Tesis de Licenciatura

Una generalización de Unsat Core para algoritmos de Sat-Solving basados en DPLL

Tesista:
Esteban Lanzarotti
L.U.: 301/03

Directores:
Lic. Juan Pablo Galeotti
Dr. Sergio Mera

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

Abstract

Identificar el núcleo de insatisfactibilidad (**unsat core**) de un modelo **Alloy** es bastante útil en varios escenarios [SSJ+03]. En este trabajo se extiende este concepto al **hot core**, una aproximación heurística al **unsat core** que permite al usuario obtener información cuando el proceso **Sat-Solving** de **Alloy** es interrumpido. El hecho de que verificar especificaciones usando SAT sea *NP-Completo* hace de **hot core** una herramienta muy interesante, dado que es bastante frecuente que el usuario interrumpa el proceso por haber excedido el tiempo de espera. Exhibimos resultados experimentales satisfactorios que validan nuestros propósitos y muestran el buen funcionamiento de la heurística.

Índice general

1. Introducción

Intuitivamente, la *expresividad* de un lenguaje formal puede entenderse como la capacidad que éste tiene para describir o *especificar* conceptos. La semántica de un lenguaje define los objetos con los que trabaja el lenguaje, y le da significado a los diferentes operadores que podemos combinar a la hora de escribir fórmulas. Por ejemplo, en el caso de la lógica proposicional, los símbolos atómicos que se emplean son proposiciones y los podemos combinar con operadores lógicos (*no*, *y*, *o*, *entonces* con a interpretación usual). Con esta lógica es posible expresar sentencias sencillas, como ser:

A = "Si *llueve* **entonces** *está nublado*".

Podemos decir que $p = llueve$ y $q = está\ nublado$, y usando la palabra "entonces" como el conectivo lógico de implicación($\Rightarrow$), nos quedaría la sentencia proposicional $A = p \Rightarrow q$. Todo lo expresable en lógica proposicional es lo que se puede poner en términos de proposiciones ($p = llueve$, $q = esta\ nublado$, ...) y conectándolas con conectivos lógicos. Pero sucede que hay razonamientos que no son expresables usando la lógica proposicional y en general suele pasar que, para cualquier lenguaje dado, existen conceptos imposibles de ser expresados en él. Por ejemplo, supongamos que queremos expresar la siguiente afirmación usando lógica proposicional,

B = "**Todos** los *días que llueve*, *está nublado*".

Podriamos decir que $p' = Todos\ los\ días\ que\ llueve$ y $q = está\ nublado$, para construir la sentencia propocisional $B = p' \Rightarrow q$. Pero se presenta un problema; supongamos que aparece una afirmación como la siguiente,

C = *"Hoy llueve* **y no** *está nublado"*.

Podriamos expresar esta afirmación con la sentencia proposicional $C = r \wedge \neg q$ con $r = Hoy\ llueve$ y nos interesaría que la sentencia $B \wedge C$ sea una *contradicción*, dado que las proposiciones *"Todos los días que llueve, esta nublado"* y *"Hoy llueve y no está nublado"* se contradicen. Pero, intuitivamente, se puede ver que no hay nada que relacione p' con r, y ambas proposiciones pueden tomar valores de verdad arbitrarios. Esta relación no es expresable en lógica proposicional. Esto se debe a que con esta lógica no podemos razonar sobre todos los días lluviosos y nublados, sino que el modificador **todos** queda "encapsulado" dentro de la proposición p' dejándonos sin la posibilidad de expresar que p' y r hablan de lo mismo. La lógica proposicional sólo puede representar hechos acerca del mundo como si fueran bloques monolíticos. Para poder expresar esta relación necesitamos otra lógica, una que provea operadores con mayor expresividad. En este caso, deberíamos poder representar los días cómo si fueran *objetos* independientes entre sí y poder cuantificarlos. Por ejemplo, con la lógica de primer orden, es posible describir el mundo usando objetos y propiedades (o predicados) que se aplican sobre estos objetos. Podemos definir el predicado $Llueve(d)$ que es verdadero si en el día d llueve y $Nublado(d)$ si el día d está nublado, y las proposiciones B y C se expresarían como B' y C' a continuación:

$$B' = (\forall d)(Llueve(d) \Rightarrow Nublado(d))$$
$$C' = Llueve(hoy) \wedge \neg Nublado(hoy)$$

Ahora sí, con estas construcciones es posible verificar que B' y C' no pueden valer simultáneamente. La lógica de primer orden agrega a la lógica proposicional cuantificadores, como el cuantificador *universal* ($\forall$), que produce un aumento en la expresividad del lenguaje. Con él, podemos hablar de todos los elementos de un determinado conjunto, en este caso, el conjunto de todos los días.

En general, cuanto más expresivo es un lenguaje, más complejos son los conceptos que se pueden describir usándolo, por lo que para definir y resolver ciertos problemas, se vuelve indispensable usar lenguajes con un alto poder expresivo.

Ahora bien, dado una lenguaje, es interesante preguntarse si es posible usar una computadora para verificar la validez de los enunciados que se pueden expresar usándolo. En el caso de la lógica proposicional, determinar el valor de verdad de una fórmula es bastante intuitivo, simplemente se prueban todas las combinaciones posibles de valores de verdad para cada proposición y se calcula el valor de verdad de la fórmula según se conecte cada proposición con los conectivos lógicos. Por ejemplo, probar cada valor de verdad para las proposiciones de la fórmula $p \Rightarrow ((\neg q) \wedge r)$ daría como resultado:

p	$\Rightarrow$	$((\neg$	$q)$	$\wedge$	$r)$
0	**1**	1	0	0	0
0	**1**	1	0	1	1
0	**1**	0	1	0	0
0	**1**	0	1	0	1
1	**0**	1	0	0	0
1	**1**	1	0	1	1
1	**0**	0	1	0	0
1	**0**	0	1	0	1

la fórmula vale verdadero para algunos valores de las proposiciones y para otros no.

Pero sucede que la complejidad de verificar la validez de las fórmulas de un lenguaje dado suele aumentar conforme aumenta la expresividad. En algunos casos, la lógica es suficientemente expresiva como para que el problema de determinar la validez de una fórmula cruce la frontera de la decidibilidad, siendo teóricamentente imposible encontrar un algoritmo que compute dichas verificaciones, o como también suele decirse, la lógica es indecidible. Este es el caso de la lógica de primer orden [Chu36] [Tur36].

Cuando el problema de decidir la validez de una sentencia de una lógica dada no es computable, se suelen buscar formas alternativas de aproximar el problema. Por ejemplo, si bien la lógica de primer orden no es decidible, se desarrollaron demostradores que funcionan bien en la práctica por más que no haya garantías de que el procedimiento termine para el caso general. Otra alternativa es tomar fragmentos de la lógica que se demuestren decidibles, por ejemplo, imponiendo límites a los "recursos" que provee el lenguaje. Por ejemplo, uno de estos recursos

pueden ser las variables; podríamos poner límites a la cantidad de variables de una fórmula, como ser usar sólo dos variables para poder expresar conceptos en lógica de primer orden. Esto da como resultado un lenguaje conocido como $\mathbf{LPO}^2$, que es decidible [Mor75] [Sco62]. De esta manera, es posible hacer uso de operadores con mayor expresividad y por más que no podamos hacer uso del lenguaje en su totalidad, podemos expresar y resolver problemas que con lógicas de menor expresividad no hubiera sido posible.

1.1. Alloy

En esta tesis trabajamos con Alloy [Jac06], un lenguaje relacional. Con **Alloy** podemos especificar una teoría T que describa un *universo* y una *propiedad* P para verificar si se deduce de T (si $T \Rightarrow P$). Resolver un problema usando **Alloy** consiste en la búsqueda de un *contramodelo* para la propiedad P. Llamamos contramodelo a un elemento del universo descripto por la teoría que no cumple con la propiedad, o sea un elemento que cumpla $T \wedge \neg P$. De esta forma si encontramos dicho contramodelo queda demostrado que $T \not\Rightarrow P$, por otro lado si podemos probar que no existe, entonces $T \Rightarrow P$.

Pero sucede que la lógica de **Alloy** no es decidible. Una forma de tratar este problema es tomar un fragmento decidible de ella. La idea a vuelo de pájaro de este fragmento es definir una cota para el tamaño del universo descripto por la teoría y así, la busqueda del contramodelo se realiza sobre un número finito de modelos. Restringir el tamaño máximo de modelos permite la existencia de una traducción (que describiremos más adelante) de las sentencias expresadas en este fragmento de alloy a sentencias de la lógica prosicional. La ventaja de poder traducir este fragmento de Alloy a la lógica proposicional es que esta última es decidible. Aún así, el problema de decidir la validez de una fórmula proposicional es *NP-Completo* [Coo71]. Esta complejidad puede resultar elevada en algunos contextos, pero se han desarrollado verificadores automáticos llamados *SAT-Solvers* que están bastante optimizados usando heurísticas que para el común de los casos funcionan bastante bien.

Verificar sobre universos finitos obliga al usuario a tomar cierto compromiso en las verificaciones automáticas, dado que si no se encuentra un contramodelo para el universo acotado, no se pueden tener garantías de la validez de la propiedad a verificar. Esto sucede dado que puede ser que exista algún contramodelo para un universo de mayor tamaño. De todas formas, se usa este esquema porque si una propiedad tiene contramodelo, en general, como menciona Daniel Jackson en [Jac06], éste suele aparecer en universos de tamaño chico.

Si un determinado análisis no encuentra un contramodelo para $T \Rightarrow P$, puede ser por dos razones:

- $T \Rightarrow P$ realmente vale aunque **Alloy** no nos de garantías, o
- $T \Rightarrow P$ tiene un contramodelo, pero en un universo de mayor tamaño.
- Hay un error en la especificación. T es muy fuerte o P es muy débil, o ambas.

Una funcionalidad interesante que provee la herramienta que realiza las verificaciones para **Alloy** (**Alloy Analyzer**) cuando el contramodelo no es encontrado,

es la posibilidad de obtener el núcleo de insatisfactibilidad (*unsat core*) una vez terminado el proceso de verificación. El unsat core es el conjunto (posiblemente minimal) de las porciones de la especificación que implican que no es posible encontrar un contramodelo para la propiedad que se quiere verificar. La idea detrás de esta funcionalidad es facilitar el desarrollo de las especificaciónes mostrando las razones por las cuales no se puede encontrar un contramodelo. Así, el usuario puede tener mas información para sobrellevar el compromiso asumido a la hora de acotar el universo y, si la especificación es muy extensa, facilita la tarea de revisar la porción de especificación que conforma el unsat core en el caso que sospeche que $T \Rightarrow P$ puede no valer.

El problema con el que nos encontramos ahora es que aún siendo posible su verificación automática, el proceso de solving puede ser intratable por los recursos que tengamos disponibles. Estos recursos involucran tanto al tiempo del que disponemos para esperar los resultados como por la memoria física de la computadora que usamos para llevar a cabo las verificaciones. El objetivo de este trabajo es estudiar el proceso de verificación a nivel de la lógica proposicional para producir cierto *feedback* en caso de que el proceso sea abortado por el usuario porque superó el tiempo máximo de espera. La idea es usar las métricas que usa el SAT-Solver para valerse de información útil a la hora de identificar el núcleo de posible insatisfactibilidad al cual llamaremos *hotcore*. Intuitivamente el hotcore es una aproximación al unsat core.

Este trabajo se organiza presentando en el segundo capítulo una breve introducción a la sintaxis y la semántica de **Alloy**. En el tercer capítulo se hace un breve resumen de cómo se traducen las propiedades de **Alloy** a lógica proposicional para efectuar su verificación. Luego, en el cuarto capítulo, se habla de SAT, se describe el problema y el *state of the art* de los solvers actuales ejemplificando con el que usamos para llevar a cabo la implementación del *hotcore*: **Minisat** [ES03]. En el quinto capítulo se formaliza la noción de unsat core para la verificación de una propiedad especificada usando **Alloy** y para una fórmula de la lógica proposicional que sea insatisfasible y se explica cómo se implementa su obtención en **Minisat**. En el sexto capítulo se describe la implementación del hotcore y se exhibe el *benchmark* con el que mostramos su funcionamiento. Finalmente en el séptimo capítulo se discuten las conclusiones y se habla del trabajo a futuro.

2. Alloy

Alloy [Jac06] es un lenguaje relacional muy usado para modelar sistemas de software y verificar propiedades sobre ellos. Su simplicidad, su naturaleza *orientada a objetos* y su entorno de verificación automática (**Alloy Analyzer**) hicieron de este lenguaje un gran exponente para satisfacer la necesidad de verificar especificaciones. Como dijimos en la introducción, con todo lenguaje formal es posible expresar conceptos a partir de la descripción de estructuras u objetos matemáticos. En el caso de **Alloy**, todos los objetos sobre los cuales podemos hablar son relaciones. Los conjuntos se definen como relaciones unarias y los escalares como conjuntos de un solo elemento (*singletones*). A su vez, los lenguajes tienen operadores que permiten construir expresiones mas complejas de forma inductiva. Los operadores de **Alloy** son operadores para usar sobre relaciones. Por ejemplo, el operador de navegación (`.`), nos permite expresar nociones que son expresables en lógica de primer orden pero de forma más sintética. Por ejemplo, se puede expresar que un nodo m de un grafo es vecino de un nodo n con la fórmula $m \in n.E$, donde E es la relación que define los ejes del grafo y $n.E$ es la imagen de n sobre la relación E. **Alloy** tambien tiene un operador de clausura transitiva ($\hat{}$), que lo ubica expresivamente por encima de la lógica de primer orden. Con él podemos expresar si existe un camino entre dos nodos n y m de un grafo con la fórmula $m \in n.\hat{E}$.

Para facilitar la descripción de **Alloy** como lenguaje, exhibiremos a modo de ejemplo la especificación de un sistema de archivos. A lo largo de esta sección, ampliaremos sucesivamente la especificación para mostrar los posibles usos del lenguaje, describir su sintaxis y presentar los operadores. Sin embargo, sólo mostraremos una breve reseña, para más detalles acerca de **Alloy** referirse a [Jac06].

2.1. Sintaxis y semántica

La forma de definir los objetos con los que trabaja **Alloy**, es mediante la definición de *signaturas*. En el ejemplo de la Figura 1 definimos los objetos `Archivo` y `Directorio` usando **sig**. Esto significa que tanto `Archivo` como `Directorio` son conjuntos disjuntos (para **Alloy**, relaciones unarias) con elementos que son archivos y directorios respectivamente. Cada elemento carece de una identidad definida con respecto a los otros elementos del mismo conjunto y la cantidad de elementos en cada conjunto dependerá del tamaño que elijamos para cada uno a la hora de verificar alguna propiedad.

