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

# Recuperando 'fórmulas culpables' de un lenguaje deóntico mediante el mapeo inverso de unsat cores

Tesis de Licenciatura en Ciencias de la Computación

Damián Ignacio Mazzini

Director: Fernando Schapachnik
Buenos Aires,

# Abstract

FormaLex es un conjunto de herramientas que tiene como objetivo modelar y analizar sistemas legales realizando de manera automatizada verificaciones de fórmulas lógicas que representan textos normativos. Como resultado de la ejecución, FormaLex determina si un documento normativo es consistente o si por el contrario se hallaron inconsistencias.

En este proceso de verificación intervienen distintas herramientas, entre ellas se encuentra NuSMV que es un model checker utilizado como motor de razonamiento.

Este trabajo se centra principalmente en la interacción entre FormaLex y NuSMV, tanto en la optimización de los archivos que sirven como input de NuSMV como en la interpretación del output del mismo.

Con respecto a la primera tarea el problema consiste en que los archivos de autómatas que se generan y que sirven de input de NuSMV pueden llegar a ser gran volumen. Esto no solo impacta en la performance de la ejecución sino que también se identificó una limitación en NuSMV que no permite inputs que superen un tamaño específico. La propuesta para abordar esta problemática se centra en realizar una reducción del tamaño de estos archivos de manera que permita a una mayor cantidad de casos ser admitidos por NuSMV.

En el caso de la segunda tarea, el problema está relacionado con que actualmente el model checker al momento de responder que existe una contradicción no da indicios de cuáles son las fórmulas involucradas en la misma. En un trabajo previo llamado *Recuperación de “fórmulas culpables” mediante análisis de unsat core* de Francisco Giménez se realizaron modificaciones a las herramientas que intervienen en el proceso de verificación de modelos, el model checker NuSMV y su interacción con el SAT solver Picosat. Dichas modificaciones permitieron formar un vínculo directo entre las variables del unsat core y las variables del model checker. Esta información brindada resulta valiosa para entender en dónde residen las inconsistencias, por lo que la propuesta de este trabajo en este aspecto consiste en tomar la información resultante de la extracción de unsat core y procesarla para identificar las variables de alto nivel involucradas en las fórmulas que hacen que un modelo legal no sea satisfacible.

**Palabras claves:** FormaLex, NuSMV, Picosat, unsat core

# Índice general

**Índice general** 1

**Introducción** 2

La herramienta FormaLex 2
El lenguaje FL
Model Checking y NuSMV 5
Problema de Satisfacibilidad Booleana (SAT) y Picosat 8
El alcance de esta tesis 10

**Traducción de variables de fórmulas** 12

**Compactación de inputs** 23

**Caso de estudio** 30

**Conclusiones** 31

**Bibliografía** 33

# 1. Introducción

## 1.1. La herramienta FormaLex

Las **normas** rigen el comportamiento de la sociedad y sus individuos mediante la formulación de obligaciones, prohibiciones y permisos. Se formulan en leyes, decretos, pero también resoluciones, disposiciones, acuerdos. En definitiva, en textos normativos. Estos textos pueden tener problemas difíciles de identificar como contradicciones, inconsistencias, huecos, ambigüedades, etc.

Por su parte, en el mundo del desarrollo de software una **especificación** define qué tiene que hacer un programa y al igual que en los textos normativos se indican los comportamientos permitidos y los prohibidos. Otra característica en común que tienen es que a ambos se les pide que sean coherentes, es decir que no tengan contradicciones y que contemplen todos los casos.

Aprovechando que la Ingeniería del Software ha trabajado desde hace tiempo en técnicas de detección de errores en las especificaciones se intenta trazar un paralelismo con los textos normativos de manera que se puedan utilizar las mismas herramientas para identificar problemas en las normas.

Existe un área conocida como *Automated Legislative Drafting* que tiene como objetivo brindar asistencia automatizada a legisladores[1] detectando inconsistencias en el momento de la elaboración de leyes, normativas o reglamentaciones. En este contexto se presenta **FormaLex** [1] que es el resultado de un amplio trabajo que viene atacando este problema basándose en un conjunto de herramientas que fueron creadas para dicho propósito.

Como parte de ese trabajo se desarrolló **FL** [1] que es un **lenguaje deóntico** orientado a expresar las estructuras de razonamiento y representación más frecuentes del ámbito legal, de manera sencilla e intentando ser lo más completo posible. Este lenguaje también hereda características de lógicas temporales modales (en particular de la **Linear Temporal Logic, LTL** [5] o [10]) lo que le permite expresar nociones de **temporalidad**.

La lógica deóntica es un tipo de lógica modal que se ocupa de conceptos normativos (permiso, prohibición, obligación, etc.). Los lenguajes basados en este tipo de lógicas, como es el caso de FL, contienen operadores deónticos que se utilizan para expresar dichos conceptos dentro de las fórmulas.

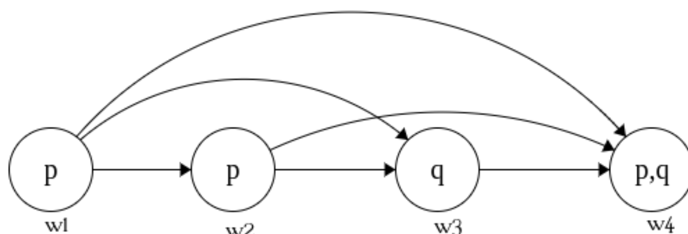
[1] No se limita solamente a representantes en el Congreso. Se refiere a toda persona de cualquier tipo de institución que escribe un texto normativo.

Entre los operadores deónticos más utilizados podemos nombrar al de obligación ***O(φ)*** que significa que la fórmula φ **se cumple siempre**, es decir, en todos los estados de todos los modelos. Por ejemplo si queremos expresar que es obligatorio votar lo haremos de la siguiente manera: *O(votar).* Por otro lado el operador de prohibición ***F(φ)*** significa que la fórmula φ está **prohibida**. Por ejemplo, para expresar que está prohibido fumar, escribimos lo siguiente: *F(fumar).* El operador de prohibición se puede ver como la obligatoriedad de lo contrario, es decir *F(φ) ≡ O(¬φ).* Por su parte el operador de permiso ***P(φ)*** sirve para representar que algo está **permitido**. Entonces para expresar que está permitido hablar lo haremos de la siguiente manera: *P(hablar).* También podemos decir que algo está permitido si no está prohibido, es decir *P(φ) ≡¬F(φ).* Por último tenemos las obligaciones **CTD** (Contrary-To-Duty Obligation) que se denotan como ***O(φ) repaired by*** ρ que significa que *φ* es obligatorio, pero si no se cumple, entonces es obligatorio ρ*.*

Por otro lado, la **lógica temporal lineal** (LTL) sirve para poder expresar cómo varían los valores de verdad de las proposiciones a lo largo del tiempo. Es una lógica modal que se interpreta sobre modelos de Kripke con las siguientes características:

- Los mundos representan instantes en el tiempo.
- Existe una única relación notada → que es la de pasaje de una cantidad indeterminada de tiempo. Es decir, si w1 → w2 significa que w2 es un instante posterior a w1.
- Además los modelos son irreflexivos, antisimétricos y transitivos.

Por eso se dice que los modelos de LTL son **lineales**.




Las fórmulas de LTL son:

- Las proposicionales.
- ♢***φ*** que se interpreta como que **existe un instante** posterior en donde vale ***φ***.
- □ ***φ*** que se interpreta como que en **todo instante** posterior vale ***φ***.
- ***φ* U ψ** que se interpreta como que ***φ*** vale hasta que empieza a valer **ψ**.

También se permiten hacer combinaciones de las proposiciones anteriores.

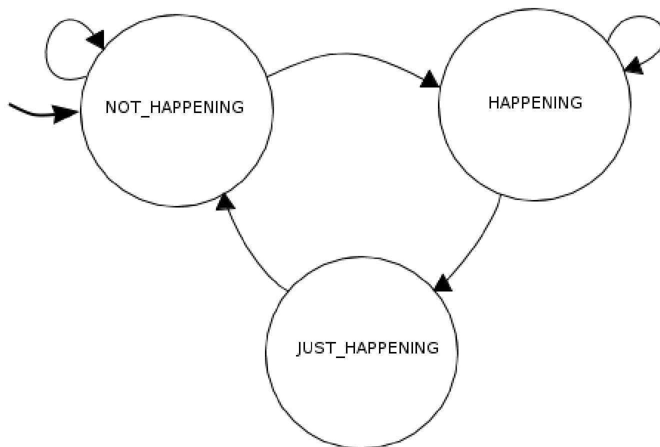
## 1.2. El Lenguaje FL

FL tiene como objetivo encontrar problemas de coherencia en el contexto de documentos normativos. La manera en que lo realiza es siguiendo una serie de premisas como por

ejemplo que un determinado comportamiento no puede ser permitido u obligatorio y, a la vez, estar prohibido para un mismo individuo.

Gracias a FL se pueden expresar los dos componentes necesarios para este proceso de verificación: un **conjunto de reglas** que son fórmulas con operadores deónticos y una **teoría marco** o *background theory* que provee mecanismos sencillos para describir la clase de modelos sobre los que predican las reglas. Los modelos son lineales y cada uno describe un comportamiento legal posible. En base a esto los comportamientos que no cumplen con las reglas normativas son descartados. Si algo es obligatorio, entonces debe mantenerse en todos los modelos legales, en todos los estados posibles y por lo tanto $O(\varphi)$ es interpretado como la fórmula LTL $\Box\varphi$. La prohibición de algo es la obligación de lo contrario $F(\varphi) \equiv O(\neg\varphi)$. Para el caso de obligaciones CTD se escriben como $O_\rho(\varphi)$ y se traducen como $\Box(\neg \varphi \rightarrow \rho)$. Por su parte $F_\rho(\varphi)$ es interpretado como $O_\rho(\neg \varphi)$. Un permiso se interpreta como ausencia de prohibición pero no se trata como un operador que modifica el conjunto de comportamientos legales, sino como un predicado que los modelos legales deben satisfacer.

