DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Aumentando la capacidad de cómputo de Alloy Analyzer mediante verificación incremental de modelos

18 de Agosto de 2010

Brian J. Cardiff
bcardiff@dc.uba.ar

Directores

Frias, Marcelo Fabian
mfrias@dc.uba.ar

Galeotti, Juan Pablo
jgaleotti@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Para Alma y Vero
por ser el abrigo de mi corazón

Índice

1. Resumen

En este trabajo se presentan estrategias para suplir los problemas de capacidad para resolver modelos Alloy[Jac06]. De forma tal que, aunque no siempre se obtengan mejoras de tiempo, sí se es capaz de resolver instancias que antes no se podían.

Las diversas alternativas exploradas son automatizables y no requieren datos adicionales con respecto al Alloy Analyzer. Se opera mediante el uso parcial e incremental de la información del modelo Alloy.

Uno de los resultados alcanzados es la resolución de un problema presentado en [Zav06] para instancias que no pudieron ser resueltas con la versión actual de la herramienta.

2. Introducción

En el área de Ingeniería de Software se investigan métodos para mejorar la calidad de los sistemas. Las técnicas empleadas suelen ser clasificadas en los siguientes grupos:

Simulación donde se realizan supuestas ejecuciones usando una abstracción o modelo del sistema. Usualmente previo a la construcción propiamente dicha del sistema en cuestión.[IEE04]

Testing donde se intenta plantear casos interesantes en los cuales evaluar el comportamiento real del sistema con respecto al esperado. Se busca que el conjunto de casos de prueba cumpla cierto criterio de cobertura, por ejemplo cobertura de decisiones, a fin de medir qué porción del sistema está siendo puesto a prueba. A diferencia de la simulación, el testing se realiza sobre el sistema concreto.[IEE04]

Demostración (semi)automática donde se realiza un análisis deductivo formal del sistema para demostrar fehacientemente la validez de alguna propiedad.[EMCJ99]

Análisis de modelos donde se analiza un modelo del sistema para averiguar si cumple cierta propiedad. Dentro del análisis de modelos se pueden diferenciar el *model checking* y la búsqueda de contraejemplos. En el primero se suele verificar la interacción de sistemas concurrentes de estados finitos, por lo general para garantizar la ausencia de *deadlocks* o alguna propiedad de *safety*. En el segundo, en cambio, se analiza un modelo dentro de un determinado tamaño a fin de garantizar la validez de alguna propiedad, por ejemplo estructural.[EMCJ99]

Las demostraciones formales brindan certeza absoluta sobre la validez de la propiedad que se está analizando, pero lamentablemente muy difícil de automatizar para el caso general. Otro factor que dificulta la adhesión de éste y otros tipos de análisis estáticos es que pueden reportar fallas espúreas. Algunas potenciales fallas que pueden encontrarse en el análisis son inválidas y por lo tanto espúreas. Esto puede ocurrir por lo estricto que es una demostración frente a algunas características del sistema.

Utilizar casos de prueba requiere datos adicionales además del sistema en cuestión. Si bien hay varios intentos por automatizar la creación de los casos de prueba, persiste el hecho de que su utilización no constituyen una garantía de la ausencia de errores. Garantía que sí se obtiene mediante las demostraciones.

Los análisis de modelos presentan un balance interesante entre automatización y calidad de la respuesta. El presente trabajo está relacionado con la búsqueda de contraejemplos. En esta técnica, se recorren todos los posibles estados dentro de ciertas cotas. En la siguiente sección se muestra qué forma tienen estas cotas y cómo es que, al disponer de ellas, se puede recorrer un conjunto de estados posible. En caso de encontrar un ejemplo del sistema en el cual la propiedad no se sostenga, se dispondrá entonces de un testigo que indica la invalidez de la propiedad en análisis. Si se agotan todos los estados, se concluye que la propiedad es valida para el sistema, pero solo dentro de estas cotas. Esta técnica puede requerir considerable capacidad de cómputo, a veces impidiendo un análisis práctico en términos de tiempo u otros recursos. Es por esto que se estudia cómo vencer dicho límite.

2.1. Análisis de Modelos - Búsqueda de contraejemplos

Una de las alternativas para tratar de saber si cierta propiedad se cumple en un sistema dado es la búsqueda de contraejemplos. Lo primero que se necesita es una descripción formal del sistema bajo análisis, esto conforma el modelo. Además se codifica la propiedad a verificar o aserción. Si no es válida, ha de existir un contraejemplo que satisface la descripción del sistema (modelo), pero en el cual la propiedad no se cumple.

Los siguientes son ejemplos de modelos y aserciones. El primero, de tinte más teórico, los siguientes relativos a implementaciones en programas y el último alejado del ambiente informático a priori, pero no por eso menos ambicioso.

1. En un grafo dirigido, no vacío, sin ciclos hay algún nodo fuente y algún nodo sumidero.
2. Estructuras de datos con operaciones y sus propiedades que se esperan invariantes.
3. Algoritmos de control de procesos que eviten *deadlocks*.
4. Dadas las reglas de control de partida de trenes en una terminal, ver que se impiden colisiones de los mismos.[Jac02]

El análisis de modelos es estudiado siempre como una técnica automática, delegando a la herramienta lidiar con el espacio de búsqueda para determinar la validez o no de la propiedad.

2.2. Alloy

Alloy es un lenguaje que permite describir modelos. Los modelos comienzan con la declaración de dominios (o signaturas). Éstos representan la clase de elementos que existirán en el modelo. A cada signatura se le adjudican campos o atributos. Un elemento de una

signatura tendrá un valor para cada campo definida en al misma. Un modelo para verificar la propiedad 1 de los ejemplos anteriores incluiría la siguiente descripción para los nodos:

```
sig Node { next : set Node }
```

En donde cada nodo tiene un conjunto de nodos conectados a él. El campo next suele ser visto como una relación entre pares de nodos. La interpretación relacional de los campos es usada ampliamente en la comunidad y permite el uso de conceptos adicionales para expresar modelos, como el uso de clausuras transitivas que se muestra más adelante. El modelo presentado describe los grafos dirigidos. Para restringir el análisis a los grafos no vacíos se debe incluir un axioma o **fact** que asegure la existencia de algún elemento dentro del conjunto de valores de Node.

```
fact { some Node }
```

De la misma forma se debe lograr que el modelo represente solamente los grafos sin ciclos. Alloy permite expresar clausuras transitivas ($\hat{}$) y clausuras reflexo-transitivas ($*$). Las primeras pueden ser usadas para expresar de manera concisa la inexistencia de ciclos. Argumentando que desde cualquier nodo no debe suceder que, recorriendo el campo next, lleguemos al mismo nodo.

```
fact NoCicles { all n : Node | not (n in n.^next) }
```

Dado que el modelo está restringido a los grafos dirigidos no vacíos sin ciclos, lo que resta es expresar la propiedad a verificar. A fin de simplificar la lectura se pueden definir predicados. Por ejemplo, para definir cuando un nodo es sumidero (*Sink*) se expresa que el conjunto de nodos conectados desde él (next) debe ser vacío.

```
pred Sink[n: Node] { no n.next }
```

De forma similar, se define que un nodo es fuente (*Source*) cuando la proyección del conjunto universal (univ) por la relación next no debe contener al nodo, o sea, que no hayan ejes que incidan en el nodo.

```
pred Source[n: Node] { not (n in univ.next) }
```

Con estos predicados la propiedad que se desea verificar se expresa de la siguiente manera mediante el uso de cuantificadores existenciales. Recordando, se desea verificar que, en la clase de grafos representados por el modelo, siempre hay un nodo fuente y un nodo sumidero.

```
assert SinkAndSource {
  (some s : Node | Source[s]) && (some s : Node | Sink[s]) }
```

Es preciso notar que en ningún lugar del modelo se expresaron cotas a la cantidad de nodos. Esta cota aparece recién al realizar el análisis mediante la sentencia **check**.

```
check SinkAndSource for 3
check SinkAndSource for 5
```

Estas sentencias, al ser ejecutadas por medio del Alloy Analyzer, verificarán la propiedad SinkAndSource para grafos de a lo sumo 3 y 5 nodos respectivamente. El autor del

modelo puede ir modificando estas cotas a fin de lograr un análisis que le ofrezca la suficiente seguridad para quedarse conforme con el resultado. Dependiendo del modelo la cota variará. Por ejemplo para este caso una cota de 1 ó 2 nodos no es adecuada, mientras que 5, si bien parece bastante pequeño, ya es suficiente. Una cota adecuada es aquella que permite contemplar un conjunto de instancias significativas del modelo, y no sólo casos borde. En esta situación, grafos de uno o dos nodos son muy particulares. Considerar hasta cinco nodos ya incluye algunos grafos significativos que aportan mayor confiabilidad al resultado del análisis. El código 1 en la página 4 muestra el modelo completo.

```
1  sig Node { next : set Node }
2  fact { some Node }
3  fact NoCicles { all n : Node | not (n in n.^next) }
4  pred Sink[n: Node] { no n.next }
5  pred Source[n: Node] { not (n in univ.next) }
6
7  assert SinkAndSource {
8    (some s : Node | Source[s]) && (some s : Node | Sink[s]) }
9  check SinkAndSource for 3
10 check SinkAndSource for 5
```

Código 1: Ejemplo introductorio a alloy

Recién al introducir las cotas es que el modelo se finitiza y pueden realizarse los procesos implementados en Alloy para su análisis efectivo. En la sección 2.3 se explican éstos procesos. En la figura 1 se muestra un esquema sobre el funcionamiento de Alloy Analyzer. Al analizar un modelo para verificar una propiedad, los resultados posibles son a) concluir que la aserción es válida dentro del *scope* elegido, o b) encontrar un contraejemplo que muestre que la propiedad no se cumple. En la sección 2.3 se aumenta el detalle del diagrama.

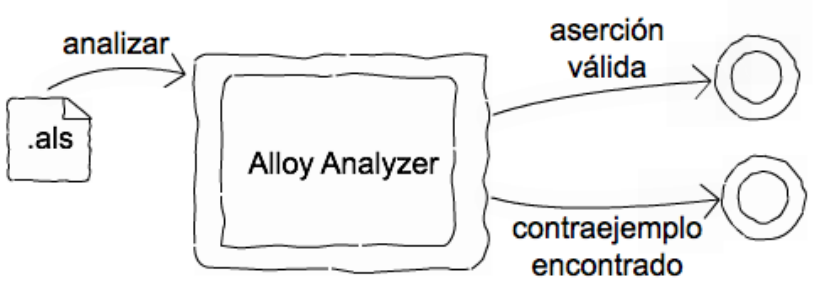


Figura 1: Caso de uso Alloy

Las siguiente son algunas propiedades destacables con las que cuenta Alloy.

1. El análisis se realiza dentro de un dominio finito. El análisis es correcto (*sound*) pero incompleto, dado que no se producen falsos positivos, pero no se analizan instancias

mayores a las cotas. A pesar de esto es válido decir que el análisis sí es "completo dentro de las cotas" ya que no pasará por alto la existencia de algún contraejemplo dentro de estos términos. El análisis en escenarios pequeños suele ser muy eficaz para la detección de errores.

2. La especificación en sí no impone ninguna restricción a las signaturas utilizadas. Las cotas de tamaño se introducen al escribir el comando de verificación.

3. El lenguaje es sumamente declarativo. Al disponer de operadores lógicos y herramientas de álgebra relacional se pueden lograr descripciones muy expresivas y claras de un sistema.

4. El análisis es totalmente automático, lo cual no sucede con otros lenguajes de especificación como Z u OCL.

5. Si bien en el contexto de este trabajo sólo se utiliza la funcionalidad de Alloy Analyzer de verificar una propiedad mediante la búsqueda de contraejemplos, el mismo también dispone de la funcionalidad de, dada una propiedad, encontrar una instancia de ejemplo donde dicha propiedad sea satisfecha.

Otra particularidad de Alloy es que el lenguaje puede ser interpretado desde tres puntos de vista. Estos puntos de vista suelen usarse por el autor del modelo para poder expresar de la mejor forma distintas fórmulas.

En primer lugar se puede usar un nivel de abstracción elevado con respecto a los mecanismos subyacentes del lenguaje y tener en mente un paradigma orientado a objetos. Como se mostró en el ejemplo la sintaxis es similar a la presente en lenguajes de programación orientados a objetos. Esto a su vez permite una mejor incorporación de la herramienta por la comunidad de programadores al encontrarse con conceptos ya conocidos.

En segundo lugar, para codificar fórmulas más expresivas se suele tener en cuenta la teoría de conjuntos para interpretar el lenguaje. Ejemplo de esto son las clausuras y la aplicación de campos a conjunto de valores en lugar de solamente un elemento.

Por último, el nivel inferior de abstracción es el que corresponde a interpretar el lenguaje como átomos y relaciones. Éste coincide con la semántica subyacente de Alloy, aunque es pertinente aclarar que rara vez es utilizado.

