEXT_FOR_JH.equals(anLTLTranslationType) ||
ActionReferencedState.IS_HAPPENING.equals(referencedState) )
                    return actionNameWithAgent + " = " +
this.getReferencedState().getValueInLtlFormula();

             // Se reemplaza el acción = JUST_HAPPENED por acción = HAPPENING & next(acción) =
NOT_HAPPENING
             return String.format("(%s = %s & X(%s) =  %s)", actionNameWithAgent,
LtlActionValue.HAPPENING.getValue(), actionNameWithAgent,
LtlActionValue.NOT_HAPPENING.getValue() );
    }
```

2.3.3.4 Decisiones de implementación

Como se describió más arriba había que hacer bastantes modificaciones al template de Velocity. Privilegiando el entendimiento y la comprensión del template, se decidió que para este objetivo haya un nuevo template en vez de tener un único template con un montón de condicionales esparcidos por todo el archivo.

Para hacer referencia a ese nuevo template se tuvo que agregar una nueva línea al archivo de configuración de la herramienta.

Otra decisión que se tuvo que tomar es quitarle la responsabilidad al método `toString()` de traducir un término FL a LTL y dársela al método `translateToLTL()`. Esto se hizo para que este método pudiera tomar el enum que indica cómo traducir un término LTL (con o sin `JUST_HAPPENED`).

2.3.3.5 Casos de estudio

En esta sección se incluyen las pruebas realizadas para cuantificar la mejora introducida.

Al igual que en el objetivo anterior las pruebas consistieron en medir y comparar la cantidad de acciones representadas con 3 estados, el tamaño del autómata y el tiempo que tarda el model checker en dar una respuesta, antes y después de realizar la eliminación del estado `JUST_HAPPENED`. Debido que la reducción varía de acuerdo a la cantidad de cláusulas que contiene la especificación se van a incluir diferentes pruebas variando la cantidad de cláusulas. El criterio que se adoptó para elegir el subconjunto de cláusulas que se incluyen en las pruebas, es el orden en que aparecen en la ley original.

A las pruebas se decidió dividirlas en dos partes. En la primera parte se van incluir ejemplos con fórmulas donde la cantidad de referencias al operador `justHappened` es mayor a 0. Con estos ejemplos lo que se busca es ver el impacto de la mejora sobre el autómata y la fórmula, ya que al haber referencias al operador `justHappened` esto implica que en la fórmula se debe reemplazar al estado `JUST_HAPPENED`. En la segunda parte se van a incluir ejemplos de fórmulas con cantidad de referencias al operador `justHappened` igual a 0, para medir únicamente el impacto de la mejora sobre el autómata, ya que las fórmulas antes y después de la mejora se mantienen sin cambios.

2.3.3.5.1 Casos de estudio con cantidad de referencias al operador justHappened mayor a 0

En esta primera parte se van incluir ejemplos con fórmulas con cantidad de referencias al operador `justHappened` mayor a 0 con el fin de medir el impacto de la mejora, que en este caso opera sobre el autómata y sobre la fórmula.

Debido a esto, adicionalmente para estas pruebas se van a contabilizar la cantidad de `JUST_HAPPENED` que hay en la fórmula LTL, el tamaño de la fórmula y la cantidad de '=' antes y después de la mejora.

Para organizar las pruebas primero se van a incluir ejemplos donde todas las referencias a las acciones en las fórmulas FL son mediante el operador `justHappened`. Debido a esto las pruebas tienen la particularidad de que la cantidad de `JUST_HAPPENED` en la fórmula LTL es muy similar a la cantidad de '='. Para ello vamos a reutilizar muchas de las pruebas que se incluyeron en el filtrado, las cuales se van a correr con el filtrado activado pero sin reductor (ya que no tendría nada que reducir al no haber referencias a acciones que estén ocurriendo).