Analizando los componentes que se presentan dentro de la teoría marco vemos que se definen **acciones**. Estas son el elemento principal de la teoría marco ya que permiten representar eventos. Estos eventos pueden ser realizados de forma impersonal (no es necesario que alguien específico esté realizando dicha acción, por ejemplo: llover) o también pueden representar acciones como *comprar* o *vender*. Este tipo de acciones deben ser llevadas a cabo por **agentes**, que son entidades que representan a las personas físicas o jurídicas que realizan las acciones. Los agentes no cuentan con una declaración explícita en la teoría marco, sino que son generados por el sistema a partir de los roles que definimos. Las acciones se representan con una variable enumerada, que puede tener los valores ***NOT_HAPPENING*** que significa que la acción no está ocurriendo, ***HAPPENING*** que indica que la acción ha empezado a suceder o ***JUST_HAPPENED*** que significa que la acción ha terminado de suceder. Estas variables que representan acciones se inicializan con el valor *NOT_HAPPENING* y puede mantener ese valor o puede pasar al estado *HAPPENING*. Una vez que se encuentra en este estado puede permanecer con ese valor o pasar al estado *JUST_HAPPENED.* De este estado debe pasar obligatoriamente al estado NOT_HAPPENING en la siguiente transición. Este ciclo se puede repetir indefinidamente y a continuación se muestra el diagrama de transiciones permitidas:




Los **roles** son formas de clasificar a los agentes en nuestro modelo. Por ejemplo, si queremos definir los roles *comprador* y *vendedor* lo podemos hacer de la siguiente manera:

***roles*** *vendedor, comprador*

Este esquema de agentes y roles permite definir **quiénes pueden ejecutar determinadas acciones**. Esto se puede declarar mediante la utilización del modificador *only performable by.* Por ejemplo podemos decir que las acciones *comprar* y *vender* solo las pueden realizar los roles *comprador* y *vendedor* respectivamente de la siguiente manera:

***action*** *comprar* ***only performable by*** *comprador*
***action*** *vender* ***only performable by*** *vendedor*

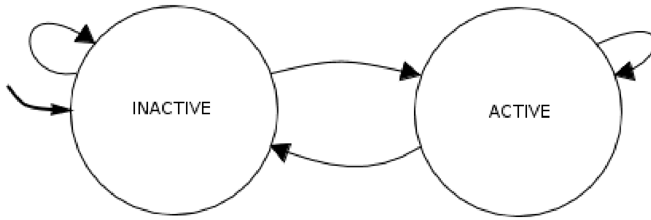
De esta manera decimos que la acción de *comprar* es solo realizable por el rol *comprador* mientras que la acción de *vender* es solo realizable por el rol *vendedor*.

En algunas circunstancias se requiere hablar de obligaciones, prohibiciones o permisos dependiendo de si sucedió alguna acción previamente, como por ejemplo para determinar si una persona tiene permitido conducir es necesario evaluar si esa persona obtuvo la licencia anteriormente. Además es necesario verificar si esa persona no tiene la licencia revocada. Para este tipo de situaciones FL incorpora la noción de **intervalo** que sirve para expresar un período temporal delimitado por una acción de inicio y una de fin. En este contexto de licencias de conducir, el intervalo se puede definir de la siguiente manera:

***interval*** *licensed* ***delimited by actions*** *GetLicense-LoseLicense*

Los intervalos también se representan con una variable enumerada que puede tener los valores ***ACTIVE*** o ***INACTIVE.*** Esta variable se inicializa con el valor *INACTIVE* y conserva ese valor hasta que ocurra la acción de inicialización (en el ejemplo *GetLicense*) cuando

toma el valor *ACTIVE*. Este valor continuará hasta que ocurra la acción de finalización (en el ejemplo *LoseLicense*) y en ese caso el valor de la variable volverá a ser *INACTIVE.*




Por último se presentan los **contadores** que son variables de tipo entero que se incrementan o decrementan cada vez que se ejecuta alguna acción relacionada. Por ejemplo si tenemos un contador de la cantidad de libros prestados *bbc* que se incrementa en uno con la acción *BorrowBook* y se decrementa con la acción *ReturnBook* se puede definir de la siguiente manera:

***counter*** *bbc* ***increases with action*** *BorrowBook* ***decreases with action*** *ReturnBook*

Por el lado de las fórmulas que componen el **conjunto de reglas** la **sintaxis de FL** se define de la siguiente manera:

Sea PROPS un conjunto infinito contable de símbolos, INTERVALS $\subseteq$ (PROPS x PROPS) un conjunto de intervalos y FORMS el conjunto de fórmulas FL con <PROPS , INTERVALS> se define:

INNER_FORMS ::= $\top$ | $\bot$ | p | $\neg\varphi$ | $\varphi_1 \wedge \varphi_2$ | $\Diamond\varphi$ | $\Diamond_i \varphi$ | INSIDE(i)
FORMS ::= O($\varphi$) | F($\varphi$) | $F_\rho$ ($\varphi$) | $O_\rho$ ($\varphi$) | $O^E$($\varphi$) | P($\varphi$)

donde

p $\in$ PROPS
$\varphi, \rho, \varphi_1, \varphi_2 \in$ INNER_FORMS
i $\in$ INTERVALS

Generalmente el conjunto de fórmulas FORMS que define el sistema normativo bajo análisis (NSUA) es finito por lo que no necesita ser definido formalmente. En la especificación se suele escribir una fórmula debajo de la otra, lo que significa la conjunción entre todas ellas.

Mostraremos la **semántica de FL** mediante una función ***Tr*** que realiza la traducción de las fórmulas de FL a LTL. Sea $\mathcal{F}$ un conjunto de fórmulas FL que definen el conjunto de modelos legales para el NSUA. Los modelos LTL son estructuras lineales infinitas que representan los caminos posibles del autómata definido por la teoría marco. En primer lugar excluimos los permisos y definimos el dominio de ***Tr*** como $\mathcal{F}_{np}$ = { $\varphi$ | $\varphi \in \mathcal{F}$ tal que $\varphi$ no es de la forma P($\psi$) }. ***Tr*** actúa como la identidad para las construcciones INNER_FORMS que no están especificadas explícitamente.

$Tr( \Diamond_i \varphi ) = \quad i = ACTIVE \rightarrow ( i = ACTIVE \; U \; Tr( \varphi) )$
$Tr( F(\varphi) ) = \quad \Box \neg Tr( \varphi )$
$Tr( F_\rho(\varphi) ) = \quad \Box ( Tr( \varphi ) \rightarrow Tr( \rho ) )$
$Tr( O(\varphi) ) = \quad \Box \; Tr( \varphi )$
$Tr( O_\rho(\varphi) ) = \quad \Box ( \neg Tr( \varphi ) \rightarrow Tr( \rho ) )$
$Tr( O^E(\varphi) ) = \quad \Diamond \; Tr( \varphi )$

Dado el autómata A definido por un la teoría marco y la clase de modelos $\mathcal{C}_A$ que representa todas las corridas posibles sobre A. Sea $\mathcal{F}$ el conjunto de fórmulas FL que representan el NSUA. La clase de modelos legales definido por $\mathcal{F}$ sobre $\mathcal{C}_A$ se define como:

$$\mathcal{C}_A^{\mathcal{F}} = \{ \mathcal{M} \in \mathcal{C}_A \mid \mathcal{M} \vDash Tr( \mathcal{F}_{np} ) \}$$

Es decir, todo modelo legal debe satisfacer las obligaciones y prohibiciones especificadas por $\mathcal{F}$. Para el caso de los permisos son una verificación que debe ser realizada sobre $\mathcal{C}_A^{\mathcal{F}}$ para asegurar la coherencia. La condición que debe cumplir $\mathcal{C}_A^{\mathcal{F}}$ es la siguiente: para cada $\varphi$ de forma $P(\psi)$ en $\mathcal{F}$ debe ser un modelo $\mathcal{M}$ en $\mathcal{C}_A^{\mathcal{F}}$ de modo que $\mathcal{M} \vDash Tr(\psi)$. Es decir, si algo está permitido entonces el resto del NSUA no impide que suceda.

A continuación mostraremos un ejemplo que servirá para presentar los elementos con los que contamos para describir los escenarios que se presenten. El ejemplo se sitúa en el contexto de la compra-venta de automóviles e intenta describir las condiciones del contrato entre la fábrica y un concesionario. El contrato define las siguientes reglas:
1) Establece un máximo de diez unidades que el comprador puede adquirir al mes.
2) Estipula un mínimo de 5 unidades al mes.
3) La fábrica cuenta con transportes que llevan de a 5 autos, y sólo permite ventas mensuales que los llenen.

El modelo descripto puede ser expresado en FL de la siguiente manera:

**Archivo del modelo expresado en FL**:

```
#Background

roles vendedor, comprador

action comprar only performable by comprador
action vender only performable by vendedor

global counter unidades_compradas_al_mes
increases with action comprar
global counter ventas_realizadas
```

```
increases with action vender

#Clauses
O(unidades_compradas_al_mes >= 5)
F(unidades_compradas_al_mes >= 10)
O(<> unidades_compradas_al_mes > 5 -> <> unidades_compradas_al_mes == 10)
```

Como se muestra en el archivo la teoría marco se encuentra en la sección ***#Background*** en donde se definen dos roles (*comprador* y *vendedor)*, dos acciones (*comprar* y *vender)* y dos contadores (*unidades_compradas_al_mes* y *ventas_realizadas).*

En la sección ***#Clauses*** se define el conjunto de reglas que el usuario quiere validar

- *O(unidades_compradas_al_mes >= 5)* significa que la variable *unidades_compradas_al_mes* tiene que eventualmente alcanzar el valor 5.
- *F(unidades_compradas_al_mes >= 10)* significa que está prohibido que la variable *unidades_compradas_al_mes* alcance valor 10 en cualquier momento.
- *O(<> unidades_compradas_al_mes > 5 -> <> unidades_compradas_al_mes == 10)* significa que en caso de que la variable *unidades_compradas_al_mes* supere el valor 5 entonces en algún momento debe tener el valor 10.

Ya presentados y analizados ambos componentes del modelo FL pasaremos a describir la herramienta involucrada en el siguiente paso del proceso de verificación: el **model checker**. En el caso de este trabajo el model checker utilizado es **NuSMV** [2] por lo que a continuación mostraremos sus características. Más adelante explicaremos cómo se traduce el modelo FL al lenguaje que entiende NuSMV y cómo encaja todo en el proceso de verificación que realiza *FormaLex.*

## 1.3. Model Checking y NuSMV

**Model checking** es una técnica de verificación que provee mecanismos algorítmicos para determinar si un modelo abstracto satisface una propiedad expresada como una fórmula en alguna lógica. Muchos model checkers son *temporales*, es decir, que utilizan una lógica temporal, como por ejemplo LTL. Además si la propiedad no es válida el método es capaz de identificar un contraejemplo que muestre el origen del problema. Mediante esta técnica se puede realizar de una manera exhaustiva la comparación entre la descripción de un sistema con su especificación formal y buscar errores sistemáticamente[2]. Estas técnicas utilizadas son muy sofisticadas pero a continuación intentaremos explicar la idea básica.
El input de un model checker LTL consta de un autómata *A* que describe el comportamiento de un sistema y una fórmula LTL $\varphi$ y el output es el siguiente:

[2] SMV utiliza bounded model checking lo que limita su capacidad de encontrar contraejemplos pero a su vez permite tiempos de uso más acotados. Si bien esto es una limitación, abordarla excede el marco de esta tesis pues se trata de una decisión arquitectónica de FormaLex.