Las especificaciones en lenguajes declarativos como Alloy pueden sufrir de algunos problemas que conllevan a un análisis no válido. Como se lo explica en [TCJ08] estos problemas podrían ocurrir en axiomas muy restrictivos, en propiedades muy débiles o cotas muy pequeñas para el análisis. Por tal motivo, en caso de que la herramienta indique que la propiedad se satisface, es necesario ver de alguna manera qué cobertura del modelo se alcanzó al indicar esto. La medida de cobertura es simplemente si cierta parte del modelo fue usada o no para deducir la insatisfacibilidad del mismo. La parte del modelo que se utiliza es denominada *unsat core*. En las secciones 2.3 y 2.5 se detalla cómo es el proceso de extracción del *unsat core*.

Por ejemplo si una propiedad siempre se la encuentra válida luego de variar las cotas de análisis, pero el *unsat core* nunca cubre nada de la aserción, debe de suceder que los axiomas

son muy restrictivos y no hay forma de encontrar siquiera una instancia del modelo. Por lo tanto la propiedad es válida, pero por vacuidad.

En [TCJ08] se identifican patrones de *unsat core* con diversas anomalías del modelo. Además puede ser usado en forma temprana por el autor del modelo para ir comprobando que no se cayó en una de estas situaciones, y en tal caso encontrar el origen. En [SSJ+03] se muestra también otros escenarios donde el uso del *unsat core* es bastante útil.

Por tal motivo es deseable mantener la capacidad de informar un *unsat core* ya que es sumamente valioso a fín de comprender la validez del análisis.

2.3. Traducción de Alloy a lógica proposicional

Como se mencionó, la verificación de una especificación Alloy es un proceso automático. En la presente sección se muestra cómo es que opera el Alloy Analyzer para lograr esto.

Las especificaciones Alloy en sí pueden describir modelos infinitos. En el caso general la verificación de modelos infinitos es indecidible, dado que corresponde a un fragmento que abarca a la lógica de primer orden.

Por lo tanto el primer paso para lograr que el análisis sea automatizable y computable desde el punto de vista teórico, o sea sin tener en cuenta limitaciones de recursos, es restringir el poder expresivo para que el proceso de verificación sólo necesite tener en cuenta un conjunto finito de instancias.

Una técnica comúnmente usada para restringir el poder expresivo es directamente cambiar el lenguaje. Alloy en cambio utiliza otro recurso, dejando un lenguaje de especificación formal sencillo, elegante pero poderoso.

Como se aprecia en los ejemplos anteriores, la verificación de una propiedad siempre se la realiza para dominios finitos. El solo hecho de acotar la cantidad de elementos en las signaturas torna la verificación computable. La restricción de dominios finitos es tan fuerte que además permite la traducción a lógica proposicional de cualquier especificación Alloy.

El limitar los dominios a ser finitos no es algo drástico ya que siempre los sistemas usan representaciones finitas, pero además es compartido que la mayoría de los errores que pueda tener un sistema, suelen aparecer ya en instancias pequeñas. Esta hipótesis es conocida como *small scope hypothesis* y es uno de los pilares en lo que se basa la utilidad del análisis de modelos. En [ADKM02] se estudia cuán certero es basarse en dicha hipótesis.

Cuando se trata de verificar una propiedad, se busca una instancia válida del modelo en la que dicha propiedad no esté satisfecha. Una instancia se define por cómo se relacionan los distintos elementos de cada dominio. Al hacer finitos los dominios, una relación $R \times S$ se puede representar como una matriz de $\#R \times \#S$ variables proposicionales que indican cuáles túplas pertenecen a la relación. El mismo recurso se utiliza para representar conjuntos y relaciones entre más de dos signaturas.

Por ejemplo en la sección anterior se trata un modelo con signatura Node que contiene un campo next. El campo puede representarse como una matriz de $\#Node \times \#Node$. Supongamos que en el modelo hay un axioma como el del código 2 en donde se indica que para todo nodo hay al menos uno relacionado a través de next. Si el análisis se lo hace para exactamente tres nodos, la relación next se representaría con las variables $n_{x,y}$ para

$0 \leq x, y < 3$ en donde $n_{x,y}$ es verdadera sí y solo si, $node_y \in node_x.next$. Entonces el axioma podría expresarse como la fórmula 2.1.

```
1 sig Node { next : set Node }
2 fact { all n : Node | some n.next }
```

Código 2: Ejemplo para traducción de axiomas

$$(n_{0,0} \vee n_{0,1} \vee n_{0,2}) \wedge (n_{1,0} \vee n_{1,1} \vee n_{1,2}) \wedge (n_{2,0} \vee n_{2,1} \vee n_{2,2}) \tag{2.1}$$

La fórmula 2.1 está en CNF (*conjunctive normal form*). Una fórmula f está en CNF si tiene la siguiente estructura:

$$f = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

donde cada cláusula C_i, $1 < i \leq n$ es una disyunción de literales.

$$C_i = l_1^i \vee l_2^i \vee \ldots \vee l_m^i$$

donde cada literal l_j^i es una variable proposicional o su negación.

Las fórmulas CNF suelen ser expresadas en notación de conjuntos dado que las operaciones a aplicar están determinadas. Por ejemplo $p \wedge (q \vee \neg p \vee \neg r)$ puede denotarse $\{ \{ p \}, \{ q, \overline{p}, \overline{r} \} \}$.

Este tipo de traducciones no concluye siempre en fórmulas CNF, pero sí en fórmulas proposicionales. Parte del proceso de traducción es pasar de una fórmula proposicional a una en CNF. No hay una única manera de hacer esta última etapa. La forma trivial de hacer la normalización tiene la desventaja de aumentar exponencialmente el tamaño de la fórmula. Es por eso que se crearon alternativas para conseguir la normalización sin aumentar exponencialmente el tamaño de la fórmula. Para lograrlo se introducen variables proposicionales adicionales. Es por eso que en las fórmulas resultantes de una traducción Alloy se distinguen la cantidad de variables primarias, que corresponden a las que determinan la instancia del modelo, y la cantidad de variables totales que incluyen además las variables creadas por la traducción [Tse68, JS04].

Si se define como T_B a la traducción de una fórmula Alloy a lógica proposicional para las cotas de signaturas B, entonces un modelo Alloy (con cotas incluidas) es equivalente a la fórmula 2.2.

$$\Big(\bigwedge_{f \in Facts} T_B(f) \Big) \rightarrow T_B(a) \tag{2.2}$$

en donde $Facts$ es el conjunto de axiomas y a es la propiedad a verificar.

La aserción será válida (dentro de las cotas B) si la fórmula 2.2 es una tautología. Para saber si dicha fórmula es una tautología se utiliza un *sat solver* para buscar una valuación en donde el antecedente sea verdadero, pero el consecuente no. O sea, que se utiliza un *sat solver* para encontrar una valuación que haga verdadera la fórmula 2.3 que es equivalente a la fórmula 2.4

$$\neg\Big(\Big(\bigwedge_{f\in Facts} T_B(f)\Big) \rightarrow T_B(a)\Big) \tag{2.3}$$

$$\bigwedge_{f\in Facts} T_B(f) \wedge \neg T_B(a) \tag{2.4}$$

Es por esto que si se encuentra una valuación para la fórmula 2.4, se corresponde a encontrar un contraejemplo de la propiedad Alloy. El valor de las variables proposicionales primarias determinan la instancia del modelo. En caso de no encontrar valuación alguna, se corresponde con que la aserción es válida para el modelo dentro de las cotas del análisis. Ésto último se ilustra en la figura 2.

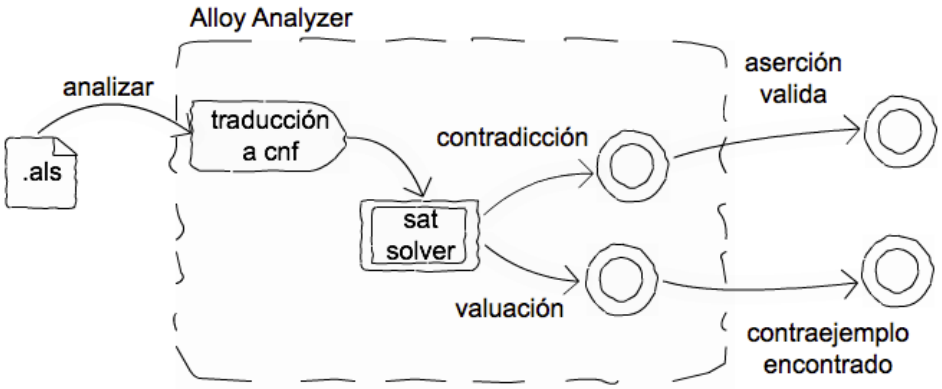


Figura 2: Caso de uso Alloy

En el caso donde se quiera contar con un *unsat core* del modelo Alloy, no basta con saber simplemente que la fórmula es insatisfasible (unsat). Además se debe contar con un *unsat core* que corresponde al conjunto de cláusulas usado para deducir la instatisfacibilidad. Cada cláusula es mapeada con la parte correspondiente del modelo para generar el *unsat core* a nivel Alloy. No todos los *sat solver* soportan la extracción de un *unsat core*. Aquellos que sí lo soportan son denominados *sat prover*. Con el Alloy Analyzer, por ejemplo se distribuyen dos versiones de MiniSat, una que soporta y otra que no soporta dicha extracción.

2.3.1. KodKod

Actualmente Alloy no traduce la especificación directamente a una fórmula proposicional. Como paso intermedio se utiliza KodKod [TJ07]. No se detallará sobre KodKod en el presente trabajo, pero al presentar Alloy es imposible no mencionarlo brevemente.

En los inicios Alloy Analyzer era una herramienta diseñada de forma monolítica que interactuaba con *sat solvers* externos. Desde la cuarta versión se lo desglosó en diversas capas dando origen así a KodKod.

La expresividad de Alloy y de KodKod son equivalentes. Difieren en cómo plantear las especificaciones. Alloy es un lenguaje más cómodo para ser utilizado por un usuario. Mientras tanto KodKod es de más bajo nivel, aunque ambos comparten la misma semántica relacional.

Ya se mencionó cómo se conforma una especificación Alloy y cómo una instancia en particular se determina por los átomos en los dominios y el valor de las relaciones que, en conjunto, satisfacen los distintos axiomas. En KodKod las relaciones y signaturas no son tipadas, éstas se representan como conjuntos de aridad adecuada con átomos genéricos. Una de las responsabilidades del Alloy Analyzer es revisar el tipado de las expresiones que conforman la especificación.

Una especificación KodKod se conforma por a) la enumeración de los posibles átomos en el universo, b) la declaración de las relaciones y c) una fórmula de lógica relacional que corresponde con los axiomas y la negación de la aserción. La declaración de las fórmulas tiene como relevante la aparición de cotas: una inferior L_R y una superior U_R para cada símbolo relacional R.

Una instancia en KodKod es una asignación concreta a los conjuntos y relaciones que además de satisfacer la fórmula de lógica relacional debe, para cada relación con símbolo R, contener todos los elementos que aparecen en L_R y contener a lo sumo todos los elementos de la relación U_R. De esta manera se da semántica a las cotas de las relaciones.

Las distintas cotas son creadas por Alloy al hacer la traducción a KodKod usando como dato la aridad de las relaciones y las operaciones aplicadas. El objetivo es reducir el espacio de búsqueda al fijar cotas los más ajustadas posibles.

Mucho conocimiento se encapsuló y se seguirá encapsulando en KodKod para permitir que otras herramientas al basarse en éste dispongan de diversas optimizaciones y mejoras logradas a lo largo de distintos trabajos. Entre otras son: mejores codificaciones para operaciones del álgebra relacional, mejores codificaciones a CNF y ruptura de simetría.

La ruptura de simetría es la eliminación de instancias del espacio de búsqueda que son esencialmente permutaciones de otras. En general, el aumento de la velocidad es importante y sin ésta incorporación muchos casos de estudio serían intratables [CGLR96, Shl07, GRLPF10].

2.4. Complejidad de SAT

SAT es el problema de encontrar una valuación que haga verdadera una fórmula proposicional. Este problema es muy importante y estudiado en el área de la ciencia de la computación. Como está demostrado en el teorema de Cook[Coo71] pertenece a la clase NP-completo. Por lo tanto en el peor caso resolver una instancia de SAT puede requerir una inmensa cantidad de recursos.

El problema SAT no es el único NP-completo, otras variantes también lo son. Resolver SAT para fórmulas CNF, sigue siendo NP-completo. Éstas últimas son las que se utilizan para verificar los modelos Alloy. Incluso restringiendo a que las cláusulas de una fórmula CNF sólo cuenten con tres literales, problema conocido como 3-SAT, se continúa en la clase NP-Completo.

A pesar de que formalmente no se pueda esperar encontrar una solución eficiente al problema, a lo largo de las últimas décadas se han elaborado heurísticas capaces de lidiar satisfactoriamente con muchas de las fórmulas que aparecen en la práctica.

Por ser impracticable en el caso general, se debe contemplar que el *sat solver* no cuente con los recursos necesarios para resolver exitosamente la búsqueda de la valuación, ya sea que la misma exista o no.

2.5. SAT Solvers

En la actualidad, varios *sat solvers* implementan variaciones del algoritmo denominado DPLL [DLL62]. MiniSat [ES03], distribuido con Alloy Analyzer y utilizado extensamente durante el presente trabajo, es un *sat solver* que extiende DPLL y será usado en esta sección para introducir el funcionamiento de los *sat solvers*.