Posteriormente se van a incluir ejemplos donde no todas las referencias a las acciones en las fórmulas FL son mediante el operador `justHappened`, es decir hay referencias al operador `isHappening`. Por consiguiente la cantidad de `JUST_HAPPENED` en la fórmula LTL es muy baja con respecto a la cantidad de '='. Para estos casos las pruebas se corrieron con el filtrado y el reductor activado.

2.3.3.5.1.1 Casos de estudio donde todas las referencias de las acciones en la fórmula son con justHappened

Nombre: LDC1_1

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	92	0	100
# de JH en la fórmula LTL	92	0	100
Tamaño de la fórmula LTL (en Kb)	4.1	8.1	-97.56
# de '=' en la fórmula LTL	92	184	-100
Tamaño del autómata (en Kb)	39.2	11.2	71.43
Tiempo que tarda el model checker (en segundos)	461.3	1.8	99.61

Nombre: LDC1_5

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	189	0	100
# de JH en la fórmula LTL	190	0	100
Tamaño de la fórmula LTL (en Kb)	9.6	18.5	-92.71
# de '=' en la fórmula LTL	190	380	-100
Tamaño del autómata (en Kb)	84.9	23.6	72.20
Tiempo que tarda el model checker (en segundos)	NO	12	

Nombre: LDC1_15

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	491	0	100
# de JH en la fórmula LTL	676	0	100
Tamaño de la fórmula LTL (en Kb)	34.2	64.8	-89.47
# de '=' en la fórmula LTL	701	1377	-96.43
Tamaño del autómata (en Kb)	269.8	112.2	58.41
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC1_25

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	815	0	100
# de JH en la fórmula LTL	1114	0	100
Tamaño de la fórmula LTL (en Kb)	65.4	118.3	-80.89
# de '=' en la fórmula LTL	1277	2391	-87.24
Tamaño del autómata (en Kb)	456.7	193.3	57.67
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_25

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	814	0	100
# de JH en la fórmula LTL	1114	0	100
Tamaño de la fórmula LTL (en Kb)	61.8	113.2	-83.17
# de '=' en la fórmula LTL	1254	2368	-88.84
Tamaño del autómata (en Kb)	438.8	175.3	60.05
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_1

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	46	0	100
# de JH en la fórmula LTL	46	0	100
Tamaño de la fórmula LTL (en Kb)	2.6	4.9	-88.46
# de '=' en la fórmula LTL	46	92	-100
Tamaño del autómata (en Kb)	23.8	7.2	69.75
Tiempo que tarda el model checker (en segundos)	1.5	0.4	73.33

Nombre: LDC3_5

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	162	0	100
# de JH en la fórmula LTL	230	0	100
Tamaño de la fórmula LTL (en Kb)	15.9	29.1	-83.02
# de '=' en la fórmula LTL	253	483	-90.91
Tamaño del autómata (en Kb)	101.1	40.5	59.94
Tiempo que tarda el model checker (en segundos)	NO	21	

Nombre: LDC3_15

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	200	0	100
# de JH en la fórmula LTL	263	0	100
Tamaño de la fórmula LTL (en Kb)	34.6	49.6	-43.35
# de '=' en la fórmula LTL	577	840	-45.58
Tamaño del autómata (en Kb)	124.9	51.5	58.77
Tiempo que tarda el model checker (en segundos)	NO	28.3	

Nombre: LDC3_25

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	278	0	100
# de JH en la fórmula LTL	314	0	100
Tamaño de la fórmula LTL (en Kb)	47	64.6	-37.45
# de '=' en la fórmula LTL	779	1093	-40.31
Tamaño del autómata (en Kb)	185.7	88.4	52.40
Tiempo que tarda el model checker (en segundos)	1131	140	87.62

Nombre: LDC3_32

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	336	0	100
# de JH en la fórmula LTL	365	0	100
Tamaño de la fórmula LTL (en Kb)	50.1	70.3	-40.32
# de '=' en la fórmula LTL	834	1199	-43.76
Tamaño del autómata (en Kb)	222.3	105.8	52.41
Tiempo que tarda el model checker (en segundos)	NO	NO	

2.3.3.5.1.2 Análisis de las pruebas de los casos de estudio

Antes de comenzar con el análisis se van a hacer unas aclaraciones con respecto a la no inclusión de algunos casos de estudio.