- true, si $\forall \tau \in L(A)$, $\tau \Rightarrow \varphi$
- false, si $\tau \in L(A)$ *pero no se cumple* $\tau \Rightarrow \varphi$

Se basa en **autómatas de Büchi**, que son parecidos a los autómatas clásicos pero que aceptan palabras infinitas.

Un autómata de Büchi *A* es una tupla $< Q, \Sigma, \delta, q0, F >$donde

- Q es un conjunto finito de estados.
- $\Sigma$ es un conjunto finito de símbolos (el alfabeto de A).
- $\delta$: Q x $\Sigma \rightarrow$ Q es la función de transición.
- $q_0 \in Q$ es el estado inicial.
- F es un conjunto finito de estados llamado la condición de aceptación.

*L(A)* (el lenguaje de A) son todas aquellas secuencias de caracteres de $\Sigma$ que comenzando desde $q_0$ pasan infinitamente seguido por estados en F.

El objetivo es verificar si todas las palabras de *L(A)* satisfacen φ y aprovechando que toda fórmula LTL puede transformarse en un autómata de Büchi que satisfacen dicha fórmula *B = buchi(*φ*)* lo que reduciría el problema a ver si ***L(A)*** ⊆***L(B)***. Sin embargo esta tarea no es sencilla por lo que en su lugar lo que se debe verificar es que ***L(A)* ∩ *L(compl(B))* = Ø** siendo *compl(B) = buchi(*¬φ*)*. Es decir, si ninguna de las palabras del sistema A coincide con las palabras que acepta la negación de la fórmula φ.

El model checker realiza la composición sincrónica de A con compl(B) y recorre todo el autómata combinando para asegurar la inexistencia de ciclos de aceptación sobre F.
Si termina de recorrer y no encuentra ninguno, entonces la intersección está vacía.
En caso de encontrar un ciclo entonces es capaz de identificar una palabra τque es aceptada por A y por compl(B), es decir, que satisface ¬φ y por ende no satisface φ.

**NuSMV** es una herramienta Open Source que es una reimplementación de SMV [3], un model checker simbólico basado en diagramas de decisión binaria (BDD [4]). Permite la representación de sistemas de estados finitos para el análisis de propiedades expresadas en CTL o LTL. Un modelo NuSMV se describe mediante una máquina de estados finita (FSM) o autómata finito definiendo los **valores iniciales** que deben tomar sus variables y las **transiciones** que determinan los posibles valores que pueden tomar dichas variables en el siguiente paso.

Veamos un ejemplo sencillo de especificación NuSMV:

```
MODULE main

VAR
        color: {Rojo, Amarillo, Verde};
INIT
        color = Amarillo;
```

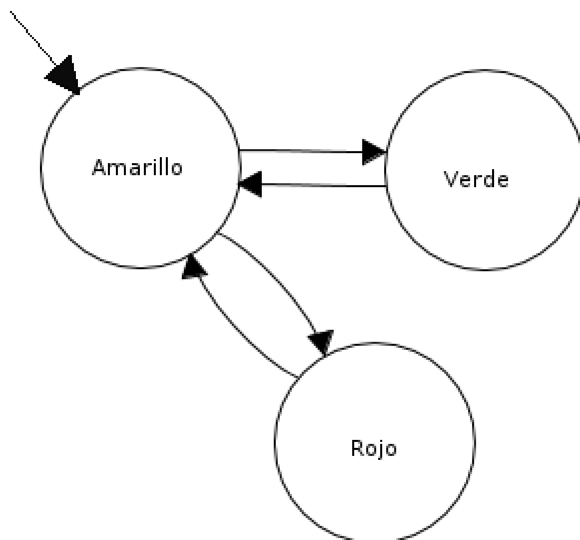
```
TRANS
	(color = Amarillo) -> (next(color) = Verde | next(color) = Rojo);
TRANS
	(color = Rojo) -> (next(color) = Amarillo);
TRANS
	(color = Verde) -> (next(color) = Amarillo);
```

En este ejemplo en la sección *VAR* se define una única variable *color* y se indica que esta puede tomar los valores “*Rojo*”, “*Amarillo*” o “*Verde*”.

Luego en la sección *INIT* se define cuál es el estado inicial que va a tomar la variable definida. En este caso la variable *color* va a comenzar con valor “*Amarillo*”.

Por último en *TRANS* se definen las transiciones que determinan los posibles valores que pueden tomar las variables en el siguiente paso dependiendo de qué valor tienen en el paso actual.

El autómata o FSM correspondiente al ejemplo se podría graficar de la siguiente manera:




Estas transiciones definen **caminos** o **trazas** posibles que puede tomar la variable *color* comenzando con el valor inicial “*Amarillo”.* Por ejemplo algunas trazas posibles serían las siguientes:

- *Amarillo → Verde → Amarillo → Rojo → Amarillo → Verde → Amarillo → Rojo → ...*
- *Amarillo → Rojo → Amarillo → Rojo→ Amarillo → Rojo → ...*
- *Amarillo → Verde → Amarillo → Verde → Amarillo → Verde → ...*

Sin embargo la siguientes trazas no son permitidas:

➢ *Amarillo → Rojo → Rojo→ ...*
➢ *Amarillo → Verde → Verde→ ...*

La función principal de NuSMV es verificar que cierta propiedad (o un conjunto finito de ellas) sea válida dentro de un determinado modelo. Con este objetivo trabaja sobre un modelo del sistema y explora exhaustivamente todos sus posibles estados con el fin de verificar si la propiedad especificada sobre el mismo es verdadera o no.

La manera en que se expresan en NuSMV las propiedades a verificar es mediante fórmulas de lógica temporal lineal (LTL) lo cual nos permite razonar sobre el paso del tiempo. Entre los conectores temporales que vamos a utilizar más frecuentemente están ***G** (Globally),* ***F** (Finally)* y ***U** (Until)*.

➢ *G(φ)* expresa que en todo momento durante la ejecución se satisface la fórmula φ.
➢ *F(φ)* expresa que en algún momento en el futuro se satisface la fórmula φ.
➢ *φ U ψ* expresa que ψ se satisface en algún momento, y para todo momento anterior a aquel φ se satisface.

Veamos ejemplos de expresiones LTL que trabajen sobre el modelo anterior:

➢ *G(color=Amarillo | color=Rojo | color=Verde).* Esta expresión indica que en todo momento la variable *color* tiene valor *Amarillo* o *Rojo* o *Verde* lo cual es **verdadero** para cualquier traza del modelo. Para este caso NuSMV responde:

*-- specification  G ((color = Rojo | color = Amarillo) | color = Verde)  is true*

➢ *G(color=Amarillo).* En este caso la expresión dice que en todo momento la variable *color* tiene el valor *Amarillo* lo cual es **falso** y se puede demostrar fácilmente con una traza en la que la variable tome otro valor distinto al *Amarillo*. Para este caso NuSMV además de responder que es falso también muestra un **contraejemplo**:

```
-- specification  G color = Amarillo  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 6.1 <-
    color = Amarillo
  -> State: 6.2 <-
    color = Rojo
  -> State: 6.3 <-
    color = Amarillo
```

➢ *F(color=Amarillo).* Esta expresión dice que en algún momento la variable *color* tiene el valor *Amarillo*. Para este caso NuSMV responde:

*-- specification  F color = Amarillo  is true*

- ➢ *F(color=Rojo)*. En esta oportunidad se expresa que la variable *color* debe tomar el valor *Rojo* en algún momento. Esto es **falso** porque existen trazas en la que la variable no alcanza ese valor. Es el caso en que las transiciones sean de *Amarillo* a *Verde* y de *Verde* a *Amarillo*. La respuesta de NuSMV en este caso es la siguiente:

```
-- specification  F color = Rojo  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 7.1 <-
    color = Amarillo
  -> State: 7.2 <-
    color = Verde
  -> State: 7.3 <-
    color = Amarillo
```

Internamente NuSMV convierte el problema de validar una fórmula en un **problema de satisfacibilidad booleana (SAT)**. Es aquí donde también entran en juego los **SAT solvers** que son programas dedicados a resolver este tipo de problemas y que pasaremos a explicar a continuación.

## 1.4. Problema de Satisfacibilidad Booleana (SAT) y Picosat

El problema conocido como **Satisfacibilidad Booleana (SAT)** consiste en determinar si una fórmula booleana es satisfacible o no. Se dice que una fórmula es **satisfacible** si existe alguna asignación de valores booleanos a las variables proposicionales que la componen de manera que esta resulte verdadera. Por ejemplo la siguiente fórmula:

$$( x_1 \lor x_2 ) \land ( x_1 \lor x_3 ) \land \neg x_2$$

es satisfacible ya que con x1 = true, x2 = false y x3= true la fórmula resulta verdadera. Pero la siguiente fórmula:

$$( x_1 \lor x_2 ) \land ( x_1 \lor \neg x_2) \land ( \neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$$

no es satisfacible ya que no existe manera de asignar valores booleanos a las variables para que la fórmula resulte verdadera.

En general las fórmulas están expresadas en **forma normal conjuntiva (CNF)** [11] lo que significa que es una conjunción de cláusulas. Se llama *cláusula* a una disyunción de literales. Un literal y su complemento no pueden aparecer en una misma cláusula. Por lo tanto una fórmula expresada en CNF significa que es una conjunción de disyunciones de

literales. Toda fórmula se puede llevar a CNF, aunque eso puede hacer que crezca su longitud.

Se han desarrollado diversos algoritmos y programas conocidos como **SAT solvers** que tienen el objetivo de resolver instancias del problema SAT. Uno de ellos es **Picosat** [6] que es un SAT solver desarrollado por un grupo de investigación de la JKU (Johannes Kepler University, Sede Linz).

La manera en que se interactúa con Picosat es a través de archivos con formato llamado **DIMACS** que es ampliamente aceptado como estándar para fórmulas booleanas expresadas en CNF. Es el formato por defecto para los parámetros de entrada de los SAT solvers.

