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

Recuperación de "fórmulas culpables" mediante análisis de unsat core.

Tesis de Licenciatura en Ciencias de la Computación

Francisco Andrés Gimenez

Director: Dr. Fernando Schapachnik

Buenos Aires, 2017

Abstract

La verificación de modelos (en inglés *model checking*) es un método automático para verificar sistemas formales, escritos generalmente en algún tipo de lógica temporal. A lo largo de los años se fueron creando nuevas técnicas dentro de esta área para poder realizar la tarea más eficientemente. Una de ellas es conocida como *Bounded Model Checking* (BMC), que en general se tratxta de utilizar internamente procesos SAT para realizar la verificación deseada sobre un determinado modelo.

FormaLex es una herramienta desarrollada para verificar coherencia de documentos normativos escritos en un lenguaje basado en lógica temporal. Internamente utiliza un model checker para poder constatar dicha coherencia. Sin embargo, existen casos en los que la información que estos brindan no es suficiente para determinar de manera certera que no existan incoherencias entre las normas legales analizadas.

Este trabajo explora el uso del núcleo de insatisfacibilidad (unsat core) para refinar el análisis de FormaLex. Se documenta una modificación que se puede realizar dentro del model checker NuSMV (invocado por FormaLex para validar el modelo generado), en conjunto con el SAT solver *Picosat*, para que se pueda analizar el unsat core de la fórmula booleana generada al realizar BMC, contribuyendo así a detectar contradicciones entre fórmulas normativas.

Palabras claves: FormaLex, BMC, NuSMV, unsat core, SAT problem, Picosat.

Índice general

1. Introducción **4**

1.1. La herramienta FormaLex 4

1.1.1. La lógica como base de la verificación de textos normativos 4

1.1.2. El proceso de verificación por dentro 6

1.2. SAT y UNSAT Core 7

1.3. Picosat 8

1.4. Bounded Model Checking 8

1.5. Compact Clause Conversion 9

1.6. NuSMV Model Checker 10

2. Trabajo realizado **12**

2.1. Investigación sobre NuSMV 12

2.2. El problema 13

2.3. Trabajo realizado sobre el NuSMV 17

2.3.1. Las estructuras de datos 18

2.3.2. Las modificaciones en el código 20

2.3.3. Generando el resultado deseado 25

3. Conclusiones **30**

4. Anexo **32**

Capítulo 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 similares 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 que sirven como asistentes al momento de redactar una norma con el fin de facilitar la verificación. FormaLex [1] es una de ellas, la cual es el resultado de un amplio trabajo que viene atancando 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[1] y su lenguaje basada en [1].

1.1.1. 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

[1]Las secciones a continuación están basadas en la tesis de grado de Carlos Faciano acerca de una optimizacion sobre FormaLex [2]

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) [3]. 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, 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 [2] (CTDs, del inglés ContraryTo-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.

- 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 razonablemente maduras. Han pasado varias décadas desde que surgieron como prototipos, y el tamaño de los modelos que son capaces de manejar, a crecido significativamente en ese tiempo, acercandose así cada vez mas a casos reales. 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 estabilizarse y obtener resultados.

El lenguaje está compuesto por dos partes

- Un conjunto de reglas, que son fórmulas con operadores deónticos para expresar

[2] 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".

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 $P(\varphi) \not\equiv \neg F(\varphi)$, por ejemplo:

"Prohibido hablar dentro de la biblioteca"

"Permitido hablar dentro de la sala parlante"

"Es obligatorio devolver los libros antes de retirarse"

- 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

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 φ, 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.

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

1.1.2. El proceso de verificación por dentro

Por cada especificación se realizan varios chequeos. 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 $\tau \models \neg\varphi$, 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 $\neg\varphi$. Si encuentra un contraejemplo, ese contraejemplo es una violación de $\neg\varphi$ y por ende satisface φ. Es decir, se trata de un "testigo" de que existe al menos una forma de cumplir todas las reglas bajo la teoría marco y por ende de que no hay contradicciones.

Si vamos al caso particular del model checker NuSMV [4], vamos a encontrar que se utiliza un SAT solver al que se le pasa una codificación del modelo NuSMV en forma de un problema SAT. Esto se verá con más detalle en la sección 1.6.

Cuando el SAT solver no puede resolver el modelo SAT significa que no pudo encontrar un contrajemplo (en términos del model checker), lo que a su vez significa que no hay un comportamiento legal en términos de FormaLex. Como no hay contraejemplo, no hay información que permita saber cuáles de las normas involucradas provocaron la contradicción que llevó a que no hubiera modelo legal. Es en este momento cuando se vuelve de interés poder entender cómo funciona el SAT solver para tratar de recuperar alguna información que permita entender cuáles son las variables involucradas. Poder realizar ese análisis presenta un grado de dificultad elevado, ya que no se sabe qué variables se corresponden con las normas del lenguaje. Para eso es necesario tener acceso a la relacion entre las variables del modelo y las utilizadas por el SAT solver. Este tema se abordará en el capítulo 2.

1.2. SAT y UNSAT Core

El problema de satisfacibilidad booleana, o lo que se conoce comúnmente como **SAT** [5], es el problema de determinar si existe alguna interpretación que satisfaga una fórmula booleana dada. Una interpretación es la asignación de valores que se le da a un conjunto de variables lógicas. Dicho de otra manera, se intenta descubrir si asignando valores de **TRUE** o **FALSE** a las variables de la fórmula, se puede lograr que ésta sea verdadera. Se dice entonces que la fórmula es *satisfacible*. En contraparte, si no existe una asignación de valores posible que hagan que la fórmula sea verdadera, se dice que es *insatisfacible*. Entonces, se puede definir a SAT como el problema de verificar si una fórmula es satisfacible o no. Por lo general, las fórmulas están expresadas en forma normal conjuntiva (CNF), lo que facilita su análisis. En lógica booleana, una fórmula está en CNF si corresponde a una conjunción de cláusulas, donde una cláusula es una disyunción de literales, donde un literal y su complemento no pueden aparecer en la misma cláusula. Por ejemplo, la siguiente fórmula se encuentra expresada en CNF

$$(X_1 \vee Y_1) \wedge (X_2 \vee Y_2) \wedge (Z_1 \vee \neg Z_2) \wedge (\neg X_2 \vee X_1)$$

Con X_i, Y_i, Z_i literales. Es importante aclarar que toda fórmula booleana tiene una equivalente expresada en CNF.

Fue uno de los primeros problemas en demostrarse que pertenecía al conjunto de problemas NPcompleto, por lo que no se conoce un algoritmo que lo pueda resolver en tiempo polinomial. En consecuencia, se desarrollaron muchas heurísticas y casos particulares del problema lo que permitió que sea aplicado en varios campos de la computación, como pueden ser la inteligencia artifical, diseño de circuitos o pruebas de teoremas automatizadas.

Cuando una fórmula es *insatisfacible*, es deseable encontrar las cláusas causantes de la insatisfacibilidad. Se quiere encontrar un **unsat core**. El **unsat core** se define como el núcleo de la insatisfacibilidad, es decir, un conjunto de clausulas que si no estuvieran como parte de la fórmula, harían que la fórmula si sea satisfacible. En particular, se quiere encontrar el **unsat core** que además sea minimal (es decir, un conjunto al que si le quitamos cualquier elemento pierde su condición de insatisfacible).

1.3. Picosat

La definición de SAT generó la creación de técnicas y algoritmos enfocados en dar soluciones válidas al problema. Mas adelante, se comenzaron a crear programas cuya meta era resolver instancias del problema, cada uno utilizando diferentes técnicas y procesos. Uno de ellos es el **Picosat** [6].