Como se mostró en la figura 1 dada una fórmula CNF, se debe encontrar, en caso de existir, una valuación que la satisfaga. De no existir valuación alguna se debe indicar que la fórmula es insatisfacible e incluir una justificación de este hecho.

El proceso recorre el espacio de búsqueda de todas las posibles valuaciones mediante el uso de asignaciones parciales. El algoritmo DPLL básico puede presentarse en forma recursiva, pero a fin de introducir algunos mecanismos usados en la actualidad se utiliza una versión iterativa del mismo. Durante la ejecución del algoritmo se mantiene el conjunto de cláusulas en cuestión y una valuación parcial de las variables que aparecen en la fórmula.

Una cláusula unitaria (formada solamente por un literal) restringe el espacio de búsqueda, pues de existir una valuación para la fórmula debería respetar el valor impuesto a la variable por el literal. Por ejemplo con las cláusulas $\{\overline{p}\}$ y $\{q\}$ ya se sabe que el único valor posible para p es 0 y para q es 1. Esta información es incorporada a la valuación. Propagando la información de los valores fijados, se simplifican las cláusulas. A partir de esto una cláusula puede a) ignorarse por estar ya satisfecha, en el ejemplo anterior sería el caso de $\{\overline{p}, r\}$. Podría generar b) una cláusula vacía que indica un conflicto y por ende la valuación no puede ser extendida para satisfacer la fórmula, sería el caso con una cláusula $\{p, \overline{q}\}$. Otra opción sería c) dar origen a cláusulas con literales de variables no asignadas, por ejemplo $\{p, s, \overline{t}\}$ se reduce a $\{s, \overline{t}\}$ pues p es 0. Dentro de éste último caso se puede distinguir cuando la propagación d) crea nuevas cláusulas unitarias, como sucedería en $\{\overline{q}, t\}$ de donde se puede incorporar que t debe ser 1. Éste proceso de denomina **propagación** y se itera hasta que no haya más información que propagar.

Si se logró obtener una valuación completa, entonces se concluye que la fórmula es satisfacible y se dispone de una valuación que sirve de testigo.

En caso de no haber detectado un conflicto, se debe continuar con el recorrido del espacio de búsqueda. Dentro de las variables que aún no están asignadas, se escoge alguna y se le asigna un valor. En caso de que esta decisión concluya luego de potencialmente varios pasos, se intentará con el otro valor para así contemplar todas las posibilidades. Cada vez que se realiza una etapa de **decisión** se dice que se pasó a un nuevo *nivel de decisión*. Estos niveles simbolizan las alternativa en el árbol de búsqueda y funcionan como

```
loop
  propagar asignaciones
  if no se detectaron conflictos then
    if todas las variables están asignadas then
      return SAT
    else
      escoger variable no asignada y un valor
    end if
  else
    analizar el conflicto
    if no hay decisiones para deshacer then
      return UNSAT
    else
      retroceder decisiones según conflicto
    end if
  end if
end loop
```

Cuadro 1: Pseudocódigo de *sat-solver*

marca para retornar a este estado. Entre un nivel de decisión y otro se pueden realizar una asignación por decisión y varias por propagación.

En caso de haber generado un conflicto, o sea alguna cláusula vacía, se procede a analizar su motivo. Si una cláusula se redujo a una vacía es porque todos sus literales son falsos según la asignación. Detectar el origen del conflicto sirve para volver al punto donde efectivamente se lo generó y además para recorrer de forma más provechosa lo que resta del espacio de búsqueda, en otras palabras, *aprender del conflicto*. Por ejemplo, si la cláusula que se redujo hasta generar una vacía era originalmente $\{p, \overline{q}, r\}$ deberíamos impedir que suceda nuevamente $\overline{p} \wedge q \wedge \overline{r}$. Ahora bien, si el valor de p fue propagado a partir de una cláusula $\{x, y, \overline{p}\}$ por ser $x = y = 0$, se puede plantear el conflicto como $\overline{x} \wedge \overline{y} \wedge q \wedge \overline{r}$, sin utilizar p. Si se incorpora la cláusula $\{x, y, \overline{q}, r\}$ se evita caer en la misma situación. Notar que estas la cláusulas aprendidas son implicaciones de las cláusulas originales. Una vez incorporada a la lista de cláusulas, se procede a retroceder en el árbol de decisiones para continuar con la búsqueda de una valuación que satisfaga la formula. Así como se analizó la propagación realizada a p, puede continuarse con el resto de las variables hasta cumplir algún criterio. En última instancia el conflicto estará expresado únicamente en variables cuyo valor fue asignado por una decisión y no por una propagación.

En un algoritmo de *backtracking* usual se retrocedería un solo nivel en el árbol de decisiones. Aprovechando el análisis del conflicto, se puede retroceder directamente hasta un nivel de decisión en donde el conflicto no este presente. Esta técnica es denominada *non-chronological backtracking*.

En el cuadro 1 se muestra un pseudo-código que corresponde a MiniSat en donde se ubican las invocaciones a las distintas etapas ya mencionadas. Hay heurísticas y hechos

importantes sobre las estructuras de datos que se utilizan detalladas en [ES03].

En caso de concluir que la fórmula es insatisfacible, se utilizan todas las cláusulas disponibles para obtener la traza de deducciones que permiten llegar a la cláusula vacía a partir de las originales. Ésta traza de deducciones constituye la justificación de por qué la formula es UNSAT. El subconjunto de cláusulas originales dentro de dicha traza dan lugar al *unsat core*. Detalles de este proceso de extracción del *unsat core* pueden encontrarse en [GKS08].

El *unsat core* hallado puede no ser minimal, en [TCJ08] se estudian distintas estrategias para intentar eliminar cláusulas innecesarias.

Situando el *sat solver* dentro de Alloy Analyzer, recordemos que: en caso de concluir que la fórmula es SAT significa que la especificación Alloy es inválida. A partir de la valuación que satisface la fórmula se arma una instancia del modelo Alloy que resulta en un contraejemplo. Por otro lado, si se concluye que la fórmula es UNSAT significa que que no existe un contraejemplo y por ende la especificación es válida (dentro de las cotas del análisis).

3. Minimización del modelo Alloy

Para verificar la validez de una afirmación se busca un contraejemplo. Dicho contraejemplo es una instancia válida del modelo, en el cual valen todas los axiomas (**fact**s) pero en el que la afirmación (**assert**) no vale. Que esta búsqueda no tenga éxito significa que el modelo es inconsistente con la negación del **assert**.

Dicha inconsistencia por lo general proviene de un subconjunto de la declaración del modelo. Por lo cual sin utilizar todos los axiomas también se suele poder verificar la propiedad. En los casos donde el análisis se hace impracticable, por motivos de tiempo o recursos, encontrar qué axiomas se pueden descartar puede permitir realizar exitosamente el análisis.

El primer intento por ampliar la capacidad de resolución del Alloy Analyzer [Cha08] es verificar la propiedad para un modelo con menos axiomas que el original. Este nuevo modelo será sintetizado de forma automática y es denominado modelo minimizado.

Como el lenguaje Alloy es sumamente declarativo, cada axioma suele ser muy expresivo. Al buscar qué axiomas se pueden ignorar para la verificación sería más propicio que los mismos tengan granularidad más fina, o sea que un predicado afecte a la menor cantidad de elementos posibles de las instancias modeladas. Para lograr axiomas menos expresivos se manipulará el modelo original mediante transformaciones sintácticas automatizadas.

En esta sección se usará como ejemplo ilustrativo una propiedad estructural de listas doblemente enlazadas. Dichas listas contendrán una referencia al primer y último nodo siempre que la misma no sea vacía. Cada nodo referencia a su vez al anterior y al siguiente nodo en caso de que éste último exista. Por lo tanto el modelo comienza con la declaración de signaturas y axiomas del código 3.

```
1 one sig List { first, last : lone Node }
2
3 sig Node { prev, next : lone Node }
```

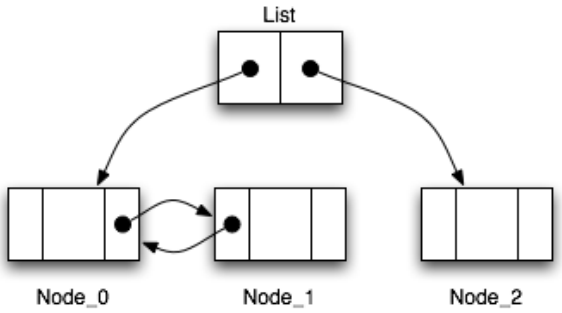


Figura 3: instancia que motiva el último axioma

```
4
5  fact { (no List.first) iff (no List.last) }
6
7  fact { no List.first.prev }
8  fact { no List.last.next }
9
10 fact { all n : Node | some n.next implies n.next.prev = n }
11 fact { all n : Node | some n.prev implies n.prev.next = n }
```

Código 3: Encabezado de dllist.als

Una instancia válida para la especificación del código 3 se muestra en la figura 3. Claramente, no corresponde a una lista doblemente enlazada. Por lo tanto se debe hacer más restrictiva la especificación. Entonces, es necesario agregar un axioma para asegurar que desde el primer nodo de la lista se puede llegar al último por medio de las referencias sucesivas de la relación next.

```
12 fact { List.last in List.first.(*next) }
```

Código 4: Continuación de dllist.als

La propiedad que se desea verificar establece que los nodos alcanzables por la relación next partiendo desde el primero de la lista, son los mismos que aquellos alcanzables por la relación prev partiendo desde el último de la lista.

```
13 assert P { List.first.(*next) = List.last.(*prev) }
14 check P for 4
```

Código 5: Continuación de dllist.als

3.1. Modelo instrumentado

La especificación dllist.als de la sección anterior puede verificarse exitosamente con Alloy Analyzer. La aserción resulta ser válida. En esta sección se introducen los conceptos básicos para modificar las especificaciones a fin de lograr axiomas con granularidad más fina. A fines demostrativos se utiliza la especificación dllist.als.

En el modelo actual los axiomas que comienzan con un cuantificador universal son aptos para modificarse. Las verificaciones se hacen para un tamaño de dominio fijo, en este caso 4, con lo que habrá un máximo de cuatro nodos y una lista (dado que es **one sig**). Dada ésta información, se introduce una signatura que permitirá hablar de todos los nodos que pueden llegar a existir.

```
one sig E { n1, n2, n3, n4 : lone Node }
fact { E.n1+E.n2+E.n3+E.n4 = Node }
```

Usando la signatura E se pueden reemplazar los axiomas de las líneas 7 y 8 obteniendo un modelo equivalente. Al modelo resultante se lo denomina "modelo instrumentado" y en el contexto del ejemplo será denominado M.

```
1  one sig List { first, last : lone Node }
2  sig Node { prev, next : lone Node }
3
4  one sig E { n1, n2, n3, n4 : lone Node }
5  fact { E.n1+E.n2+E.n3+E.n4 = Node }
6
7  fact { (no List.first) iff (no List.last) }
8
9  fact { no List.first.prev }
10 fact { no List.last.next }
11
12 fact { some E.n1.next implies E.n1.next.prev = E.n1 }
13 fact { some E.n2.next implies E.n2.next.prev = E.n2 }
14 fact { some E.n3.next implies E.n3.next.prev = E.n3 }
15 fact { some E.n4.next implies E.n4.next.prev = E.n4 }
16 fact { some E.n1.prev implies E.n1.prev.next = E.n1 }
17 fact { some E.n2.prev implies E.n2.prev.next = E.n2 }
18 fact { some E.n3.prev implies E.n3.prev.next = E.n3 }
19 fact { some E.n4.prev implies E.n4.prev.next = E.n4 }
20
21 fact { List.last in List.first.(*next) }
22
23 assert P { List.first.(*next) = List.last.(*prev) }
24 check P for 4
```

Código 6: Modelo instrumentado M

En la sección 3.4.2 se muestran otras transformaciones aplicables en función del modelo original para obtener el modelo instrumentado.

3.2. Análisis iterativo

En lugar de verificar todo el modelo instrumentado directamente con Alloy Analyzer, teniendo en cuenta que la fórmula CNF resultante a partir de él puede ser demasiado

grande para ser tratada, se inicia un análisis quitando todos los axiomas propios de las listas enlazadas. De esta manera el modelo inicial M_0 es el siguiente.

```
1 one sig List { first, last : lone Node }
2 sig Node { prev, next : lone Node }
3 one sig E { n1, n2, n3, n4 : lone Node }
4
5 fact { E.n1+E.n2+E.n3+E.n4 = Node }
6
7 assert P { List.first.(*next) = List.last.(*prev) }
8 check P for 4
```

Código 7: Modelo inicial M_0

Usando Alloy Analyzer se puede buscar un contraejemplo para esta propiedad. Notar que al eliminar los axiomas de las listas enlazadas, los contraejemplos pueden ser espúreos dado que pueden no ser instancias del modelo original. Para M_0, el contraejemplo es el de la figura 4.