Debido a que las pruebas se corren con el filtro activado y como se había dicho en el análisis de las pruebas del filtrado que las pruebas LDC2_1, LDC2_5 y LDC2_15 con el filtrado son iguales a las LDC1_1, LDC1_5 y LDC1_15 respectivamente, se decidió no incluir a las pruebas LDC2_1, LDC2_5 y LDC2_15 para evitar repeticiones.

Otras pruebas que no se incluyeron son las LDC1_28 y LDC2_28, en este caso porque las pruebas LDC1_25 y LDC2_25 con la optimización hacían que se supere el largo de la fórmula LTL que puede soportar el model checker, por lo tanto no tenía sentido incluir dos pruebas más en las que ya sabemos que no van a poder correr por la limitación mencionada anteriormente.

Ésta es la primera optimización en la cual algunas métricas son negativas después de la optimización. En este caso hablamos de las métricas 'Tamaño de fórmula LTL' y 'cantidad de ='. Esto se debe a que en la optimización se reemplaza en la fórmula LTL a acción = JUST_HAPPENED por acción = HAPPENING & X(acción) = NOT_HAPPENING, es decir se está reemplazando un término por dos. Por consiguiente después de la optimización el tamaño de la fórmula LTL y la cantidad de '=' crece, y en el peor de los casos se duplica, por ejemplo en LDC1_1. Parte del análisis consiste en ver si esto pesa más que la reducción en el tamaño del autómata. Al momento del análisis se deben tener en cuenta dos cosas: la primera es que que la semántica del operador X es intrínsecamente más compleja para los model checkers (ver [11]) y la segunda es que en estas pruebas la cantidad extra de '=' equivale a la cantidad de operadores X incorporados por la optimización.

En la prueba LDC1_1 se ve una importante reducción en el tiempo que tarda la herramienta en devolver resultados, pasa de tardar más de 7 minutos a menos de 2 segundos.

Y más importante aún es el resultado de la prueba LDC1_5 que mediante la optimización hace que devuelva resultados (en 12 segundos) cuando antes no lo hacía.

En la prueba LDC1_15 empieza a pesar más el crecimiento de la cantidad de operadores X de la fórmula que la reducción en el tamaño del autómata. Por otra parte en las pruebas LDC1_25 y LDC2_25 el crecimiento del tamaño de la fórmula LTL impide correr las pruebas ya que los tamaños exceden al máximo soportado por el model checker.

En las pruebas LDC3_1 y LDC3_25 se puede ver una gran reducción en el tiempo que tarda la herramienta en devolver resultados, pero más importante son los resultados de las pruebas LDC3_5 y LDC3_15 las cuales terminan en menos de 30 segundos cuando antes de la optimización no terminaban.

Lamentablemente la prueba LDC3_32 no se puede correr porque el tamaño de la fórmula LTL (70.3 Kb) excede el límite soportado por el model checker.

Si analizamos los porcentajes de crecimiento en la cantidad de '=' (que en estas pruebas equivale al crecimiento del operador X) después de la optimización, se puede ver que en las pruebas LDC3_XX tuvo un piso del 40%, mientras que en las demás pruebas el piso fue del 87%. Esto explica en parte el mejor comportamiento de las pruebas LDC3_XX por sobre las demás pruebas.

Haciendo una análisis más general las pruebas parecen indicar que la optimización vale la pena, es decir, que se ahorra tiempo por más que crezca la cantidad de operadores X de la fórmula después de la optimización. Para comprobarlo mejor habría que encontrar una alternativa que permita superar la limitación de los 65 Kb del model cheker, ya que debido a esta limitación hay casos que no se pudieron ejecutar.