Un archivo con formato DIMACS comienza con una línea opcional de comentario cuyo primer carácter debe ser la letra c. Ejemplo: *c <comment>*

Luego debe seguir con una línea en donde se especifica la cantidad de variables y la cantidad de cláusulas. Esta línea debe comenzar con la letra p. Ejemplo:
*p cnf <num_vars> <num_clauses>*

Por último se encuentran las líneas que especifican las cláusulas. Se define una línea por cada cláusula de la siguiente manera: si la cláusula tiene un literal positivo se denota por el número correspondiente, y si tiene un literal negativo se denota por el número negativo correspondiente. El último número en una línea debe ser cero. Veamos cómo sería el archivo correspondiente a las fórmulas del ejemplo.

Para la fórmula $( x_1 \lor x_2 ) \land ( x_1 \lor x_3 ) \land \neg x_2$ el archivo DIMACS correspondiente sería el siguiente:

*c Ejemplo de fórmula satisfacible*
*p cnf 3 3*
*1 2 0*
*1 3 0*
*-2 0*

Cuando se ejecuta este archivo con Picosat el resultado es el siguiente:
*s SATISFIABLE*
*v 1 -2 3 0*

Por otro lado para la fórmula $( x_1 \lor x_2 ) \land ( x_1 \lor \neg x_2) \land ( \neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$ el archivo seria:

*c Ejemplo de fórmula no satisfacible*
*p cnf 2 4*
*1 2 0*
*1 -2 0*

*-1 2 0*
*-1 -2 0*

Al ejecutar este archivo con Picosat el resultado es:
*s UNSATISFIABLE*

Al encontrarnos con una fórmula insatisfacible resulta valioso identificar cuáles son las cláusulas responsables de dicha insatisfacibilidad. A este grupo de cláusulas se lo denomina **núcleo de insatisfacibilidad (unsat core)** de una fórmula, que es el conjunto minimal de cláusulas que de no ser parte de la fórmula la harían satisfacible.

Picosat, a diferencia de otros SAT solvers, tiene la posibilidad de identificar el unsat core de una fórmula insatisfacible y devolverlo en su respuesta. Más adelante veremos en detalle cuál es el beneficio que aporta esta característica al proceso de verificación que realiza FormaLex.

## 1.5. El alcance de esta tesis

El trabajo de esta tesis se centra principalmente en la manera en que interactúan FormaLex y NuSMV, analizando los archivos generados que sirven como input de NuSMV e interpretando la información que se presenta en el output del mismo.

En referencia a la información que devuelve NuSMV en su output, el problema que se presenta está relacionado a que actualmente el model checker al momento de responder que existe una contradicción no da indicios de cuáles son las fórmulas involucradas. Este problema motivó a otro trabajo llamado "*Recuperación de "fórmulas culpables" mediante análisis de unsat core"* de Francisco Giménez [8] en donde se realizaron modificaciones a las herramientas que intervienen en el proceso de verificación que permitieron formar un vínculo directo entre las variables del unsat core y las variables de NuSMV. En este caso se propuso como trabajo futuro realizar la integración de la versión modificada de NuSMV a la herramienta FormaLex utilizando la información que se agregó en la salida del model checker. El presente trabajo parte de ese punto y continúa con la integración de la versión modificada de NuSMV a la herramienta FormaLex utilizando la información incorporada. Esta información se utiliza para realizar el mapeo de variables de manera tal que se puedan recuperar las variables involucradas en las fórmulas que hacen que un modelo legal no sea satisfacible. Esta información es valiosa para poder determinar incoherencias entre las normas legales analizadas. En la sección *2 - Traducción de variables de fórmulas* se describe cómo se realiza esta integración y cómo se muestra el resultado de este proceso.

Con respecto a los archivos que se generan para el input de NuSMV, en un trabajo previo realizado por Carlos Faciano llamado "*Optimizando el rendimiento de la herramienta FormaLex de análisis de documentos normativos"* [9] se implementaron optimizaciones y también se analizaron casos de estudio en los cuales se evidenciaron problemas que impidieron la ejecución del proceso. Uno de los problemas detectados fue que el tamaño de

estos archivos excedían los límites aceptados por NuSMV. En este mismo trabajo se propuso como objetivo a futuro realizar optimizaciones que reduzcan los tamaños de los archivos de manera que los casos puedan ser ejecutados. En la sección 3 - *Compactación de inputs* se explicará el mecanismo que se propone para poder llegar a cumplir con el objetivo de reducir el tamaño de los archivos mencionados.

# 2. Traducción de variables de fórmulas

El objetivo que se va a abordar en esta sección tiene que ver con la información que surge en la interacción entre FormaLex, el model checker y el SAT solver. Esta información, que comprende a las variables de los modelos de las herramientas mencionadas, resulta útil para analizar las fórmulas que presenta el usuario y en particular las que contienen inconsistencias. Como resultado final del proceso se van a mostrar por pantalla las variables que son "culpables" de esas inconsistencias.

La tarea de **FormaLex** es modelar y analizar normas legales que plantea un usuario. Mediante la utilización de herramientas informáticas su objetivo es comprobar que dichas normas admiten **al menos un comportamiento legal**. En ese proceso de verificación se utiliza como motor de razonamiento a **model checkers** que son herramientas que ya han sido ampliamente probadas en diferentes ámbitos académicos y profesionales. Por lo tanto resulta indispensable entender el funcionamiento de los model checkers para comprender cómo es el proceso de verificación que realiza FormaLex. Como dijimos anteriormente para este trabajo se utilizó como model checker a **NuSMV**.

Recordemos que el input de NuSMV consiste en un **autómata** y una **fórmula LTL**, y su tarea principal es determinar si la fórmula es **satisfacible** dentro de la clase de modelos descripta por dicho autómata.

En nuestro caso la fórmula se construye de la siguiente manera: Siendo $f_1, f_2, \ldots, f_n$ las fórmulas que resultan de la traducción a LTL de las reglas que plantea el usuario se conforma una fórmula $F$ como la conjunción de todas las fórmulas $f$ es decir $F = f_1 \land f_2 \land \ldots \land f_n$. El model checker toma como parámetros de entrada por un lado a la negación de la fórmula $F$ ($\neg F$) y por otro lado toma un autómata que describe la clase de modelos correspondiente a la teoría marco definidas en FL. Como respuesta el model checker informa acerca de la satisfacibilidad de $\neg F$.

En caso de que la respuesta sea que $\neg F$ no es satisfacible significa que $F$ sí lo es y por lo tanto existe un comportamiento en el que se pueden cumplir todas las normas legales planteadas. En cambio en el caso contrario si la respuesta dice que no hay un contraejemplo[3] para $\neg F$ implica que es satisfacible y por lo tanto, en el contexto de Formalex, significa que $F$ no lo es. Esto significa que algunas de las $f$ que conforman a $F$ son **insatisfacibles**. El problema que se presenta es que el model checker al momento de responder que existe una contradicción **no da indicios de cuáles son las fórmulas involucradas en la contradicción**.

Para poder visualizar el problema mencionado volvamos a utilizar el ejemplo que se presentó en la introducción realizando paso a paso el camino que hace FormaLex.

[3] Las herramientas de model checking llaman contraejemplo a una asignación de valores a las variables que hace que se falsee la fórmula cuya validez se intenta verificar.

El objetivo de presentar este ejemplo es mostrar de una manera sencilla situaciones en donde existen contradicciones entre las reglas formuladas. También intenta mostrar de qué manera el proceso de verificación es capaz de detectar y exponer estas contradicciones.

El contexto en el que se sitúa el caso es un contrato simplificado entre una fábrica de automóviles y un concesionario en el cual se especifican una serie de pautas y condiciones que se deben cumplir. En primer lugar se establece que el comprador se compromete a ordenar mensualmente un mínimo de 5 unidades, lo que le permite al vendedor manejar mejor sus costos y garantizar cierto precio de venta. Luego también estipula un máximo de 10 unidades que el comprador puede adquirir al mes. Por último, debido a que la fábrica dispone de transportes que tienen capacidad para 5 vehículos y que solo se permite que operen si cuentan con su capacidad completa, en caso de que el comprador supere las 5 unidades tiene la obligación de completar la compra de 10 unidades antes de finalizar el mes.

A continuación mostraremos el archivo correspondiente al modelo mencionado expresado en FL:

```
#Background

roles vendedor, comprador

action comprar only performable by comprador
action vender only performable by vendedor

global counter unidades_compradas_al_mes
increases with action comprar
global counter ventas_realizadas
increases with action vender

#Clauses
O(unidades_compradas_al_mes >= 5)
F(unidades_compradas_al_mes >= 10)
O(<>unidades_compradas_al_mes >=5 -> <> unidades_compradas_al_mes == 10)
```

Recordemos qué significa lo que plantean las reglas expresadas en la sección *#Clauses:*

- *O(unidades_compradas_al_mes >= 5)* significa que la variable *unidades_compradas_al_mes* tiene que eventualmente alcanzar el valor 5.
- *F(unidades_compradas_al_mes >= 10)* significa que está prohibido que la variable *unidades_compradas_al_mes* alcance el valor 10 en cualquier momento.

- *O(<>unidades_compradas_al_mes > 5 -> <> unidades_compradas_al_mes == 10)* significa que en caso de que la variable *unidades_compradas_al_mes* supere el valor 5 entonces en algún momento debe tener el valor 10.

El conjunto de reglas definido contiene un **error de límites** ya que por un lado una regla indica que si se superan las 5 unidades es obligatorio llegar a 10 unidades, sin embargo, al mismo tiempo otra regla prohíbe que se alcancen las 10 unidades. Estas reglas que presentan una contradicción son las que **FormaLex debe detectar y exponer**.

El primer paso que realiza FormaLex es parsear el archivo que contiene el modelo y las reglas que se quieren validar. El siguiente paso es **traducir** la teoría marco y las reglas al lenguaje que entiende el model checker NuSMV.