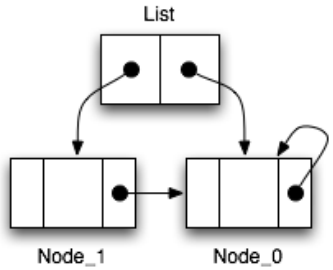


Figura 4: Contraejemplo

El siguiente paso es encontrar, si existe, algún axioma de M no incluído en M_0 que descarte el contraejemplo de la figura 4. Para encontrar dicho axioma se puede usar Alloy creando un modelo nuevo que contenga, además de código 6, axiomas que fijen los elementos de la signatura Node y todas las relaciones existentes a fin de representar el contraejemplo encontrado. El modelo resultante se muestra en el código 8 y se lo denomina V_0. La signatura C es utilizada para nombrar los elementos del contraejemplo. Los axiomas de las líneas 24-35 son para fijar el contraejemplo. Es necesario notar que el modelo, si bien contiene al modelo original instrumentado, se espera que sea analizable con menos recursos. La signatura C y los axiomas para fijar el contraejemplo deberían orientar al analizador en el recorrido del espacio de búsqueda.

```
1 one sig List { first, last : lone Node } /* signaturas originales */
2 sig Node { prev, next : lone Node }
3 one sig E { n1, n2, n3, n4 : lone Node } /* signatura de instrumentación */
4 fact { E.n1+E.n2+E.n3+E.n4 = Node }
5
6 fact { (no List.first) iff (no List.last) }  /* todos los facts de M */
```

```
 7 fact { no List.first.prev }
 8 fact { no List.last.next }
 9 fact { some E.n1.next implies E.n1.next.prev = E.n1 }
10 fact { some E.n2.next implies E.n2.next.prev = E.n2 }
11 fact { some E.n3.next implies E.n3.next.prev = E.n3 }
12 fact { some E.n4.next implies E.n4.next.prev = E.n4 }
13 fact { some E.n1.prev implies E.n1.prev.next = E.n1 }
14 fact { some E.n2.prev implies E.n2.prev.next = E.n2 }
15 fact { some E.n3.prev implies E.n3.prev.next = E.n3 }
16 fact { some E.n4.prev implies E.n4.prev.next = E.n4 }
17 fact { List.last in List.first.(*next) }
18
19 one sig C {     /* codificacion del contraejemplo del modelo M0 */
20   Node_0 : Node,
21   Node_1 : Node
22 }
23
24 fact { C.Node_1 != C.Node_0 }
25
26 fact { no C.Node_0.prev }
27 fact { C.Node_0.next = C.Node_0 }
28 fact { no C.Node_1.prev }
29 fact { C.Node_1.next = C.Node_0 }
30 fact { List.first = C.Node_1 }
31 fact { List.last = C.Node_0 }
32 fact { E.n1 = C.Node_1 }
33 fact { E.n2 = C.Node_0 }
34 fact { E.n3 = C.Node_0 }
35 fact { E.n4 = C.Node_1 }
36
37 assert P { List.first.(*next) = List.last.(*prev) } /* propiedad */
38 check P for 4
```

Código 8: Modelo V_0 para verificar si un contraejemplo es espúreo

Para el modelo V_0 no se encuentra contraejemplo. Utilizando el *unsat core*, la inexistencia de un contraejemplo se debe a la inconsistencia entre las líneas 8, 27 y 31. Precisamente, en el modelo, se indica que el último nodo de la lista no debe tener siguiente elemento, y en el contraejemplo de M_0 (figura 4), el último nodo de la lista se referencia a él mismo. De este *unsat core* se concluye que al incluir el axioma de la línea 8, el modelo M_1 del código 9 excluirá el contraejemplo de la figura 4.

```
1 one sig List { first, last : lone Node }
2 sig Node { prev, next : lone Node }
3 one sig E { n1, n2, n3, n4 : lone Node }
4
5 fact { E.n1+E.n2+E.n3+E.n4 = Node }
```

```
6
7  fact { no List.last.next }
8
9  assert P { List.first.(*next) = List.last.(*prev) }
10 check P for 4
```

Código 9: Modelo M_1 correspondiente a la segunda iteración.

Con el modelo M_1 se repite el proceso de analizar la validez de la propiedad. Si la propiedad resulta válida, entonces también lo será para el modelo M (código 6) por inclusión, así como para el modelo original por equivalencia con el anterior. Pero si la propiedad resulta inválida, se dispondrá de un contraejemplo. Se debe averiguar si este contraejemplo es espúreo o no con respecto al modelo M, para lo cual se arma un modelo V_1, similar al V_0 (código 8) que incluya todos los axiomas del modelo M_1.

En total se necesitan nueve iteraciones para minimizar el modelo instrumentado y concluir que la propiedad es válida dentro de las cotas del análisis. En la figura 5 se muestran los contraejemplos que se obtuvieron en las distintas iteraciones salvo en la última donde se concluye que el modelo instrumentado es válido por medio del modelo M_9 (código 10). El modelo M_9 es también el modelo minimizado y contiene cuatro axiomas menos que el instrumentado.

A lo largo de las iteraciones los axiomas que se fueron incluyendo van descartanto los contraejemplos espúreos de la figura 5. En las últimas iteraciones se puede apreciar que los contraejemplos (5f), (5g) y (5h) son solamente permutaciones. Ésto se debe a que cuando se analiza si un contraejemplo es espúreo o no, se usa fuertemente el nombre de los elementos (ver líneas 32 a 35 de código 8).

```
1  one sig List { first, last : lone Node }
2  sig Node { prev, next : lone Node }
3  one sig E { n1, n2, n3, n4 : lone Node }
4
5  fact { E.n1+E.n2+E.n3+E.n4 = Node }
6
7  fact { (no List.first) iff (no List.last) }
8  fact { no List.first.prev }
9  fact { no List.last.next }
10 fact { some E.n1.next implies E.n1.next.prev = E.n1 }
11 fact { some E.n2.next implies E.n2.next.prev = E.n2 }
12 fact { some E.n3.next implies E.n3.next.prev = E.n3 }
13 fact { some E.n4.next implies E.n4.next.prev = E.n4 }
14 fact { List.last in List.first.(*next) }
15
16 assert P { List.first.(*next) = List.last.(*prev) }
17 check P for 4
```

Código 10: modelo correspondiente a la última iteración

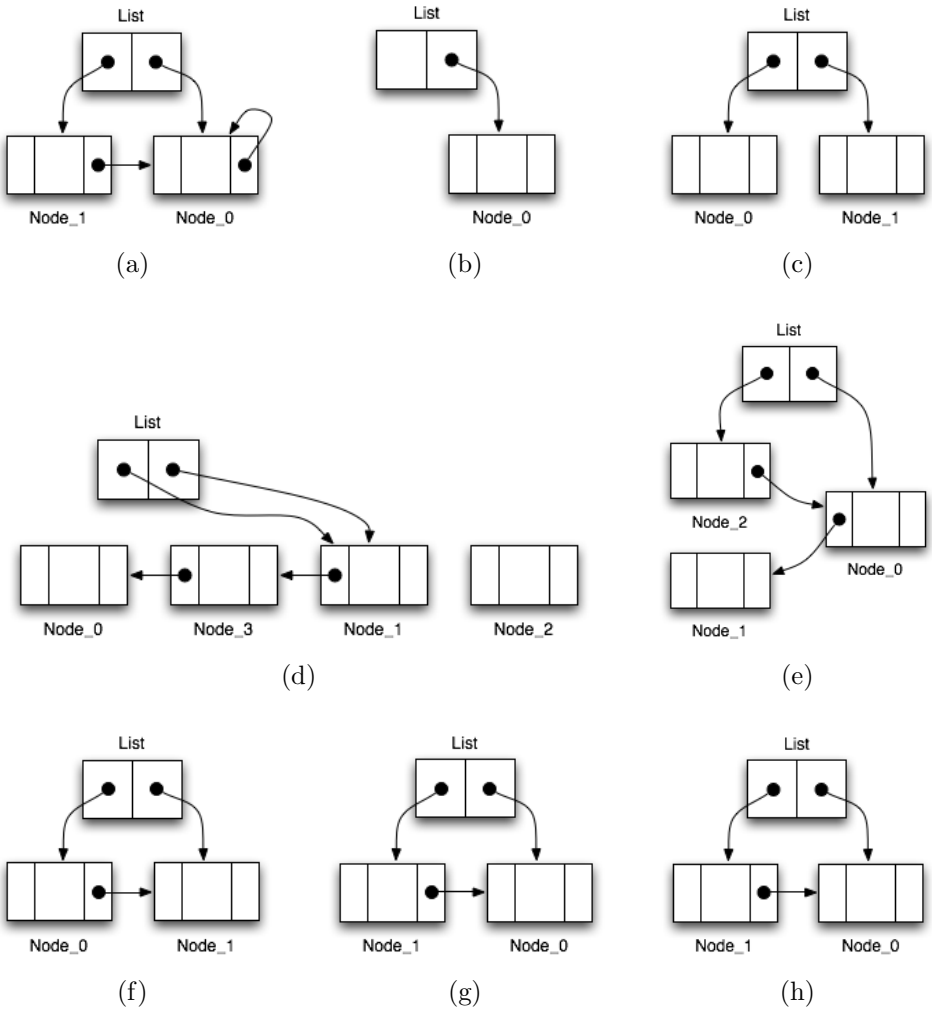


Figura 5: Contraejemplos correspondientes a cada iteración

3.3. Terminación

Para asegurarse que el proceso de minimización de modelos es un algoritmo efectivo, se debe garantizar su terminación. Primero, se debe notar que la cantidad de axiomas en el modelo instrumentado es finita. Segundo, en cada iteración del proceso, se comienza analizando un modelo más débil que el instrumentado. Si el modelo es insatisfacible, el instrumentado también y el proceso termina. En caso de que el modelo sí sea satisfacible, se procede a verificar si el contraejemplo es espúreo o no con respecto al modelo instrumentado. Si no lo es, entonces se encontró un contraejemplo del modelo instrumentado y el proceso termina. En cambio, si el contraejemplo es espúreo, se conoce como testigo, al menos un axioma del modelo instrumentado que no se encontraba en el modelo correspondiente al inicio de la iteración. Este testigo es incluido en el modelo para la siguiente iteración. Dado que la cantidad de axiomas en el modelo instrumentado es finita, estas iteraciones pueden realizarse a lo sumo un número finito de veces, asegurando de esta manera su terminación.

3.4. Experimentación

3.4.1. Caso de estudio

El caso de estudio considerado es el análisis de la especificación presentada en [Zav06]. En el trabajo anterior se detallan los objetivos y la motivación por los cuales se decide estudiar el impacto de la resolución de identificadores en una red para distintos servicios. Para los detalles de la especificación se sugiere consultar [Zav06]. A lo largo del trabajo la especificación original será denominada M^{cb}.

El interés por la especificación no es por el sistema que representa, sino por los resultados que obtuvieron los autores al analizarla. Las signaturas que aparecen son Domain, Path, Agent y Identifier . La propiedad se trata de verificar para distintos tamaños de Identifier , dejando al resto constante como se muestra en el código 11. Resolver los casos con $\#Identifier$ 9 y 11 resulta impracticable, no por que la fórmula CNF resultante sea grande, sino porque al parecer, es demasiado compleja para el estado del arte de los *sat solvers*. En el trabajo [Zav06] donde se introduce la especificación en estudio se intuye ésto último basado en los tiempos de análisis y en la longitud de la fórmula resultante conseguidos por el MIT.

```
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 3 Identifier
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 5 Identifier
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 7 Identifier
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 8 Identifier
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 9 Identifier
```

```
check StructureSufficientForPairReturnability for 2 but
    2 Domain, 2 Path, 3 Agent, 11 Identifier
```

Código 11: Comandos correspondientes al caso de estudio.

Inicialmente se evalúa el desempeño de la minimización de modelos con una versión simplificada de M^{cb} en donde se verifica parte de la propiedad original. Luego, el modelo que es usado como entrada de la técnica de minimización será denotado como M_s^{cb}. El modelo M_{si}^{cb} es el resultado obtenido luego de aplicar los pasos de instrumentación descriptos en la siguiente sección.

En el cuadro 2 se muestran los tiempos obtenidos al analizar la especificación M^{cb} en el mismo entorno en donde se realizarán todas las mediciones a lo largo del trabajo: Linux AMD Athlon 64 X2 Dual Core 2.3 GHz 2GB RAM. Para el análisis se utilizó MiniSat prover. En el mismo cuadro se puede apreciar el tamaño de la fórmula CNF generada y cómo crece dramáticamente a medida que # Identifiers aumenta.

# Identifiers	Resultado	Sat solver	Generación CNF	Variables primarias	Variables totales	Cláusulas
3	UNSAT	0.4 s	0.4 s	196	2979	5464
5	UNSAT	0.7 s	0.6 s	400	7079	13962
7	UNSAT	51.7 s	1.1 s	684	13787	28168
8	UNSAT	14.1 m	1.2 s	856	18031	37310
9	ERROR	14.9 h	1.2 s	1048	26825	56812
11	n/a			1492	43351	93267

Cuadro 2: Análisis del modelo M^{cb} usando MiniSat prover.

Medir la cantidad de cláusulas, de variables primarias y de variables totales ayuda a tener noción sobre qué tamaño tiene la fórmula CNF que es usada como entrada al *sat solver*. No constituyen un índice preciso sobre el esfuerzo necesario para analizarlas, pero son una buena aproximación si uno analiza un ejemplo en particular variando las cotas del análisis. Estudiar el crecimiento de algunas de estas métricas en función de las cotas del análisis ayuda a reconocer cuándo se llega al límite de lo que se puede analizar a efectos prácticos.