Picosat es un desarrollo de un grupo de investigación de la JKU (Johannes Kepler University, Sede Linz), y es un intento de optimizar la performance de bajo nivel en comparación con otros SAT solvers, y a la fecha continuan generando nuevas versiones del programa, agregando funcionalidad y mejorandolo.

A diferencia de varios SAT solvers, Picosat tiene la particularidad que permite realizar la extracción de núcleos de clausulas dada una instancia de SAT. Gracias a esto, es posible generar un conjunto de clausulas (el core propiamente dicho) de una formula booleana expresada en CNF. Para este trabajo en particular, sólo interesan las variables involucradas en ése conjunto, más que las cláusulas en sí.

1.4. Bounded Model Checking

Como se dijo anteriormente, model checking (MC) es una técnica que se utiliza para validar modelos de sistemas reactivos. Dada una especificación de un sistema (denominada modelo), un verificador es capaz de encontrar errores o contradicciones del mismo utilizando diferentes técnicas. La especificación del problema es expresada utilizando lógicas temporales (no significa que toda técnica de MC lo haga, pero es el caso en este trabajo), y el sistema es modelado por una maquina de estados finita (FSM).

De la misma forma en la que se pueden validar propiedades en un modelo, también es posible encontrar contraejemplos para propiedades inválidas. Decir que una propiedad o fórmula f es válida dentro del modelo M es decir que f es válida en todas las trazas de la FSM del modelo. Entonces, si f es inválida, el verificador nos proporciona una traza de estados del sistema que hacen que f no se cumpla.

Una particularidad de MC es que la cantidad de estados que puede llegar a tener la máquina hacen que el análisis del modelo sea muy costoso, bordeando lo imposible. A medida que se agregan componentes al sistema, la cantidad de estados de la FSM pueden aumentar de manera exponencial, por lo que fue necesario desarrollar nuevas técnicas de MC. *Symbolic Model Checking* [7], es una variante del MC tradicional. Gracias a que utiliza un diagráma de decisión binario (BDD) [8], es capáz de analizar sistemas con 10^{20} estados diferentes [9], siendo una gran ventaja por sobre el resto de las técnicas.

Pese a todo ese avance, los BDD generados para sistemas cada vez mas complejos seguían siendo lo suficientemente grandes para el poder de cómputo disponible.

Los procesos de decisión proposicional (como se mostró en 1.2) no tienen la necesidad de recurrir a generar BDDs, además de que no sufren del crecimiento exponencial característico de otras herramientas en su cantidad de estados. Son capaces de resolver problemas de satisfacibilidad de magnitudes mucho mayores y de manera más eficiente, a costa de sacrificar capacidad expresiva. Sin embargo, y pese a ésta perdida, muchos problemas son tratables con la capacidad que brindan estas herramientas. De esta idea es de donde surge el *Bounded Model Checking* [10].

La idea detrás de BMC es la siguiente: Dados el modelo y una propiedad, considerar contraejemplos de longitud k y generar una fórmula proposicional que sea satisfacible si y solo si dicho contraejemplo existe. La longitud máxima que puede alcanzar un contraejemplo representa el bound (límite) que se esta analizando. Una de las ventajas de BMC es que gracias a utilizar los SAT solvers para evaluar si la fórmula es satisfacible o no, es mucho mas rápido que en muchos otros casos, aunque, sin embargo, cabe mencionar que una gran desventaja de utilizarlo como técnica para encontrar contraejemplos es que está acotado por el valor de k, lo que hace que modelos de mayor maginitud no sean tenidos en cuenta para la solución.

1.5. Compact Clause Conversion

Como se mencionó en la sección 1.2, la Forma Normal Conjuntiva (CNF) es una manera en la cual una expresión booleana puede estar escrita. Cada fórmula proposicional puede convertirse en una fórmula equivalente que esté en CNF. Esta transformación se basa en reglas sobre equivalencias lógicas: la doble negación, las leyes de De Morgan, y la distributividad [11].

Sin embargo, hay algunos casos en que dicha conversión puede producir un crecimiento

exponencial del tamaño de la fórmula. Mientras más grande sea la fórmula, más costoso será analizar si es satisfacible o no.

Una de las soluciones a este problema es la *conversión compacta de cláusulas* [12]. Realiza una transformación de las cláusulas internas de una fórmula booleana, reemplazando algunas de ellas por variables que no aparezcan previamente en la fórmula, manteniendo la equivalencia entre la original y la transformada. Sin entrar en los detalles técnicos de cómo la conversión realiza el reemplazo de variables, cabe destacar que como en el fondo se está trabajando con SAT solvers (un programa que intenta resolver el problema SAT), solo se lleva a cabo el reemplazo si la fórmula (o sub-fórmula en este caso) tiene una menor cantidad de cláusulas que la original.

El resultado de la conversión es una fórmula equivalente a la anterior, con un número menor de clausulas, lo que hace que el análisis sea mucho más rápido y eficiente.

1.6. NuSMV Model Checker

NuSMV [4] es una reimplementación y extensión del verificador de modelos simbólico **SMV**, la primera herramienta de validación basada en diagramas de decisión binaria (BDD). NuSMV fue diseñado como una arquitectura abierta para verificación de modelos. Apunta a una validación confiable de diseños de tamaño industrial, para ser usado como un sistema de fondo para otras herramientas de verificación y también como una herramienta para la investigación de nuevas técnicas de validación de modelos. Fue desarrollado como un projecto en conjunto de ITC-IRST (Istituto Trentino di Cultura in Trento, Italia), Universidad Carnegie Mellon, la Universidad de Genoa y la Universidad de Trento.

La función principal del NuSMV es verificar que cierta propiedad (o un conjunto de ellas) sea válida dentro de un determinado modelo. Una mejora en su nueva versión es la integración efectiva de técnicas basadas en satisfacibilidad proposicional (SAT), sumadas al conjunto previo de técnicas basadas en BDD. Ambas técnicas pueden ser vistas como metodologías complementarias, ya que representan diferentes clases de problemas. Internamente, viene configurado con dos SAT solvers para poder realizar el SAT model checking: CHAFF [13] y SIM [14].

NuSMV es capaz de procesar archivos escritos en una extensión del lenguaje del **SMV**. Dicho proceso consta de diferentes etapas. Las primeras fases son las que se encargan de cargar los datos del archivo en una manera ordenada y prolija para la construcción del modelo. Se comienza por la descripción de un modelo M y un conjunto de propiedades $P_1, ..., P_n$. El primer paso, denominado *flattening* (aplanamiento), es el proceso por el cual se produce un nuevo modelo M^f en donde cada variable tiene un nombre absoluto (es decir, es identificada unequivocamente). El segundo paso, llamado *boolean encoding* (codificación booleana) es el encargado de mapear el modelo aplanado a un modelo booleano $M^f b$, en donde las variables escalares fueron reemplazadas por variables booleanas. Los mismos

pasos son aplicados sobre las propiedades P_i, obteniendo así las versiones aplanadas y booleanas ${P_i}^{fb}$.

Todo ese preproceso puede hacerse independientemente del motor utilizado para hacer la verificación del modelo. Para el caso de verificación basada en BDD, NuSMV genera una representación basada en BDD de la máquina de estados finita, para después utilizar diferentes tipos de verificación.

En el caso de verificación basada en SAT, NuSMV genera una representación del modelo basada en una forma simplificada de Circuitos Booleanos Reducidos (RBC), con el cual es capaz de realizar *Bounded Model Checking* sobre alguna fórmula previamente definida. Dada una cota (*bound*) para la longitud del contraejemplo, NuSMV convierte el problema de validar la proposición en un problema SAT. Si se encuentra un modelo válido para SAT, corresponde al contrajemplo para el modelo original. Para por utilizar este modo de verificación basado en SAT, se le debe proveer a NuSMV la cota deseada, es decir, el tamaño máximo del contraejemplo buscado.