En el caso de las **reglas**, las mismas se traducen formalmente a **fórmulas de la lógica temporal LTL**. En el ejemplo la traducción de las reglas quedaría de la siguiente manera:

**Fórmula resultante de la traducción de las reglas:**

```
!( G ( unidades_compradas_al_mes >= 5 ) &
G ( !( unidades_compradas_al_mes >= 10 ) ) &
G ( ( unidades_compradas_al_mes >= 5 -> ( F(unidades_compradas_al_mes = 10) ) ) ) )
```

Por otro lado la **teoría marco** es traducida al lenguaje de especificación del model checker mediante un autómata. El siguiente es un extracto del archivo que contiene el autómata correspondiente al ejemplo:

**Archivo del autómata NuSMV**:

```
--Agente: roles asignados
--agent_1: vendedor
--agent_2: vendedor, comprador
--agent_3: comprador
MODULE main
VAR
        agent_1.vender: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        agent_3.comprar: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        agent_2.comprar: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        unidades_compradas_al_mes: 0..20;
INIT
        agent_3.comprar = NOT_HAPPENING &
        agent_2.comprar = NOT_HAPPENING &
        unidades_compradas_al_mes =  0 &  TRUE
```

```
TRANS
(agent_3.comprar = JUST_HAPPENED -> next(agent_3.comprar) = NOT_HAPPENING) &
(agent_3.comprar = HAPPENING -> next(agent_3.comprar) != NOT_HAPPENING ) &
...(continua)
```

Luego de la traducción es el turno de NuSMV quien toma el problema y lo transforma en un problema de **satisfacibilidad booleana (SAT)**[4]. Para resolverlo utiliza un SAT solver quien recibe una codificación del modelo NuSMV en forma de problema SAT.

Recordemos que el análisis se realiza sobre la negación de la fórmula, es decir **¬*F*.** Si el SAT solver puede resolver el modelo significa que encontró un contraejemplo. Ese contraejemplo, para el SAT solver significa que existe una forma de asignar valores a las variables de manera que satisfaga la fórmula es decir, que el problema SAT es satisfacible.

Si el problema SAT no es satisfacible significa que no pudo encontrar un contraejemplo. Esto no le preocupa al model checker pero sí a FormaLex ya que implica que no existe comportamiento legal y por lo tanto es necesario identificar cuáles son las normas involucradas que provocaron la contradicción. Para esto es necesario reconocer cuáles son las variables que hicieron que el problema SAT no sea solucionable.

Al analizar la manera en que interaccionan los model checkers con los SAT solvers vemos que NuSMV puede devolver los resultados intermedios en archivos de formato **DIMACS** que es el formato por defecto para los parámetros de entrada de los SAT solvers. Según la documentación oficial de NuSMV para poder generar la salida en formato DIMACS la invocación al mismo se debe realizar con este comando:

*gen_ltlspec_bmc -p "formula" -o filename*

Donde *"formula"* corresponde a la fórmula a validar y *filename* especifica el nombre del archivo de salida en formato DIMACS. Por lo tanto para poder obtener este formato se realizaron modificaciones en FormaLex de manera que el archivo de comandos de ejecución de NuSMV sea generado de la forma mencionada.

Los archivos DIMACS están formados por un encabezado que incluye la relación entre las variables del modelo NuSMV con las variables numéricas de la fórmula booleana expresada en **forma normal conjuntiva** (CNF). Veamos cómo es el archivo DIMACS generado por NuSMV correspondiente al ejemplo mencionado.

Extracto del archivo **DIMACS** generado por **NuSMV** que sirve de entrada para el SAT solver:

---

[4] NuSMV basa sus análisis en *bounded* model checking, lo que le permite recurrir a SAT como metodología de análisis.

```
c BMC problem generated by NuSMV
c Time steps from 0 to 10, 9 State Variables, 0 Frozen Variables and 0 Input Variables
c Model to Dimacs Conversion Table
c
...
c @@@@@ Time 2
c CNF variable 80 => Time 2, Model Variable agent_2.comprar.0
c CNF variable 85 => Time 2, Model Variable unidades_compradas_al_mes.0
...
c @@@@@ Time 10
c CNF variable 687 => Time 10, Model Variable agent_1.vender.1
c CNF variable 688 => Time 10, Model Variable agent_1.vender.0
...
(continúa)
```

Como se ve en el ejemplo, el encabezado incluye el mapeo entre las variables numéricas CNF y las variables del modelo de NuSMV. Por ejemplo la variable *80* se corresponde con la variable *agent_2.comprar*del modelo NuSMV y la variable *85* se corresponde con la variable *unidades_compradas_al_mes* del modelo NuSMV.

Al momento de encontrarnos con un problema SAT no satisfacible es necesario reconocer cuáles son las variables que hacen que no lo sea. Es por ello que para este trabajo se utiliza a **Picosat** como SAT solver debido a que tiene una característica que resulta fundamental para identificar las variables culpables de generar la insatisfacibilidad de una fórmula no satisfacible. En caso de que una fórmula sea insatisfacible Picosat puede generar un archivo en donde se vuelcan las cláusulas que pertenecen al núcleo de la insatisfacibilidad, es decir, las cláusulas que de no ser parte de la fórmula harían que esta sea satisfacible. Este conjunto de cláusulas se lo denomina **unsat core** [7] y es un listado de variables numéricas. Para el caso del ejemplo el archivo de unsat core generado por Picosat queda conformado de la siguiente manera:

Archivo de **unsat core** generado por **Picosat**

```
84
85
86
...
753
757
```

**A partir de este listado la tarea principal de este trabajo es encontrar las entidades del modelo FL que están relacionadas al mismo. Luego de analizar la trazabilidad que**

**existe entre ambos vamos a tener como output un mapeo entre estas variables numéricas del unsat core con las entidades del modelo FL**.

Analizando el ejemplo vemos que el listado de variables del unsat core incluye la variable 85 que mencionamos anteriormente. Pero también vemos que en este listado existen variables numéricas que no se encuentran en el mapeo del encabezado del archivo DIMACS, como por ejemplo las variables 753 y 757. Esto se debe a que con la finalidad de mejorar los tiempos de ejecución y encontrar soluciones más rápidamente NuSMV realiza **conversiones** de las variables del modelo antes de formar la fórmula en CNF. Esto puede generar un problema ya que estas variables convertidas pueden potencialmente ser parte del unsat core, como en el ejemplo sucede con las variables 753 y 757. Para alcanzar el objetivo de encontrar todas las variables culpables de la contradicción necesitamos tener la trazabilidad de todas las variables del listado del unsat core, incluso las convertidas.

En un trabajo previo llamado "*Recuperación de "fórmulas culpables" mediante análisis de unsat core"* de Francisco Giménez [8] se exploró el uso del núcleo de insatisfacibilidad (unsat core) y a través de una modificación del model checker NuSMV se logró vincularlo con sus variables del modelo. Gracias al trabajo de Francisco disponemos de la información acerca de cómo NuSMV realiza las conversiones. Veamos en el ejemplo cómo queda el archivo DIMACS con esta modificación realizada a NuSMV.

Archivo **DIMACS** generado por **NuSMV** que incluye el mapeo de las **conversiones** realizadas:

*c BMC problem generated by NuSMV*
*c Time steps from 0 to 10, 18 State Variables, 0 Frozen Variables and 0 Input Variables*
*c Model to Dimacs Conversion Table*
*...*
*c @@@@@ **Converted values***
*c converted **753** : [687, -688]*
*c converted **757** : [39, 37, 41, 43, 753, 754, 756, 695, 696, ...]*
*c converted **756** : [-698, -699]*
*... (continua)*
*c BMC problem generated by NuSMV*
*c Time steps from 0 to 10, 36 State Variables, 0 Frozen Variables and 0 Input Variables*
*c Model to Dimacs Conversion Table*
*c*
*c @@@@@ Time 0*
*c CNF variable 73 => Time 0, Model Variable agent_6.vender.1*
*c CNF variable 74 => Time 0, Model Variable agent_6.vender.0*
*...*
*c CNF variable 135 => Time 1, Model Variable compras_realizadas.4*
*c CNF variable 136 => Time 1, Model Variable compras_realizadas.3*
*c CNF variable 137 => Time 1, Model Variable compras_realizadas.2*
*c CNF variable 138 => Time 1, Model Variable compras_realizadas.1*

```
c CNF variable 139 => Time 1, Model Variable compras_realizadas.0
...
c CNF variable 802 => Time 7, Model Variable compras_realizadas.4
c CNF variable 803 => Time 7, Model Variable compras_realizadas.3
c CNF variable 804 => Time 7, Model Variable compras_realizadas.2
c CNF variable 806 => Time 7, Model Variable compras_realizadas.1
c CNF variable 805 => Time 7, Model Variable compras_realizadas.0
...
c CNF variable 931 => Time 8, Model Variable compras_realizadas.4
c CNF variable 932 => Time 8, Model Variable compras_realizadas.3
c CNF variable 933 => Time 8, Model Variable compras_realizadas.2
c CNF variable 934 => Time 8, Model Variable compras_realizadas.1
c CNF variable 935 => Time 8, Model Variable compras_realizadas.0
...
c CNF variable 1095 => Time 9, Model Variable compras_realizadas.4
c CNF variable 1096 => Time 9, Model Variable compras_realizadas.3
c CNF variable 1097 => Time 9, Model Variable compras_realizadas.2
c CNF variable 1098 => Time 9, Model Variable compras_realizadas.1
c CNF variable 1099 => Time 9, Model Variable compras_realizadas.0
c CNF variable 1100 => Time 9, Model Variable ventas_realizadas.4
c CNF variable 1101 => Time 9, Model Variable ventas_realizadas.3
c CNF variable 1102 => Time 9, Model Variable ventas_realizadas.2
c CNF variable 1103 => Time 9, Model Variable ventas_realizadas.1
c CNF variable 1104 => Time 9, Model Variable ventas_realizadas.0
...
```

Como vemos el archivo es muy similar al anteriormente presentado pero este último incluye además las **conversiones de variables que realizó NuSMV**. Esto se puede visualizar en la sección *Converted values.*

Ahora nuestro trabajo consiste en mapear estas variables a entidades FL. Volviendo al listado de variables del unsat core y con la información de las conversiones vamos a analizar los casos que se pueden presentar y plantear un procedimiento para el mapeo de estos casos.

Como primer caso presentamos el que resulta de realizar un mapeo directo de la variable CNF con la variable del modelo FL. En el ejemplo se puede ver el casos de la variable *85* que es parte del unsat core. Esta variable tiene un mapeo directo con una variable del modelo FL, en este caso *unidades_compradas_al_mes* . En estos casos no es necesario hacer ningún proceso extra ya que el mapeo es directo.

El segundo caso es del mapeo de una variable CNF convertida. Este es el caso de la variable *753,* que es parte del unsat core y además es una variable convertida. En este caso para obtener su mapeo se debe hacer el camino inverso analizando la lista de sus

conversiones: 687 y 688. Todas estas variables tienen un mapeo directo con la variable *agent_1.vender* del modelo FL por lo que la variable *753* también tiene el mismo mapeo.