3.4.2. Instrumentaciones

A continuación se detallan las distintas transformaciones realizadas para obtener un modelo instrumentado. Las reglas se aplican de manera reiterada hasta que no se puedan seguir aplicando.

Cada una de estas manipulaciones no solo preservan la satisfacibilidad del modelo. Además permiten, en el caso de encontrar un contraejemplo para el modelo instrumentado, traducirlo a un contraejemplo del modelo original.

Separar axiomas por conjunciones

Cuando una axioma es de la forma **fact** { P **and** Q } se lo substituye por los axiomas **fact** { P } y **fact** { Q }.

Codificación de antecedentes en la propiedad como axiomas

Cuando la propiedad a verificar es de la forma **assert** { P **implies** Q } se la substituye por el axioma **fact** { P } y por la propiedad **assert** { Q }.

Se puede ver que las anteriores reglas son correctas por la semántica de los axiomas y la aserción. Recordar la fórmula 2.4 de la página 8.

Expandir predicados

Tanto en los axiomas como en la propiedad a verificar expandir la declaración de los predicados puede permitir aplicaciones adicionales de las reglas de instrumentación.

Agregar variables de instrumentación

Se agrega un **one sig** E {...} que dispondrá de campos para referenciar a todos los elementos del modelo. Para eso se deben interpretar correctamente las cotas del análisis y la relación entre las signaturas. Por ejemplo para $\#Identifier = 3$ las variables de instrumentación son las mostradas en el código 12.

```
one sig E {
    d1, d2: lone Domain,
    p1, p2: lone Path,
    g1, g2, g3: lone Agent,
    i1, i2, i3: lone Identifier
}

fact { (E.d1 + E.d2) = Domain }
fact { (E.p1 + E.p2) = Path }
fact { (E.g1 + E.g2 + E.g3) = Agent }
fact { (E.i1 + E.i2 + E.i3) = Identifier }
```

Código 12: Variables de instrumentación para $\#Identifier = 3$

Para que esta regla sea correcta, se debe interpretar cómo se aplican las cotas a las signaturas definidas en el modelo. Cada signatura estará limitada a tener a lo sumo tantos elementos como se listan en el axioma correspondiente.

Codificación de invariantes de signaturas como axiomas

En Alloy las signaturas puede declararse con invariantes. El esquema de dicha declaración se muestra en el código 13. La fórmula Inv se interpreta en el contexto de un valor de S.

```
sig S { c : TC } { Inv }
```

Código 13: Esquema de declaración de invariantes de signaturas

Luego, para codificar el invariante Inv como axioma se puede seguir el esquema indicado en el código 14. En donde Inv' resulta de reemplazar c por s.c en Inv.

```
sig S { c : TC }
fact { all s : S | Inv' }
```

Código 14: Esquema de expresar invariantes como axiomas

Eliminación de ∀ en los axiomas

Muchos axiomas suelen ser expresados como **fact** { **all** t : T | P[t] }. Al disponer de variables de instrumentación E.t1 ... E.tn para la signatura T se puede expresar el axioma original mediante n axiomas de granularidad más fina **fact** { P[E.ti] } para cada $1 \leq i \leq n$.

Dado que las variables de instrumentación enumeran todos los posibles elementos de un signatura T, una forma de expresar el axioma con el cuantificador universal, es substituir la variable predicada por cada uno de los elementos de la signatura. Notar que es necesario saber la cota aplicada a la signatura.

También se presenta la situación en donde, para signaturas T y S con $T \subset S$, al estar fijadas las cotas para la signatura S, se disponen de variables de instrumentación E.s1 ... E.sn para S. En esta caso un axioma **fact** { **all** t : T | P[t] } se puede expresar como **fact** { E.si in T => P[E.si] } para cada $1 \leq i \leq n$.

En este caso la motivación es la misma que la anterior. Pero al no disponer un conjunto exacto de variables de instrumentación de la signatura T, se aprovechan las de S. Para mantener la semántica del axioma original se agrega la restricción de pertenencia a la signatura T.

Eliminación de ∀ en la propiedad

Cuando la propiedad a verificar es de la forma **assert** { **all** t : T | P[t] }, basta que contemos con un elemento de T, digamos Q.t1, para un **one sig** Q { t1 : T }, para expresar la propiedad de manera equivalente como **assert** { P[Q.t1] }.

Como no se fijaron restricciones para Q.t1 más que pertenezca a los valores de cierta signatura, si no se encuentra contraejemplo para la propiedad instrumentada, entonces se mantiene para todos los valores de la signatura. En caso de sí encontrarse un contraejemplo para la versión instrumentada, entonces para al menos un valor de la signatura la propiedad no se mantiene y por consiguiente tampoco se mantiene la propiedad original.

Como se enunció al principio éstas reglas mantienen la satisfacibilidad entre modelos. En cada regla se muestran argumentos de por qué son correctas, aunque una demostración formal no se está realizando.

Para ver que un contraejemplo de un modelo instrumentado corresponde a uno del original basta quedarse con los valores de las signaturas originales. El valor de verdad para cada uno de los axiomas del modelo original puede deducirse usando el valor de verdad de los correspondientes en el modelo instrumentado.

3.4.3. Resultados

Como se mencionó, se evaluó el desempeño de la técnica propuesta analizando el modelo M_s^{cb} con las cotas indicadas en el código 11 de la página 19.

Algunos resultados se indican en el cuadro 3. En el mismo se lista en cada fila el comportamiento del proceso para un determinado tamaño de dominio (se recuerda que la única variación es la cantidad de elementos de la signatura Identifier). Respectivamente en la distintas columnas se detallan a) la cantidad de iteraciones, b) el tiempo consumido por el sat solver a lo largo de todas las iteraciones, c) el tiempo insumido en la generación de las fórmulas CNF, d) el tiempo total, e) la proporción de tiempo de generación de las fórmulas CNF con respecto al tiempo total y, por último f) la proporción del *sat solving* de la última iteración con respecto del tiempo total de *sat solving*.

# Identifiers	iteraciones	sat solvers [seg]	generación CNF [seg]	total	% CNF/total	% último sat solving
3	14	4	4	8 s	45.51 %	0.92 %
5	18	7	6	13 s	44.54 %	2.19 %
7	30	18	11	30 s	37.52 %	24.30 %
8	29	173	10	3 m	5.67 %	92.20 %
9	>28	error		> 3.5h		
11		no ejecutado				

Cuadro 3: Mediciones de la minimización de modelos con M_s^{cb}.

El incremento del tiempo total de ejecución delata lo impracticable del método tal como fue descripto. Recordar que es una versión simplificada de M^{cb} a fin de ver como se comporta la minimización de modelos en un escenario más controlado.

A pesar de no lograr un resultado favorable se puede extraer como conclusión que es demasiado el tiempo insumido en la generación reiterada de las fórmulas CNF. Desde ya no es el cuello de botella dado que porcentaje del tiempo que lleva la generación de la fórmula CNF con respecto al tiempo total de análisis va disminuyendo a medida que se aumenta el tamaño del dominio a analizar.

La conclusión más importante sería el aumento en la última columna. Es aquí donde se aprecia la falta de escalabilidad y el motivo por el cuál la instancia #Identifiers=9 finalizó debido a un error. Si bien se van logrando aumentos pequeños en el tamaño del modelo iteración tras iteración, este incremento hace que el modelo se torne demasiado difícil de pronto.

Los pequeños incrementos hechos al modelo en cada iteración son uno o algunos pocos axiomas del modelo instrumentado. Por lo mencionado antes es necesario hacer incrementos mucho más finos o pequeños, al mismo tiempo que sigan siendo automatizables. En la sección 4 se trata como lograr esto.

Antes de continuar con la siguiente alternativa se presentan algunas mediciones más que, además de contribuir al entendimiento de la técnica actual, introducen métricas usuales

de dificultad sobre modelos Alloy. Como el *sat solver* opera sobre la fórmula CNF, se suelen ver algunas propiedades generales de éstas últimas. Éstas son a) la cantidad de variables primarias, b) la cantidad de variables totales y c) la cantidad de claúsulas. En la sección 2.3 se explica por qué algunas variables son primarias y otras no. Es importante destacar que ninguna de estas métricas es absoluta al momento de categorizar el esfuerzo en resolver el problema SAT para una fórmula en particular. No son las únicas métricas disponibles, pero son las más usadas en la comunidad.

Lo que se busca con las métricas es comparar, para los distintos tamaños de dominios, cuánto se redujo la fórmula CNF que se utiliza para analizar el modelo. Las fórmulas que se miden son las correspondientes a los siguientes modelos: a) el modelo de entrada M_s^{cb}, b) el modelo instrumentado M_{si}^{cb} y el modelo de correspondiente a la última iteración. Notar que todos estos modelos son equivalentes.

# Identifiers	Variables primarias			Variables totales			Cláusulas		
	M_s^{cb}	M_{si}^{cb}	ult. it.	M_s^{cb}	M_{si}^{cb}	ult. it.	M_s^{cb}	M_{si}^{cb}	ult. it.
3	200	209	209	2754	3005	2128	5064	5661	3865
5	404	425	425	6692	7587	6484	13237	15370	12897
7	688	729	729	13218	15101	13636	27024	31917	28865
8	860	914	914	17343	20407	17757	35989	43711	37676
9	1052	1121	1121	25970	30185	>25810	55160	65909	>55477
11	1496	1601	n/a	42243	49531	n/a	91042	110023	n/a

Cuadro 4: Mediciones sobre fórmulas CNF principales

El incremento de variables entre M_s^{cb} y M_{si}^{cb} se debe a la signatura E y axiomas de instrumentación (ver sección 3.1). En general se ve que el modelo de la última iteración es una apreciable simplificación de M_{si}^{cb} pero desafortunadamente no compensa el incremento debido a la instrumentación.

Los únicos casos en los que se lograron cierta mejora con respecto a M_s^{cb} son los de # Indentifiers 3 y 5. Lo más significativo hubiese sido lograr un descenso en la cantidad de variables primarias, desafortunadamente no se obtuvo.

Por último se hace notar que el modelo obtenido en la última iteración sirve de *unsat core* aunque quizás no sea minimal.

3.5. Conclusiones

En los experimentos realizados se nota que la técnica de minimización de modelos no escala y sufre de algunas desventajas para ser práctica. a) Se introducen variables al modelo original para realizar la instrumentación, b) los axiomas no resultan tan granulares como se necesita para que la inclusión de uno solo no concluya en un modelo intratable, c) mucho costo de entrada-salida por la continua traducción de modelos Alloy a fórmulas CNF y su posterior análisis.

4. Resolución incremental

A pesar de los resultados obtenidos con la técnica anterior, la motivación de usar paulatinamente la información del modelo continúa siendo una vía interesante para aumentar la escalabilidad de la verificación de modelos.

En la siguiente sección se tratan un conjunto de estrategias basadas en la utilización paulatina de la información del modelo. Pero, en lugar de ir incluyendo axiomas Alloy del modelo instrumentado, se manipula directamente la fórmula CNF que resulta de la traducción del modelo original.

Las técnicas que se presentan en esta sección son totalmente automáticas y se espera no sufran de falencias similares a las de la minimización de modelos.

4.1. Motivación

La motivación es esencialmente la misma que antes. Recordar que para saber si en un modelo se satisface cierta propiedad, se genera una fórmula CNF de lógica proposicional f. Si f resulta insatisfacible, entonces la propiedad se mantiene para dominios correspondientes a las cotas impuestas. Si se encuentra una valuación que satisfaga f, de ésta se puede deducir una instancia del modelo en el cual la propiedad no se cumple, o sea un contraejemplo.

Usualmente se utilizaría algún *sat solver* para encontrar una valuación de f directamente. Pero en este caso son los sat solvers los que agotan los recursos disponibles para realizar esta tarea.

Es gracias a que f es una conjunción de cláusulas que se pueden usar las siguientes propiedades como pilares de las estrategias a proponer.

Teorema 4.1 *Sean f y g fórmulas de lógica proposicional en CNF tales que $g \subset f$ entonces si g es insatisfacible, f también lo es. Donde $\cdot \subset \cdot$ representa la inclusión de conjuntos.*

Teorema 4.2 *Sean f y g fórmulas de lógica proposicional en CNF tales que $g \subset f$. Sea v una valuación tal que $v \models g$. Si $v \models f \setminus g$ entonces $v \models f$. Donde $\cdot \setminus \cdot$ representa la diferencia de conjuntos y $\cdot \models \cdot$ la satisfacibilidad de lógica proposicional.*

Ambos teoremas se deducen directamente de la estructura de la fórmula. Al estar en CNF, si $f = \{c_i\}_{i \in I}$, como $g \subset f$, se sabe que $g = \{c_j\}_{j \in J}$ con $J \subset I$.