Dentro de cada signatura es posible definir atributos. Los atributos son interpretados como relaciones que contienen tuplas de la forma $(r_0, r_1, \ldots, r_k)$, donde la primer componente (r_0) corresponde a la signatura sobre la cual está definido el atributo y las demas componentes $(r_1, \ldots, r_k)$ corresponden a los elementos que especifica su declaración. En nuestro ejemplo `contenidos` $\subseteq$ (`Directorio` $\times$ (`Archivo` $\cup$ `Directorio`)), con una cantidad variable de elementos expresada por el modificador `set`. Para definir la cardinalidad de estas relaciones se pueden usar modificadores como:

```
// Hay Archivos.
sig Archivo{}

// Hay Directorios.
sig Directorio{
        // Los directorios contienen más archivos y directorios.
        contenidos: set Archivo + Directorio
}

// El directorio Raíz.
one sig Root extends Directorio{}
```

Figura 1. Modelado en **Alloy** de archivos y directorios: Signaturas

- `set`: Para decir que puede haber uno o más elementos en la relación.
- `one`: Para que pueda haber un solo elemento.
- `lone`: Para a lo sumo uno.
- `some`: Para uno o más elementos.

También es posible definir subsignaturas como es el caso de `Root`. Usando **extends** podemos declarar signaturas que están contenidas dentro de otras y con **one** definimos singletones. En nuestro ejemplo, `Root` es un conjunto con un solo elemento, incluido dentro del conjunto de directorios. Las subsignaturas o subtipos heredan también los atributos definidos en las clases que extienden y es posible redefinirlos en las nuevas declaraciones.

Como venimos mencionando, además de los objetos que usamos para razonar y expresar conceptos, cada lenguaje posee un conjunto de operadores. A continuación enumeramos los operadores de **Alloy** que son de más interés:

- *Navegación o Junta* (`.`): con el operador de junta podemos referirmos al conjunto de todos los archivos y directorios que hay dentro de un determinado directorio `d`, como `d.contenidos`.
- *Clausura Transitiva* (`^`) y *Clausura Reflexo-transitiva* (`*`): con la clausura transitiva podemos expresar, por ejemplo, el conjunto de todos los archivos y directorios a los que se puede llegar desde `Root` como `Root.^contenidos`. La clausura reflexo-transitiva contiene a la transitiva y además incluye a la relación identidad.
- *Traspuesta* (`~`): con este operador podemos referirnos al directorio donde está un determinado archivo `a` como `a.~contenidos`, navegando la traspuesta de la relación que refiere a los contenidos de todos directorios, a través del archivo `a`.
- Operadores booleanos de la lógica proposicional: *Negación*(`!` o `not`), *Conjunción*(`&&` o `and`), *Disyunción*(`||` o `or`), *Implicación* (`=>` o `implies`) y *Si y solo si* (`<=>` o `iff`).
- Operaciones de la teoría de conjuntos: *Unión*(`+`), *Intersección*(`&`), *Resta*(`-`), *Inclusión y Pertenencia*(`in`), *Cardinalidad*(`#`), *Igualdad*(`=`) y *Desigualdad*(`!=`)

- Cuantificadores de primer orden: *Universalidad*(`all`), *Vacuidad*(`no`), *Existencia*(`some`), *Unicidad*(`one`), *Al menos uno*(`lone`).

Una vez que definimos la signatura de los objetos que participan en la definición del sistema que estamos especificando, debemos imponer ciertas restricciones. No todos los modelos del sistema que planteamos, definiendo solo las relaciones, satisfacen los aspectos deseados. Por ejemplo, no estaría bien pensar que podemos llegar desde el directorio `Root`, navegando a través de los contenidos de sus subdirectorios, otra vez al directorio `Root`. En general, la relación `contenidos` debe ser *acíclica*. Esto se puede describir en **Alloy**, como se puede ver en la Figura 2, usando `fact` para agregar *axiomas* o *hechos* que deben valer para todo modelo de la teoría.

```
sig Archivo{}
sig Directorio{
      contenidos: set Archivo + Directorio
}
one sig Root extends Directorio{}

// La navegación por directorios es aciclica.
fact { no ^contenidos & iden }

// Un archivo o un Directorio está contenido si pertenece a los
// contenidos de uno y solo un directorio.
pred contenido(ad:Archivo+Directorio) { one ad.~contenidos }

// Todos los archivos y directorios con excepción del root
// deben estar contenidos.
fact { all ad: Archivo + Directorio - Root | contenido[ad] }

// Root no está contenido en ningún directorio.
fact { no contenidos.Root }
```

Figura 2. Modelado en **Alloy** de archivos y directorios: Hechos

Expresamos la aciclicidad de una relación diciendo que su clausura tiene intersección vacía con la relación `iden`. La relación `iden`, es una constante que provee el lenguaje para hacer referencia a la relación identidad: la relación que sólo contienen los pares (x, x) para cada elemento x perteneciente al universo.

También, quisiéramos expresar que todos los archivos y directorios, excepto el directorio `Root`, deben estar contenidos en un y sólo un directorio. Para eso definimos el predicado `contenido`, usando `pred`, que expresa dicho concepto para cualquier archivo o directorio. Luego construimos con él un hecho que enuncia que todos los archivos y directorios excepto el `Root` cumplen con dicha propiedad.

Por último, restrigimos que Root no esté contenido en ningún directorio, diciendo que no hay elementos en el conjunto de los directorios que contienen a

root.

En definitiva, todo modelo expresado en **Alloy** se puede estructurar usando:

- Definiciones de las Signaturas: usando `sig` podemos definir las entidades sobre las cuales vamos a predicar, tal como lo hicimos en el ejemplo de la Figura 1.
- Definiciones de Predicados, Funciones y Hechos: usando `fact` podemos escribir fórmulas, en lógica relacional, que caracterizan las restricciones que deseamos que tenga nuestro modelo. A su vez, podemos definir predicados y funciones auxiliares con `pred` y `fun`.
- Aserciones, Validaciones y Ejecuciones: usando los comandos `check` y `run` podemos poner a prueba propiedades lo cual será explicado en breve. También podemos usar `assert` para definir predicados especiales (aserciones) para hacer validaciones.

Con estas estructuras que provee el lenguaje podemos, ahora sí, expresar conceptos bastante interesantes. Las signaturas con sus respectivas restricciones describen la existencia y el comportamiento de las entidades que deseamos modelar. La pregunta ahora es: estas entidades cumplen con las propiedades que garantizan la correcta interpretación del problema que nos interesa resolver?

2.2. El análisis

Cuando hablamos de poner a prueba propiedades, nos referimos a que podemos escribir fórmulas usando la lógica relacional que provee **Alloy** y validarlas usando un motor de resolución que será descripto en más detalle en la Sección 3. Veamos ahora cómo podemos verificar que el sistema de archivos que venimos construyendo se comporta adecuadamente, poniendo a prueba propiedades que nuestra especificación debe satisfacer.

Como mencionamos en la introducción, el motor de inferencia de **Alloy** explora un universo de modelos cuya cardinalidad está acotada. La forma en la que opera **Alloy** consta de dos partes: *la especificación de un problema*, escribiendo las signaturas y restricciones, y *la elección de un scope*, definiendo las cotas para la cantidad máxima de elementos en el universo a la hora de validar una propiedad. Podemos especificar dichas cotas usando `for` en la sintaxis de `check`. Por ejemplo, si ponemos `for 3 Archivo, 5 Directorio`, se procederá a la verificación considerando un universo con a lo sumo 3 archivos y a lo sumo 5 directorios como máximo. Si solamente ponemos `for 3`, se verificará usando 3 elementos como cota para cada signatura.

El análisis consiste en la búsqueda exhaustiva sobre todo el espacio de combinaciones posibles de elementos del universo, con la finalidad de obtener un modelo de la teoría que no satisfaga la propiedad en cuestión, o sea, un contraejemplo.

La primer propiedad a verificar sobre nuestro sistema de archivos expresa que no es posible alcanzar un directorio comenzando la exploración desde él mismo.

```
sig Archivo{}
sig Directorio{
      contenidos: set Archivo + Directorio
}
one sig Root extends Directorio{}
fact { no (^contenidos & iden) }
pred contenido(ad: Archivo+Directorio) { one ad.~contenidos }
fact { all ad: Archivo+Directorio-Root | contenido[ad] }
fact { no contenidos.Root }

// No se puede llegar a un directorio navegando desde sí mismo.
check { no d: Archivo+Directorio | d in d.^contenidos } for 3

// Todos los archivos y directorios con excepción del Root deben
// ser alcanzables desde el Root.
check { all a: Archivo+Directorio-Root | a in Root.^contenidos } for 3

// Ningún directorio contiene más de 6 archivos o carpetas
check { no d: Directorio | #(d.contenidos)>6 } for 3
```

Figura 3. Modelado en **Alloy** de archivos y directorios: Verificaciones

Esto se debería implicar de la regla de aciclicidad de la relación `contenidos`, por lo que la verificación de esta propiedad no producirá una instancia contraejemplo. La segunda propiedad a verificar, expresa que todos los archivos son alcanzables desde el directorio `Root`, esta propiedad tampoco produce un contraejemplo dado que definimos la regla que expresa que todos los elementos menos `Root` están contenidos en algún directorio. La tercer propiedad dice que no hay directorios con más de 6 elementos contenidos en él. Esto claramente no tendría que ser cierto, sin embargo el análisis no produce contraejemplo. El problema acá es la elección del scope. A la hora de realizar la verificación usamos `for 3` en el comando `check`, esto quiere decir que el análisis se va a llevar a cabo teniendo en cuenta 3 archivos y 3 directorios. Dejando de lado modelos con a lo sumo 6 elementos lo que hace imposible que se presente un contraejemplo a la propiedad que estamos verificando. Sin embargo, podríamos usar `for 4` para que el universo tenga a lo sumo 4 archivos y 4 directorios, y ahora sí el análisis produciría un contraejemplo como el que se puede ver en la Figura 4.

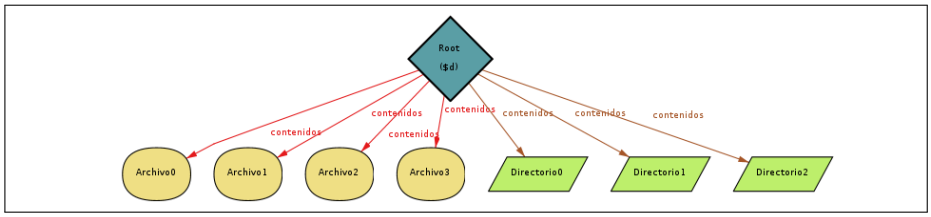

Figura 4. Contraejemplo para la tercer propiedad verificada con scope 4

3. Alloy Analyzer

Alloy Analyzer es una herramienta que implementa la verificación de teorías definidas en **Alloy**. Como dijimos anteriormente, el problema de determinar si una fórmula de **Alloy** es válida, no es computable. En lugar de eso, **Alloy Analyzer** realiza verificaciones finitistas. Supone que las variables del universo sólo pueden tomar valores en dominios de tamaño acotado, previamente definidos por el usuario. Con esta restricción es posible usar lógica proposicional para expresar, y luego verificar, las fórmulas definidas en **Alloy** [Jac00].

El proceso de verificación consiste en usar un buscador de modelos relacional llamado **KodKod** que se encarga de traducir la especificación a lógica proposicional y verificarla usando un **Sat-Solver**, como se muestra en la Figura 5. A fines prácticos, el lenguaje de **Alloy** y el de **KodKod** pueden pensarse como equivalentes. Vamos a dar más detalle sobre **KodKod** en la siguiente sección. A partir de una especificación S definida por el usuario usando **Alloy**, se construye una especificación S' en el lenguaje de **KodKod**. **KodKod** toma la cota c especificada por el usuario y aplica a S' una traducción que preserva equivalencia y obtiene una fórmula $Tr_c(S')$ en lógica proposicional. Esta traducción garantiza que si hay alguna asignación A que satisfaga $Tr_c(S')$, se puede construir una traducción Tr'_{SAT}, a partir de Tr, que dada una asignación para la fórmula proposicional obtiene el modelo **Alloy** correspondiente $Tr'_{SAT}(A)$ que se presentará al usuario si éste lo solicita. En el caso de no haber tal asignación, porque la fórmula no es satisfacible, el **Sat-Solver** provee una traza de resolución T de la cual hablaremos en la Sección 5. Es posible (usando el mismo principio que para Tr'_{SAT}) construir una traducción Tr'_{UNSAT} que toma T y obtiene la correspondiente porción de especificación **Alloy** $Tr'_{UNSAT}(T)$ que hace imposible hallar un contramodelo. A esta porción de especificación la llamamos **unsat core** y también hablaremos de ella con más detalle en la Sección 5.

3.1. KodKod.

Si bien **Alloy Analyzer** comenzó como una pieza de software monolítica, la complejidad de la herramienta hizo que se presente la necesidad de encapsular el motor de resolución en una pieza de software independiente: **KodKod**. **KodKod** [TJ07] tiene su primer aparición en la versión 4 de **Alloy Analyzer** y hoy en día funciona como una biblioteca independiente. A diferencia de **Alloy**, con el que se puede ejecutar la verificación automática de una propiedad sobre una teoría dada, **KodKod** provee una interfaz para especificar una teoría y buscar modelos que la satisfagan explorando un universo finito.

La sintaxis de **KodKod** se puede pensar como un subconjunto de **Alloy**. Los objetos matemáticos con los que trabaja **KodKod** también son relaciones. Con la diferencia de que, en **Alloy**, las relaciones se describen definiendo las signaturas y sus atributos y, a su vez, las signaturas definen jerarquías de tipos. En cambio, **KodKod** no hace diferencia entre las relaciones unarias (signaturas) y las no unarias (atributos), y además, las relaciones no son tipadas. Tampoco tiene convenciones sintácticas como los predicados (`pred`) o los hechos (`fact`).

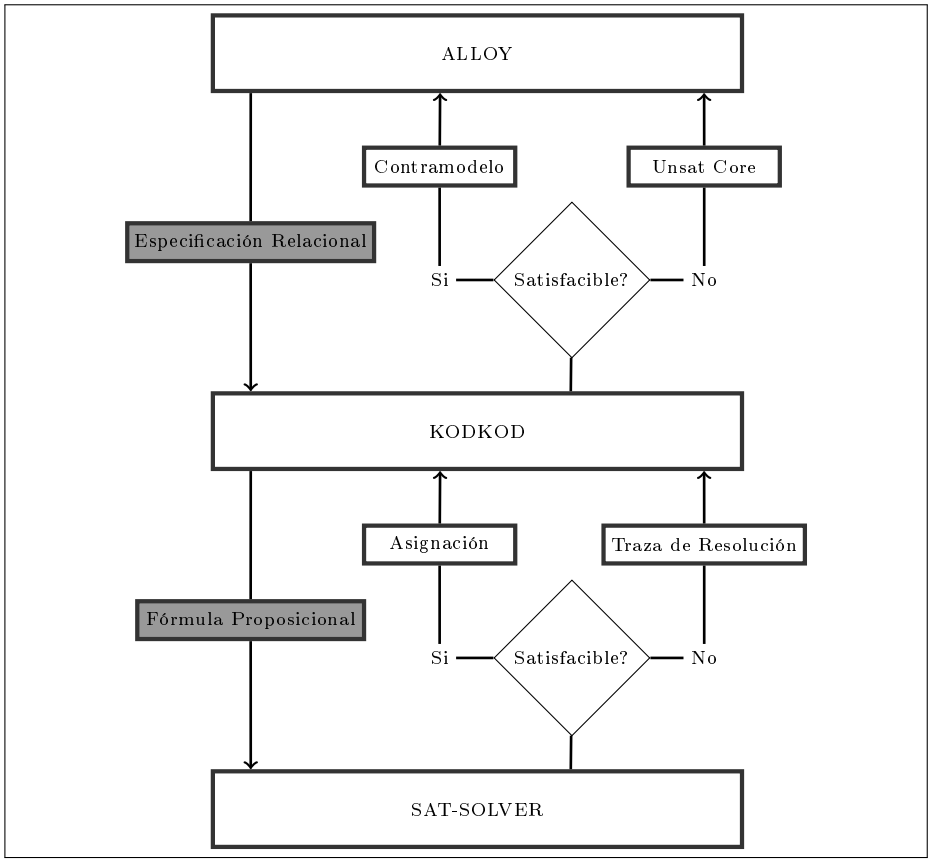

Figura 5. Arquitectura de Alloy Analyzer

Se puede pensar que la lógica de **KodKod** es como la de **Alloy** solo que sin *syntactic sugar*.