Cada problema SAT, es representado por un RBC internamente, por lo que se convierte a formato CNF (valga la redundancia) y se lo pasa como argumento al SAT solver interno. Aternativamente, también se cuenta con la opción de exportar los resultados intermedios en formato CNF a un archivo de salida en formato **DIMACS** [15]. El formato *DIMACS* es una forma de conveniente en la cual se pueden escribir fórmulas booleanas que se encuentran en CNF. Es el formato por defecto para los parametros de entrada de los SAT solvers.

Gracias a la posibilidad de poder obtener resultados intermedios, es posible procesar las fórmulas del modelo con otros SAT solvers diferentes a los que posee NuSMV, como puede ser el caso de Picosat.

Capítulo 2

Trabajo realizado

2.1. Investigación sobre NuSMV

El objetivo final que se intentaba alcanzar era lograr poder armar una relación entre las variables del unsat core y las variables del modelo original.

Como punto de partida del trabajo se realizaron pruebas con modelos básicos para entender qué resultados daba NuSMV y qué archivos eran posible generar a partir de un problema. Se crearon diferentes modelos a probar, y las propiedades que se querían validar eran siempre incorrectas, para que de esa manera se puediese encontrar un contraejemplo válido. Tomando como base los tutoriales del programa, se definió la siguiente secuencia de pasos:

1. Leer la especificación del modelo desde un archivo de texto plano.
2. Configurar y construir una representación booleana del modelo utilizando, con una base de BDD. Esto lo hace automaticamente NuSMV al leer el modelo.
3. Indicarle al **NuSMV** que funcione en modo BMC.
4. Elegir el conjunto de fórmulas que se desean validar.
5. Exportar el resultado en un archivo de formato *dimacs*
6. Utilizando **Picosat**, se realiza el core extraction de la fórmula booleana del paso anterior.
7. Con las variables del *unsat core* y las del encabezado del archivo *dimacs*, ya se puede armar la relación deseada.

Los pasos enumerados se realizan de manera automática con el uso de scripts de desarrollo propio y Picosat.

2.2. El problema

Al realizar los pasos listados en la sección anterior, se notó que aparecían variables dentro del **unsat core** que no pertenecían al encabezado del archivo generado por **NuSMV**.

Para poder explicar los motivos por los cuales este problema ocurre, se muestra a continuación un ejemplo de un modelo que NuSMV acepta como entrada:

Figura 2.1: Modelo ejemplo

```
1  MODULE main
2  VAR
3       y : 0..15;
4       z : 0..2;
5  ASSIGN
6       init(y) := 0;
7       init(z) := 0;
8  TRANS
9       case
10           y = 7 : next(y) = 0;
11           TRUE : next(y) = ((y + 1) mod 16);
12           z = 0 : next(z) = 0;
13      esac
14
15 LTLSPEC NAME alpha := F(y = 2 | z = 3)
```

Este modelo cuenta con 2 variables z e y, que toman los valores de 0 a 2 y de 0 a 15, respectivamente. El siguiente valor de y está dado por la por la ecuación

TRUE : next(y) = ((y + 1) mod 16);

La variable incrementará su valor al siguiente, excepto cuando alcance el valor 7, caso en el cual reinicia desde 0. El único valor que tiene z para cualquier estado del modelo es 0.

La fórmula *alpha* expresa que en toda traza de ejecución del modelo, se alcanza un estado en el que o bien y tiene valor 2, o z tiene valor 3. Se puede ver que *alpha* es trivialmente válida, ya que sin importar que z nunca alcanze el valor 3, existe un estado dentro de la FSM en el cual y alcanza el valor 2.

Como ya se explicó en el capítulo anterior, NuSMV es capaz de devolver los resultados intermedios en archivos de formato **dimacs**. Dichos archivos cuentan con un encabezado que está formado por comentarios en donde se muestra la relación entre las variables del problema con las variables de la formula booleana expresada en forma normal conjuntiva

(CNF). Es necesario indicarle a NuSMV cuál debe ser la cota por la cuál realizar **BMC** para generar este archivo (además de la propiedad a validar), por lo que se eligió a 3 como la longitud máxima de traza que puede utilizar el validador (i.e $\mathbf{k = 3}$). Recordar que internamente NuSMV negará la propiedad elegida para ver si es capaz de encontrar un contraejemplo válido.

Veamos un ejemplo del encabezado del archivo **dimacs** que se corresponde con el modelo previamente mostrado

Figura 2.2: Dimacs Header

```
1   c @@@@@ Time 0
2   c CNF variable 13 => Time 0, Model Variable y.3
3   c CNF variable 14 => Time 0, Model Variable y.2
4   c CNF variable 15 => Time 0, Model Variable y.1
5   c CNF variable 16 => Time 0, Model Variable y.0
6   c CNF variable 17 => Time 0, Model Variable z.1
7   c CNF variable 18 => Time 0, Model Variable z.0
8   c
9   c @@@@@ Time 1
10  c CNF variable 19 => Time 1, Model Variable y.3
11  c CNF variable 20 => Time 1, Model Variable y.2
12  c CNF variable 21 => Time 1, Model Variable y.1
13  c CNF variable 22 => Time 1, Model Variable y.0
14  c CNF variable 23 => Time 1, Model Variable z.1
15  c CNF variable 24 => Time 1, Model Variable z.0
16  c
17  c @@@@@ Time 2
18  c CNF variable 25 => Time 2, Model Variable y.3
19  c CNF variable 26 => Time 2, Model Variable y.2
20  c CNF variable 27 => Time 2, Model Variable y.1
21  c CNF variable 28 => Time 2, Model Variable y.0
22  c CNF variable 29 => Time 2, Model Variable z.1
23  c CNF variable 30 => Time 2, Model Variable z.0
24  c
25  c @@@@@ Time 3
26  c CNF variable 31 => Time 3, Model Variable y.3
27  c CNF variable 32 => Time 3, Model Variable y.2
28  c CNF variable 33 => Time 3, Model Variable y.1
29  c CNF variable 34 => Time 3, Model Variable y.0
30  c CNF variable 45 => Time 3, Model Variable z.1
31  c CNF variable 46 => Time 3, Model Variable z.0
```

Un análisis del encabezado permite ver que cada variable del modelo esta asociada a una variable de la fórmula generada. La relación completa se puede ver en el cuadro 2.1.

Notar que donde $X.i$ representa a la variable X del modelo (en el ejemplo z o y) en el estado i. Notar que i no alcanza un valor mayor a 3, ya que esa fue la cota elegida.

El resto del archivo **dimacs** detalla la fórmula booleana generada para el modelo. Cada línea del archivo está formada por una secuencia de valores numéricos que representan

Variable (dimacs)	Variable Modelo
13	y.3
14	y.2
15	y.1
16	y.0
17	z.1
18	z.0
19	y.3
20	y.2
21	y.1
22	y.0
23	z.1
24	z.0
25	y.3
26	y.2
27	y.1
28	y.0
29	z.1
30	z.0
31	y.3
32	y.2
33	y.1
34	y.0
45	z.1
46	z.0

Cuadro 2.1: Relación entre variables del modelo y de la fórmula

una cláusula y las variables involucradas en la misma, cada una con su signo propio. Por ejemplo, veamos algunas líneas del archivo **dimacs** generado para el modelo de la figura 2.1

-46	-45	-55	0
-28	54	-55	0
27	54	-55	0

se deben intepretar como las cláusulas

$$(-46 \vee -45 \vee -55 \vee 0) \wedge (-28 \vee -54 \vee -55 \vee 0) \wedge (-27 \vee -54 \vee -55 \vee 0)$$

Como ahora se cuenta con el modelo expresado como una única fórmula booleana, es posible realizar una extracción del **unsat core** de la misma. La motivación detrás de esto es poder discernir qué variables del modelo se corresponden con las variables que aparecen dentro del núcleo de insatisfacibilidad.