Ejemplo

A continuación se muestra como podría verificarse que la fórmula 4.1 es insatisfacible usando de manera incremental la misma. Es fácil demostrar la validez de la fórmula y que hay una cláusula que es innecesaria para eso: $\{\ \overline{r},\ s\ \}$. A lo largo del ejemplo se ve este hecho.

$$f = \quad p \wedge (p \to q) \wedge (q \to r) \wedge (r \to s) \wedge \neg r \tag{4.1}$$

$$f = \quad \{\ \{\ p\ \},\ \{\ \overline{p},\ q\ \},\ \{\ \overline{q},\ r\ \},\ \{\ \overline{r},\ s\ \},\ \{\ \overline{r}\ \}\ \} \tag{4.2}$$

Se inicia con una fórmula g_0 en CNF tal que $g_0 \subset f$, por ejemplo:

$$g_0 = \emptyset = \top$$

La fórmula g_0 es una aproximación a f, con el paso de las iteraciones las fórmulas g_i irán incluyendo cláusulas de f de forma tal que $g_{i-1} \subset g_i \subset f$. Se sabe que g_0 es satisfacible trivialmente, por ejemplo con la valuación:

$$v_0 = \{\ p/0,\ q/1,\ r/1,\ s/1\ \}$$

Como $v_0 \nvDash f$, no se puede asegurar nada sobre la satisfacibilidad de f. Se pasa a generar una fórmula g_1 tal que $v_0 \nvDash g_1$ y $g_0 \subset g_1 \subset f$. Para esto se pueden incluir la primer y última cláusula de f, ya que son las cláusulas no satisfechas por la valuación v_0:

$$g_1 = \{\ \{\ p\ \},\ \{\ \overline{r}\ \}\ \}$$

Notar que por construcción $v_0 \nvDash g_1$ ya que se incluyeron cláusulas que no eran satisfechas por v_0. Se procede a analizar la satisfacibilidad de g_1 y en efecto g_1 es satisfacible, por ejemplo con la valuación:

$$v_1 = \{\ p/1,\ q/0,\ r/0,\ s/1\ \}$$

Nuevamente no se puede asegurar nada sobre la satisfacibilidad de f ya que $v_1 \nvDash f$. Se procede a generar la siguiente aproximación g_2 de f tal que $v_1 \nvDash g_2$ y $g_0 \subset g_1 \subset g_2 \subset f$. Notar que $v_0 \nvDash g_2$ pues $g_1 \subset g_2$. En esta ocasión se consigue al incluir la segunda cláusula de f, ya que es la no satisfecha por v_1:

$$g_2 = \{\ \{\ p\ \},\ \{\ \overline{r}\ \},\ \{\ \overline{p},\ q\ \}\ \}$$

En este paso la valuación que satisface g_2 es

$$v_2 = \{\ p/1,\ q/1,\ r/0,\ s/0\ \}$$

Se debe hacer una iteración más, para generar g_3 ya que $v_2 \nvDash f$. Recordar que g_3 debe cumplir que $g_2 \subset g_3 \subset f$ y $v_2 \nvDash g_3$. Esto se logra si incluimos la tercer cláusula de f.

$$g_3 = \{\ \{\ p\ \},\ \{\ \overline{r}\ \},\ \{\ \overline{p},\ q\ \},\ \{\ \overline{q},\ r\ \}\ \}$$

Al analizar g_3 se obtiene que es insatisfacible y por consiguiente f lo es.

Notar que g_3 no solo tiene menos cláusulas que f sino que, además menos variables. El ejemplo es solo motivador y armado con las valuaciones adecuadas para poder ignorar la cláusula $\{\ \overline{r},\ s\ \}$ durante todo el proceso.

En caso de que para algún i, la valuación v_i cumpliese que $v_i \models f \setminus g$, entonces se hubiese concluido que f es satisfacible. En este supuesto caso, se hubiese conseguido una valuación que satisface f sin haber analizado la fórmula completa. De esta valuación de variables proposicional se puede armar un contraejemplo de la especificación usando los mecanismos propios de Alloy.

4.2. Algoritmo

El proceso iterativo de la sección anterior se describe en el algoritmo del cuadro 5.

```
Require: fórmula CNF f
  g ← fórmula inicial a partir de f
  loop
    res ← Analizar g
    if res es UNSAT then
      return f es UNSAT
    else
      v ← Valuación de res
      c ← Cláusulas de f no satisfechas por v
      if c = ∅ then
        return f es SAT
      else
        Incorporar c a g
      end if
    end if
  end loop
```

Cuadro 5: Algoritmo para resolución incremental de una fórmula CNF

En éste algoritmo queda sin especificar cómo realizar algunos pasos:

FórmulaInicial $g \leftarrow$ fórmula inicial a partir de f

Analizar $res \leftarrow$ Analizar g

NuevasCláusulas $c \leftarrow$ Cláusulas de f no satisfechas por v. Donde v es una valuación que satisface g en caso de ser SAT.

En la sección 4.3 se estudian distintas estrategias, donde cada una fija cómo se realizan estas etapas del algoritmo. Las mismas fijan la eficiencia de todo el proceso y fue la evolución del presente trabajo una búsqueda constante por lograr un mejor aprovechamiento de los recursos y un aumento en la escalabilidad.

En el paso NuevasCláusulas podría indicarse $f \setminus g$ en lugar de sólo f, pero este detalle se decide ocultar para una mejor lectura.

Asumiendo alguna implementación correcta (y que termine) de estos tres pasos se puede estudiar la correctitud y terminación para el algoritmo principal.

4.2.1. Correctitud

Al inicio del ciclo se sabe que $g \subset f$, que es la postcondición más débil para FórmulaInicial. El que $g \subset f$ se mantiene invariante a lo largo de todas las iteraciones y permite aplicar en distintos puntos los teoremas 4.1 y 4.2 de la página 25.

Dada una iteración cualquiera, si res indica que g es insatisfacible, por el teorema 4.1, como $g \subset f$, se concluye que f es insatisfacible.

Por el contrario, si res indica que g es satisfacible, se prueba la valuación testigo v con la fórmula f. Si todas las cláusulas de f son satisfechas, entonces f es satisfacible y v es, al menos, una valuación que lo hace. Usando el teorema 4.2 se puede analizar si v satisface a f usando solo las cláusulas de $f \setminus g$ ya que $v \models g$.

En caso de haber cláusulas de f no satisfechas por v, se incluyen algunas de estas. Dado que $v \models g$, las cláusulas no satisfechas por v, pertenecen a $f \setminus g$. Incluyendo algunas de estas, se garantiza que $g \subset f$ a lo largo de todas las iteraciones.

4.2.2. Terminación

A lo largo del ciclo tres cosas pueden ocurrir: a) concluir que f es insatisfacible, b) concluir que f es satisfacible, c) agregar a g alguna cláusula de f que no estaba previamente en g.

En los dos primeros, el proceso termina. La única opción para que cicle indefinidamente es que siempre caiga en el tercer caso. Pero la fórmula f contiene finitas cláusulas. Como en cada iteración se incorpora al menos una cláusula que antes no estaba en g, el tercer caso se puede repetir una cantidad finita de veces. Por lo tanto el proceso siempre termina.

4.3. Caso de estudio

En esta sección se detallan los distintos experimentos realizados para buscar una estrategia que logre lidiar con el modelo M^{cb}. El análisis se hace con las mismas cotas que se muestran en el código 11 de la página 19. El primer paso como se menciona en la sección anterior es obtener la fórmula CNF correspondiente a la traducción del modelo original. Dicha traducción es realizada por los mismos mecanismos que usa Alloy. Se remarca que aquí se trabaja con la especificación Alloy M^{cb} tal cual fue presentada en [Zav06], sin simplificar ni instrumentar.

En [Zav06] se indica que para $\#Identifiers = 9$ y $\#Identifiers = 11$ no se pudo determinar la validez de la propiedad. Como se indicó con anterioridad, uno de los objetivos del presente trabajo no es lograr mejorar con respecto al tiempo requerido para analizar un modelo, pero sí incrementar la escalabilidad. Los tiempos de las distintas estrategias hay que compararlos con los mostrados en el cuadro 2 de la página 20.

Estrategia mm0a

La primer estrategia a evaluar se denomina mm0a. Para FórmulaInicial simplemente se elige, como en el ejemplo, el conjunto vacío. Para la etapa Analizar se usa algún sat solver por ejemplo MiniSat. Como la fórmula g debería ser sencilla de resolver en la mayoría de los casos, se espera que esta etapa no insuma muchos recursos.

Para el paso correspondiente a NuevasCláusulas se opta por utilizar un mecanismo análogo en la minimización de modelos, donde se utilizaba el modelo instrumentado con todos

los axiomas pero con el agregado de unas signaturas y axiomas adicionales que fijaban el valor y las relaciones entre los distintos elementos de las signaturas (Ver código 8 de la página 15).

En este caso se usará la fórmula original f con unas cláusulas extra que fijarán el valor de las variables proposicionales según la valuación v de la iteración actual. Estas cláusulas adicionales se denominan $tc(v)$. El objetivo es que al fijar la valuación el sat solver propague rápidamente estas restricciones a las demás cláusulas y el análisis restante se realice con mayor eficiencia.

En caso de que $f \wedge tc(v)$ sea satisfacible, c será el conjunto vacío. Pero si es insatisfacible, se utiliza el *unsat core* (ver sección 2.3) para obtener qué cláusulas de f no son satisfechas por v. Notar que como $tc(v)$ es satisfacible, el *unsat core* de $f \wedge tc(v)$ debe incluir alguna cláusula de f. Son estas últimas las cláusulas de f no satisfechas por v que se asignan a c.

Formalmente se define $tc(v)$ para una valuación v como:

$$tc(v) = \bigwedge_{p \in Vars} tt_v(p)$$

$$tt_v(p) = \begin{cases} p & \text{si } v \models p \\ \neg p & \text{si } v \models \neg p \end{cases}$$

donde $Vars$ es el conjunto de variables proposicionales.

En resumen, la estrategia mm0a está constituida por la siguiente elección:

FórmulaInicial $g \leftarrow \emptyset$

Analizar Aplicar MiniSat solver sobre g

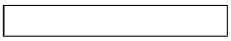
NuevasCláusulas Aplicar MiniSat prover con $f \wedge tc(v)$

Es solo en la etapa de NuevasCláusulas que se necesita el *unsat core* y por eso es aplica un MiniSat prover. En la etapa de Analizar basta con un MiniSat solver pues se debe saber si es UNSAT o SAT y una valuación en este último caso.

El tiempo que insume esta estrategia para llevar a cabo el análisis es prohibitivo para verificar la propiedad con el *scope* deseado. En el cuadro 6 se comparan los tiempo de ejecución total para instancias disponibles. En la última columna se muestra una pequeña visualización para ilustrar las proporciones de tiempo. En este caso el tiempo de MiniSat es despreciable frente al de mm0a.

Estrategia mm0

La estrategia anterior insume más del tiempo esperado en la etapa de NuevasCláusulas. Se decidió comparar con una segunda estrategia para corroborar si efectivamente se está usando la información de v para analizar f.

# Identifiers	MiniSat ■	mm0a □	MiniSat vs. mm0a
3	0.42 s	1.3 m	
5	0.73 s	11.5 m	
7	51.73 s	74.7 h	
8	14 m	n/a	
9	n/a	n/a	
11	n/a	n/a	

Cuadro 6: Tiempos MiniSat vs. Estrategia mm0a

En la estrategia mm0a las cláusulas de $tc(v)$ se ubican a continuación de las de f. En la presente estrategia se altera solamente, con respecto a la anterior que, las cláusulas de $tc(v)$ se ubican antes de las de f.

En resumen, la estrategia mm0 está constituida por la siguiente elección:

FórmulaInicial $g \leftarrow \emptyset$

Analizar Aplicar MiniSat solver sobre g

NuevasCláusulas Aplicar MiniSat prover con $tc(v) \wedge f$

En el cuadro 7 se muestra la mejora lograda al variar el orden de las cláusulas. Los tiempos continúan siendo prohibitivos para un análisis completo. La diferencia de tiempos que se obtuvo en todos los experimentos da indicios de una falta de aprovechamiento de la información que hay en las cláusulas dependiendo del orden de las mismas. No se puede suponer que el orden de las cláusulas en la nueva estrategia mm0 sea el óptimo para lograr un menor tiempo de análisis.

# Identifiers	mm0a ■	mm0 □	mm0a vs. mm0
3	1.3 m	50 s	
5	11.5 m	7.2 m	
7	74.7 h	26 h	
8	n/a	n/a	
9	n/a	n/a	
11	n/a	n/a	

Cuadro 7: Tiempos Estrategia mm0a vs. Estrategia mm0

Estrategia mm

En la etapa de FórmulaInicial se está usando hasta el momento una alternativa trivial, que no depende de la fórmula de entrada f. Cualquier subconjunto de cláusulas de f es uno

válido para iniciar el procedimiento. En algunos trabajos de análisis de modelos guiados por contraejemplos, así como en la minimización de modelos presentada en la sección 3, se suele trabajar de forma incremental con el modelo empezando solamente con la propiedad a verificar e ignorando inicialmente el resto de los axiomas del modelo. Algo similar podría hacerse en esta alternativa si se obtienen cuales de las cláusulas provienen de la traducción de la propiedad a verificar.

Otra alternativa, trivial pero que usa información de la fórmula inicial f, es usar como fórmula inicial aquella que resulta de escoger las cláusulas con un solo literal. Dado que la estrategia mm0 opera mejor que la mm0a, es a partir de ésta que se constituye la siguiente estrategia denominada mm.