KodKod también requiere que las variables relacionales se liguen a cotas antes de ejecutar la búsqueda de modelos. Pero, a diferencia de **Alloy**, no acota los dominios usando límites enteros para la cantidad de elementos en cada relación, sino que éstas deben estar acotadas por debajo (*lower bounds*) y por encima (*upper bounds*) por dos conjuntos fijos de constantes relacionales definidas por el usuario.

Por ejemplo, la especificación de nuestro sistema de archivos en **KodKod** sería como se describe en la Figura 6. Primero definimos los elementos del universo, los cuales son indistintos y no están tipados. En nuestro caso usamos tres átomos para los archivos ($a1$, $a2$ y $a3$), y 3 átomos para los directorios ($d1$, $d2$ y r). Luego declaramos las relaciones con sus cotas, las cuales definen los posibles formas de conformar cada relación. Por ejemplo, **KodKod** buscará modelos asumiendo que las relaciones pueden estar conformadas por todas las combinaciones posibles de las tuplas definidas por los upper bounds y cada una de estas combinaciones debe contener a las tuplas definidas por los lower bounds, por ejemplo, la relación `Directorio` puede contener cualquier tupla mientras cumpla con $\{(\texttt{r})\} \subseteq \texttt{Directorio} \subseteq \{(\texttt{d0}), (\texttt{d1}), (\texttt{r})\}$. Finalmente, escribimos una especificación que restringe los modelos, teniendo en cuenta que algunas propiedades implícitas en la sintaxis de **Alloy** deben ser tenidas en cuenta, por ejemplo la inclución de la relación `contenidos` que se desprende de haberla definido como atributo de `Directorio` y la inclución de la relación `Root` que se desprende de la jerarquía de tipos descripta por `extends`.

Como dijimos antes, para proveer a **KodKod** con esta información, **Alloy** transforma levemente la especificación que define el usuario. Tomando los scopes y la definición de las signaturas, se definen el universo de átomos U y la declaración de las relaciones con sus respectivas cotas D. Se construye una única fórmula φ a partir de negar la propiedad a verificar y ponerla en conjunción con los axiomas (**facts**) que describen la teoría. Con estos parámetros, **KodKod** aplica a φ la traducción Tr que mencionamos anteriormente usando U y D para acotar los dominios, y obtiene una nueva fórmula $Tr(\varphi, U, D)$ en lógica proposicional con la que alimenta al **Sat-Solver**.

En la Figura 7 se presenta la arquitectura de **KodKod** donde se pueden apreciar las etapas de la traducción. Dado un problema $P = (\varphi, U, D)$, la traducción Tr comienza por un proceso de *skolemnización* donde se eliminan de φ los cuantificadores existenciales y sobre el cual no entraremos en detalles, obteniendose una fórmula skolemizada $Sk(\varphi)$. Usando las declaraciones de las relaciones D, se traduce $Sk(\varphi)$ a una fórmula proposicional $Prop(Sk(\varphi), D)$ usando matrices ralas como define [Jac00] y mas tarde [TJ07]. A partir de U y D, se contruye un predicado de *ruptura de simetrías* $SBP(U, D)$ que explicaremos con más detalle en breve, se pone en conjunción con la fórmula traducida y se expresa en *Forma Normal Conjuntiva* (o CNF, es una forma de reescribir las fórmulas de la lógica proposicional de la cual hablaremosen la Sección 4) obteniendo la fórmula final

Universo

```
{a0, a1, a2, d0, d1, r}
```

Declaración de Relaciones y Cotas

```
Archivo:1 [{}, {(a0), (a1), (a2)}]
Directorio:1 [{(r)}, {(d0), (d1), (r)}]
Root:1 [{(r)}, {(r)}]
contenidos:2 [{}, {(r, r), (r, d0), (r, d1), (r, a0), (r, a1),
                   (r, a2), (d0, r), (d0, d0), (d0, d1), (d0, a0),
                   (d0, a1), (d0, a2), (d1, r), (d1, d0),
                   (d1, d1), (d1, a0), (d1, a1), (d1, a2)}]
```

Fórmula

```
// La inclusión de la relación contenidos. Esto se
// deduce, en Alloy , de la declaración del campo contenidos.
contenidos in (Directorio -> (Directorio+Archivo)) &&
// La inclusión de la relación Root. Esto, en Alloy , se
// deduce del uso de extends en la declaración de Root.
Root in Directorio &&
// La aciclicidad de la relación contents.
no (^contenidos & iden) &&
// Los archivos y directorios excepto Root, estan contenidos en
// un y solo un directorio
(all ad2: (Directorio+Archivo)-Root | one (ad2.~contenidos)) &&
// Root no está contenido en ningún directorio
no contenidos.Root
```

Figura 6. Sistema de archivos especificado en **KodKod**

$Tr(\varphi, U, D) = CNF(Prop(Sk(\varphi), D) \wedge SBP(U, D))$ lista para ser presentada a un **Sat-Solver**.

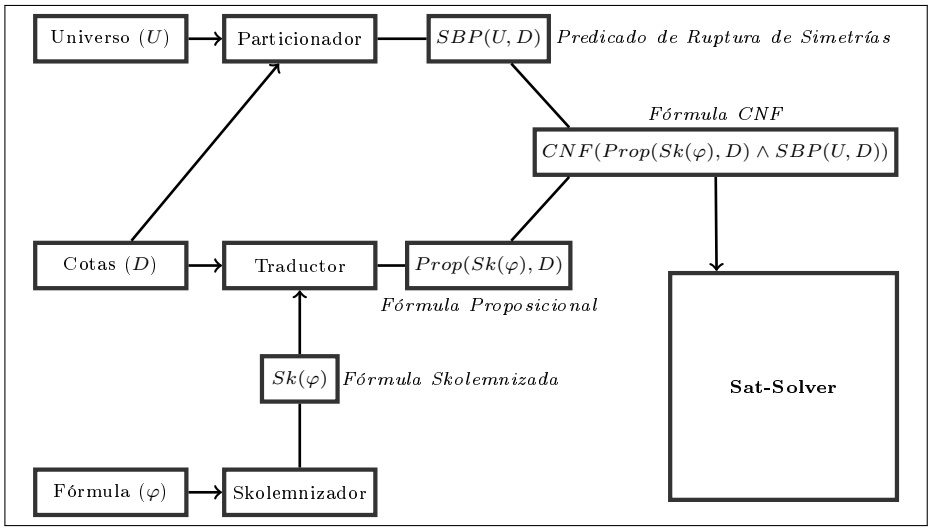

Figura 7. Arquitectura de **KodKod**

Traducción La traducción consiste es interpretar las relaciones como matrices y los operadores del lenguaje como operaciones entre matrices. Dada una declaración de relación $r :_k [Lb, Ub]$ sobre un universo $U = a_0, \ldots, a_n$, se construye una matriz booleana M de k dimensiones definida como

$$M[i_1, \ldots, i_k] = \begin{cases} 1 & \Leftrightarrow (a_{i_1}, \ldots, a_{i_k}) \in Lb \\ \texttt{variableFresca}() & \Leftrightarrow (a_{i_1}, \ldots, a_{i_k}) \in Ub - Lb \\ 0 & \textit{en otro caso} \end{cases}$$

con $i_1, \ldots, i_k \in [0, n]$. Por ejemplo, las relaciones de nuestro sistema de archivos se representarían como se muestra en el Figura 8. En primer lugar, se define una matriz por cada relación descripta en la especificación. La dimensión de estas matrices dependen de la cardinalidad de la relación a la que representan. Si la relación fuera unaria la matriz tendría dimensión $n \times 1$ o simplemente n, donde n es la cantidad de átomos del universo. Si la relación fuera binaria la dimensión de la matriz asociada sería $n \times n$ y si ésta fuera ternaria la matriz tendría dimensión $n \times n \times n$.

En el caso de las relaciones unarias, cada posición de cada matriz representa la existencia de un átomo del universo en un conjunto dado. Con la variables frescas se define que dicho átomo puede estar o no en los modelos sobre los cuales **KodKod** realizará la busqueda, mientras que las constantes 1 y 0 definen si un

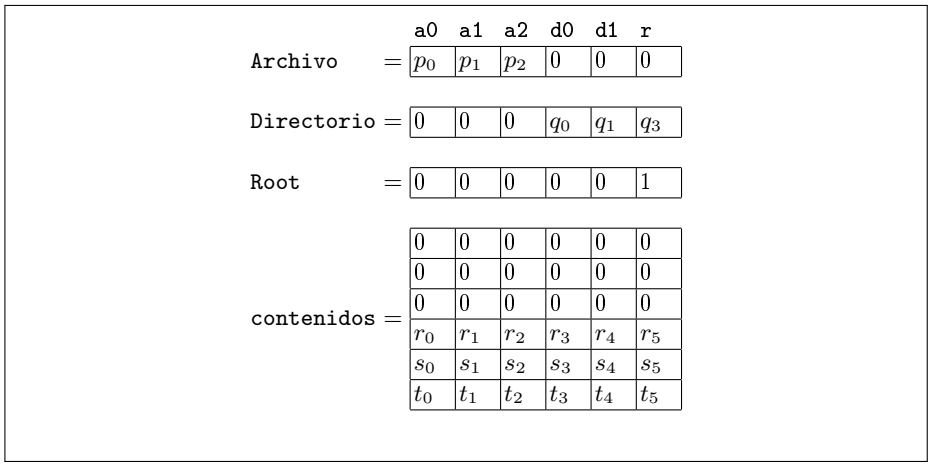

Figura 8. Interpretación Matricial de la Relaciones del Sistema de Archivos

átomo estará o no en todos los modelos que serán tenidos en cuenta. A su vez, para las relaciones de mayor cardinalidad, cada elemento en la matriz tiene una semántica análoga. Ya sea que en una determinada posición haya una variable fresca o una constante, ésta representa la pertenencia de un determinado par ordenado a la relación en cuestión. Por ejemplo, en el caso de *contenidos*, la variable r_5 define si el directorio d_0 contiene o no a *Root* y la variable t_0 define si *Root* contiene o no al archivo a_0.

Con esta interpretación, todos los operadores que se usen para escribir fórmulas con el lenguaje de **KodKod** tienen operaciones matriciales asociadas definidas en [TJ07]. Por ejemplo, el operador de navegación aplicado a dos expresiones `Root.contenidos` es traducido como el producto matricial entre las matrices correspondientes a `Root` y `contenidos` y la expresión cuantificada `some Root.contenidos` es traducida como la disyunción de todas las posiciones de la matriz correspondiente a `Root.contenidos`.

Ruptura de Simetrías Se les dice simetrías a las permutaciones de los modelos de una teoría que también son modelos. Por ejemplo, dada la teoría que define nuestro sistema de archivos presentada anteriormente, un modelo posible es como el que se puede ver en la Figura 9 (izquierda), donde `archivo0` pertenece a los contenidos de Root, y uno simétrico sería el de la Figura 9 (derecha), donde el `archivo0` está en el directorio que no es Root. Como se puede ver, a los fines del análisis, con algún representante de los dos modelos alcanza. El análisis explora casos que no es necesario tener en cuenta, dado que si uno es modelo de la teoría, el otro tambien lo será. Es por esto que el predicado de ruptura de simetrías produce una optimización importante a la hora de verificar la fórmula usando el **Sat-Solver**. En general, aumenta la velocidad de resolución por varios órdenes de magnitud y muchos problemas interesantes serían intratables sin ruptura de simetrías [CGLR96] [Shl07].

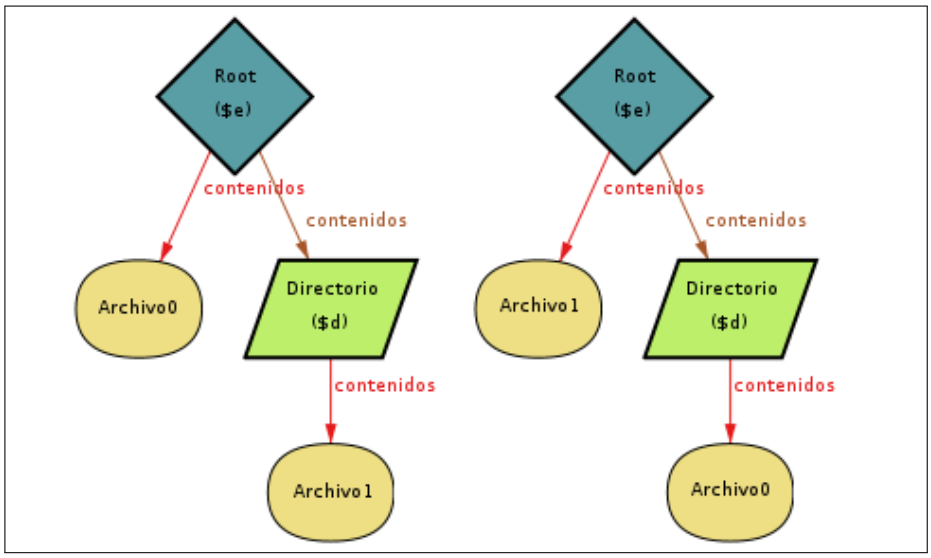

Figura 9. Ejemplo de simetría

Para detectar las simetrías se busca particionar el universo en clases de equivalencia de manera tal que todos los lower y upper bounds puedan ser expresados como una unión de productos de estas clases. Particionar el universo de esta manera no tiene una solución polinomial [BKL83], por lo que **KodKod** emplea un algoritmo goloso que funciona bien para la mayoría de los casos [TJ07]. Una vez particionado el universo se usan las particiones para contruir el predicado de ruptura de simetrías como se explica en [Shl07].