Continuando con el ejemplo y partiendo del archivo **dimacs** generado por NuSMV, es posible encontrar las variables involucradas en el unsat core del modelo, como se listan a continuación (sin tener en cuenta el signo en el cual aparecen en la fórmula del núcleo)

19 20 21 22 25 26 27 28 37 39 40 49 55

Acá es donde nos encontramos con el problema en intentar hacer el camino inverso al que hace NuSMV: Dentro del unsat core aparecen variables que no pertenecen al encabezado previamente calculado, por lo que no es posible diferenciar que variables del modelo están involucradas en la insatisfacibilidad de la fórmula. En la cuadro 2.1 se puede ve que no hay entrada válida para la variable '49', pero la misma pertenece al **unsat core** (como se mostró en el listado anterior).

Como se explicó en la sección 1.5, NuSMV realiza una conversión de las variables antes de formar la fórmula en CNF, para mejorar los tiempos de ejecución y encontrar soluciones más rápidamente. Estas variables que aparecen en el *unsat core* son el resultado de dicho proceso. No hay resultado intermedio u opción dentro de la configuración del NuSMV que nos permita averiguar qué conjunto de variables se corresponden con las que aparecieron.

2.3. Trabajo realizado sobre el NuSMV

En la sección 2.2 se explicó cuál es la naturaleza del problema a resolver: Se desea encontrar la relación entre las variables presentes dentro del **unsat core** que no tienen entrada alguna dentro de la cabecera del archivo **dimacs** generado por NuSMV. Estas variables son generadas por la conversión compacta que realiza el validador (como se explicó en la sección 1.5), para poder mejorar el tiempo en el que se encuentra una solución, por lo que es necesario hallar el segmento de código fuente dentro de NuSMV en donde

se realiza dicho reemplazo. Posteriormente, se debe modificar el programa para que la relación entre las variables del modelo y las de la fórmula en CNF sean visibles por el usuario del programa en el resultado final.

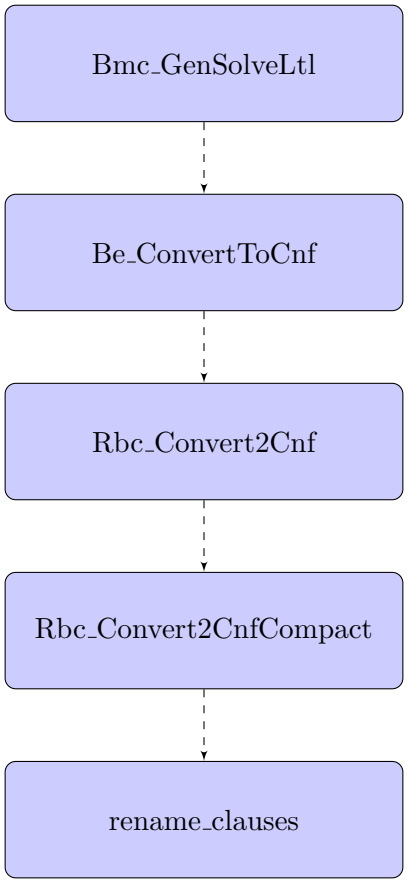
2.3.1. Las estructuras de datos

Vale primero comentar cuáles son las estructuras de datos principales a la hora de intentar validar una propiedad dentro del modelo. Conceptualmente, representan momentos distintos en la resolución del problema.

Figura 2.3: RbcManager

```
1  struct RbcManager {
2    NuSMVEnv_ptr environment;
3    Dag_Manager_t* dagManager;
4    Rbc_t** varTable;
5    int varCapacity;
6    Rbc_t* one;
7    Rbc_t* zero;
8
9    LRUCache_ptr inlining_cache;
10
11   /* splitted cache mapping in two sets (model, cnf) */
12   hash_ptr rbcNode2cnfVar_model;
13   hash_ptr rbcNode2cnfVar_cnf;
14
15   hash_ptr cnfVar2rbcNode_model;
16   hash_ptr cnfVar2rbcNode_cnf;
17
18   int maxUnchangedRbcVariable;
19   int maxCnfVariable;
20
21   int stats[RBCMAX_STAT];
22 };
```

Como su nombre lo indica, el **RbcManager** representa al modelo (junto con sus propiedades a evaluar) como RBC. Aún no está expresado en CNF y todavía no se hizo la conversión de las variables. A continuación, se puede ver la sucesivas llamadas hasta llegar a la mencionada conversión



A lo largo de todo el proceso hasta llegar al modelo en forma normal conjuntiva, el **RbcManager** mantendrá las modificaciones que se vayan realizando hasta poder cargar todo el contexto del problema en la siguiente estructura.

Figura 2.4: Representación del problema como una fórmula en CNF

```
1  typedef struct Be_Cnf_TAG {
2    be_ptr originalBe; /* the original BE problem */
3    Slist_ptr cnfVars;     /* The list of CNF variables */
4    Slist_ptr cnfClauses; /* The list of CNF clauses */
5    int     cnfMaxVarIdx;  /* The maximum CNF variable index */
6
7    /* literal assigned to whole CNF formula. (It may be negative)
8       If the formula is a constant, see Be_Cnf_ptr. */
9    int formulaLiteral;
10
11 } Be_Cnf;
```

La estructura **Be_Cnf** representa al problema como una fórmula booleana en CNF. La sustitución de variables ya fue realizada, y ya se puede pasar a la siguiente etapa en la cual un SAT solver intentará comprobar la satisfacibilidad de la misma, dependiendo de la configuración que se le haya dado al model checker (acepta tanto la conversión compacta mostrada en 1.5 como la de Tseiten [16]). En este punto es donde también, si se desea, se puede imprimir el resultado intermedio en un archivo **dimacs**. En términos de contexto de ejecución, la estructura **Be_Cnf** es utilizada después que el **RbcManager**.

2.3.2. Las modificaciones en el código

Antes de encontrar en qué segmento de código se realiza la conversión de las variables, se realizaron modificaciones a las estructuras de datos comentadas en la sección anterior. Tanto para la estructura **Be_Cnf**, como para el **RbcManager**, se incluyó en la declaración de sus elementos internos la siguiente definición

```
1 /* Set of renamed variables in the conversion */
2 Sset_ptr renamed_values;
```

Sset es una implementación de la interfaz *conjunto ordenado* dentro del proyecto NuSMV. Fue desarrollado como parte de las estructuras utilitarias de sistema, y está incluida en el código fuente del mismo. Internamente está definido como un árbol AVL, que es un árbol de búsqueda balanceado en altura y que conserva esa propiedad a medida que se le agregan elementos. Tiene un orden de búsqueda, inserción y borrado de *O(log(n))*, lo cual lo hace una estructura muy útil. Al tratarse de un conjunto ordenado, permite que la inserción sea un par **(clave, significado)**, y que se pueda recuperar el contenido total del mismo recorriendo el conjunto de llaves y obteniendo sus significados. Por último, **Sset_ptr** es un renombre de puntero a **Sset**.

La idea detrás de *renamed_values* es que mantenga un registro de los reemplazos de variables que se van haciendo a lo largo del proceso de crear la fórmula en CNF. Se agregaron dentro de cada implementanción de las estructuras los métodos necesarios para inicializar, borrar y los set/get como parte de la interfaz pública de la misma.