Ya está demostrado que es correcto escoger cualquier subconjunto de f como fórmula inicial. La motivación es sencilla. Siendo p una variable proposicional, si $\{ p \} \in f$ entonces, cualquier valuación v tal que $v \models f$, debe cumplir con $v \models p$. Análogamente con $\neg p$. Por lo tanto, el valor asignado por la valuación v a la variable proposicional p ya está fijo.

FórmulaInicial $g \leftarrow \{ c \in f \mid \#c = 1 \}$

Analizar Aplicar MiniSat solver sobre g

NuevasCláusulas Aplicar MiniSat prover con $tc(v) \wedge f$

En el cuadro 8 se muestra la mejora lograda por la reciente modificación. En el caso de #Identifiers=7, hay un cambio considerable aunque no hay un cambio con respecto a las instancias tratables.

# Identifiers	mm0 ■	mm □	mm0 vs. mm
3	50 s	48 s	
5	7.2 m	7.2 m	
7	26 h	19.1 h	
8	n/a	n/a	
9	n/a	n/a	
11	n/a	n/a	

Cuadro 8: Tiempos Estrategia mm0 vs. Estrategia mm

Las estrategias vistas hasta el momento necesitan realizar muchas iteraciones. Como se puede ver en el cuadro 9, dado que en cada iteración, que no sea la última, se incorpora al menos una cláusula, se sabe que se incluye solamente de a una. Basta observar la relación entre las columnas que cuentan las iteraciones y la cantidad de cláusulas. En el caso de la estrategia mm se debe considerar que la fórmula g inicialmente contiene alguna cláusula y por eso la sutil diferencia entre el número de cláusulas y el de iteraciones.

También en el cuadro 9 se puede apreciar cuántas cláusulas ya se están descartando para poder concluir que la fórmula f es insatisfacible. Si bien no se logra mejor capacidad de

resolución o tiempo, sí se disponen ejemplo concretos en donde se analiza la satisfacibilidad de una fórmula usando parte de sus cláusulas. Sobre este logro se continúan buscando mejoras.

# Identifiers	f	mm0a		mm0		mm	
	Cláusulas	It.	C.	It.	C.	It.	C.
3	5464	3791	3791	4156	4156	3912	3914
5	13962	11497	11497	11889	11889	11729	11731
7	28168	23975	23975	24872	24872	23972	23974

Cuadro 9: Comparación de cantidad de cláusulas finales en g y cantidad de iteraciones

4.3.1. Conclusiones primera etapa

A continuación se analiza los resultados obtenidos y se plantea la dirección en la que continua el trabajo.

Se sabe que hay oportunidad de simplificar la fórmula, al menos en términos de cantidad de cláusulas. Desafortunadamente los tiempos son prohibitivos para realizar un análisis efectivo.

La fórmula de entrada f se asume muy compleja para que MiniSat puede tratarla. En la etapa de NuevasCláusulas se está utilizando MiniSat con f y cláusulas que fijan la valuación que interesa. Luego del primer experimento se concluye que no se debe estar aprovechando la información provista por estas cláusulas. Dado esto, la etapa de NuevasCláusulas no debe estar lidiando con una fórmula mucho más sencilla que f. Al usar MiniSat se está analizando la relación entre las mismas cláusulas de f, lo cual puede ser útil pero es necesario darle más importancia a las cláusulas que fijan la valuación.

En cada iteración se modifica levemente la fórmula g. Se debe buscar una estrategia más agresiva, pues la cantidad de cláusulas ya es suficientemente grande como para no desear un incremento tan sutil.

Las estrategias presentadas están motivadas fuertemente por la minimización de modelos, en donde se usan axiomas para fijar el modelo. Los próximos pasos serán mejorar la etapa de NuevasCláusulas buscando un mecanismo diseñado directamente para fórmulas CNF, aprovechando que estas son bastante simples de manipular y evitando los problemas que aparecieron al intentar usar MiniSat prover con todas las cláusulas de f.

A partir de la valuación v en dicha etapa ya se puede determinar directamente qué cláusulas no están satisfechas. Las siguientes estrategias reemplazan a MiniSat en la última etapa por un procedimiento que solamente detecta las cláusulas no satisfechas por la valuación v.

Estrategia mj

Utilizando un procedimiento codificado en Java, sin generar llamadas de entrada/salida o comunicación entre procesos que puedan demorar la etapa de NuevasCláusulas se evalúa cada cláusula por separado de f utilizando la valuación v, que ya fija un valor para cada variable. El procedimiento es estrictamente $O(|f|)$ dado que se recorre linealmente las cláusulas de f y sus literales accediendo directamente al valor de la variable del literal en $O(1)$. Luego la estrategia mj se diferencia de mm por cómo se lleva a cabo la etapa NuevasCláusulas.

FórmulaInicial $g \leftarrow \{\ c \in f \mid \#c = 1\ \}$

Analizar Aplicar MiniSat solver sobre g

NuevasCláusulas Procedimiento Java para extraer cláusulas de f no satisfechas por v

En el cuadro 10 se aprecia una dramática mejora de tiempo. El resultado satisfactorio en este momento es que se concluye que para # Identifiers = 9 la fórmula es insatisfacible y la propiedad válida. Si bien para las cotas menores se requiere más tiempo que un análisis directo sobre el modelo Alloy usando MiniSat, se ha aumentado la capacidad de cómputo.

Al operar con fórmulas más pequeñas MiniSat puede estar procesándola más tiempo sin caer en un error por falta de memoria. Es por esto que al analizar el modelo Alloy directamente con MiniSat se produce un error a las 15 horas mientras que con la presente estrategia se están 54.3 horas, de las cuales 50 horas corresponden a la ejecución de MiniSat de la última iteración.

# Identifiers	mm ■	mj □	mm vs. mj
3	48 s	1 s	■
5	7.2 m	17 s	■
7	19.1 h	8.0 m	■
8	n/a	53.8 m	□
9	n/a	54.3 h	□
11	n/a	n/a	

Cuadro 10: Tiempos Estrategia mm vs. Estrategia mj

También se disminuyen considerablemente la cantidad de iteraciones necesarias tal como se muestra en el cuadro 11. En lugar de incluir una cláusula por iteración se incluyen cerca de diez cláusulas en promedio. Otra mejora lograda fue que la fórmula de la última iteración tiene varias cláusulas menos.

El cambio introducido es satisfactorio en todo aspecto: consumo de memoria, tiempo, escalabilidad. Al no aplicar MiniSat prover, se reduce la memoria utilizada. Ya que para obtener un *unsat core* se debe llevar un registro del árbol de ejecución usado para concluir

la insatisfacibilidad de la fórmula. Incluso el consumo de memoria al aplicar un MiniSat solver suele ser mucho menor que el aplicar un MiniSat prover a la misma fórmula.

La idea es sencilla de implementar pero cabe poner en duda si no es demasiado agresiva. Al principio del procedimiento muy pocas variables aparecen en g, luego, el valor asignado a cada variable de f que no aparece en g en el paso Analizar es escogido por MiniSat solver usar cláusula alguna. Por lo que las cláusulas no satisfechas por v están determinadas solamente por esta elección, por más que otro valor se pudiese asignar a la variable sin afectar la satisfacibilidad. A pesar de esto, se han obtenido notables mejoras. La siguiente estrategia propone una alternativa menos agresiva planteando una sencilla extensión a la estrategia mj. En la sección 7 se desarrolla otra idea motivada por el hecho de que en v, se fijan valores para variables que no figuran en la fórmula.

# Identifiers	f	mm		mj	
	Cláusulas	It.	C.	It.	C.
3	5464	3912	3914	181	3139
5	13962	11729	11731	1076	9087
7	28168	23972	23974	2280	18529
8	37310	n/a	n/a	3259	25479
9	56812	n/a	n/a	5034	38066

Cuadro 11: Comparación de cantidad de cláusulas finales en g y cantidad de iteraciones

Estrategia mjpα

En esta oportunidad, en lugar de incluir todas las cláusulas no satisfechas por v, se incluyen sólo algunas. En la etapa NuevasCláusulas se utiliza el mismo procedimiento que en mj, pero si una cláusula no está satisfecha, se la agrega a g con una probabilidad α sobre 100. De aquí la denominación mjpα con $0 < \alpha < 100$. A fin de garantizar la correctiud y progreso del procedimiento de resolución incremental es necesario asegurar que si hay cláusulas no satisfechas, al menos una sea escogida para ser agregada a g.

FórmulaInicial $g \leftarrow \{\ c \in f \mid \#c = 1\ \}$

Analizar Aplicar MiniSat solver sobre g

NuevasCláusulas Procedimiento Java para extraer cláusulas de f no satisfechas por v con probabilidad α sobre 100

En el cuadro 12 se muestran los tiempos obtenidos al evaluar la nueva familia de estrategias mjpα con respecto a la anterior. No hubo una notable mejora. En la mayoría de los casos, al ser menos agresiva se requirió más tiempo y más iteraciones. El otro aspecto que cabe ver es la cantidad de cláusulas de g en la última iteración, para tener una medida de cuánto se ha simplificado la fórmula. En el cuadro 13 se muestran los

resultados obtenidos hasta el momento. En las celdas figura la cantidad de cláusulas de g con respecto a la cantidad de cláusulas de la fórmula de entrada f. Incluso para la instancia con # Identifiers = 8, la diferencia de un 4 % está en el orden de las 1500 cláusulas. Dicha diferencia no es sustancial como para asegurar que conviene ser menos agresivo.

# Identifiers	mj	mjp5	mjp10	mjp25	mjp50	mjp75
3	1 s	7 s	5 s	2 s	1 s	1 s
5	17 s	1.4 m	1.3 m	56 s	33 s	23 s
7	8.0 m	37.6 m	45.9 m	24.1 m	11 m	6.9 m
8	53.8 m	5.9 h	7.1 h	3 h	1.6 h	1 h
9	54.3 h	n/a	n/a	53.4 h	n/a	n/a
11	n/a	n/a	n/a	n/a	n/a	n/a

Cuadro 12: Tiempos Estrategia mj vs. Estrategias mjpα

# Identifiers	mm	mm0	mj	mjpα	
3	72 %	76 %	57 %	55 %	mjp25
5	84 %	85 %	65 %	60 %	mjp5
7	85 %	88 %	66 %	65 %	mjp10
8	n/a	n/a	68 %	64 %	mjp5
9	n/a	n/a	67 %	64 %	mjp25
11	n/a	n/a	n/a	n/a	n/a

Cuadro 13: Cláusulas en la última iteración con respecto a la fórmula de entrada

Sería deseable ver el comportamiento de las estrategias, no solo para una ejecución, sino para una serie. Para así ver alguna tendencia. Lamentablemente los tiempos requeridos para esto son prohibitivos, el análisis en algunos casos demora casi dos días.

4.3.2. Conclusiones segunda etapa

Aún se está lejos de equiparar los tiempos originales de MiniSat, pero se ha logrado resolver una instancia de mayor tamaño usando la misma máquina. En las últimas estrategias se han logrado achicar las fórmulas que es necesario darle a MiniSat a un 55 % y 64 % en términos de cláusulas.

Un hecho que no se hizo notar hasta el momento es que la última iteración consume considerablemente más tiempo que las anteriores. O sea que, MiniSat es capaz de encontrar rápidamente valuaciones que satisfagan la fórmula. Encontrar una valuación no suele requerir recorrer todo el espacio de búsqueda. Como se muestra en el cuadro 14, para las estrategias más veloces, teniendo en cuenta la cantidad de iteraciones, es notoria la diferencia. En las siguientes estrategias se muestra nuevamente este efecto con mayor detalle.

# Identifiers	Estrategia	# It.	It. no finales ■	Ult. it. □	no finales vs. ult.
3	mj	181	1 s	0 s	
5	mj	1076	17 s	0 s	
7	mjp75	2939	6 m	54 s	
8	mj	3259	27.2 m	26.6 m	
9	mjp25	5034	11.6 h	41.8 h	

Cuadro 14: Comparación de tiempo requerido por distintas iteraciones

Estrategias mij, mijpα

A lo largo de las diferentes estrategias se han alterado las etapas de FórmulaInicial y NuevasCláusulas. Ya se mencionó otras alternativas y en la sección 7 se agregan otras. En esta ocasión se intenta mejorar la etapa Analizar.

Las anteriores estrategias hacen uso de MiniSat para analizar la fórmula g. Una característica importante de esta etapa a lo largo de las iteraciones es que a g sólo se van agregando cláusulas. Si denominamos g_i a la fórmula g de la i-ésima iteración, se cumple que $g_{i-1} \subseteq g_i$. Luego, dada una valuación v tal que $v \nvDash g_{i-1}$, se sabe que $v \nvDash g_i$.

Los sat solvers, en la medida que recorren el espacio de búsqueda para hallar una valuación que satisfaga la fórmula, van descartando aquellas que no lo hacen.

El objetivo es encontrar una forma de aprovechar todo el espacio de búsqueda recorrido para g_{i-1} al momento de analizar g_i ya que todas las valuaciones que fueron descartadas para g_{i-1} deberían ser descartadas en la evaluación de g_i.