Mientras se comparaba la prueba LDC3_5 de esta sección con la LDC3_5 del filtrado se detectó una inconsistencia en el tiempo que tarda la herramienta en devolver resultados.

La prueba incluida en esta sección dice que sin esta última optimización (pero sí con el filtrado) la herramienta no devuelve resultados, en cambio la misma prueba incluida en la sección del filtrado dice que la herramienta devuelve resultados en poco más de una hora.

Analizando por qué pasa esto se descubrieron dos cosas:

1. El autómata y la fórmula LTL que se le pasa al model checker varían entre corridas de la aplicación. La variación consiste en el **orden** en el que aparecen los agentes, acciones, intervalos contadores en el autómata y en el **orden** de los términos de la fórmula. Esa diferencia se origina en el uso de conjuntos como estructuras de datos, cuyos recorridos varían entre corridas.
2. Esa variación en el orden hace que en algunos casos como en el LDC3_5 se produzca una drástica diferencia en el tiempo que tarda la herramienta en devolver resultados.

En la sección *"Conclusiones y trabajo futuro"* se incluyen algunas ideas para resolver este tema.

Una observación más sobre este tema: dada la inconsistencia presentada uno podría poner en duda la fiabilidad de todas las pruebas realizadas. Es importante aclarar que una vez detectada esta inconsistencia las pruebas se corrieron por segunda vez regenerando los autómatas y las fórmulas LTL. Los resultados en esta segunda prueba fueron muy similares, es decir las pruebas que en la primer corrida terminaban, en la segunda también. Sólo hubo algunas diferencias de tiempo insignificantes en ciertos casos. Y nuevamente se volvió a dar la inconsistencia en el ejemplo LDC3_5, en una prueba terminó y en la otra no. La variedad de ejemplos analizados, sumado al hecho de que los resultados se mantuvieran en una segunda corrida nos brinda un buen nivel de confianza en que las mejoras que se atribuyen a las optimizaciones realmente lo sean. De todas formas, en la sección *"Conclusiones y trabajo futuro"* se proponen estrategias para aumentar este nivel de certeza.

Por último vamos a hacer un comentario sobre la métrica *"cantidad de '='"*. Viendo los resultados podría preguntarse por qué al comienzo de las pruebas se duplican -por ejemplo en LDC1_1- y después de a poco va descendiendo el porcentaje.

La duplicación se debe a que en la fórmula todas las referencias son a acciones sin output values, es decir la optimización lo que hace es reemplazar a `acción = JUST_HAPPENED` por `acción = HAPPENING & X(acción) = NOT_HAPPENING`, es decir se reemplaza un *'='* por dos. En cambio en los ejemplos en los que el aumento de la cantidad de '=' después de la optimización es menor al 100%, se debe a que en la fórmula hay referencias a acciones con output values. Por ejemplo, dada la siguiente fórmula LTL antes de la optimización, donde `accion2` es una acción con output values

```
G ( accion1 = JUST_HAPPENED &!( accion3 = JUST_HAPPENED -> accion2_OUTPUT = unValorDeSalida ) )
```

después de la optimización la fórmula queda así:

```
G ( (accion1 = HAPPENING & X(accion1) =  NOT_HAPPENING) & !((accion3 = HAPPENING &X(accion3) =
NOT_HAPPENING) -> accion2_OUTPUT = unValorDeSalida ) )
```

En la fórmula antes de la optimización hay tres '=' mientras que en la fórmula después de la optimización hay cinco, eso se debe a que `accion1` y `accion3` son acciones sin output values por lo tanto la cantidad de '=' después de la optimización se duplica, mientras que en `accion2` que es una acción sin output values no. En este caso por ejemplo el porcentaje de aumento de la cantidad de '=' es de un 66.67%.