3.2. Resumen

Para efectuar esta verificación se llevan a cabo los siguientes procedimientos:

1. Dada la especificación de una teoría definida en **Alloy**, se la traduce al lenguaje de **KodKod**.
   a) A partir de las signaturas y del scope se definen el universo U y las relaciones con sus respectivas cotas D.
   b) Con los facts y la negación de la propiedad a verificar se construye una fórmula φ.
2. Se provee a **KodKod** con la especificación (φ, U, D) donde se traduce a lógica proposicional para verificarla usando un **Sat-Solver** –
   a) Con U y D, se construye el predicado de ruptura de simetrías $SBP(U, D)$
   b) Se skolemniza la fórmula φ y se traduce a lógica proposicional mediante una representación matricial usando D, obteniéndose la fórmula $Prop(Sk(\varphi), D)$.

   c) Esta última fórmula se pone en conjunción con el predicado de ruptura de simetrías y se expresa en forma normal conjuntiva, dando como resultado la fórmula traducida $Tr(\varphi, U, D) = CNF(Prop(Sk(\varphi), D) \wedge SBP(U, D))$ con la que se alimenta al **Sat-Solver**.

3. A partir de la salida del **Sat-Solver** se hace el camino inverso para exhibir los resultados al usuario.
   a) Si $Tr(\varphi, U, D)$ tiene una asignación A que la satisface, se aplica la traducción en sentido inverso para construir el modelo **Alloy** que le corresponde $Tr'_{SAT}(A)$.
   b) Si no hay tal asignación, se construye el **unsat core** $Tr'_{UNSAT}(T)$, a partir de la traza de resolución T que ofrece el **Sat-Solver**.

4. Sat-Solving

4.1. Lógica proposicional

La lógica proposicional es una herramienta muy útil para expresar problemas. Tiene la particularidad de ser muy sencilla y sobre todo, es computable. Los símbolos atómicos con los que trabaja la lógica proposicional son las *proposiciones*. Las proposiciones son sentencias que pueden tomar el valor de verdadero o falso y se usan *variables proposicionales* para representarlas. Por ejemplo $p =$ "*Connor MacLeod es inmortal*", en este caso p es la variable proposicional que representa la proposición "*Connor MacLeod es inmortal*" que vale verdadero si MacLeod es inmortal y falso en caso contrario.

Sintaxis Definiremos ahora un lenguaje que, bajo la interpretación adecuada, define a la lógica proposicional. Al conjunto de fórmulas de éste lenguaje lo llamaremos $Form$ y contruiremos cada una de ellas usando los siguientes 6 símbolos: x, $'$, $($, $)$, $\neg$ y $\Rightarrow$. Al símbolo $\neg$ de *negación* y al símbolo $\Rightarrow$ de *implicación* los llamaremos *conectores lógicos*, que nos servirán para contruir fórmulas mas complejas a partir otras fórmulas de $Form$. Diremos que las apariciones del símbolo x seguido de una cantidad finita de tildes, $x', x'', x''', \ldots$ son variables proposicionales. Con estos símbolos construiremos inductivamente las fórmulas de la lógica proposicional de la siguiente manera:

- Cada elemento de $Var = \{x', x'', x''', \ldots\}$ es una fórmula de $Form$.
- Sea φ una fórmula de $Form$, $\neg\varphi$ es una fórmula de $Form$.
- Sean φ y ψ dos fórmulas de $Form$, $(\varphi \Rightarrow \psi)$ es una fórmula de $Form$.

Abreviaremos las variables proposicionales con un subíndice que representa la cantidad de tildes, dando como resultado el conjunto infinito $\{x_k \mid k > 0\} = \{x_1, x_2, x_3, \ldots, x_i, \ldots\}$ de variables proposicionales al que llamaremos Var y nombraremos como $Var(\varphi)$ al conjunto de variables proposicionales de una fórmula φ. También, ampliremos el conjunto de conectores lógicos agregando la *conjunción* ($\wedge$) y la *disyunción* ($\vee$) para abreviar ciertas fórmulas como se describe a continuación.

Sean φ y ψ fórmulas de $Form$:

- Escribiremos $(\varphi \wedge \psi)$ en lugar de $\neg(\varphi \Rightarrow \neg\psi)$.
- Escribiremos $(\varphi \vee \psi)$ en lugar de $(\neg\varphi \Rightarrow \psi)$.

Semántica La interpretación de una fórmula de $Form$ consiste en asignarle un valor de verdad ($Verdadero$ o $Falso$, T o F, 0 o 1, etc) a dicha fórmula según un criterio específico. Para asignar valores de verdad a las fórmulas de $Form$, primero es necesario asignar valores de verdad a las variables proposicionales en Var. Para esto, llamaremos *asignación* para una fómula φ con n variables proposicionales a un subconjunto de las variables de φ con sus respectivos valores

de verdad asignados, por ejemplo $A = \{(x_1, 0), (x_7, 1), (x_9, 0)\}$. No pueden aparecer en una asignación dos ocurrencias de la misma variable proposicional con distintos valores de verdad asignados. Una asignación A para φ se dice *completa* si $|A| = |Var(n)|$, de lo contrario se la llama *parcial*.

Ahora sí, es posible dar valores de verdad a una fórmula φ de $Form$ con n variables proposicionales, mediante la definición de una función de valuación $f^{\varphi} : (Var(\varphi) \times \{0,1\})^n \to \{0,1\}$ asociada a φ. Esta función toma una asignación completa para las variables de la fórmula φ y la *evalúa* según las siguientes reglas.

Sean ψ y ρ fórmulas de $Form$ y $A = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ una asignación completa con v_i un valor de verdad para $1 \leq i \leq n$ y B y C dos asignaciones completas para ψ y ρ respectivamente definidas cómo

- $B = A - ((Var(\rho) - Var(\psi)) \times \{0,1\})$ y
- $C = A - ((Var(\psi) - Var(\rho)) \times \{0,1\})$

entonces:

- si $\varphi = x_k$ con $1 \leq k \leq n$, entonces

$$f^{\varphi}(A) = \begin{cases} 1 \; si \; (x_k, 1) \in A \\ 0 \; si \; (x_k, 0) \in A \end{cases}$$

- si $\varphi = \neg\psi$, entonces $f^{\varphi}(A) = 1 - f^{\psi}(A)$
- si $\varphi = \psi \Rightarrow \rho$, entonces

$$f^{\varphi}(A) = \begin{cases} 0 \; si \; f^{\psi}(B) = 1 \; y \; f^{\rho}(C) = 0 \\ 1 \quad en \; otro \; caso \end{cases}$$

- si $\varphi = \psi \vee \rho$, entonces

$$f^{\varphi}(A) = \begin{cases} 0 \; si \; f^{\psi}(B) = 0 \; y \; f^{\rho}(C) = 0 \\ 1 \quad en \; otro \; caso \end{cases}$$

- si $\varphi = \psi \wedge \rho$, entonces

$$f^{\varphi}(A) = \begin{cases} 1 \; si \; f^{\psi}(B) = 1 \; y \; f^{\rho}(C) = 1 \\ 0 \quad en \; otro \; caso \end{cases}$$

Problema de satisfactibilidad El *Problema de Satisfactibilidad* (SAT) consiste en encontrar una asignación completa A para las variables de una fórmula φ de manera tal que $f^{\varphi}(A) = 1$, o demostrar que dicha asignación no existe. SAT fue el primer problema identificado como perteneciente a la clase de complejidad *NP-completo* [FM]. La intuición detrás de esto es que la cantidad de asignaciones posibles que resultan de una fórmula de n variables proposicionales es 2^n y siempre hay un caso en donde se deben explorar todas. Por ejemplo, para una fórmula con una variable hay dos asignaciones posible $A_1 = \{(x_1, 0)\}$ y $A_2 = \{(x_1, 0)\}$ mientras que para una fórmula con dos variables hay cuatro: $A_1 = \{(x_1, 0), (x_2, 0)\}$, $A_2 = \{(x_1, 0), (x_2, 1)\}$, $A_3 = \{(x_1, 1), (x_2, 0)\}$ y

$A_4 = \{(x_1, 1), (x_2, 1)\}$. Hasta hoy, si bien han sido estudiadas heurísticas que guían el proceso de búsqueda y funcionan muy bien en la práctica, la busqueda de una asignación que satisfaga una fórmula dada, debe recorrer todo el espacio de asignaciones posible para poder asegurar que dicha asignación no existe. Es por esto que los algoritmos de busqueda tiene un orden exponencial en función de la cantidad de variables de la fórmula para el peor caso.

Forma Normal Conjuntiva Una fórmula de la lógica proposicional se dice estar en *Forma Normal Conjuntiva* (CNF) si es una conjunción de la forma $w_1 \wedge \ldots \wedge w_m$, donde cada w_i es una disyunción $l_{1_i} \vee \ldots \vee {}_{k_i}$, siendo cada l_{j_i} la ocurrencia de una variable(x_{j_i}) o su negación($\neg x_{j_i}$). Toda fórmula de la lógica proposicional puede ser expresada en CNF mediante el procedimiento descripto intuitivamente a continuación:

- Primero quitamos las implicaciones con la regla $(\varphi \Rightarrow \psi) = (\neg\varphi \vee \psi)$
- Luego, introducimos todas las negaciones lo mas cerca posible de las varibles proposicionales usando las leyes de De Morgan que dicen que:
  - $\neg(\varphi \wedge \psi) = (\neg\varphi \vee \neg\psi)$ y
  - $\neg(\varphi \vee \psi) = (\neg\varphi \wedge \neg\psi)$
- A continuación, eliminamos las negaciones que sobran sabiendo que $\neg\neg x_k = x_k$, para que las variables aparezcan solas o afectadas por una sola negación.
- Finalmente usamos la propiedad distributiva de la disyunción frente a la conjunción que dice que $\varphi \vee (\psi \wedge \rho) = (\varphi \vee \psi) \wedge (\psi \vee \rho)$, para que la fórmula quede expresada como una conjunción de disyunciones de variables o variables negadas.

En general, se usa esta forma para el análisis de satisfactibilidad dado que simplifica el proceso. El problema es que este procedimiento para pasar cualquier fórmula a CNF puede explotar exponencialmente. En lugar de este método se usa otro que introduce variables nuevas y preserva satisfactibilidad. En particular, **KodKod**, usa también un método llamado *compact boolean circuits* para eliminar redundancias en la fórmula producidas por la representación matricial de las relaciones de una especificación. No entraremos en detalle acerca de estos métodos, lo que cabe recalcar es que existe un método que hace posible expresar una fórmula de la lógica proposicional en CNF usando tiempo polinomial, para mas información, ver [TJ07].

En adelante, nos referiremos tanto a las variables como a las variables negadas como *literales*, a las disyunciones de literales como *cláusulas* y a la conjunción de cláusulas como la *base de cláusulas*. Ahora bien, al evaluar una fórmula φ en CNF, según una asignación, se genera una partición en el conjunto de cláusulas: las cláusulas satisfechas (que evalúan a 1), cláusulas insatisfechas (que evalúan a 0) y, si la asignación en cuestión es parcial, también aparecen cláusulas sin resolver (que evalúan a un valor X indeterminado, con $X \in \{0, 1\}$). Llamaremos *literales libres* a lo literales sin asignar de una cláusula sin resolver y a las cláusulas con un solo literal libre las llamaremos *cláusulas unitarias*.

4.2. Algoritmo de Búsqueda

Hoy en día los algoritmos que resuelven SAT consisten en un proceso de búsqueda exhaustiva por backtracking sobre el espacio de todas las asignaciones posibles que finaliza cuando se encuentra una asignación que evalúe la fórmula a 1 o se termina de recorrer el espacio sin encontrarla. Estos algoritmos están basados en la regla de resolución que se enuncia como la siguiente tautología

$$(A \vee \neg x) \wedge (B \vee x) \Leftrightarrow (A \vee B)$$

siendo A y B dos disyunciones de literales. Con esta propiedad en mente, se desarrolló un algoritmo presentado en [DP60], que años mas tarde fue el principal esquema de resolución adoptado por muchos resolvedores de SAT y demostradores de teoremas. Este algoritmo lleva el nombre de DP y el esquema es como se describe a continuación

```
Para cada variable v en la fórmula
     Para cada cláusula c conteniendo v y cada cláusula n conteniendo ¬v
          Resolver c y n usando la regla de resolución y
          agregar el resultado a la fórmula general
     Remover de la fórmula todas cláusulas que contienen v y ¬v
```

Este algoritmo fue mejorado en [DLL62], al que nos referiremos como DPLL, y es con ésta mejora con la que trabajan los demostradores actuales. Fundamentalmente, el algoritmo DPLL, consiste en llevar una asignación parcial que se extiende eligiendo literales y resolver usando un esquema llamado *propagación* que también es una extensión del método de resolución. Las etapas del algoritmo de DPLL son

- *Decisión*: Elegir una variable no asignada para extender la asignación actual. Llamaremos a las asignaciones de variables hechas en este paso *asignaciones por decisión*, o simplemente *decisiónes*. Cada vez que se tome una decisión diremos que entramos en un nuevo *nivel de decisión*. Extenderemos la definición de asignación para mantener información acerca del nivel de decisión donde fue asignada. Por ejemplo $A = \{(x_1, 0@1), (x_7, 1@2), (x_9, 0@3)\}$, significa que x_1 fue asignada en en nivel de decisión 1, x_7 en el nivel 2 y x_9 en el nivel 3.
- *Propagación*: Se extiende la asignación actual a través de la propagación de la última decisión tomada. Este proceso se detalla mas adelante. La idea básica es, por ejemplo, dada la cláusula $\{x_1, \neg x_2, x_3\}$ con la asignación $x_1 = 0$ y $x_2 = 1$ se deduce que $x_3 = 1$. Llamaremos a estas, *asignaciones por implicación*, o simplemente *implicaciones*. Este proceso de deducción puede llevar a que una o mas cláusulas queden insatisfechas, en este caso decimos que se alcanzo un *conflicto* y se identifican las decisiónes que fueron tomadas hasta el momento como la *causa del conflicto*.

- *Retroceso*: Si hubo un conflicto en el paso anterior, se deshace la asignación actual hasta un nivel de decisión en que el conflicto desaparezca para poder probar nuevas asignaciones.

No entraremos en detalle acerca de cómo se toman las decisiónes, es decir, de cómo de eligen las variables y que valores se asignan. La estrategia más popular es usar una heurística llamada *Dynamic Largest Individual Sum* (DLIS) en la cual se elige el literal que aparece en mas cláusulas sin resolver. Algunos otros como [MMZ+01], [GN02] y [ES03] usan una heurística llamada *Variable State Independent Decaying Sum* (VSIDS), que se basa en la cantidad de conflictos que producen las cláusulas a las que pertenece cada literal. Para mas información acerca de la importancia de métricas que acompañen la toma de decisiónes ver [Sil99].