Por último vemos el caso en que una variable convertida tiene en su lista de conversiones a variables que a su vez son convertidas. Este caso requiere un procesamiento recursivo para resolver el mapeo. En el ejemplo vemos la variable *757* que forma parte del conjunto del unsat core y que en su lista contiene a la variable *756* que a su vez es una variable convertida[5]. Por lo tanto es necesario resolver el mapeo de 756 en primer lugar para luego poder completar el mapeo de la variable 757.
Este procedimiento de mapeo se va a iterar hasta transformar toda la lista de variables del unsat core en un conjunto de variables correspondientes al modelo de NuSMV.

A continuación se muestra el pseudocódigo del algoritmo propuesto para resolver el problema:

[5] En el ejemplo se ve que algunas variables aparecen negadas pero es sólo una indicación de polaridad. Para los efectos de nuestro trabajo representan lo mismo que las positivas por lo que vamos a expresarlas siempre en positivo.

```
mapUnsat
        def cnfVariablesMap ← Se parsean las variables CNF y se vuelcan en el mapa
        def unsatVariables ← Se parsean las variables unsat y se vuelcan en una lista
        def convertedVariablesMap ← Se parsean las conversiones que realizó NuSMV y se vuelca
en un mapa

        def completedMap // Mapa en donde se va a devolver el resultado final del mapeo
        def pendingQueue // Cola que va a guardar las variables pendientes de resolver el mapeo

        Para todas las unsatVar de unsatVariables
                Si unsatVar tiene mapeo directo (se encuentra en el cnfVariablesMap)
                        se vuelca en el completedMap todo el mapeo correspondiente a la unsatVar
                Si unsatVar no tiene mapeo directo (es una variable convertida)
                        listaConversiones ← convertedVariablesMap.get(unsatVar)
                        se agrega a la pendingQueue la tupla <unsatVar, listaConversiones>
                fin Si
        fin Para todos

        Mientras existan elementos en la pendingQueue
                se desencola tupla <unsatVar, listaConversiones> de la pendingQueue
                def hayQueReencolar ← false  // booleano que indica si hay que reencolar la tupla
                Para todas las convertedVar de listaConversiones
                        Si la convertedVar tiene todo su mapeo resuelto
                                se vuelca todas los mapeos que se corresponden a unsatVar
                        Si la convertedVar no tiene su mapeo resuelto
                                hayQueReencolar ← true
                        fin Si
                fin Para todos
                Si hayQueReencolar
                        se vuelve a encolar la tupla <unsatVar, listaConversiones>
                Si no hayQueReencolar
                        se vuelca en el completedMap todo el mapeo correspondiente a la unsatVar
                fin Si
        fin Mientras
        devuelve completedMap

fin mapUnsat
```

El algoritmo en primer lugar recolecta toda la información necesaria parseando los archivos del modelo NuSMV y del unsat core generado por PicoSat.

El archivo del modelo generado por NuSMV es un archivo DIMACS del cual se pueden extraer por un lado las **variables CNF** y su mapeo a **variables del modelo en el autómata NuSMV** y por el otro del mismo archivo se obtienen las **conversiones realizadas por NuSMV**. Por su parte el archivo generado por PicoSat contiene las variables del **unsat core**.

Luego del parseo de los archivos vamos a tener un mapa (**cnfVariablesMap**) que relaciona las variables CNF del archivo DIMACS con las variables del autómata de NuSMV, otro mapa

(**convertedVariablesMap**) que contiene la relación entre las variables convertidas por NuSMV con las variables originales y un conjunto (**unsatVariables**) que contiene las variables “culpables” de la insatisfacibilidad o sea las que están involucradas en el unsat core.

En primer lugar se recorre el conjunto unsatVariables y a medida que se van completando los mapeos de las variables se irán guardando en un mapa llamado **completedMap** en donde las claves serán las variables unsat y los valores serán las entidades FL relacionadas.

Como vimos en el ejemplo dentro del conjunto unsatVariables pueden existir tanto variables originales de modelo NuSMV como variables convertidas. Para el caso de las variables originales el mapeo con entidades FL es directo lo cual no presenta problemas. En el ejemplo sería el caso de la variable 85 la cual se agrega al completedMap de la siguiente manera:

*{*
*85 => [ unidades_compradas_al_mes ]*
*}*

En cambio en el caso de las variables convertidas es necesario hacer el camino inverso de conversión para identificar a cuáles variables originales corresponden y recién ahí poder hacer el mapeo a las entidades FL. Con ese fin en una cola llamada **pendingQueue** se colocan tuplas conformadas por las variables pendientes de resolver junto con sus variables relacionadas en la conversión. En el ejemplo presentado la cola pendingQueue quedaría conformada de la siguiente manera:

*< 753, [687, 688] >*
*< 757, [39, 37, 41, 43, 753, 754, 756, 695, 696, ...] >*
*< 756, [698, 699] >*
Mientras existan elementos en la pendingQueue se va a tomar el primero y se va a tratar de resolver todos sus mapeos.
En el ejemplo se toma el primer elemento que es la tupla *< 753, [687, 688] >*
La variable que se debe resolver en este caso es la 753 y como todos sus variables relacionadas ya tienen su mapeo resuelto (se encuentran todas en el mapa cnfVariablesMap) entonces se agrega al completedMap que queda de la siguiente manera:

*{*
*85 => [ unidades_compradas_al_mes ],*
*753 => [ agent_1.vender ]*
*}*

Luego obtenemos el siguiente elemento de la pendingQueue que en este caso sería la tupla < 757, [*39, 37, 41, 43, 753, 754, 756, 695, 696, ...*] >. La variable a resolver en este caso es la 757y como no todas sus variables relacionadas tienen resuelto su mapeo (en este caso la

variable 756 aún no ha sido resuelta) entonces se vuelve a encolar en una tupla junto con la variable relacionada que aún resta resolver.
La pendingQueue quedaría de la siguiente manera:

*< 756, [698, 699] >*
*< 757, [756] >*

Como la siguiente variable a resolver es la 756 cuando vuelva a ser el turno de la 757 ya va a tener todas sus variables relacionadas con su mapeo resuelto y por lo tanto podrá pasar al completedMap.

Cuando la pendingQueue queda vacía quiere decir que en el completedMap van a quedar todas las variables del unsat core mapeadas con las entidades del modelo FL correspondientes. En el ejemplo el completedMap queda conformado de la siguiente manera:

*{*
*85 => [ unidades_compradas_al_mes ],*
*753 => [ agent_1.vender ],*
*756 => [ unidades_compradas_al_mes ],*
*757 => [ unidades_compradas_al_mes, agent_1.vender ]*

*}*

Como resultado final se muestran en pantalla el listado de las entidades FL identificadas en el proceso descripto como se ve a continuación:

*No se ha encontrado un comportamiento legal para las normas.*
*Entidades involucradas en el unsat:*
*ventas_realizadas*
*agent_1 (roles: [vendedor]), entidad: vender*
*agent_2 (roles: [vendedor, comprador]), entidad: vender*
*agent_3 (roles: [comprador]), entidad: comprar*
*unidades_compradas_al_mes*
*agent_2 (roles: [vendedor, comprador]), entidad: comprar*

En este caso se visualiza la entidad *unidades_compradas_al_mes* que como explicamos anteriormente está involucrada en la contradicción de las reglas planteadas por el usuario[6].

[6] Se propone como trabajo futuro que en lugar de que el output muestre agentes y roles, que son una representación interna de FormaLex, el mapeo final se realice con las cláusulas originales que escribió el usuario de manera que sea más inteligible para el mismo.

# 3. Compactación de inputs

Como se explicó en la sección anterior en una de las etapas del proceso de verificación que realiza FormaLex se invoca a un **model checker** que para el caso de este trabajo se trata de **NuSMV**. También mencionamos que el input del mismo consiste en un **autómata** que se genera a partir de la especificación de la teoría marco y una **fórmula LTL** la cual se necesita verificar si es satisfacible o no. Tanto el autómata como la fórmula están representados por **archivos de texto plano** y cuyo contenido respeta el formato que requiere el model checker. La herramienta utilizada para la generación de estos archivos es **Velocity**[7] que es un motor de templates basado en Java.

Estos archivos que se generan (el del autómata y el de la fórmula) pueden llegar a ser de **gran volumen** lo cual impacta negativamente en la **performance** de la ejecución del model checker e incluso **sobrepasar el límite** de tamaño aceptado por el mismo. Mientras se realizaban las pruebas para la tesis de grado *"Optimizando el rendimiento de la herramienta FormaLex de análisis de documentos normativos"* de Carlos Faciano [9] se identificó una **limitación** que posee NuSMV que consiste en que sólo puede procesar fórmulas LTL de hasta **65 kb** de tamaño. Esta limitación de NuSMV no se encuentra documentada sin embargo en pruebas empíricas se observaron problemas en el parseo de los archivos al superar dicho límite. Como se detalla en la tesis en varios casos se superó este límite lo cual impidió la ejecución de esas pruebas. Si bien NuSMV es una herramienta open source, lo que permite analizar y hacer modificaciones que optimicen la limitación nombrada, realizar esta tarea no parece ser menor. Es por esto que se planteó una solución alternativa que realiza una **optimización** que se focaliza en **reducir el tamaño** de los archivos input de manera que permita llevar a cabo las pruebas que no se pudieron realizar por la limitación mencionada.

Analizando el contenido del archivo del autómata vemos que en él se describen las **variables** que representan las entidades del modelo junto con los **estados** que pueden tomar y las ecuaciones que definen las posibles transiciones entre un estado y otro.

Una técnica de compactación posible es mediante el **renombre de variables y estados** por nombres más cortos manteniendo la semántica que describe el modelo. Veamos qué tipos de entidades se pueden presentar y qué optimización podemos realizar en cada uno de esos casos.

En primer lugar podemos hablar de las **acciones**. Estas se representan con variables que pueden tomar los valores ***HAPPENING***, ***JUST_HAPPENED*** o ***NOT_HAPPENING***. Por ejemplo si en el modelo tenemos las acciones *comprar* y *vender* la declaración e inicialización de las variables correspondientes en el autómata sería de la siguiente manera:

*MODULE main*
*VAR*

[7] http://velocity.apache.org

```
        comprar: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        vender: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
INIT
        comprar = NOT_HAPPENING &
        vender = NOT_HAPPENING &
        TRUE
```

Otro tipo de entidades son los **timers**. Los mismos se declaran e inicializan en el autómata de manera similar a las acciones.