2.3.3.5.1.3 Casos de estudio donde no todas las referencias de las acciones en la fórmula son con justHappened

Nombre: LDC1_25'

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	204	0	100
# de JH en la fórmula LTL	93	0	100
Tamaño de la fórmula LTL (en Kb)	61.3	65.3	-6.53
# de '=' en la fórmula LTL	1277	1370	-7.28
Tamaño del autómata (en Kb)	254.6	193.3	24.08
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC2_25’

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	204	0	100
# de JH en la fórmula LTL	139	0	100
Tamaño de la fórmula LTL (en Kb)	57.9	64.4	-11.23
# de ‘=’ en la fórmula LTL	1254	1393	-11.08
Tamaño del autómata (en Kb)	239.6	175.3	26.84
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_5’

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	46	0	100
# de JH en la fórmula LTL	23	0	100
Tamaño de la fórmula LTL (en Kb)	15.1	16.4	-8.61
# de ‘=’ en la fórmula LTL	253	276	-9.09
Tamaño del autómata (en Kb)	58.1	40.5	30.29
Tiempo que tarda el model checker (en segundos)	5.8	4.1	29.31

Nombre: LDC3_15’

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	81	0	100
# de JH en la fórmula LTL	47	0	100
Tamaño de la fórmula LTL (en Kb)	33.8	36.4	-7.69
# de ‘=’ en la fórmula LTL	577	624	-8.15
Tamaño del autómata (en Kb)	80.8	51.5	36.26
Tiempo que tarda el model checker (en segundos)	12.4	9.7	21.77

Nombre: LDC3_25'

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	120	0	100
# de JH en la fórmula LTL	56	0	100
Tamaño de la fórmula LTL (en Kb)	46	49.1	-6.74
# de '=' en la fórmula LTL	779	835	-7.19
Tamaño del autómata (en Kb)	128.5	88.4	31.21
Tiempo que tarda el model checker (en segundos)	178.2	28.4	84.06

2.3.3.5.1.4 Análisis de las pruebas de los casos de estudio

Antes de comenzar con el análisis se van a hacer unas aclaraciones con respecto a la no inclusión de algunas pruebas. Las pruebas LDC1_1', LDC1_5', LDC2_1', LDC1_5' y LDC3_1' no se incluyeron debido a que con el reductor (optimización anterior) todas las acciones del autómata ya se representan con dos estados, es decir no hay espacio para la mejora ni para empeorar (porque tampoco se agregarían operadores X en la fórmula).

Como se puede ver en las pruebas el crecimiento del tamaño de la fórmula LTL es bastante menor a los de las pruebas de la sección anterior, esto se debe a que las referencias a las acciones de la fórmula no son todas al estado `JUST_HAPPENED`. Es decir, hay referencias al estado `IS_HAPPENING` y por lo tanto en este último caso no hay que hacer el reemplazo de un término por dos. En estas pruebas en el peor de los casos no se llega al 12% en contraposición al 97% de las pruebas anteriores. Esta característica favorece a que las pruebas LDC3_5', LDC3_15' y LDC3_25' reduzcan los tiempos, sobre todo la LDC3_25' que pasa de casi 3 minutos a menos de medio minuto. Por otra parte en las pruebas LDC1_25' y LDC2_25' la optimización no alcanza para que la herramienta devuelva resultados.

2.3.3.5.2 Casos de estudio con cantidad de referencias al operador justHappened igual a 0

En esta sección se incluyen las pruebas de los casos en los que la fórmula que se le pasa al model checker no contiene referencias al estado `JUST_HAPPENED`, por lo tanto la fórmula antes y después de la optimización es la misma (lo que cambia es el autómata).