Con respecto al proceso de propagación, el mecanismo por el cual se derivan implicaciones de una base de cláusulas CNF se lo conoce como *Boolean Constraint Propagation* (BCP). Como se explicó en el ejemplo anterior, de las asignaciones a los literales pueden surgir *cláusulas unitarias*. Si todos los literales de la cláusula unitaria, con excepción del literal libre, fueron asignado a 0, se deduce que este último debe asignarse a 1. De lo contrario la cláusula quedaría sin satisfacer y por lo tanto la fórmula entera también. Diremos que el nivel de decisión de esta asignación es el de la última decisión tomada. Estas implicaciones se propagan iterativamente hasta agotar todas las cláusulas unitarias o dar con un conflicto. En caso de agotar las cláusulas unitarias se continúa con el proceso aumentando la asignación actual con una nueva decisión. Ahora bien, recorrer toda la base de cláusulas buscando cláusulas unitarias en cada asignación puede tomar mucho tiempo. En lugar de eso se usa una estrategia llamada *watched literals* que consiste en mantener una estructura que se indexa por literal y tiene referencias a las cláusulas que contienen al literal que funciona como índice como un literal libre. Con esta estructura es posible obtener directamente todas las implicaciones que se deducen de asignar un literal. Es necesario mantener esta estructura actualizada durante el proceso de propagación, lo cual insume un tiempo de cómputo adicional pero solo del orden del largo de las cláusulas y no de la cantidad de cláusulas total. Para más información acerca de esta estrategia ver [MMZ+01]

Por el contrario, si se encontró un conflicto, se procede a retroceder en el árbol de decisiónes. Este retroceso es conocido con el nombre de *non-Chronological Backtracking*. Consiste en volver atras a través de las decisiónes tomadas hasta que desaparezca el conflicto. Puede pasar que el nivel de decisión al que se regrese no sea el inmediatamente el anterior al actual, por ejemplo si se prueban las dos asignaciones posibles para un determinado literal y en ambos casos se obtiene un conflicto, se deduce que éste es producido por decisiónes tomadas en niveles anteriores al nivel de decisión actual, que las propagaciones no detectaron, en este caso se dice que el retroceso es no-cronológico.

En la Figura 10 tenemos un grafo dirigido que da una ilustración de las decisiónes tomadas y las implicaciones inferidas. Cada decisión puede dar lugar a una o más implicaciones dependiendo cómo sea la base de cláusulas. En el

ejemplo, decidir que x_1 sea asignada a 0 en el nivel 1 implica que x_2 sea asignada a 0 en el nivel 1. De esta manera cada implicación tiene asociada una cláusula unitaria a la que llamaremos la *razón de implicación*, la cual es la responsable directa de la asignación de la variable. En el ejemplo de la Figura 10, la razón de $x_2 = 0@1$ sería ω_1 y de $x_6 = 1@3$, ω_3.

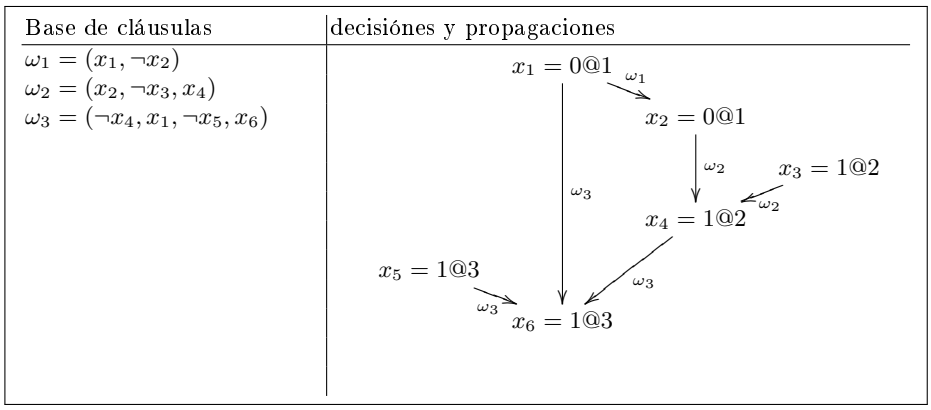

Figura 10. decisiónes y Propagaciones

Cláusulas Aprendidas Ahora bien, los motores de resolución de SAT actuales trabajan con un agregado interesante. Producen *cláusulas aprendidas* que se agregan a la base de datos inicial a modo de información inferida durante el proceso de resolución. A este acercamiento se lo clasifica como un mecanismo de búsqueda con aprendizaje dinámico. Estas cláusulas aprendidas son la columna vertebral de la heurística para la obtención del **hot core**. La idea a vuelo de pájaro detrás del concepto de cláusula aprendida es que una cierta cadena de decisiónes pueden llevar a un conflicto que debe ser tenido en cuenta en las próximas decisiónes que tome el solver.

Supongamos que al ejemplo de la Figura 10 se le agrega la cláusula $\omega_4 = \{\neg x_3, \neg x_6\}$. La implicación en el nivel de decisión 3 que fuerza a asignar x_6 a 1 propagaría un conflicto dado que x_3 fue asignado a 1 por una decisión tomada en el nivel 2 y como $x_3 = 1@2$ y $x_6 = 1@3$, se deduce que la cláusula ω_4 no puede ser satisfecha por la asignación actual, arrivando a un conflicto ilustrado en la Figura 12 que denominaremos κ. Cuando se alcanza un conflicto se procede a una nueva etapa del proceso de resolución a la cual llamaremos *Análisis* que consiste en analizar el conflicto y llevar al proceso de resolución a nivel de decisión adecuado (*Non-Chronological Backtracking*). El pseudo-código para el proceso de análisis se muestra en la Figura 11. Primero, si el nivel de decisión actual es 0, no hay forma de resolver el conflicto y se devuelve -1 informando que la fórmula no es satisfacible. En caso contrario, se aplica iterativamente un paso de resolución entre la cláusula conflictiva y la razón de implicación de alguna variable de la

cláusula conflictiva. La función *elegir_literal(c)* elige el literal de la cláusula c que fue asignado más recientemente. La función *resolver(c_1, c_2, v)* devuelve una cláusula que contiene todos los literales que aparecen en c_1 y c_2 exceptuando los que corresponden a la variable v. Notar que la cláusula conflictiva tiene todos los literales asignados a 0 y la cláusula razón de implicación tiene todos los literales asignados en 0 excepto el literal que está implicando (por haber sido una cláusula unitaria) el cual debe estar asignado a 1, por lo que la cláusula resultante de aplicar un paso de resolución también es una cláusula conflictiva dado que todos sus literales están asignados en 0, así el análisis continúa iterativamente. El análisis termina cuando la cláusula aprendida tiene un solo literal asignado en el nivel de decisión actual y todos los demás en niveles de decisión menores. En este punto se agrega la cláusula aprendida a la base de cláusulas y se calcula el nivel de retroceso como el máximo nivel menor al actual. Después del retroceso esta cláusula será unitaria y dicho literal será forzado a asumir el valor opuesto llevando el proceso de búsqueda a un nuevo espacio. Esta variable asumirá el nivel de decisión del punto donde se retroceda y el proceso de búsqueda continuará.

```
 1: if el nivel de decisión actual es 0 then
 2:    devolver -1
 3: end if
 4: conflicto = cláusula_conflictiva()
 5: c_aprendida = conflicto
 6: loop
 7:    l = elegir_literal(c_aprendida)
 8:    v = variable_del_literal(l)
 9:    conflicto = razón(v)
10:    c_aprendida = resolver(c_aprendida, conflicto, v)
11:    if un solo literal en c_aprendida fue asignado en el nivel actual then
12:       Agregar c_aprendida a la base de cláusulas
13:       devolver nivel_retroceso(c_aprendida)
14:    end if
15: end loop
```

Figura 11. Pseudo-Código del proceso de Análisis

Teorema 1 *Todas las cláusulas aprendidas se deducen de la base de cláusulas inicial.*

No daremos una demostración de este teorema, para ver una se puede referir a [BKS03]. Pero, la idea es que como toda cláusula aprendida se construye a partir de la negación de un conflicto, cualquier asignación que no satisface una cláusula aprendida, produce un conflicto que, por definición, no satisface la base de cláusulas inicial.

	Nivel 0	Nivel 1	Nivel 2	Nivel 3
$\omega_1 = (x_1, \neg x_2)$	$(\bot, \bot)$	$(0, 1)$	$(0, 1)$	$(0, 1)$
$\omega_2 = (x_2, \neg x_3, x_4)$	$(\bot, \bot, \bot)$	$(0, \bot, \bot)$	$(0, 0, 1)$	$(0, 0, 1)$
$\omega_3 = (\neg x_4, x_1, \neg x_5, x_6)$	$(\bot, \bot, \bot, \bot)$	$(\bot, 0, \bot, \bot)$	$(0, 0, \bot, \bot)$	$(0, 0, 0, 1)$
$\omega_4 = (\neg x_3, \neg x_6)$	$(\bot, \bot)$	$(\bot, \bot)$	$(0, \bot)$	$(0, 0) = \kappa$

Figura 12. Conflicto

Para cada cláusula aprendida hay un conjunto de cláusulas incluído en la unión entre la base de cláusulas inicial y la base de cláusulas aprendidas, que representan lo que llamaremos las *cláusulas antecedentes* de la clasula aprendida. En nuestro ejemplo, las cláusulas antecedentes de la cláusula ω_5, son ω_1, ω_2, ω_3 y ω_4. Más adelante, en la Sección 5 hablaremos de la importancia de mantener esta información y de cómo nos puede servir para calcular la prueba de insatisfactibilidad y su consecuente core de insatisfactibilidad.

Finalmente, en la Figura 13 tenemos la estructura principal del algoritmo para resolver SAT fundamentado en DPLL y con el agregado de las cláusulas aprendidas.

```
1: loop
2:    Propagación()
3:    if no hay conflicto then
4:       if todas las variables están asignadas then
5:          devolver SATISFACIBLE
6:       else
7:          Decisión() - Nueva asignación de variable
8:       end if
9:    else
10:      - Analizar el conflicto y añadirlo a la base de cláusulas
11:      nivel_retroceso = Análisis()
12:      if nivel_retroceso == −1 then
13:         devolver NO SATISFACIBLE
14:      else
15:         Retroceso(nivel_retroceso)
16:      end if
17:   end if
18: end loop
```

Figura 13. Pseudo-código del algoritmo para resolver SAT

4.3. MiniSat

Minisat [ES03] es un **Sat-Solver**. Uno de los que usa **Alloy Analyzer** para realizar la verificación automática con la fórmula CNF producto de traducir la especificación **Alloy**.

Este **Sat-Solver** implementa heurísticas interesantes como *Non-Chronological Backtraking*, *Boolean Constraint Propagation* y algunas más sobre las cuales no entraremos en detalles. Lo interesante de **Minisat** es que es un **Sat-Solver** minimalista pensado para hacer fáciles adaptaciones. Además, como veremos en la siguiente sección, con **Minisat** es posible obtener la traza de resolución de la resolución de la fórmula CNF, lo que nos permitirá obtener el **hot core**.

Ahora bien, el conjunto de cláusulas aprendidas puede aumentar mucho durante el proceso de resolución, volviéndose imposible mantenerlo en memoria. Para esto, **Minisat** usa una heurística que mantiene una métrica que mide la actividad de las cláusulas aprendidas y reduce la base de cláusulas en función de esta actividad. Esta actividad se calcula como la participación de las cláusulas en los conflictos durante la propagación y resulta ser el pilar principal de la heurística para determinar el **hot core**.

5. Unsat Core

Como Dijsktra hizo notar una vez, el *testing* puede mostrar la presencia de errores pero nunca su ausencia. Probar la ausencia de errores en un sistema ha sido una gran motivación para muchos estudios realizados en la ingeniería del software. Entre ellos el desarrollo de herramientas como Alloy, que a pesar de las ventajas que nos provee a la hora de detectar *bugs*, no siempre queda claro si, en realidad, quedan algunos más por ser detectados.

La razón por la cual no se puede estar seguro de la remanencia de bugs en un modelo es, en esencia, que no podemos tener la seguridad a la hora de verificar una propiedad si ésta captura correctamente la noción que estamos interesados en modelar [Bro87]. Puede ser que simplemente no hayamos diseñado bien la representación de nuestro problema.

Para esto, se puso en práctica la noción de **unsat core**. A nivel especificación alloy, el **unsat core** es la porción de dicha especificación que produce una trivialización de la propiedad a verificar, que **Alloy Analyzer** es capaz de demostrar como válida (o sea, no encuentra una instancia contraejemplo). Mostrar el **unsat core** puede servirle al usuario para identificar, en la teoría, una posible falla de modelado. Notar que a nivel **Alloy** el nombre **unsat core** es confuso dado que puede llevar a pensar que corresponde a la porción de especificación que hace imposible que la propiedad sea válida. Sin embargo es el caso inverso, proviene de la imposibilidad de hallar una asignación que satisfaga la fórmula en el nivel de la lógica proposicional, lo que en este caso corresponde a que no se pudo encontrar un contraejemplo para la propiedad **Alloy**.

5.1. Unsat Core en Alloy

Recordemos el esquema de verificación que plantea **Alloy**. Dada la especificación de una teoría T y una propiedad P en términos de T, **Alloy Analyzer** realiza una busqueda exhaustiva sobre todos los modelos de T en un universo acotado. Esta búsqueda tiene como objetivo encontrar un contraejemplo para P, o sea, un modelo de T que muestre que no vale $T \Rightarrow P$. Ahora bien, que la búsqueda no encuentre un contramodelo para la propiedad no es condición suficiente para decir que es válida esto puede ocurrir por dos razones,

- T es muy fuerte o P es muy debil, o ambas, debido a un error de especificación en el momento de definirlas, o
- $T \Rightarrow P$ realmente vale aunque alloy no nos de garantías dado que puede haber contraejemplos en universos de mayor tamaño.

El segundo caso es el deseado por el usuario, mientras que el primero es el que se busca evitar. Es una tarea difícil identificar cuándo un determinado análisis cae en un caso o el otro. Para enfrentar esta dificultad es que se propuso el **unsat core** como una forma de tener mas información acerca de una especificación.

Haciendo una división más sutil para los casos en los que **Alloy Analyzer** no puede encontrar un contraejemplo y produce un **unsat core**, podemos identificar los siguientes

1. El modelo es muy fuerte.
2. La propiedad es muy débil.
3. El modelo y la propiedad son válidos.
4. El scope es demasiado chico.

De los cuatro casos, sólo el tercero es el que dejaría conforme al usuario, el cuarto forma parte del compromiso que se asume al usar un buscador de modelos finito y el primer caso, así como el segundo, delatan posibles problemas de diseño. Para explicar mejor los cuatro escenarios, supongamos que disponemos de la siguiente especificación de nuestro sistema de archivos, sobre la cual introdujimos un error de diseño: el segundo **fact** dice "**all** `ad:Archivo+Directorio`" en lugar de "**all** `ad:Archivo+Directorio-Root`" como figuraba en la Sección 2

Si verificamos la primer propiedad, **Alloy Analyzer** no encontrará un contraejemplo. Sin embargo, tenemos el resaltado de la especificación que representa el unsatcore como se ve a continuación.

Se puede notar que la propiedad a verificar no está siendo resaltada. Esto es síntoma del primer caso de **unsat core**: el modelo es muy fuerte. Al tener en cuenta al directorio `Root` entre los directorios y archivos sobre los que predica el segundo **fact**, estamos diciendo que todos los directorios deben estar contenidos en algún otro. Esto entra en contradicción con el tercer **fact** que dice que no puede haber un directorio que contenga a `Root`.