Por otro lado está el caso de los **contadores** que se representan con variables que pueden tomar valores numéricos. Como ejemplo podemos tener dos contadores *compras_realizadas* y *ventas_realizadas* que se se declaran e inicializan de la siguiente manera:

```
MODULE main
VAR
        compras_realizadas: 0..20;
        ventas_realizadas: 0..20;
INIT
        compras_realizadas =  0 &
        ventas_realizadas =  0 &
        TRUE
```

Para el caso de los **intervalos** se representan en el autómata con variables que pueden tener los valores ***ACTIVE*** o ***INACTIVE***. Por ejemplo si tenemos dos intervalos *semana* y *concurso* se representan en el autómata de la siguiente forma:

```
MODULE main
VAR
        semana: {ACTIVE, INACTIVE};
        concurso: {ACTIVE, INACTIVE};
INIT
        semana = INACTIVE &
        concurso = INACTIVE &
        TRUE
```

Ya que conocemos qué tipo de entidades se pueden presentar y en qué forma lo hacen pasamos a plantear de qué manera se pueden realizar **reemplazos** con el fin de reducir el tamaño del archivo del autómata.

Por un lado tenemos los valores ***HAPPENING***, ***JUST_HAPPENED***, ***NOT_HAPPENING, ACTIVE*** e ***INACTIVE.*** Si queremos reducir el tamaño de estas cadenas de caracteres podríamos realizar los siguientes reemplazos:

*NOT_HAPPENING → NH*

*HAPPENING → HA*
*JUST_HAPPENED → JH*
*ACTIVE → AC*
*INACTIVE → IA*

Por su parte los nombres de **variables** de las entidades (acciones, timers, intervalos y contadores) también pueden reducirse en tamaño. Una manera posible es hacer los reemplazos generando variables compuestas por una letra y un número incremental[8]. Por ejemplo un formato de nombre de variable de reemplazo podría ser $i_1, i_2, i_3, \ldots, i_n$. Si en nuestro modelo tuviéramos acciones *comprar* y *vender* y además contadores *compras_realizadas* y *ventas_realizadas* los reemplazos de los nombres de sus variables quedarían de la siguiente manera:

*comprar* → $i_0$
*vender* → $i_1$
*compras_realizadas* → $i_2$
*ventas_realizadas* → $i_3$

Veamos un ejemplo de archivo de autómata al cual posteriormente se le realizará el proceso de compactación sugerido.

**Archivo del Autómata**:

```
MODULE main
VAR
        comprar: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        vender: {HAPPENING, NOT_HAPPENING, JUST_HAPPENED};
        compras_realizadas: 0..20;
        ventas_realizadas: 0..20;
INIT
        comprar = NOT_HAPPENING &
        vender = NOT_HAPPENING &
        compras_realizadas =  0 &
        ventas_realizadas =  0 &
  TRUE

TRANS
        (comprar = JUST_HAPPENED -> next(comprar) = NOT_HAPPENING) &
        (comprar = HAPPENING -> next(comprar) != NOT_HAPPENING ) &
        (comprar = NOT_HAPPENING -> next(comprar) != JUST_HAPPENED) &
        (vender = JUST_HAPPENED -> next(vender) = NOT_HAPPENING) &
        (vender = HAPPENING -> next(vender) != NOT_HAPPENING ) &
        (vender = NOT_HAPPENING -> next(vender) != JUST_HAPPENED) &
```

[8] Se propone como trabajo futuro modificar la base utilizada para la numeración del subíndice, por ejemplo a base hexadecimal, lo que lograría aún más compactación.

```
        (next(comprar) = JUST_HAPPENED  & (compras_realizadas + 1 <= 20) &
(compras_realizadas + 1 >= 0) -> next(compras_realizadas) = compras_realizadas + 1) &
(next(comprar) = JUST_HAPPENED  & (ventas_realizadas + 1 <= 20)
        & (ventas_realizadas + 1 >= 0) -> next(ventas_realizadas) = ventas_realizadas + 1) &

... (continua)
```

Aplicando los reemplazos propuestos el archivo resultante queda de la siguiente manera:

**Archivo del autómata luego de la compactación:**

```
MODULE main
VAR
        i0: {HA, NH, JH};
        i1: {HA, NH, JH};
        i2: 0..20;
        i3: 0..20;
INIT
        i0 = NH &
        i1 = NH &
        i2 = 0 &
        i3 =  0 &
   TRUE

TRANS
        (i0 = JH -> next(i0) = NH) &
        (i0 = HA -> next(i0) != NH ) &
        (i0 = NH -> next(i0) != JH) &
        (i1 = JH -> next(i1) = NH) &
        (i1 = HA -> next(i1) != NH ) &
        (i1 = NH -> next(i1) != JH) &
        (next(i0) = JH  & (i2 + 1 <= 20) &
        (i2 + 1 >= 0) -> next(i2) = i2 + 1) &
        (next(i0) = JH  & (i3 + 1 <= 20) &
        (i3 + 1 >= 0) -> next(i3) = i3 + 1) &
... (continua)
```

Como vemos luego de realizar todos los reemplazos se logra **reducir** considerablemente la cantidad de caracteres y por ende el tamaño del archivo. Sin embargo se mantiene la **equivalencia semántica** con el archivo original. Este nuevo archivo compactado será entonces el input del model checker.

Algo importante a tener en cuenta es que además de tener un mecanismo que realice estos reemplazos es necesario tener otro que sirva para restablecer los nombres originales de

estados y variables, es decir un mecanismo para hacer el **proceso inverso**. Es por eso que al momento de hacer los reemplazos también se genera un **archivo** en donde se registra cómo fueron hechos dichos reemplazos así es posible **restablecer** los nombres originales cuando sea necesario.

Veamos más en detalle cómo se realiza la **compactación** y cómo se incorpora en el proceso de verificación que realiza FormaLex. Como dijimos anteriormente en una de las etapas de este proceso se genera el archivo del autómata. Luego en caso de que esté habilitada la opción de compactación se genera en memoria un **mapa de reemplazos** de la manera que describimos anteriormente. Este mapa es generado a partir de la información que brinda la teoría marco y además de quedar almacenado en memoria también queda guardado en un archivo de reemplazos.

En el siguiente paso se crea una entidad ***automatonCompactor*** a partir del mapa de reemplazos generado en el paso anterior. Esta entidad es la encargada de realizar la **compactación del autómata** a través de su método ***compact***. Este método carga el contenido del archivo en un String y luego aplica los reemplazos correspondientes generando como resultado el archivo con el autómata compactado.

Para aplicar los reemplazos se utilizan las clases del paquete ***java.util.regex*** de Java que proporciona funcionalidades que permiten realizar **operaciones sobre cadena de caracteres** en base a **expresiones regulares**. Estas expresiones regulares se utilizan para **buscar todas las apariciones** de un **patrón de caracteres** especificado por metacaracteres especiales. Los **metacaracteres** permiten crear una expresión regular única que encontrará todas las ocurrencias de un patrón básico.

En nuestro caso el patrón de la expresión regular se especifica a través de un String que posteriormente determinará dónde se deben hacer los reemplazos. Veamos un ejemplo que nos permita mostrar cómo se conforma el patrón de nuestro caso. Supongamos que tenemos el mapa de reemplazos del ejemplo anterior:

*NOT_HAPPENING → NH*
*HAPPENING → HA*
*JUST_HAPPENED → JH*
*ACTIVE → AC*
*INACTIVE → IA*
*comprar* → $i_0$
*vender* → $i_1$
*compras_realizadas* → $i_2$
*ventas_realizadas* → $i_3$

El patrón de búsqueda se debe conformar en base a las **claves del mapa de reemplazos**. Para el caso del ejemplo quedaría conformado de la siguiente manera:

*"\bNOT_HAPPENING\b | \bJUST_HAPPENED\b | \bHAPPENING\b | \bACTIVE\b | \bINACTIVE\b | \bcomprar\b | \bvender\b | \bcompras_realizadas\b | \bventas_realizadas\b"*

El metacaracter **|** expresa el conector lógico OR y ***\b*** indica el límite de palabra (word boundary). Por lo tanto este patrón indica que debe encontrar una ocurrencia cada vez que la cadena de caracteres coincide con algunas de las claves del mapa de reemplazos.

El fragmento de código que define el patrón es el siguiente:

```
StringBuffer pattern = new StringBuffer();

for (String toBeReplaced : replacementMap.keySet()) {
        pattern.append("|\\b"+toBeReplaced+"\\b");
}

Pattern p = Pattern.compile(pattern.toString());
```

El ciclo for va conformando el string del patrón (*pattern*) a partir de las claves del mapa de reemplazos (*replacementMap*) de la manera que explicamos anteriormente. Luego de construir el string del patrón este es compilado mediante el método ***compile*** que devuelve una instancia de la clase ***Pattern****.* La clase *Pattern* es la representación compilada de una expresión regular a partir de la cual podremos realizar las **operaciones de búsqueda y reemplazo de caracteres**. Estas operaciones de búsqueda se realizan en la siguiente porción de código:

```
StringBuffer stringWithReplacementsBuffer = new StringBuffer();

Matcher m = p.matcher(originalString);

while (m.find()) {
        m.appendReplacement(stringWithReplacementsBuffer,
        replacementMap.get(m.group()));
}
m.appendTail(stringWithReplacementsBuffer);
```

La clase ***Matcher*** es un tipo de objeto que se crea a partir de un patrón mediante la invocación del método ***matcher***. Este objeto es el que nos permite realizar **operaciones** sobre la secuencia de caracteres que queremos buscar en función del patrón especificado.

Como se ve en el código hay un ciclo while cuya condición corresponde al resultado de la invocación del método ***find****.* Cada vez que este método encuentra una **coincidencia** en la entrada (*originalString*) se invoca al método ***appendReplacement*** que agrega a la salida (*stringWithReplacementsBuffer*) los caracteres que se encuentran entre la posición de la coincidencia anterior y la actual y a continuación el texto de reemplazo. Este ciclo se repite hasta que no se encuentren más coincidencias en la entrada.

Luego de salir del ciclo una parte del texto de entrada aún no se habrá copiado en el StringBuffer de salida. Estos caracteres son los que se encuentran desde la posición del

final de la última coincidencia hasta el final del texto de entrada. Llamando al método ***appendTail*** se agregan estos últimos caracteres al StringBuffer de salida.

El **output** de este procedimiento es un String con todos los reemplazos de nombres de variables y estados el cual será el contenido del archivo del **autómata compactado**. El tamaño del archivo compactado resulta considerablemente menor al tamaño del archivo original.