Nombre: LDC1_15'

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	65	0	100
Tamaño del autómata (en Kb)	133.5	112.2	15.96
Tiempo que tarda el model checker (en segundos)	619.8	71.6	88.45

Nombre: LDC2_28 (con default isHappening)

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	111	0	100
Tamaño del autómata (en Kb)	213.3	179.2	15.99
Tiempo que tarda el model checker (en segundos)	NO	NO	

Nombre: LDC3_32 (con default isHappening)

	Con JH	Sin JH	Porcentaje de reducción
# acciones con 3 estados	73	0	100
Tamaño del autómata (en Kb)	127.1	105.8	16.76
Tiempo que tarda el model checker (en segundos)	97	52	46.39

2.3.3.5.2.1 Análisis de las pruebas de los casos de estudio

El objetivo de estas pruebas era ver la influencia que tiene la reducción en la representación de las acciones de 3 a 2 estados en el tiempo que tarda la herramienta en devolver resultados, ya que como se dijo previamente la optimización no influye en el tamaño de la fórmula. Los resultados obtenidos son alentadores, en las pruebas LDC1_15' y LDC3_32 la reducción del tiempo es alrededor del 88% y 46% respectivamente. En cambio en la prueba LDC2_28 no alcanza para ver mejoras en el tiempo, probablemente lo que haya influido en esta prueba es el tamaño del autómata que es el más grande de los tres, al igual que el tamaño de la fórmula LTL que es de 61.6 Kb contra 31.5 Kb y 48.6 Kb.

3 Conclusiones y trabajo futuro

Se han presentado e implementado tres propuestas de optimización sobre la herramienta. A su vez con la inclusión de tres casos de estudios reales se pudieron medir las mejoras alcanzadas mediante estas tres optimizaciones.

Haciendo un resumen de los resultados de las pruebas incluidas, podemos decir que por medio de las tres optimizaciones se logra que la herramienta dé resultados en 8 de las 15 primeras pruebas (las que no tienen referencias al operador `isHappening`) y en 11 de las 15 segundas pruebas (las que sí tienen referencias al operador `isHappening`). Es decir de 30 pruebas que no terminaban sin las optimizaciones, mediante éstas se logra que terminen 19. Un dato más a agregar es que el máximo tamaño de autómata y de fórmula LTL en la que la herramienta devuelve resultados es 185 Kb y 47 Kb respectivamente.

Por consiguiente, creemos haber logrado una mejora en la performance de la herramienta, aunque somos conscientes que quedan cosas por mejorar. En resumen se podría decir que el trabajo consistió en hacer optimizaciones “gruesas”, quedando para más adelante las optimizaciones “finas”.

Con respecto a las optimizaciones “finas”, una alternativa a explorar sería la siguiente: tomar de cada caso de estudio la prueba con más cláusulas que haya terminado e ir agregando cláusulas de a una para analizar hasta qué punto se puede correr.

Otra técnica más para investigar sería la de automatizar la división de un conjunto de normas en subconjuntos independientes, lo cual fue mencionado en la sección “2.1.4.2.3.1 Análisis de las pruebas del caso de estudio”. En este caso vale la aclaración realizada en la sección “2.1.4.2.2 Caso de estudio 2: Ley de Defensa del Consumidor - primera parte” de que hay que asegurarse de que al dividir las partes sean efectivamente independientes.

En la sección “2.1.4.2.3.1 Análisis de las pruebas del caso de estudio” se hace referencia a que hay dos pruebas para las cuales no se encontró un comportamiento legal válido. Hoy en día no es posible aún determinar cuáles son las fórmulas que generan que eso suceda. Un alternativa a explorar sería usar enfoques iterativos al estilo CEGAR (Counterexample-Guided Abstraction Refinement) [10] para ir refinando el modelo en busca de las fórmulas problemáticas.

Durante la realización de las pruebas se encontró una limitación del model checker con respecto al tamaño máximo de la fórmula LTL que puede soportar (alrededor de 65 Kb). Algunas acciones que se podrían realizar para tratar de evitar esta limitación son las siguientes:

1. Reemplazar el término `HAPPENING` por 1 y el término `NOT_HAPPENING` por 0, por cada referencia a cada uno de estos términos se ahorrarían 8 y 12 bytes respectivamente. Otra opción válida sería reemplazar `HAPPENING` por `H` y `NOT_HAPPENING` por `NH` obteniéndose un ahorro menor en el caso de `NOT_HAPPENING` pero siendo un poco más legible.