El segundo caso de **unsat core** se da cuando la propiedad no produce contraejemplo por haber sido subespecificada. Por ejemplo, la segunda propiedad enuncia que todos los directorios tienen algún archivo o tienen al menos un archivo. Está claro que esta propiedad es tautológica y que vale para cualquier modelo de nuestro sistema de archivos. El **unsat core** que produce **Alloy Analyzer** sería

En la especificación corregida, nuevamente escrita como está en la Figura 3, no se tiene en cuenta al directorio `Root` entre los directorios sobre los que predica el segundo **fact**. Podemos entonces verificar otra vez la primer propiedad y obtener el siguiente **unsat core**

Ahora sí, este **unsat core** entra en el tercer caso, donde la especificación del sistema es consistente y la propiedad no presenta ningún contraejemplo.

Finalmente, al verificar la tercer propiedad podemos ver que no produce contraejemplo sino que la herramienta resalta el siguiente **unsat core**.

La propiedad dice que no hay un directorio con mas de 6 elementos contenidos en él. Esto tiene sentido dado que la probamos usando `for 3`, o sea, con 3 directorios y 3 archivos. En este escenario no puede haber más de 6 elementos en todo el universo. La propiedad cae claramente en el cuarto caso de **unsat core**. Si aumentamos el tamaño del universo obtendríamos un contraejemplo como el de la Figura 4.

Ahora bien, para obtener el **unsat core** de la verificación de una propiedad que no produce contraejemplo, es necesario obtener el **unsat core** de un conjunto de cláusulas CNF como explicaremos en breve en la Sección 5.2. Este conjunto debe ser un subconjunto de la base de cláusulas inicial. Una vez obtenido este subconjunto de cláusulas es posible, usando la traducción de **Alloy** a CNF,

obtener la correspondiente porción de especificación **Alloy** que representa el **unsat core**

Unsat Core Minimal Cabe recalcar que el **unsat core** que se obtiene como explicaremos en breve puede producir, en la especificación **Alloy**, un conjunto de restricciones no minimal. Se entiende por **unsat core** minimal a un conjunto de restricciones de la especificación tales que si alguna restricción se elimina, la especificación se vuelve satisfacible. Sobre esto, en [TCJ08] se exhiben algunos algoritmos que buscan minimizar el **unsat core** a traves de varias corridas del proceso de sat-solving. El algoritmo más sencillo se llama *naive core extraction* y consiste en ir quitando iterativamente restricciones de la especificación, siempre y cuando ésta siga siendo insatisfacible, hasta que no se puedan quitar mas sin que **Alloy Analyzer** encuentre un contraejemplo. Una vez removidas todas las restricciones con las que la especificación sigue siendo insatisfacible, se exhibe el **unsat core** al usuario. Esta estrategia requiere de muchas ejecuciones del proceso de sat-solving y en dicho trabajo se busca minimizar esta cantidad de ejecuciones usando una estrategia más intersante llamada *recycling core extraction*. Sin embargo en este trabajo comparamos el desempeño del **hot core** con un **unsat core** sin minimizar, el cual es la primer salida del proceso de sat-solving al que nos referiremos como en [TCJ08] con el nombre de *One-Step core extraction*.

5.2. Unsat Core en Lógica Proposicional

Recordando el proceso de resolución de un **Sat-Solver** basado en DPLL, a medida que el proceso de resolución avanza, éste se encuentra con conflictos producidos por las decisiones tomadas. Estos conflictos se agregan a la base de cláusulas como cláusulas aprendidas a modo de información implicada de la base de cláusulas original. Como mencionamos en la Sección 4, cada cláusula aprendida tiene un conjunto de cláusulas asociado conocido como sus cláusulas antecedentes y denotado con $Ante(w)$ siendo w una cláusula aprendida. Estas cláusulas son aquellas que entraron en contradicción para alguna asignación parcial en un determinado momento del proceso de resolución, dando lugar al conflicto a partir del cual se construye w. Con esto en mente, se puede llevar cuenta del proceso de resolución manteniendo una *traza de resolución* como se explica en [ZM03]. Esta traza se puede definir usando un grafo dirigido como se ve en la Figura 19

Cada nodo del grafo representa una cláusula, los nodos raíz representan las cláusulas originales de la fórmula CNF y los nodos intermedios representan las cláusulas aprendidas durante el proceso de resolución. Si de un nodo a hay un eje orientado hacia b decimos que a es el antecedente de b en el proceso de resolución.

Más formalmente, la traza de resolución se define como

Definición 1 *(Traza de resolución)*

Sea C y R dos conjuntos de cláusulas tales que C `no se puede satisfacer` *y R contiene la cláusula vacía ($c_\varnothing$). Sea E un conjunto de ejes que va de $C \cup R$ a R. Un grafo dirigido $G = (C \cup R, E)$ es una traza de resolución* `sii`

- *En C sólo están todos los nodos con* `grado de insidencia` *igual a cero.*
- *Para cada nodo $r \in R$ se cumple que hay un nodo $c \in C$ tal que $r \in E^*(c)$.*

A la luz de esta definición es posible definir el **unsat core** como el conjunto $\{c \in C | c_\varnothing \in E*(c)\}$. O sea, todas las cláusulas pertenecientes a la base de cláusulas inicial tales que se puede encontrar, en la traza de resolución, un camino desde ellas a $c_\varnothing$. Así, es posible recorrer la traza de resolución desde $c_\varnothing$, hacia atrás hasta los nodos raíz obteniendo el **unsat core** como se muestra con los ejes y nodos más grandes en la Figura 19.

```
sig Archivo{}
sig Directorio{ contenidos: set Archivo + Directorio }
one sig Root extends Directorio{}

fact { no (^contenidos & iden) }
pred contenido(ad:Archivo+Directorio) { one ad.~contenidos }
fact { all ad:Archivo+Directorio | contenido[ad] }
fact { no contenidos.Root }

check { all a: Archivo+Directorio-Root | a in Root.^contenidos } for 3
check { all d: Directorio | some d.contenidos or lone d.contenidos } for 3
check { no d: Directorio | #(d.contenidos)>6 } for 3
```

Figura 14. Modelado en **alloy** de archivos y directorios: Unsat Core

```
sig Directorio{ contenidos: set Archivo + Directorio }
pred contenido(ad:Archivo+Directorio) { one ad.~contenidos }
fact { all ad:Archivo+Directorio | contenido[ad] }
fact { no contenidos.Root }
```

Figura 15. Unsat Core para la primer propiedad: Primer caso de **unsat core**

```
check { all d: Directorio | some d.contenidos or lone d.contenidos } for 3
```

Figura 16. Unsat Core para la segunda propiedad: Segundo caso de **unsat core**

```
sig Directorio{ contenidos: set Archivo + Directorio }
fact { no (^contenidos & iden) }
pred contenido(ad:Archivo+Directorio) { one ad.~contenidos }
fact { all ad:Archivo+Directorio | contenido[ad] }
check { all a: Archivo+Directorio-Root | a in Root.^contenidos } for 3
```

Figura 17. Unsat Core para la primer propiedad: Tercer caso de **unsat core**

```
sig Directorio{ contenidos: set Archivo + Directorio }
fact { no (^contenidos & iden) }
pred contenido(ad:Archivo+Directorio) { one ad.~contenidos }
fact { all ad:Archivo+Directorio | contenido[ad] }
fact { no contenidos.Root }
check { all a: Archivo+Directorio-Root | a in Root.^contenidos } for 3
```

Figura 18. Unsat Core para la tercer propiedad: Cuarto caso de **unsat core**

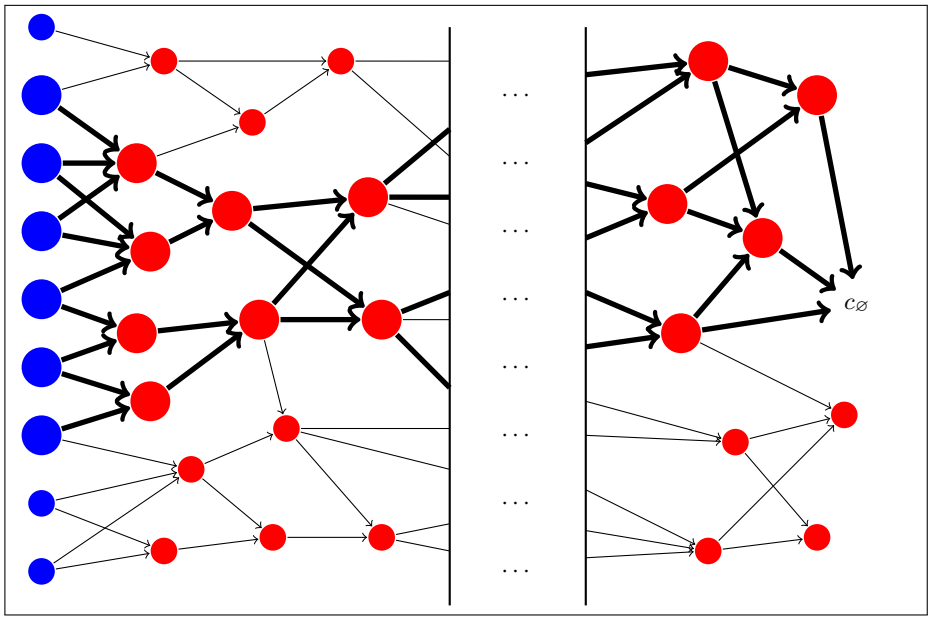

Figura 19. Traza de Resolución

6. Hotcore

El unsat core es una herramienta muy útil para quien desarrolla o diseña software. El problema que intenta atacar es cuando se presentan propiedades que requieren tiempos muy altos o bien puede pasar que la cantidad de memoria necesaria para llevar a cabo el cómputo exceda los recursos disponibles. Tanto en un caso como en el otro nos encontramos con la necesidad de detener el proceso de verificación antes de haber llegado a la solución final. El **hot core** es una generalización del **unsat core** que da respuesta a dicha problematica. Con él es posible detener el proceso de verificación y proveer al usuario con una aproximación del **unsat core** representada, con el resaltado de porciones de la especificación en **Alloy**. Para esto se necesita que el motor de resolución (el **Sat-Solver**) esté basado en DPLL como se explicó en la Sección 4. Esto es importante porque el algoritmo utilizado se basa en que la resolución de una fórmula de la lógica proposicional está pensada como un análisis sobre cláusulas CNF que producen aprendizajes dinámicos en forma de nuevas cláusulas. Dada una métrica que exprese la actividad de estas cláusulas aprendidas durante el proceso de resolución, es posible identificar aquellas que tienen altas chances de pertenecer al **unsat core**.

6.1. Motivación

Identificamos hasta tres posibles usos del **hot core**, los cuales presentamos a continuación:

- *Aproximación del Unsat Core*: Para lograr más seguridad en la validez de una fórmula para la cual **Alloy Analyzer** no produce un contraejemplo en un scope determinado, el usuario puede aumentarlo para buscar contraejemplos en un dominio de mayor tamaño. El análisis llevado a cabo por **Alloy Analyzer** recae en la resolución de una fórmula en lógica proposicional. Esta fórmula alcanza tamaños más grandes a medida que aumenta el scope del problema especificado en **Alloy**, por lo que el tiempo de resolución también suele aumentar exponencialmente para el peor caso. Dado que el análisis toma mucho tiempo mas, no es raro que el usuario interrumpa el proceso antes de alcanzar el final del proceso de resolución, por haber excedido el tiempo esperado. Como **Alloy Analyzer** no alcanzó a recorrer todo el espacio en la búsqueda de un contraejemplo, **HotCore** resaltará las porciones de especificación que más tiempo de cómputo están tomando. En este caso, el **hot core** se puede ver como una aproximación al **unsat core**. Una vez obtenido el **hot core** es posible identificar falencias en el modelo, corregirlas y volver a comenzar el análisis sin haber esperado a que la herramienta identifique el **unsat core**, ni decrementar el scope para no tener que esperarlo tanto tiempo.
- *Optimización de la Especificación*: Es bien sabido que fórmulas equivalentes escritas en **Alloy** pueden producir diferentes fórmulas CNF. A su vez, dadas dos fórmulas CNF equivalentes, el tiempo de cómputo también puede variar

bastante dependiendo de factores como el órden de las cláusulas, el largo promedio de cada una de ellas, la distribución de los literales, etc. Reescribir las fórmulas **Alloy** para que el resultado de traducir a CNF sea más compacto o esté mejor organizado no es tarea fácil. Pero, conociendo las propiedades que toman más trabajo analizar, un usuario de **Alloy** experimentado puede reescribirlas y lograr una especificación que tome menos tiempo en las resoluciones. Nótese que en este caso, el uso de **HotCore** es independiente de la validez de la propiedad. No es necesario que la propiedad carezca de contraejemplos para obtener las porciones de especificación que más tiempo le toman al Sat-Solver resolver.

- *Mas Confianza en Scopes de Mayor Tamaño*: A pesar de la hipótesis de scopes chicos [Jac06] que dice que gran parte de los contraejemplos pueden ser encontrados sin tener que aumentar mucho el tamaño del problema, suele pasar que la mayoría de los usuarios de **Alloy** tienden a agrandar los scopes hasta donde alcancen los recursos. Desde la perspectiva del usuario, un scope más grande implica mayor confianza en la validez de la propiedad que se está verificando. Como fue mencionado antes, el tiempo de resolución puede incrementarse dramáticamente conforme aumente el tamaño del problema. De hecho, este incremento puede ir desde minutos a horas aumentando sólo un poco el tamaño del problema. Así, mientras la herramienta prueba que una determinada propiedad es válida para un tamaño no muy grande, puede ser que nunca llegue a probar su validez en un tamaño más grande. En este caso, en el que no es posible obtener el **unsat core**, sólo disponemos del **hot core**. Si el **hot core** coincide con el **unsat core** de un tamaño más chico un usuario de **Alloy** puede concluir que la mayor cantidad del tiempo de cómputo está siendo utilizado en verificar propiedades que también son insatisfacibles para el tamaño de problema más chico. Con esta información, podemos intuir que la propiedad se implica de las mismas premisas tanto para el scope chico como para el scope grande. Está claro que esto no puede ser una demostración de la validez de la propiedad en un scope más grande, sólo aporta más seguridad al usuario de **Alloy** a la hora de no poder obtener la solución final para un tiempo de espera dado.

6.2. Implementación

Buscando Cláusulas con Mucha Actividad El procedimiento para extraer el **hot core** es parecido al que se usa para extraer el **unsat core**. Recordando el proceso de verificación, éste comienza por la traducción de la especificación en **Alloy** a una fórmula S en CNF. A partir de S se construye una base de cláusulas C con la que se alimenta un **Sat-Solver**. Durante el proceso de resolución, se agregan a C nuevas cláusulas a las que llamamos cláusulas aprendidas, siendo cada una de ellas, consecuencias de S. El agregado de estas cláusulas se da debido a conflictos encontrados durante la exploración del espacio de búsqueda.