Para poder enteder cómo y por qué funcionan los cambios en el código que se hicieron, veamos como comienza la secuencia de llamadas hasta llegar a la función que realiza la sustitución de las variables

Figura 2.5: Función Be_ConvertToCnf

```
1  cnf = Be_Cnf_Create(f);
2  max_var_idx = Rbc_Convert2Cnf(GET_RBC_MGR(manager),
3                          RBC(manager, f),
4                           polarity, alg,
5                           Be_Cnf_GetClausesList(cnf),
6                           Be_Cnf_GetVarsList(cnf),
7                          &literalAssignedToWholeFormula);
8
9  nusmv_assert(literalAssignedToWholeFormula >= INT_MIN);
10 rbc_mgr = GET_RBC_MGR(manager);
11 Be_Cnf_SetConvertedValues(cnf,
12                                  Sset_copy(rbc_mgr->renamed_values));
```

Lo importante a destacar en este segmento de código es que se está inicializando la variable *cnf* de tipo **Be_Cnf**, en donde se guardara el problema en el formato del mismo nombre (como se ve en la figura 2.4). En la variable *manager* se encuentra el *RbcManager* (figura 2.3), responsable de representar el problema hasta que éste pase a la variable *cnf*. La función **Rbc_Convert2Cnf** hace la conversion entre el problema representado en RBC y el problema representado en CNF. En la última línea del segmento se ve como se hace una copia del conjunto de variables reenombradas y se almacena dentro de *cnf*.

El siguiente punto a comentar ocurre dentro de **Rbc_Convert2Cnf**. Dejando de lado la definición entera de la función, las líneas pertinentes son

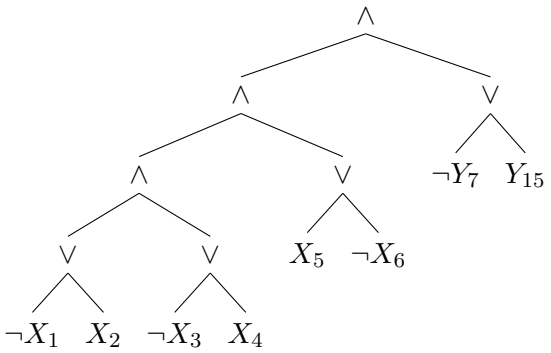
```
1  case RBC_SHERIDAN_CONVERSION:
2    result = Rbc_Convert2CnfCompact(rbcManager,
3                                    f, polarity,
4                                    clauses, vars,
5                                    literalAssignedToWholeFormula);
6    break;
```

Estamos dentro del caso de la conversión SHERIDAN[1], y se llama a **Rbc_Convert2CnfCompact** para que termine de generar el problema en CNF, además de realizar el reemplazo de las variables. Los parámetros *clauses* y *vars* son pasadas por referencia para poder almacenar los resultados de la conversión ahí mismo.

[1]La conversión de SHERIDAN es otra forma de llamar a la conversión compacta, ya que fue creada por Daniel Sheridan [12]

Rbc_Convert2CnfCompact es el segmento de código responsable de recorrer el árbol que representa a la fórmula e ir calculando los cambios de variables posibles junto con la polaridad misma de cada subcláusula y realizar dicha modificación[2]. Internamente tiene funciones que lo asisten para hacer un recorrido utilizando la técnica de *DFS* (Depth First Search) a lo largo del espacio representado. Luego, para cada nodo recorrido y de acuerdo a la polaridad de su nodos hijos, evalúa si puede hacer un reempleazo de las variables almacenadas dentro de ellos por una nueva variable.

A modo de ejemplo, supongamos que contamos con el siguiente árbol binario

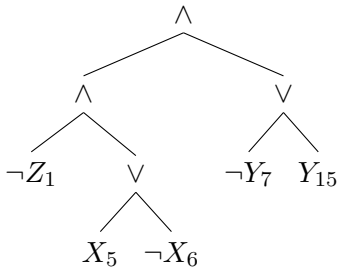


Se puede ver que la cláusula representada es

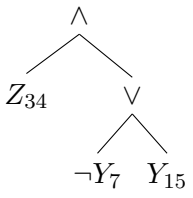
$$(\neg X_1 \vee X_2) \wedge (\neg X_3 \vee X_4) \wedge (X_5 \vee \neg X_6) \wedge (\neg Y_7 \vee Y_{15})$$

y que se encuentra en CNF.

Si el algoritmo decide que puede realizarse una conversión de variables que mantenga la satisfacibilidad de la fórmula, va a realizar dicha conversión. Entonces puede suceder que el árbol sea modificado con una nueva variable como se ve a continuación



Esta nueva cláusula puede ser reducida aun más



[2]La **polaridad** de una fórmula booleana puede ser positiva o negativa, siendo la última antepuesta por el símbolo ¬

Lo que se quiere lograr es poder almacenar de alguna manera la relación entre las variables

$$\neg Z_1 = (\neg X_1 \vee X_2) \wedge (\neg X_3 \vee X_4) \tag{2.1}$$

$$Z_{34} = \neg Z_1 \wedge (X_5 \vee \neg X_6) \tag{2.2}$$

En caso de que sea posible realizar una conversión de variables, se llama a la función de la figura 2.6

Figura 2.6: Función rename clauses

```
1  static void rename_clauses(ClgManager_ptr clgManager,
2                             clause_graph* clauses, int var,
3                             clause_graph* saved)
4  {
5    clause_graph clause, lit;
6
7    nusmv_assert(0 != var);
8
9    lit = Clg_Lit(clgManager, -var);
10   clause = Clg_Disj(clgManager, lit, *clauses);
11   *saved = Clg_Conj(clgManager, *saved, clause);
12   *clauses = Clg_Lit(clgManager, var);
13 }
```

clgManager es el encargado de representar los grafos directos que representan a las cláusulas. Los autores definen al **clgManager** como: *Estructura de datos compacta para representar los conjuntos de cláusulas que comparte una estructura en común. Es un grafo de conjunciones y disjunciones que son convertidas utilizando la conversion a CNF estandar para obtener las cláusulas requeridas.* Dicha extructura permite la creación de nuevos árboles que tengan un nodo raíz de conjuncion o de disjunción, tomando 2 grafos como hijos izquierdo y derecho respectivamente. El propósito de la función es renombrar un set de cláusulas, y almacenarlas en el parametro *saved*.

Finalmente nos encontramos en la sección de código fuente (figura 2.6) de NuSMV que se ocupa de renombrar las cláusulas según el criterio previamente mencionado.

Figura 2.7: Función modificada para guardar cambios de variables

```
1
2  static void rename_clauses(ClgManager_ptr clgManager,
3                             clause_graph* clauses, int var,
4                             clause_graph* saved,
5                             Sset_ptr renamed_clauses)
6  {
7    clause_graph clause, lit;
8    lsList list;
9    lsGen gen;
10   lsGeneric data;
11
12   nusmv_assert(0 != var);
13
14   list = lsCreate();
15   Clg_Get_Labels(*clauses, &list);
16   Sset_insert(renamed_clauses,
17                  PTR_FROM_INT(Sset_key, -var),
18                  list);
19
20   lit = Clg_Lit(clgManager, -var);
21   clause = Clg_Disj(clgManager, lit, *clauses);
22   *saved = Clg_Conj(clgManager, *saved, clause);
23   *clauses = Clg_Lit(clgManager, var);
24 } /* End of rename_clauses. */
```

En la figura 2.7, se puede ver como el parámetro *renamed_clauses* es el conjunto en el cuál se van guardando los resultados intermedios para cada nodo visitado. Se inicializa una lista vacía y se la llena con etiquetas almacenadas en *clauses* llamando a la función **Clg_Get_Labels** que recorre el árbol de nodos, y si no es un nodo que representa un **and** o un **or**, inserta la etiqueta al final de *list* (el código completo se puede ver en la figura 2.8). Finalmente se agrega al conjunto *renamed_clauses* el par **(-var, list)**, ya que ése es el nuevo literal creado para reemplazar a las variables en *clauses*.