2. Hacer un proceso que traduzca los nombres originales (los que usa el usuario) de acciones, intervalos y contadores a una codificación eficiente en la cantidad de caracteres que se le pasa al model checker. Algunos ejemplos de nombres largos de acciones son: *“NoComunicarGastosAdicionalesAlPresupuesto”, “ofrecerPublicamenteCosasDeficientes” y “PrestarServicioDeReparacionyMantenimiento”*
3. Reducción en base a equivalencias lógica. Por ejemplo se podría aplicar la ley de la distribución para reducir el tamaño de los términos. En ese caso se puede reemplazar ***(p and q) or (p and r)*** por ***p and (q or r)***.

Otra cosa que se detectó mientras se hacían las pruebas (en este caso de la última optimización) es que dada una teoría marco y una fórmula se pueden generar varias representaciones de estas variando el orden de los elementos que la componen. Esto se debe a que en la herramienta se utilizan conjuntos de agentes, acciones, intervalos, contadores y fórmulas. En base a esos conjuntos se arma el autómata y la fórmula LTL que se le pasan al model checker. Una característica que tienen los conjuntos es que los elementos no tienen un orden, debido a esto en diferentes corridas de la herramienta se generan diferentes representaciones. Una solución para esto sería establecer un orden entre los elementos del conjunto, por ejemplo el orden lexicográfico, esto serviría para generar siempre una sola representación sin importar cuantas corridas se hagan.

Un problema que trae aparejado el tener distintas representaciones de una teoría marco y de una fórmula es que en algunos casos las diferencias en tiempos son significativas. Durante el desarrollo de la tesis se detectó un caso, el LDC3_5 que con una representación termina más o menos en una hora y con la otra no. La propuesta para este caso sería realizar pruebas asignándole diferentes órdenes a los elementos y analizar cuál de estos órdenes conviene de acuerdo al tiempo que tarda la herramienta en devolver resultados. Dado que el problema de elegir un buen ordenamiento no es algo trivial, es un tema para seguir explorando con mayor profundidad (ver el párrafo siguiente).

Algo que puede ayudar a mejorar la performance es detenerse a estudiar con más profundidad el model checker que se está utilizando actualmente (NuSMV) y tratar de sacarle el máximo provecho. Tiene bastantes parámetros de configuración y quizás alguno pueda servir para mejorar la performance. Por ejemplo durante la tesis se encontró que agregando el parámetro *-dynamic* la performance en tiempo mejoraba significativamente y además ayudaba a hacer un mejor uso de la memoria como se menciona en la FAQ de NuSMV, se pueden ir refinando órdenes de variables de manera iterativa, posibilidad que no se llegó a explorar en esta tesis. Otra cosa interesante para averiguar es si el model checker puede configurarse para usar múltiples procesadores, durante la tesis se observó que usa sólo un procesador el cual la mayoría del tiempo está al 100%. En caso de que no se pueda configurar esto, una alternativa sería que nuestra herramienta pueda brindar esa característica. Otra posibilidad sería probar con otro model checker.

Una funcionalidad deseable para agregar a la herramienta sería que se le pueda establecer un límite de tiempo en el cual la herramienta debe devolver resultados. Algo similar se podría hacer con poner un límite a la máxima cantidad de memoria a utilizar.

Con respecto a la optimización del filtrado, otra alternativa a explorar sería que se haga un pre-filtrado a nivel de roles, es decir eliminar todos los roles que no son referenciados en las fórmulas. Esto implicaría que el pre-filtrado se haga antes de la expansión de los componentes de la teoría marco. Lo cual tiene como ventaja evitar la explosión combinatoria que se puede generar al crear los agentes a partir de los roles. El peor escenario es cuando se tienen n roles sin el modificador *disjoint* por lo cual se generan 2^n agentes.