El **Sat-Solver** mantiene un valor de actividad para cada cláusula que se incrementa cada vez que éstas participan en los conflictos desde su agregado a la base de cláusulas y durante el resto del proceso de resolución. Este valor aumenta

según el momento del proceso en el que se dan los conflictos. Si una cláusula participa en algún conflicto en un estadío avanzado del proceso de resolución, el aumento en su actividad será más importante que para una cláusula que participe durante el estadío inicial. El cálculo de la actividad sería como se puede ver en la Figura 20, la línea 8 aumenta dicha actividad para cada cláusula involurada en el conflicto y en la Figura 21, la línea 17 aumenta el incremento de la actividad cada vez que un propagación produce un conflicto. El valor inicial de incremento de actividad es de 1 y aumenta a razon de $0{,}999^{-1}$ luego de cada propagación que produzca un conflicto. Ambos valores fueron elegidos por los desarrolladores de **Minisat**, que como mencionamos antes, usan esta métrica para reducir la base de cláusulas cuando aumenta mucho su tamaño.

```
 1: if el nivel de desición actual es 0 then
 2:    devolver -1
 3: end if
 4: conflicto = cláusula_conflictiva()
 5: c_aprendida = conflicto
 6: loop
 7:    aumentar_actividad(conflicto)
 8:    l = elegir_literal(conflicto)
 9:    v = variable_del_literal(l)
10:    conflicto = razón(v)
11:    c_aprendida = resolver(c_aprendida, conflicto, v)
12:    if un solo literal en c_aprendida fue asignado en el nivel actual then
13:       Agregar c_aprendida a la base de cláusulas
14:       devolver nivel_retroceso(c_aprendida)
15:    end if
16: end loop
```

Figura 20. Agregados al algoritmo para el proceso de Análisis

Esta métrica coincide con la identificación de cláusulas con altas chances de pertenecer al **unsat core**. Nosotros nos aprovechamos de esta métrica para emprender la identificación del **hot core** y vemos que tiene un buen comportamiento empírico.

Cuando la verificación es interrumpida por el usuario, elegimos el conjunto $H \subset C$ de las cláusulas aprendidas con mayor actividad, o sea, las cláusulas más "difíciles" o "problemáticas" para el proceso de resolución. Hoy en día sólo capturamos las 1000 clásulas más activas. Con estas cláusulas identificadas, la idea es calcular el conjunto $H' \subset S$ tal que todas las cláusulas en H son consecuencias de H', recorriendo la traza de resolución en sentido inverso como se explicó en la sección 5.2. De esta manera, H' es el subconjunto de la base de cláusulas inicial, que identificamos que tiene altas chances de ser insatisfacible.

```
 1: loop
 2:    Propagación()
 3:    if no hay conflicto then
 4:       if todas las variables están asignadas then
 5:          devolver SATISFACIBLE
 6:       else
 7:          Decisión() - Nueva asignación de variable
 8:       end if
 9:    else
10:       - Analizar el conflicto y añadirlo a la base de cláusulas
11:       nivel_retroceso = Análisis()
12:       if nivel_retroceso == −1 then
13:          devolver NO SATISFACIBLE
14:       else
15:          Retroceso(nivel_retroceso)
16:       end if
17:       aumentar_incremento_de_actividad()
18:    end if
19: end loop
```

Figura 21. Agregados al algoritmo para resolver SAT

Extrayendo el hot core Una vez identificado el conjunto H de clásulas con altas chances de ser insatisfacible, quisieramos obtener la correspondiente porción de especificación **Alloy**. Esto nos permitirá presentar al usuario con un resaltado de especificación **Alloy** de la misma manera que se hace con el **unsat core**.

Como comentamos en la Sección 3, **KodKod** toma la traza de resolución y obtiene la porción de especificación **Alloy** que representa el **unsat core**. Para esto, recorre en sentido inverso el grafo dirigido de resolución desde el nodo que representa la cláusula vacía hasta los nodos raíz del grafo, los cuales representan las cláusulas que usa para alimentar la traducción inversa (de CNF a **Alloy**).

Lo mas efectivo es aprovechar la infraestructura, el problema es que en el instante en que el usuario interrumpe el proceso de resolución, no disponemos de una traza de resolución completa dado que todavía no se alcanzó una cláusula vacía. Para abordar esta situación, intervenimos en la implementación de **Minisat** creando una cláusula vacía ficticia c_H que tiene como cláusulas antecedentes las cláusulas más activas en el momento que el usuario interrumpió el proceso de resolución como se muestra en la Figura 22 y alimentamos esta traza de resolución retocada

6.3. Experimentos y Resultados

Para probar el funcionamiento de **HotCore** diseñamos experimentos separados básicamente en

- casos con **unsat core** conocido, con una buena convergencia de **hot core**.
- casos donde el **hot core** no funciona correctamente.

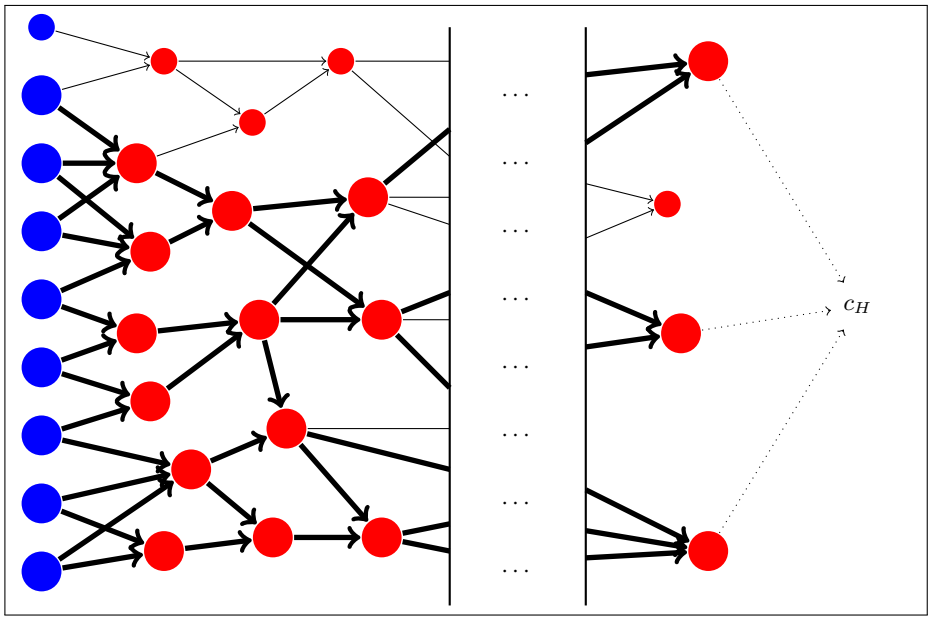

Figura 22. Grafo de Resolución Incompleto con una Cláusula Vacía Ficticia

Para el primer caso usamos 10 ejemplos clásicos extraídos de la distribución de **Alloy Analyzer**. Estos problemas vienen de una gran variedad de dominios, exhiben un amplio rango de comportamientos y están descriptos en detalle en [Jac06]. Cada uno fue corrido usando **HotCore** con varios scopes diferentes. Todos las comparaciones fueron hechas contra el **unsat core** más sencillo obtenido con *One-Step core extraction*.

Para medir la calidad de la aproximación, definimos dos métricas: *Hit* y *Error*. *Hit* representa que tan bien el **hot core** aproxima el **unsat core**, y con *Error* medimos el error de esa aproximación. Ambos se definen como se describe a continuación.

Definición 2 *Dados dos conjuntos U y H que representan un **unsat core** y un **hot core** respectivamente, resultado del análisis de una determinada propiedad, Hit y Error se definen como*

$$Hit = \#(H \cap U)/\#(U) * 100$$

$$Error = \#(H - U)/\#(H) * 100$$

Es posible usar ambas métricas tanto a nivel especificación **Alloy** como a nivel proposicional. En el primer caso calculamos sobre la especificación **Alloy** las posiciones de los caracteres ASCII que intervienen en cada identificación de **hot core** y los comparamos con los correspondientes al **unsat core** previamente conocido. Dado que **HotCore** se presenta al usuario como el resaltado de la especificación de entrada, usamos esta métrica como una medida de lo bueno que

puede ser para el usuario el uso de la herramienta. En el segundo caso usamos el conjunto de cláusulas CNF las cuales identificamos con un *id*. Comparamos el conjunto de cláusulas de cada identificación de **hot core** con el conjunto de cláusulas correspondiente al **unsat core** conocido previemente. Usamos esta métrica dado que de la traducción de la especificación **Alloy** a CNF se pueden obtener varias cláusulas CNF que no tiene su contrapartida en la especificación (como por ejemplo, el predicado de ruptura de simetrías), es por esto que presentamos la convergencia a nivel proposicional para obtener información mas detallada. Llamaremos a ambas métricas con un subíndice según corresponda a cada caso, usaremos p para referirnos a proposicional y a para **Alloy**.

Problema	Tamaño	Scope	Unsat	Hot	Hit_a	Error_a	Hit_p	Error_p	$\frac{\text{Hot}}{\text{Unsat}}$
lists - reflexive	21	10	153	16	50	0	75	3	10 %
lists - symmetric	21	8	8	1	44	0	82	8	13 %
hotel2	65	15	5	1	100	0	78	4	20 %
hotel4	61	17	166	48	100	0	81	2	29 %
lights	20	200	8	4	100	0	96	1	50 %
addressBook1h	21	30	117	74	100	0	93	93	63 %
ringElection2	27	14	5	4	100	0	98	0	80 %
sets2	11	36	444	256	100	0	88	1	58 %
mediaAssets	61	30	31	19	91	1	90	1	61 %
p306-hotel	40	18	43	31	100	0	91	3	72 %
Promedio									45.6 %

Figura 23. Resultados para criterio de parada de 75 % Hit_p

En la Figura 23 se pueden ver los resultados obtenidos en las corridas de hotcore para los problemas mencionados anteriormente. Las primeras cuatros columnas resumen la descripción del problema: nombre, tamaño del modelo (en lineas de codigo), scope del análisis y el tiempo requerido para computar el **unsat core** (en segundos). En la quinta columna se presenta el instante de tiempo en que el análisis fue interrumpido (también en segundos), o sea el instante en que se identifica el **hot core**. El criterio de parada fue alcanzar como mínimo el 75 % de Hit_p. En las columnas restantes presentamos Hit y $Error$ como en la Definición 2 y en la última columna el ratio entre el tiempo insumido para computar el **unsat core** y el **hot core**.

Observamos que, en general, se alcanza el 75 % de Hit_p bastante rápido: en promedio, **HotCore** necesita menos de la mitad del tiempo del proceso de resolución. Ademas, estos resultados viene acompañados de un $Error_p$ bastante bajo. Con respecto a Hit_a y $Error_a$ esperabamos una gran correlación entre los resultados a nivel **Alloy** y a nivel proposicional, lo cual fue confirmado. El ejemplo AddressBook1h tuvo un comportamiento particular dado que presenta una diferencia significativa en las dos métricas teniendo 93 % de $Error_p$ y 0 % de $Error_a$. Este es un caso en que el conjunto de cláusulas identificdas fuera del **unsat core** no tiene una contraparte en la especificación **Alloy**. Suponemos que estas cláusulas tienen que haber sido introducidas por procesos que optimizan la fórmula, como ser el predicado de ruptura de simetrías.

Cambiando la perspectiva a la del usuario final, también analizamos los resultados en términos de **Alloy**. Mostramos los resultados obtenidos sobre los mismos problemas pero tomando pero tomando como criterio de parada el nivel de especificación **Alloy** en lugar del nivel proposicional, usamos también un umbral del 75 % pero para Hit_a. Estos resultados se muestran en la Figura 24 y se puede ver que el tiempo neesario para obtener el 75 % de Hit_a es incluso menor al de Hit_p. en promedio **HotCore** necesitó del 20,6 % del tiempo total de resolución. La mejora se puede pensar en términos de la explosión exponencial en la traducción de **Alloy** a lógica proposicional, junto con el comportamiento permisivo del esquema de resaltado de **Alloy**.

Problem	Size	Scope	Unsat	Hot	Hit_a	Error_a	Hit_p	Error_p	$\frac{\text{Hot}}{\text{Unsat}}$
lists - reflexive	21	10	153	52	100	0	91	3	34 %
lists - symmetric	21	8	8	5	97	0	96	7	63 %
hotel2	65	15	5	1	97	0	28	12	20 %
hotel4	61	17	166	7	99	0	44	3	4 %
lights	20	200	8	3	100	0	37	2	38 %
addressBook1h	21	30	117	23	91	0	4	95	20 %
ringElection2	27	14	5	1	94	0	8	14	20 %
sets2	11	36	444	1	100	0	3	14	0.23 %
mediaAssets	61	30	31	2	88	0	7	0	6 %
p306-hotel	40	18	43	1	98	0	19	17	2 %
Average									20.6 %

Figura 24. Resultados para criterio de parada de 75 % Hit_a

En la Figura 25 (izquierda) y Figura 25 (derecha) respectivamente se puede ver como Hit_p y $Error_p$ evolucionan en el tiempo para un subconjunto de los casos estudiados. El eje X representa el porcentaje de tiempo (en escala logarítmica) mientras que el eje Y muestra la métrica correspondiente. Mirando la Figura 25 (izquierda), por un lado la convergencia del **hot core** crece exponencialmente rápido y por otro lado tiene pocas ocsilaciones mostrando un comportamiento casi monótono. Con respecto a la Figura 25 (derecha) se puede ver que después del 10 % del tiempo total de resolución el error es menor al 15 % para todos los casos.

Todo el análisis presentado hasta ahora representa un conjunto de casos donde el **hot core** se comportó bastante bien. Pero para terminar de validar y observar este comportamiento de aproximación al **unsat core** quisimos encontrar algunos ejemplos que muestren sus debilidades. Como no encontramos este comportamineto en lo ejemplos provistos por [Jac06], decidimos contruir casos *ad-hoc* que hagan fallar la heurística. Combinamos dos modelos **Alloy** independientes el uno del otro en uno sólo: uno que presente contraejemplo (o sea que la fórmula CNF sea satisfacible) y que sea difícil en términos de tiempo de resolución con uno que no sea satisfacible de manera tal que el modelo final tampoco lo sea, pudiendose obtener un **unsat core**. La intuición detras de esta mezcla es que exista la posibilidad de frenar el proceso de resolución justo cuando se resuelven cláusulas relacionadas con la porción satisfacible de la fórmula, dando

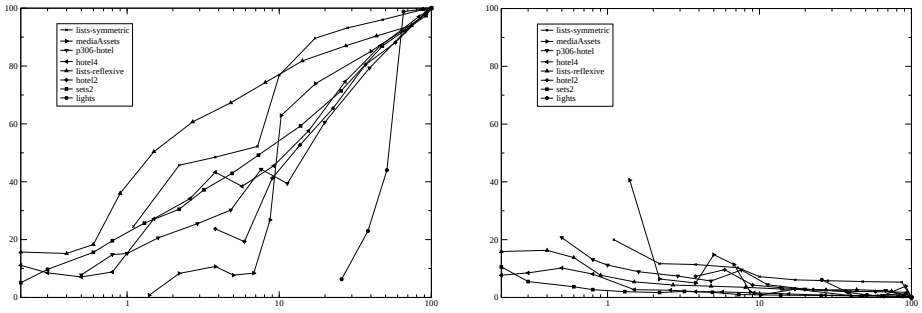

Figura 25. evolución de Hit_a y $Error_a$

como resultado un **hot core** que nada tiene que ver con el **unsat core** real. Usamos dos problemas **Alloy** satisfacibles presentados en [JGF], que se mostraron computacionalmente difíciles de resolver. Los combinamos con el problema RingElection, que sabemos que no es satisfacible. Lo que esperábamos se confirmó, como se puede ver en las Figuras 26 y 27. El **hot core** para los casos de prueba tuvo un *Hit* cercano a 0 % y un *Error* cercano a 100 % durante casi todo el proceso de resolución. El **hot core** coincide con el **unsat core** sólo por el final, cuándo ya se alcanzó alrededor del 70 % del tiempo total. Sin embargo la dificultad a la hora de encontrar estos ejemplos son una señal del buen comportamiento de la heurística.