Figura 2.8: Función Clg_Get_Labels

```
1  void Clg_Get_Labels(clause_graph node, lsList *l)
2  {
3      int status;
4      int *labelCopy;
5      if (node == NULL)
6          return;
7
8      Clg_Get_Labels(node->left, l);
9
10     if(10 != node->label && 11 != node->label)
11     {
12         labelCopy = ALLOC(int, 1);
13         nusmv_assert(labelCopy != (int*) NULL);
14         *labelCopy = node->label;
15         // label is not and "AND" nor a "OR";
16         status = lsNewEnd(*l, (lsGeneric)labelCopy, LS_NH);
17     }
18
19     Clg_Get_Labels(node->right, l);
20 }
```

Finalmente, y volviendo al código de la figura 2.5, podemos afirmar que ya contamos con una representación del problema en forma de una fórmula booleana en CNF, y que además se cuenta con un conjunto en donde están almacenadas las variables que representan el dominio y la imagen de la conversión compacta realizada.

2.3.3. Generando el resultado deseado

Como ya se explicó en la sección anterior, ya contamos con una estructura de datos en donde ya se evaluo la satisfacibilidad de las fórmulas del modelo, y además se tiene las variables que reemplazaron a cada cláusula dentro de la conversión compacta. Lo que falta por hacer es lograr que esa información sea transmitida al usuario de alguna manera.

La decisión que se tomó al respecto fue la de agregar la relación entre las nuevas variables y las clausulas del problema al archivo **dimacs**. Como se explicó en la sección 1.6, el archivo **dimacs** es un posible resultado intermedio que genera NuSMV para mostrar la fórmula booleana en CNF. Posee además una primera sección comentada (denominada *encabezado*) en donde se explica que variables del archivo se corresponden con las del

problema. Un ejemplo del encabezado es la figura 2.2.

Gracias a los cambios realizados en el código fuente de NuSMV (detallados en la sección 2.3.2), ahora queda almacenada dentro de la estructura **Be_Cnf** la relación entre variables del modelo y variables de la fórmula booleana. Lo que resta por hacer es un último cambio en el código para que dicha informacón se escriba en el archivo de salida.

Figura 2.9: Función Bmc_Dump_DimacsProblem

```
1  conversion = (Sset_ptr) Be_Cnf_GetConvertedValues(cnf);
2  fprintf(dimacsfile, "c @@@@@ Converted values \n");
3  for (set_iter = Sset_first(conversion);
4          Ssiter_is_valid(set_iter) ;
5          set_iter = Ssiter_next(set_iter))
6  {
7    fprintf(dimacsfile,
8              "c converted %d : ",
9              PTR_TO_INT(Ssiter_key(set_iter))
10   lvalues = (lsList) Ssiter_element(set_iter);
11
12   if (lsFirstItem(lvalues, &data, LS_NH) == LS_OK) {
13       fprintf(dimacsfile, "[%d", *((int*) data));
14   }
15
16   gen = lsStart(lvalues);
17   // We already printed the first element
18   lsNext(gen, &data, LS_NH);
19   while (lsNext(gen, &data, LS_NH) == LS_OK) {
20       fprintf(dimacsfile, ", %d", *((int*) data));
21   }
22   fprintf(dimacsfile, "]\n");
23   lsFinish(gen);
24 }
```

Se puede ver la figura 2.9 cómo primero se inserta un comentario para indicar que las líneas a seguir muestran la relación entre las variables del modelo y las clausulas a las cuales reemplazaron. Luego, para cada par dentro del conjunto de cláusulas reemplazadas, se imprime la nueva variable y la lista de cláusulas original. Cabe aclarar que una variable creada por NuSMV (es decir, que no formaba parte del modelo) puede formar parte de una cláusula que es cambiada por una nueva variable.

Tomemos nuevamente el ejemplo de la figura 2.1. Gracias a las modificaciones realizadas dentro del código fuente, es posible generar nuevo contenido dentro del archivo **dimacs**, como se puede ver en la figura 2.10

Figura 2.10: Nuevo contenido del archivo **dimacs**

```
1  c @@@@@ Converted values
2  c converted -55 : [-18, -17, -16, -15, -13, -14, 19, -20, 21, 22, 25,
       ↪ -26, 27, 28, 48, 47, -46, -45, -34, -33, -31, -32, -36, 19, -20,
       ↪  -21, -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20,
       ↪ 21, -22, 49, -50, -50, -25, -26, -27, 28, -31, -32, -33, -34,
       ↪ 25, 26, 27, -28, 54, -46, -45]
3  c converted -54 : [-25, -26, -32, -27, -33, 28, 34, -28, -34, 27, 51,
       ↪ 26, 53, -31, 25, 31, -26, 53, 26, -32, 52]
4  c converted -53 : [32, 52]
5  c converted -52 : [-27, 51, 27, -33, 44]
6  c converted -51 : [33, 44]
7  c converted -49 : [-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21,
       ↪ 38, 20, 40, 19, 25, -20, 40, 20, -26, 39]
8  c converted -48 : [-25, 31, 25, -31, -26, 32, 26, -32, -27, 33, 27,
       ↪ -33, 44, -29, 45, 29, -45, -30, 46, 30, -46]
9  c converted -47 : [-19, 31, 19, -31, -20, 32, 20, -32, -21, 33, 21,
       ↪ -33, -22, 34, 22, -34, -23, 45, 23, -45, -24, 46, 24, -46]
10 c converted -46 : [46]
11 c converted -45 : [45]
12 c converted -44 : [-28, 34, 28, -34]
13 c converted -43 : [-18, -17, -16, -15, -13, -14, -36, 19, -20, -21,
       ↪ -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20, 21,
       ↪ -22, 41, -29, -30, 19, -20, 21, 22, 42, -30, -29, -28, -27, -25,
       ↪  -26]
14 c converted -42 : [37, -19, 25, 19, -25, -20, 26, 20, -26, -21, 27, 21,
       ↪  -27, -23, 29, 23, -29, -24, 30, 24, -30]
15 c converted -41 : [-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21,
       ↪ 38, 20, 40, 19, 25, -20, 40, 20, -26, 39]
16 c converted -40 : [26, 39]
17 c converted -39 : [-21, 38, 21, -27, 37]
18 c converted -38 : [27, 37]
19 c converted -37 : [-22, 28, 22, -28]
20 c converted -35 : [-23, -24, 19, -20, -21, -22, -24, -23, -22, -21,
       ↪ -19, -20, -18, -17, -16, -15, -13, -14]
21 c converted -34 : [34]
22 c converted -33 : [33]
23 c converted -32 : [32]
24 c converted -31 : [31]
25 c converted -28 : [28]
```

```
26 c converted -27 : [27]
27 c converted -26 : [26]
28 c converted -25 : [25]
29 c converted -22 : [22]
30 c converted -21 : [21]
31 c converted -20 : [20]
32 c converted -19 : [19]
33 c converted 19 : [-19]
34 c converted 20 : [-20]
35 c converted 21 : [-21]
36 c converted 22 : [-22]
37 c converted 23 : [-23]
38 c converted 24 : [-24]
39 c converted 25 : [-25]
40 c converted 26 : [-26]
41 c converted 27 : [-27]
42 c converted 28 : [-28]
43 c converted 29 : [-29]
44 c converted 30 : [-30]
45 c converted 31 : [-31]
46 c converted 32 : [-32]
47 c converted 33 : [-33]
48 c converted 34 : [-34]
49 c converted 36 : [-23, -24]
50 c converted 45 : [-45]
51 c converted 46 : [-46]
52 c converted 50 : [-29, -30]
```

Gracias a la nueva salida del NuSMV y al **unsat core** extraído del modelo, es posible generar la relacion completa entre todas las variables involucradas, como se puede ver en el cuadro 2.2.