Es importante aclarar que el pre-filtrado no modificaría la información que se le envía al model checker. En todo caso lo que modifica es la entrada al filtrado, ya que al dejar únicamente los roles que son referenciados en las fórmulas la cantidad de agentes a filtrar va a ser menor pero con o sin pre-filtrado la salida del proceso de filtrado es la misma. Como se dijo, el único objetivo es reducir el tiempo y el espacio en la generación de agentes.

Si bien no está muy relacionado con el trabajo realizado en esta tesis, se menciona otra mejora, de expresividad en este caso. Al comienzo de la tesis cuando me quería familiarizar con el lenguaje FL y las posibilidades que éste brindaba armé un ejemplo ficticio conteniendo una acción impersonal que “únicamente ocurre” en un intervalo local para ver cómo se comportaba la herramienta. La herramienta no estaba preparada para ese comportamiento y rompía con una excepción por null. En ese momento pensé si la herramienta debía soportar ese comportamiento y la verdad no encontraba ejemplos reales en que necesitara modelar esto. Entonces en aquel momento se decidieron dos cosas: la primera fue realizar una modificación al código para que manejara esta situación de una manera más prolija, es decir evitar que tire la excepción por null y la segunda fue dejar este tema para más adelante (con más conocimiento y más ejemplos vistos) y ver si valía la pena que la herramienta soporte este comportamiento. Al finalizar la tesis el ejemplo que se me ocurrió es el siguiente:

```
action ComienzaGarantia
action FinalizaGarantia
local interval EnGarantia defined by actions ComienzaGarantia - FinalizaGarantia
impersonal action paso_un_dia_reparandose only occurs in scope EnGarantia
```

En el ejemplo se pretende modelar el concepto de tener un producto en garantía, mediante el ejemplo se puede ver que la acción impersonal `paso_un_dia_reparandose` *sólo ocurre cuando se está en el intervalo de* `EnGarantía`. Por esta razón sería importante que la herramienta soporte este comportamiento.

4 Referencias

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

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

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

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

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

[6] María Celeste Gunski and Melisa Gabriela Raiczyk. FormaLex - Mejorando el poder expresivo del lenguaje FL para la detección de defectos normativos. Tesis de Licenciatura en Ciencias de la Computación, Marzo de 2016

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

[8] Cristian Prisacariu & Gerardo Schneider (2009): C L : An Action-Based Logic for Reasoning about Contracts. In: WoLLIC ’09: Proceedings of the 16th International Workshop on Logic, Language, Information and Computation. Springer-Verlag, Berlin, Heidelberg, pp. 335–349. Available at http://dx.doi.org/10.1007/978-3-642-02261-6_27.

[9] Sznur, S., Ruffa, M. B., Martínez, M., Brun, J. M., & Schapachnik, F. (2014). Análisis de Consistencia de la Legislación de Defensa del Consumidor mediante métodos formales. Disponible en http://ciiddi.org/congreso2014/images/documentos/anlisis%20de%20consistencia%20de%20la%20legislacin%20de%20defensa%20del%20consumidor%20mediante%20mtodos%20formales%20sznur.pdf

[10] D’Argenio, Pedro R., et al. "Reachability analysis of probabilistic systems by successive refinements." Process Algebra and Probabilistic Methods. Performance Modelling and Verification. Springer Berlin Heidelberg, 2001. 39-56.

[11] Kucera, Antonín, and Jan Strejček. "The Stuttering Principle Revisited: On the Expressiveness of Nested X and⋃ Operators in the Logic LTL." Computer Science Logic. Springer Berlin Heidelberg, 2002.

[12] Cimatti, Alessandro, et al. "Nusmv 2: An opensource tool for symbolic model checking." Computer Aided Verification. Springer Berlin Heidelberg, 2002.

[13] Baier, Christel, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.