Time	% Hit (Spec)	% Miss (Spec)	% Hit (CNF)	% Miss (CNF)
370	0.0	100.0	0.0	100.0
616	0.0	100.0	0.0	100.0
1027	1.1	99.4	1.8	94.6
1248	1.8	99.0	2.5	94.6
1645	0.0	100.0	0.0	100.0
2779	0.0	100.0	0.0	100.0
4063	1.1	99.7	1.1	98.7
5817	1.1	99.7	2.9	97.3
10187	0.0	100.0	0.0	100.0
18636	0.0	100.0	0.0	100.0
31921	0.0	100.0	0.0	100.0
59953	0.0	100.0	0.0	100.0
118365	0.0	100.0	0.0	100.0
161724	100.0	0.0	100.0	0.0

Figura 26. Problema **check_fright_13_null** junto con **ringElection**

Para finalizar, cabe mencionar algunos detales técnicos acerca de la implementación. Corrimos los ejemplos arriba mencionados variando la cantidad de cláusulas más activas que se seleccionan para formar parte del **hot core**. Como se puede ver en las Figuras 28, 29, 30, 31, 32, 33 y 34, los resultados no difieren significativamente cuando se usan mas de 100 cláusulas. La aproximación comienza a ser de menor calidad cuando usamos menos de 10, 20 o 50 cláusulas.

Time	% Hit (Spec)	% Miss (Spec)	% Hit (CNF)	% Miss (CNF)
305	0.0	100.0	0.0	100.0
543	0.0	100.0	0.0	100.0
927	0.0	100.0	0.0	100.0
1388	0.0	100.0	0.0	100.0
2631	0.0	100.0	0.0	100.0
3785	0.0	100.0	0.0	100.0
5820	0.0	100.0	0.0	100.0
8882	0.0	100.0	0.0	100.0
17100	0.0	100.0	0.0	100.0
30368	0.0	100.0	0.0	100.0
53340	100.0	82.3	30.3	90.3
67573	100.0	0.0	100.0	0.0

Figura 27. Problema **check_fright_10_11** junto con **ringElection**

Después de muchos experimentos, elegimos 200 como el tamaño fijo de cláusulas. Esto puede ser una prueba de lo bien ajustada que está la heurística de **Minisat** y que un conjunto relativamente chico de cláusulas puede alcanzar. Además la heurística que usamos prácticamente no agrega overhead, y esto se debe a que construimos la heurística para **hot core** fundamentándonos en la heurística de **Minisat**.

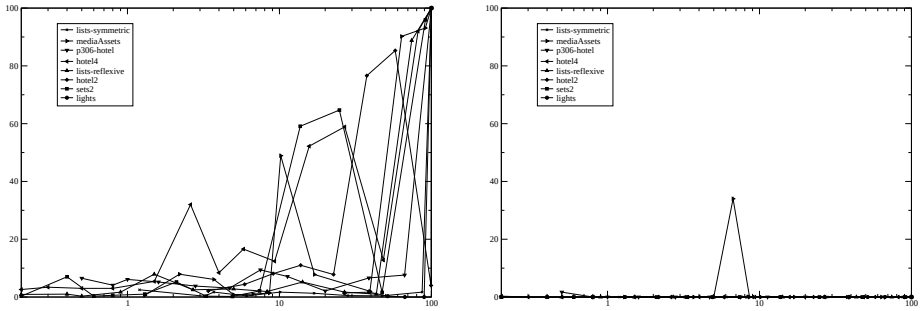

Figura 28. Hit_a y $Error_a$ para 10 cláusulas

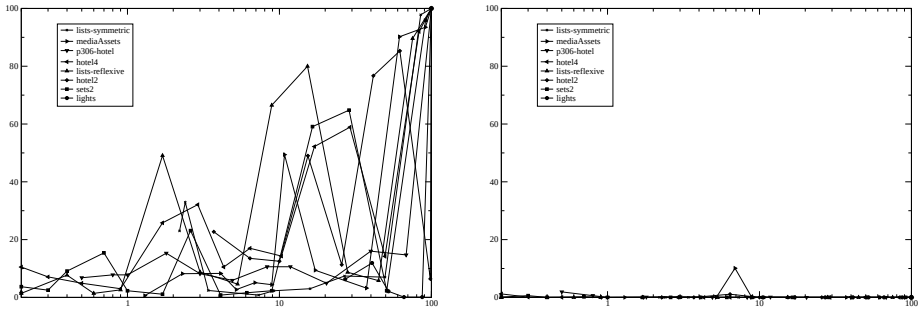

Figura 29. Hit_a y $Error_a$ para 20 cláusulas

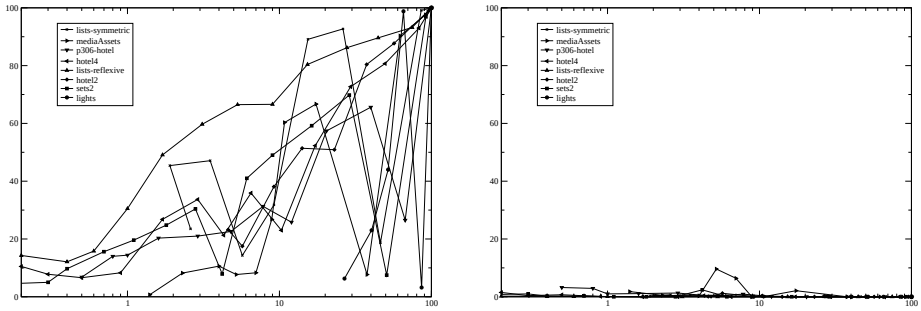

Figura 30. Hit_a y $Error_a$ para 50 cláusulas

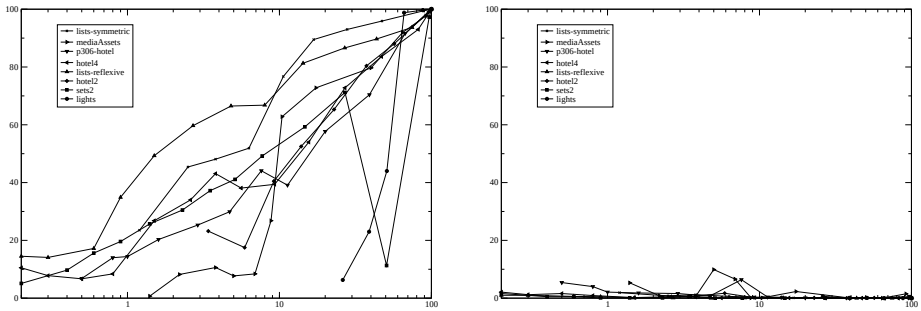

Figura 31. Hit_a y $Error_a$ para 100 cláusulas

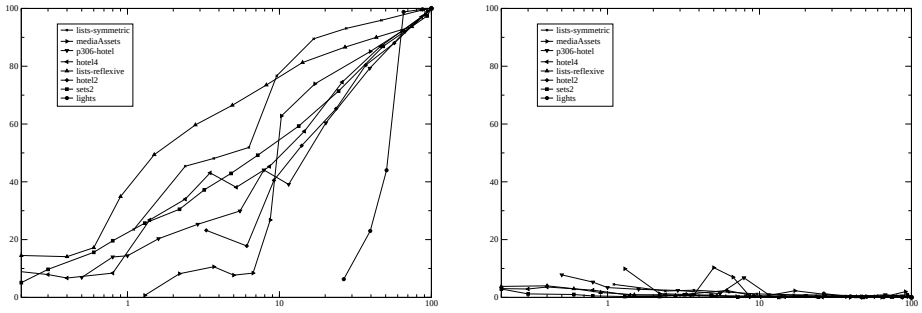

Figura 32. Hit_a y $Error_a$ para 200 cláusulas

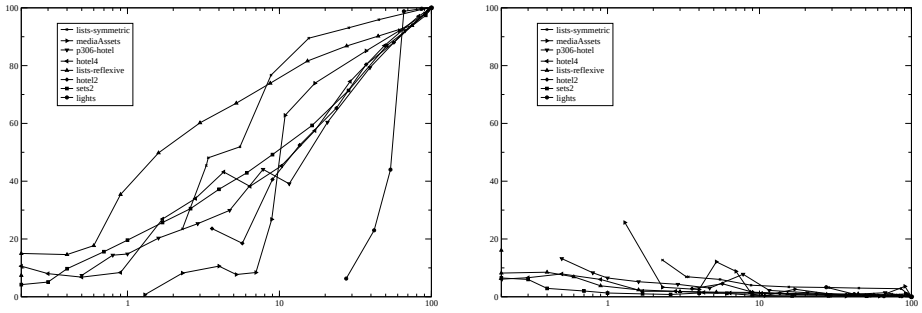

Figura 33. Hit_a y $Error_a$ para 500 cláusulas

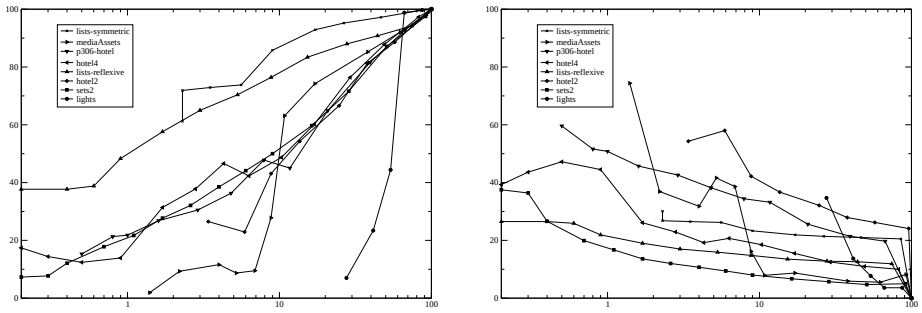

Figura 34. Hit_a y $Error_a$ para 10000 cláusulas

7. Conclusiones y trabajo futuro

En este trabajo de tesis mostramos que cuando un proceso de verificación iniciado por **Alloy Analyzer** es interrumpido, es posible devolver información acerca de la propensión de ciertas porciones de la especificación a pertenecer al **unsat core**. Alteramos levemente la implementación actual de **Alloy Analyzer** para realizar experimentos que resultaron satisfactorios, tanto en términos de la calidad de las aproximaciónes realizadas como en los tiempos en que se obtuvieron dichas aproximaciones.

Vimos también que usar la actividad de las cláusulas aprendidas definida en términos de la participación en los conflictos producidos durante el proceso de resolución es una buena heurística.

Además, el buen comportamiento de esta técnica sugiere pensar que también puede servir para problemas SAT que no provengan necesariamente de **Alloy**. Incluso abordar el problema de solucionar SAT de forma incompleta o aproximada como lo hacen herramientas como **GSAT** o **WalkSAT**, las cuales están enfocadas en términos de una cantidad finita de iteraciones, con **hot core** no hace falta especular con una cantidad máxima de iteraciones sino que en un determinado instante es posible interrumpir el proceso para obtener una solución aproximada, en este caso en términos de aproximación al **unsat core**.

Por útlimo notamos que en la traducción de **Alloy** a CNF es posible identificar una gran diferencia en cuanto a su granularidad, o sea, la convergencia en **Alloy** es bastante más veloz que a nivel CNF. Esto se debe a que cada fórmula **Alloy** se traduce en varias cláusulas CNF. Mientras que el **Sat-Solver** debe tener en cuenta todas las cláusulas para asegurar insatisfactibilidad, más de una de esas cláusulas corresponde a la misma fórmula **Alloy**. Esto hace pensar que podría ser posible resaltar la especificación **Alloy** con distintos colores teniendo en cuenta los diferentes niveles de actividad de las cláusulas CNF, con el fín de obtener una visualización más exacta del proceso de resolución a nivel **Sat-Solver**.

Sin embargo queda mucho trabajo por hacer como por ejemplo:

- Investigar heurísticas nuevas para identificar el **hot core**. La heurística actual sólo identifica un conjunto de tamaño fijo. Es de esperar que el tamaño de este conjunto debería depender del tamaño de la base de cláusulas inicial. Por lo que podríamos identificar un conjunto de cláusulas con una actividad por encima de un determinado percentil.
- En este trabajo sólo comparamos contra los **unsat core** producidos por un algoritmo de extracción bastante sencillo. También quisieramos comparar la heurística frente a **unsat cores** obtenidos con otro algoritmos de extracción.
- Evaluar la posibilidad de que intervenir en **Alloy Analyzer** de manera tal que sea posible un resaltado de la especifición en tiempo real, de manera tal no sea necesario interrumpir el proceso de resolución para obtener información acerca de cómo se esta llevando el proceso de resolución a nivel del **Sat-Solver**.

- Finalmente, en contraparte al caso en que la especificación sea insatisfacible, para el caso en que el análisis si produzca un contraejemplo, estamos interesados en investigar la posibilidad de producir un *posible contraejemplo*, que se presente al usuario dela misma menara en se presentan los contraejemplos en **Alloy Analyzer** luego determinado el proceso de resolución.

Referencias

[BKL83] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of finite simple groups. In *SFCS '83: Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 162–171, Washington, DC, USA, 1983. IEEE Computer Society.

[BKS03] Paul Beanie, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1194–1201, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[Bro87] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[CGLR96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. pages 148–159. Morgan Kaufmann, 1996.

[Chu36] A Church. A note on the entscheidungsproblem. *J. of Symbolic Logic*, (1), 1936.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[ES03] Niklas Een and Niklas SÃűrensson. An extensible sat-solver [ver 1.2]. 2003.

[FM] John Franco and John Martin. *A History of Satisfiability*, chapter 1, pages 3–74.

[GN02] E. Goldberg and Y.Ñovikov. Berkmin: A fast and robust sat solver, 2002.

[Jac00] Daniel Jackson. Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes*, 25(6):130–139, 2000.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. 2006.

[JGF] Carlos Lopez Pombo Juan Galeotti, Nicolas Rosner and Marcelo Frias. Distributed sat-based analysis of object oriented code.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[Mor75] M Mortimer. On languages with two variables. zeitsch. f. math. logik und grundlagen d. *Math*, (21), 1975.

[Sco62] D Scott. A decision method for validity of sentences in two variables. *Journal of Symbolic Logic*, (27):37–7, 1962.

[Shl07] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 155(12):1539–1548, 2007.

[Sil99] ao P. Marques Silva, Jo˙ The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA '99: Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, London, UK, 1999. Springer-Verlag.

[SSJ+03] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *19th IEEE International Conference on Automated Software Engineering*, 2003.

[TCJ08] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[TJ07] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '07)*, 2007.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.

[ZM03] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.