En conclusión: en el modelo mostrado en la figura 2.1 pudimos ver cómo se define un modelo para NuSMV. La verificación sobre la fórmula *alpha* se realiza utilizando tecnicas que definen una fórmula booleana en CNF, como se mostró anteriormente. Dicha fórmula tiene un unsat core con variables que no aparecen en el encabezado del archivo **dimacs**. Ahora podemos ver cómo dichas variables del unsat core aparecen dentro de las líneas del archivo **dimacs** modificado.

Variable **unsat core**	Variables **dimacs** que reemplaza	Variables del modelo involucradas
37	-22, 28, 22, -28	y
39	-21, 38, 21, -27, 37	y
40	26, 39	y
49	-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21, 38, 20, 40, 19, 25, -20, 40, 20, -26, 39	y
55	-18, -17, -16, -15, -13, -14, 19, -20, 21, 22,1 25, -26, 27, 28, 48, 47, -46, -45, -34, -33, -31, -32, -36, 19, -20, -21, -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20, 21, -22, 49, -50, -50, -25, -26, -27, 28, -31, -32, -33, -34, 25, 26, 27, -28, 54, -46, -45	y

Cuadro 2.2: Nueva relación entre variables del unsat core del modelo

c converted -39 : [-21, 38, 21, -27, 37]
c converted -38 : [27, 37]
c converted -37 : [-22, 28, 22, -28]
c converted -40 : [26, 39] c converted -49 : [-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21, 38, 20, 40, 19, 25, -20, 40, 20, -26, 39]
c converted -55 : [-18, -17, -16, -15, -13, -14, 19, -20, 21, 22, 25, -26, 27, 28, 48, 47, -46, -45, -34, -33, -31, -32, -36, 19, -20, -21, -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20, 21, -22, 49, -50, -50, -25, -26, -27, 28, -31, -32, -33, -34, 25, 26, 27, -28, 54, -46, -45]

Lo cual quiere decir que cada nueva variable surge de la conversión de cláusulas que involucran a las variables listadas, y que a su vez dichas variables se corresponden con la variable y en diversos estados de la FSM del modelo original. Se puede ver cómo la variable z es dejada de lado completamente, aunque también forma parte de la fórmula *alpha*. Por ende, la variable y del modelo de NuSMV tiene que ver con la no existencia, o al menos la imposibilidad de hallar, un contraejemplo. Esa conclusión indica que el problema de no encontrar una solución a la validación del modelo está relacionado con las fórmulas en las que interviene dicha variable.

Se puede observar el archivo **dimacs** completo en el capítulo 4.

Capítulo 3

Conclusiones

FormaLex es una herramienta que permite realizar un análisis lógico de proposiciones legales. En este trabajo se realizó una introducción al aspecto téorico de la herramienta, además de un detalle técnico de cómo funciona la herramienta cuando nos encontramos ante fórmulas legales que se contradicen entre si. Dado que FormaLex utiliza internamente a un model checker que, a su vez, utiliza técnicas SAT, en este marco se intenta solucionar la problemática relacionada con la validación de las normas legales provistas.

Se mostró como NuSMV es capaz de generar resultados que pueden ser tomados por un sat solver (**Picosat**, por ejemplo) para ser evualuados de forma separada. El problema nace al notar que NuSMV genera nuevas variables a la hora de definir la fórmula booleana en CNF, y que éstas se corresponden con varias del modelo original, pero no hay conocimiente acerca de cuales son. Esto sucede porque NuSMV realiza internamente una conversión de variables, denominada conversión compacta, que tiene como fin bajar los costos computacionales de hallar la forma normal conjuntiva (CNF) de la nueva fórmula booleana.

Como primera parte del trabajo fue necesario investigar cómo NuSMV genera el archivo con la fórmula en CNF y cómo podía ser tomada por **Picosat** para realizar la extracción de unsat core. Una vez logrado ese objetivo, se realizó la modificación del código fuente de NuSMV para poder almacenar los reemplazos de las variables realizados y poder hacer llegar dicha información al usuario del sistema.

Gracias a esas modificaciones, ahora es posible formar un vínculo directo entre las variables del unsat core de la fórmula del archivo **dimacs** y las variables del modelo representado. Esto permite hacer un análisis más profundo de las propiedades del modelo, además de generar un indicio de cuáles son las variables involucradas que hacen que no sea posible encontrar un contraejemplo de la propiedad, pudiendo generar nuevas pruebas con mayor enfoque e información.

Como trabajo futuro se buscará integrar la versión modificada de NuSMV a la herramienta FormaLex de manera tal de que puedan recuperarse las variables involucradas en

las fórmulas que hacen que un modelo legal no sea satisfacible.