El model checker además del autómata toma como parámetro una **expresión** que representa la **fórmula a verificar**. Esta expresión también contiene nombres de variables del modelo por lo cual es necesario aplicarle los mismos reemplazos que se aplicaron al autómata. Un ejemplo de fórmula puede ser la siguiente:

*!( G ( !( compras_realizadas = 4 ) ) & G ( ventas_realizadas = 2 -> ( F(compras_realizadas = 4) ) ) & G ( F(ventas_realizadas = 2) ) )*

Luego de aplicarle los reemplazos correspondientes quedaría de la siguiente forma:

*!( G ( !( i2 = 4 ) ) & G ( i3 = 2 -> ( F(i2 = 4) ) ) & G ( F(i3 = 2) ) )*

Por lo tanto al momento de invocar al model checker se le pasa como parámetros al **autómata y la expresión compactados** y de esta manera al ser archivos más livianos benefician al funcionamiento del model checker.

Por último luego de la invocación al model checker es probable que se requiera **interpretar** y mostrar en pantalla el resultado que este retorna. Es el caso de cuando no se encuentra un comportamiento legal y se necesita identificar cuáles fueron las variables culpables de que esto suceda. Es aquí donde resulta fundamental referirnos a los **nombres originales** de las variables involucradas en el resultado. Por lo tanto se aplica el **proceso inverso** de reemplazos al output del model checker antes de mostrarlo en pantalla.

# 4. Casos de estudio

En la sección 2 se explicó la traducción de fórmulas y en ella se utilizó un ejemplo que sirvió para mostrar de manera detallada todos los pasos que lleva a cabo el algoritmo propuesto para resolver el problema planteado. Por lo tanto podemos tomar este ejemplo como un caso de estudio que verifica el correcto funcionamiento del proceso incorporado a FormaLex.

Con respecto a la compactación de inputs, para evaluar la efectividad del trabajo realizado se construyeron los casos de estudio en base a los presentados en la tesis de Carlos Faciano [9] en donde se analizaron en detalle los tamaños de los archivos generados que son input del model checker NuSMV (autómata y fórmula LTL). De los casos presentados en el trabajo mencionado se seleccionaron solo los que tuvieron problemas en la ejecución debido a la limitación de tamaño de los inputs que tiene NuSMV (65 Kb).

Se estudiaron 15 casos de los cuales se muestran a continuación 4 de ellos junto con el resultado obtenido luego de la compactación:

| Caso LDC1_1 | Sin compactar | Compactado | Porcentaje reducción |
|---|---|---|---|
| Tamaño del autómata (en Kb) | 39,1 | 13 | **66,75%** |
| Tamaño de la fórmula (en Kb) | 4,1 | 1,3 | **68,29%** |

| Caso LDC1_25 | Sin compactar | Compactado | Porcentaje reducción |
|---|---|---|---|
| Tamaño del autómata (en Kb) | 457,7 | 174,7 | **61,83%** |
| Tamaño de la fórmula (en Kb) | 65,5 | 27,9 | **57,40%** |

| Caso LDC1_25 | Sin compactar | Compactado | Porcentaje reducción |
|---|---|---|---|
| Tamaño del autómata (en Kb) | 193,3 | 85,4 | **55.82%** |
| Tamaño de la fórmula (en Kb) | 118,3 | 47,8 | **59.59%** |

| Caso LDC2_28 | Sin compactar | Compactado | Porcentaje reducción |
|---|---|---|---|

| Tamaño del autómata (en Kb) | 179,5 | 74,4 | **58,55%** |
| --- | --- | --- | --- |
| Tamaño de la fórmula (en Kb) | 122,2 | 48,4 | **60,39%** |

Como se ve en el cuadro el resultado del proceso de compactación generó una reducción del tamaño de los archivos de alrededor del **60%**.

Algo que se logró identificar es que en los casos ejecutados sin la compactación y que superan el límite **se presentó el siguiente error**:

*file <command-line>: line 2386: at token ";": syntax error*
*Parsing error: expected an "LTL" expression.*

Luego de aplicar la compactación a los mismos casos todos los archivos correspondientes quedaron por debajo del límite. Al ejecutar los casos con la versión compactada se verificó que **no se manifestó el error** anteriormente mencionado.

Sin embargo, si bien se consiguió una importante reducción de los archivos, **ninguno de los casos ejecutados** luego de aplicada la optimización lograron finalizar su ejecución.

# 5. Conclusiones

La principal meta de este trabajo era contribuir a que FormaLex logre detectar y exponer contradicciones que se puedan presentar en el conjunto de reglas que el usuario quiere validar. Para lograr ese objetivo se tomó como punto de partida el trabajo realizado en [8] y como primera tarea se llevó a cabo la integración de la versión modificada de NuSMV a FormaLex.

La siguiente tarea consistió en tomar la nueva información que aporta esta modificación y utilizarla para poder determinar las variables del modelo que originan las contradicciones que se puedan presentar. Para llevar a cabo esta tarea se estudió más en detalle el funcionamiento del model checker (NuSMV) y el SAT solver (Picosat) y la manera que interactúan entre ellos.

Utilizando un ejemplo se hizo un recorrido de punta a punta del proceso de verificación partiendo del modelo expresado en FL pasando por la transformación al modelo NuSMV y finalizando con la traducción al formato DIMACS necesario para que lo pueda entender Picosat. En este recorrido se presentaron diversos problemas como por ejemplo la conversión de variables que realiza NuSMV y que gracias a las modificaciones realizadas en [8] se dispone de la información necesaria para entender cómo se realizan estas conversiones.

A partir del unsat core y la información de conversiones se logró presentar un algoritmo que permite realizar la trazabilidad de las variables que aparecen en el recorrido lo que permitió a su vez mostrar en pantalla cuáles son las entidades FL que son culpables de la contradicción en caso de que exista.

Como trabajo futuro se propone modificar el output del proceso de manera que la información que se muestre sea más cercana al dominio del usuario de la herramienta. La manera en que la versión actual presenta las partes que generan la contradicción es a través de los objetos que intervienen en la misma. Estos objetos, como por ejemplo agentes y roles, son una representación interna de FormaLex lo cual no resulta inteligible para el usuario. Una alternativa más útil es que en el output se identifiquen claramente cuáles son las fórmulas pertenecientes al conjunto de reglas que están involucradas en la contradicción. Para ejemplificar la idea de la propuesta vamos a contrastar cómo se muestra actualmente el output con lo que se espera a futuro. Como vimos anteriormente el output del proceso se muestra como sigue:

*No se ha encontrado un comportamiento legal para las normas.*
*Entidades involucradas en el unsat:*
*ventas_realizadas*
*agent_1 (roles: [vendedor]), entidad: vender*
*agent_2 (roles: [vendedor, comprador]), entidad: vender*
*agent_3 (roles: [comprador]), entidad: comprar*
*unidades_compradas_al_mes*
*agent_2 (roles: [vendedor, comprador]), entidad: comprar*

En su lugar, la propuesta sería que se identifiquen las fórmulas que intervienen en la contradicción con un estilo similar al siguiente:

> *No se ha encontrado un comportamiento legal para las normas.*
> *Fórmulas involucradas en la contradicción:*
> *O(unidades_compradas_al_mes >= 5)*
> *F(unidades_compradas_al_mes >= 10)*
> *O(<>unidades_compradas_al_mes >=5 -> <> unidades_compradas_al_mes == 10)*

Como se ve el resultado habla de fórmulas en lugar de entidades lo que permite ver de una forma más precisa la información correspondiente a las contradicciones.

Para cumplir este objetivo se plantea el desafío de poder encontrar la forma de discriminar cuáles de las fórmulas que contienen las entidades unsat son parte de la contradicción. Otro problema que abordó esta tesis fue el detectado en [9] en donde se encontró una limitación de NuSMV con respecto al tamaño máximo de la fórmula LTL que este puede soportar (65 Kb). Si bien esta limitación no está documentada se logró identificar que si se ejecutan casos que la superan lanzan un error de parseo del archivo.

En relación a esta limitación el objetivo que se planteó en este caso fue el de reducir el tamaño de los archivos que sirven de input de NuSMV de manera que esta mejora permita abarcar una mayor cantidad de casos exitosos.

Para cumplir con este propósito de reducir el tamaño de los archivos se presentó un algoritmo que desarrolla una técnica de compactación que consiste en realizar reemplazos de nombres de variables por nombres más cortos. Este algoritmo además de realizar los reemplazos correspondientes también brinda un mecanismo que permite recuperar los nombres de variables originales.

Analizando el cuadro con los resultados que se obtienen luego de aplicar la compactación se observa que la reducción de los archivos representa en promedio el 60% del tamaño original.

Sin embargo, a pesar de que las reducciones del tamaño de los archivos fueron importantes los casos analizados igualmente no lograron finalizar. Como trabajo futuro se propone analizar otro tipo de optimizaciones como por ejemplo modificar la base utilizada para la numeración de los subíndices que actualmente se encuentran en base 10. Si en su lugar se utilizara base hexadecimal se lograría aún más compactación lo que a su vez podría permitir correr casos que aún no lograron finalizar.

# Bibliografía


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

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

[3] Kenneth Lauchlin McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[4] S. B. Akers. Binary decision diagrams. IEEE Trans. Comput., 27(6):509–516, June 1978.

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

[6] Armin Biere. Picosat essentials. JSAT, 4:75–97, 2008.

[7] Cimatti, A., Griggio, A., & Sebastiani, R. (2011). Computing small unsatisfiable cores in satisfiability modulo theories. Journal of Artificial Intelligence Research, 40, 701-728.

[8] Francisco Andrés Gimenez. Recuperación de “fórmulas culpables”mediante análisis de unsat core, 2017. Tesis de Licenciatura.

[9] Carlos Augusto Faciano. Optimizando el rendimiento de la herramienta formalex de análisis de documentos normativos, 2016. Tesis de Licenciatura.

[10] A. Pnueli, "The temporal logic of programs," 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), Providence, RI, USA, 1977, pp. 46-57, doi: 10.1109/SFCS.1977.32.[11] M. N. Velev. Efficient translation of Boolean formulas to CNF in formal verification of microprocessors.
In Asia and South Pacific Design Automation Convference (ASP-DAC ’04), January 2004.
o
S.K. Sobolev. Conjuntive Formal Form. http://www.encyclopediaofmath.org/index.php?title=Conjunctive_normal_form&oldid=35078, 2008.