Capítulo 4

Anexo

Figura 4.1: Archivo **dimacs** completo

```
1  c BMC problem generated by NuSMV
2  c Time steps from 0 to 3, 6 State Variables, 0 Frozen Variables and 0
       ↪ Input Variables
3  c Model to Dimacs Conversion Table
4  c
5  c @@@@@ Time 0
6  c CNF variable 13 => Time 0, Model Variable y.3
7  c CNF variable 14 => Time 0, Model Variable y.2
8  c CNF variable 15 => Time 0, Model Variable y.1
9  c CNF variable 16 => Time 0, Model Variable y.0
10 c CNF variable 17 => Time 0, Model Variable z.1
11 c CNF variable 18 => Time 0, Model Variable z.0
12 c
13 c @@@@@ Time 1
14 c CNF variable 19 => Time 1, Model Variable y.3
15 c CNF variable 20 => Time 1, Model Variable y.2
16 c CNF variable 21 => Time 1, Model Variable y.1
17 c CNF variable 22 => Time 1, Model Variable y.0
18 c CNF variable 23 => Time 1, Model Variable z.1
19 c CNF variable 24 => Time 1, Model Variable z.0
20 c
21 c @@@@@ Time 2
22 c CNF variable 25 => Time 2, Model Variable y.3
23 c CNF variable 26 => Time 2, Model Variable y.2
24 c CNF variable 27 => Time 2, Model Variable y.1
25 c CNF variable 28 => Time 2, Model Variable y.0
26 c CNF variable 29 => Time 2, Model Variable z.1
27 c CNF variable 30 => Time 2, Model Variable z.0
28 c
```

```
c @@@@@ Time 3
c CNF variable 31 => Time 3, Model Variable y.3
c CNF variable 32 => Time 3, Model Variable y.2
c CNF variable 33 => Time 3, Model Variable y.1
c CNF variable 34 => Time 3, Model Variable y.0
c CNF variable 45 => Time 3, Model Variable z.1
c CNF variable 46 => Time 3, Model Variable z.0
c
c Beginning of the DIMACS dumping
c model 24
c 24 23 46 30 45 29 34 33 32 31 28 27 26 25 22 21 20 19 14 13 15 16 17
    ↪ 18 0
p cnf 55 105
55 0
-46 -45 -55 0
-28 54 -55 0
27 54 -55 0
26 54 -55 0
25 54 -55 0
-34 -25 -26 -27 28 -55 0
-33 -25 -26 -27 28 -55 0
-32 -25 -26 -27 28 -55 0
-31 -25 -26 -27 28 -55 0
-50 -55 0
-50 -55 0
-22 49 -55 0
21 49 -55 0
20 49 -55 0
19 49 -55 0
-28 -19 -20 -21 22 -55 0
-27 -19 -20 -21 22 -55 0
-26 -19 -20 -21 22 -55 0
-25 -19 -20 -21 22 -55 0
-36 -55 0
-22 -55 0
-21 -55 0
-20 -55 0
19 -55 0
-36 -55 0
-32 47 48 -55 0
-31 47 48 -55 0
-33 47 48 -55 0
-34 47 48 -55 0
-45 47 48 -55 0
```

```
 72 | −46  47  48  −55  0
 73 | 25  −26  27  28  −55  0
 74 | 19  −20  21  22  −55  0
 75 | −14  −55  0
 76 | −13  −55  0
 77 | −15  −55  0
 78 | −16  −55  0
 79 | −17  −55  0
 80 | −18  −55  0
 81 | 52  26  25  −54  0
 82 | −32  26  25  −54  0
 83 | −26  53  25  −54  0
 84 | 31  25  −54  0
 85 | −31  −25  −54  0
 86 | 26  53  −25  −54  0
 87 | 27  51  −26  −25  −54  0
 88 | −28  −34  −27  −26  −25  −54  0
 89 | 28  34  −27  −26  −25  −54  0
 90 | −33  −27  −26  −25  −54  0
 91 | −32  −26  −25  −54  0
 92 | 52  −53  0
 93 | 32  −53  0
 94 | 44  27  −52  0
 95 | −33  27  −52  0
 96 | −27  51  −52  0
 97 | 44  −51  0
 98 | 33  −51  0
 99 | 50  −29  −30  0
100 | 39  20  19  −49  0
101 | −26  20  19  −49  0
102 | −20  40  19  −49  0
103 | 25  19  −49  0
104 | 20  40  −19  −49  0
105 | 21  38  −20  −19  −49  0
106 | −22  −28  −21  −20  −19  −49  0
107 | 22  28  −21  −20  −19  −49  0
108 | −27  −21  −20  −19  −49  0
109 | −26  −20  −19  −49  0
110 | −25  −19  −49  0
111 | 39  −40  0
112 | 26  −40  0
113 | 37  21  −39  0
114 | −27  21  −39  0
115 | −21  38  −39  0
```

```
116 37 -38 0
117 27 -38 0
118 22 -28 -37 0
119 -22 28 -37 0
120 36 -23 -24 0
121 30 -46 -48 0
122 -30 46 -48 0
123 29 -45 -48 0
124 -29 45 -48 0
125 44 -48 0
126 27 -33 -48 0
127 -27 33 -48 0
128 26 -32 -48 0
129 -26 32 -48 0
130 25 -31 -48 0
131 -25 31 -48 0
132 24 -46 -47 0
133 -24 46 -47 0
134 23 -45 -47 0
135 -23 45 -47 0
136 22 -34 -47 0
137 -22 34 -47 0
138 21 -33 -47 0
139 -21 33 -47 0
140 20 -32 -47 0
141 -20 32 -47 0
142 19 -31 -47 0
143 -19 31 -47 0
144 28 -34 -44 0
145 -28 34 -44 0
146 c @@@@@ Converted values
147 c converted -55 : [-18, -17, -16, -15, -13, -14, 19, -20, 21, 22, 25,
    ↪ -26, 27, 28, 48, 47, -46, -45, -34, -33, -31, -32, -36, 19, -20,
    ↪  -21, -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20,
    ↪ 21, -22, 49, -50, -50, -25, -26, -27, 28, -31, -32, -33, -34,
    ↪ 25, 26, 27, -28, 54, -46, -45]
148 c converted -54 : [-25, -26, -32, -27, -33, 28, 34, -28, -34, 27, 51,
    ↪ 26, 53, -31, 25, 31, -26, 53, 26, -32, 52]
149 c converted -53 : [32, 52]
150 c converted -52 : [-27, 51, 27, -33, 44]
151 c converted -51 : [33, 44]
152 c converted -49 : [-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21,
    ↪ 38, 20, 40, 19, 25, -20, 40, 20, -26, 39]
```

```
153 c converted -48 : [-25, 31, 25, -31, -26, 32, 26, -32, -27, 33, 27,
      ↪ -33, 44, -29, 45, 29, -45, -30, 46, 30, -46]
154 c converted -47 : [-19, 31, 19, -31, -20, 32, 20, -32, -21, 33, 21,
      ↪ -33, -22, 34, 22, -34, -23, 45, 23, -45, -24, 46, 24, -46]
155 c converted -46 : [46]
156 c converted -45 : [45]
157 c converted -44 : [-28, 34, 28, -34]
158 c converted -43 : [-18, -17, -16, -15, -13, -14, -36, 19, -20, -21,
      ↪ -22, -36, -19, -20, -21, 22, -25, -26, -27, -28, 19, 20, 21,
      ↪ -22, 41, -29, -30, 19, -20, 21, 22, 42, -30, -29, -28, -27, -25,
      ↪  -26]
159 c converted -42 : [37, -19, 25, 19, -25, -20, 26, 20, -26, -21, 27, 21,
      ↪  -27, -23, 29, 23, -29, -24, 30, 24, -30]
160 c converted -41 : [-19, -25, -20, -26, -21, -27, 22, 28, -22, -28, 21,
      ↪ 38, 20, 40, 19, 25, -20, 40, 20, -26, 39]
161 c converted -40 : [26, 39]
162 c converted -39 : [-21, 38, 21, -27, 37]
163 c converted -38 : [27, 37]
164 c converted -37 : [-22, 28, 22, -28]
165 c converted -35 : [-23, -24, 19, -20, -21, -22, -24, -23, -22, -21,
      ↪ -19, -20, -18, -17, -16, -15, -13, -14]
166 c converted -34 : [34]
167 c converted -33 : [33]
168 c converted -32 : [32]
169 c converted -31 : [31]
170 c converted -28 : [28]
171 c converted -27 : [27]
172 c converted -26 : [26]
173 c converted -25 : [25]
174 c converted -22 : [22]
175 c converted -21 : [21]
176 c converted -20 : [20]
177 c converted -19 : [19]
178 c converted 19 : [-19]
179 c converted 20 : [-20]
180 c converted 21 : [-21]
181 c converted 22 : [-22]
182 c converted 23 : [-23]
183 c converted 24 : [-24]
184 c converted 25 : [-25]
185 c converted 26 : [-26]
186 c converted 27 : [-27]
187 c converted 28 : [-28]
188 c converted 29 : [-29]
```

```
c converted 30 : [-30]
c converted 31 : [-31]
c converted 32 : [-32]
c converted 33 : [-33]
c converted 34 : [-34]
c converted 36 : [-23, -24]
c converted 45 : [-45]
c converted 46 : [-46]
c converted 50 : [-29, -30]
c End of dimacs dumping
```

Bibliografía

[1] María Celeste Gunski and Melisa Gabriela Raiczyk. *Mejorando el poder expresivo del lenguaje FL para la detección de defectos normativos*. PhD thesis, Buenos Aires, Argentina, 2016. Tesis de Licenciatura.

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

[3] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA, 2006.

[4] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv2:an open source tool for symbolic model checking nusmv 2: an opensource tool for symbolic model checking. 2002.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

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

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

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

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on*

Tools and Algorithms for Construction and Analysis of Systems, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[11] S.K. Sobolev. Conjuntive Formal Form. http://www.encyclopediaofmath.org/index.php?title=Conjunctive_normal_form&oldid=35078, 2008.

[12] Daniel Sheridan. The Optimality of a Fast CNF Conversion and its Use with SAT. *In SAT (SAT04)*, 2004.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[14] Enrico Giunchiglia, Massimo Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, pages 347–363, London, UK, UK, 2001. Springer-Verlag.

[15] International Conference on Theory and Applications of Satisfiability Testing. Submission format. http://www.satcompetition.org/2009/format-benchmarks2009.html, 2009.

[16] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.