Esta situación es conocida como *incremental sat solving* o *effective sat solving*[Sör08]. La manera en la que se aprovecha el espacio de búsqueda recorrido es mediante la reutilización de cláusulas inferidas. Las cláusulas se infieren al llegar a una refutación mientras se busca una valuación que satisfaga la fórmula. Cada cláusula inferida es una consecuencia lógica de las anteriores y eso las hace válidas para una fórmula que incluya las cláusulas de la anterior.

No todos los sat solvers cuentan con soporte de *incremental sat solving*. MiniSat es uno de los que sí soporta dicha funcionalidad, lo cual resulta ventajoso para medir el beneficio de reutilizar las cláusulas inferidas.

A continuación se trabaja con una modificación de las estrategias mj y mjpα en donde se cambia la etapa Analizar para utilizar MiniSat aprovechando el soporte de *incremental sat solving*. Las nuevas estrategias se denominan mij y mijpα respectivamente. En el cuadro 15 se comparan las dos primeras. Con respecto a la simplificación de la fórmula de entrada se logran resultados similares entre ambas estrategias. En la mayoría de los casos, la ganancia en tiempo es dramática. La nueva estrategia requiere entre un 10 % y 30 % de tiempo con respecto a la anterior. Lamentablemente para la instancia # Identifiers=9 hubo un retroceso. Se tardó 5 veces más en resolver la instancia.

La incorporación del *incremental sat solving* puede no ser positiva siempre, a pesar de parecerlo desde el punto de vista teórico. El espacio de búsqueda se recorre de manera

# Identifiers	mj ■	mij □	mj vs. mij	Cl. mj	Cl. mij
3	1 s	<1 s		57 %	58 %
5	17 s	3 s		65 %	64 %
7	8 m	56 s		66 %	68 %
8	53.8 m	18.1 m		68 %	67 %
9	54.3 h	12 d		67 %	65 %
11	n/a	n/a		n/a	n/a

Cuadro 15: Tiempos y proporción de cláusulas. Estrategia mj vs. Estrategia mij

distinta si se usa *incremental sat solving*, el ya conocer valuaciones que pueden ser ignoradas no disminuye obligatoriamente el tiempo de análisis.

Como se mostró antes, es la última iteración la más costosa. Para aportar otro punto de comparación entre las estrategias mj y mij en el cuadro 16 se muestran los tiempos requeridos durante todas las iteraciones salvo la última.

# Identifiers	mj ■	mij □	mj vs. mij
3	1 s	<1 s	
5	17 s	3 s	
7	7 m	23 s	
8	27.2 m	3 m	
9	4.3 h	17 m	
11	n/a	n/a	

Cuadro 16: Tiempos hasta la última iteración. Estrategia mj vs. Estrategia mij

En esta ocasión siempre hubo una mejora para todos los casos. En la sección 7 se remarca por qué es útil analizar el tiempo requerido para alcanzar la última iteración.

En particular se resalta el comportamiento para # Identifiers = 9, en donde el análisis total pasó de 54.3 horas a 12 días, mientras que el tiempo necesario para alcanzar la última iteración pasó de 4.3 horas a 17 minutos.

Para comparar las estrategias de las familias mjpα y mijpα se escogen las mejores de cada una, en términos de tiempo y luego en cantidad de cláusulas finales. En el cuadro 17 se las contrasta con los resultados obtenidos para las estrategias de la familia mjpα, sin usar el análisis incremental de MiniSat. En este caso, hubo mejora constante por la incorporación de la última modificación. Incluso se alcanzó una nueva marca en los tiempos para resolver la instancia con #Identifiers = 9, mejorando de 54.3 horas a 22.1 horas totales.

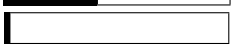
Como se mencionó en la sección 4.3.2, es la última iteración la que insume considerable tiempo con respecto a las otras y, en algunos casos, al análisis completo. En el cuadro 18 se analizan éstas proporciones para las estrategias más veloces conseguidas hasta el momento. Dada la mejora lograda por la última modificación la diferencia es menos notoria que la del

# Identifiers	mjpα ■	mijpα □	mjpα vs. mijpα	Cl. mjpα	Cl. mijpα	
3	<1 s	<1 s		56 %	56 %	mjp50 mijp50
5	23 s	5 s		65 %	65 %	mjp75 mijp75
7	7 m	45 s		67 %	66 %	mjp75 mijp75
8	1 h	12.6 m		66 %	66 %	mjp75 mijp25
9	53.4 h	22.1 h		64 %	64 %	mjp25 mijp25
11	n/a	n/a		n/a	n/a	n/a n/a

Cuadro 17: Tiempos y proporción de cláusulas Estrategias mjpα vs. Estrategias mijpα

cuadro 18, sin embargo no deja de ser válida. Para los casos #Identifiers 3 y 5 el problema se resuelve en tan poco tiempo que carece de importancia éste análisis.

A fin de disponer de una visión más abarcativa sobre el tiempo de cada iteración, en las figuras 6, 7 y 8 se grafica qué proporción del tiempo total se insume en cada iteración. El tiempo se grafica en unidades de progreso (entre 0 y 1) donde la última iteración corresponde al 1. Como se puede apreciar la diferencia entre todas las iteraciones es ínfima, dejando como excepciones a algunas de las iteraciones finales, en especial la última siendo la mayor. Nuevamente hay que tener en cuenta la cantidad de iteraciones frente a las medidas de tiempo que se están tratando, no alcanza con observar solamente las proporciones.

# Identifiers	Estrategia	# It.	It. no finales ■	Ult. it. □	no finales vs. ult.
3	mijp50	319	<1 s	0 s	
5	mij	989	3 s	0 s	
7	mijp75	2693	40 s	5 s	
8	mijp25	8399	5.2 m	7.4 m	
9	mijp25	13042	32.6 m	21.5 h	

Cuadro 18: Comparación de tiempo requerido por distintas iteraciones

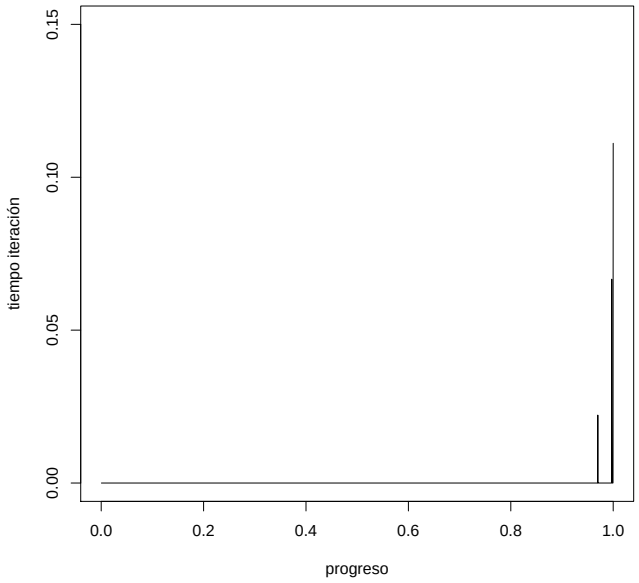


Figura 6: Tiempo de cada iteración para # Identifiers=7 y estrategia mijp75

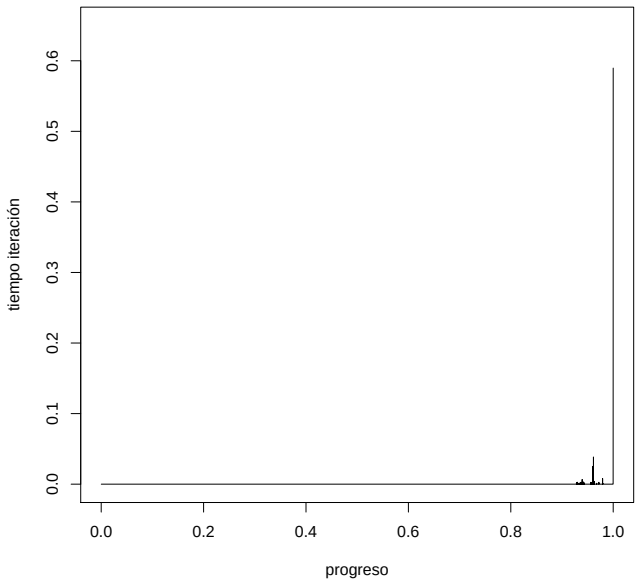


Figura 7: Tiempo de cada iteración para # Identifiers=8 y estrategia mijp25

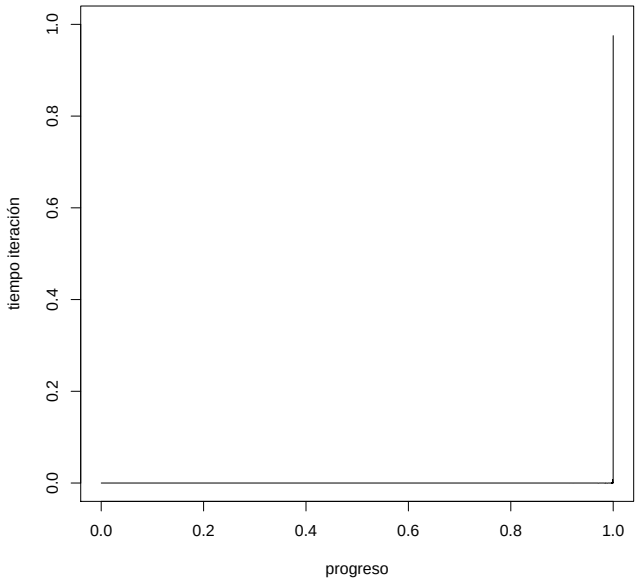


Figura 8: Tiempo de cada iteración para # Identifiers=9 y estrategia mijp25

Otra faceta a analizar es la reducción de variables. En el cuadro 19 se muestra cuántas variables se han logrado eliminar, tanto primarias como totales. En la primer columna se muestran los valores iniciales que corresponden a la fórmula f. En la segunda, los valores para las estrategias en donde se lograron eliminar la mayor cantidad de variables primarias. Finalmente, en la tercera, se muestra el resultado conseguido para las estrategias más veloces hasta el momento, a fin de poner en la balanza cuánto se sacrifica por escoger un técnica más veloz.

# Identifiers	Variables originales		Mayores eliminaciones			Más veloces		
	Primarias	Totales	Primarias	Totales		Primarias	Totales	
3	196	2979	24	500	mijp10	18	523	mij
5	400	7079	23	899	mjp5	11	900	mij
7	686	13787	21	1168	mijp10	16	1347	mijp75
8	856	18031	36	1628	mijp5	15	1331	mijp25
9	1048	26825	29	2136	mijp5	23	2196	mijp25
11	1492	43351						

Cuadro 19: Comparación de eliminación de variables

4.3.3. Conclusiones tercer etapa

A partir del último cambio realizado se alcanzan tiempos comparables, aunque mayores que utilizar directamente MiniSat en aquellas instancias que son tratables por éste último. Dado que estamos utilizando MiniSat en forma iterativa varias veces por cada especificación a analizar, resulta improbable obtener mejores tiempos en aquellas instancias que originalmente son tratables. El haber alcanzado los tiempos que se muestran en el cuadro 17 es muy satisfactorio.

Desde la segunda etapa se logró resolver una instancia nueva del problema. Este hecho se mantiene luego de las modificaciones de la tercer etapa y la ganancia en eficiencia lograda se hace notoria por las magnitudes de tiempo que se presentan.

El haber podido resolver instancias de mayor tamaño está estrechamente relacionado con haber podido ignorar cláusulas de la fórmula a analizar. En general se termina operando con un 55 % / 67 % de la fórmula original en términos de cláusulas. Éste es uno de los logros más notorios del trabajo. Se resalta la proporción de cláusulas ignoradas frente al aumento del tamaño de la fórmula original a medida que se modifican las cotas del análisis. El ignorar cláusulas de la fórmula original puede interpretarse como un concepto análogo al de *program slicing* [Wei81].

Relativo a la reducción de la fórmula, se logran eliminar unas cuantas variables primarias, aunque la proporción baja drásticamente a medida que aumenta el tamaño de la fórmula. A diferencia de la reducción de cláusulas, en donde se mantienen en una proporción similar a lo largo de los distintos experimentos.

Es dramática la distribución del tiempo necesario en cada una de las distintas iteraciones. En al sección 7 se detalla cómo podría aprovecharse la estimación de la última iteración a partir del tiempo transcurrido. El utilizar *incremental sat solving* permite discriminar aún más las iteraciones finales de las anteriores.

4.4. Experimentos en DynAlloy

4.4.1. DynAlloy

El lenguaje Alloy está orientado a modelos estáticos en donde no hay cambio de estado. El lenguaje DynAlloy [FLPB$^+$05] es una extensión al lenguaje Alloy que permite describir la evolución del estado de un sistema y predicar propiedades sobre las posibles ejecuciones del sistema. Un programa y su contrato es un ejemplo de los sistemas que podrían modelarse de esta manera.

El lenguaje DynAlloy está compuesto por a) acciones atómicas, las cuales constituyen la unidad de transición por medio de una precondición y una postcondición. Además permite la definición de programas mediante composición de acciones atómicas y otros programas en forma de b) composición secuencial, c) número arbitrario de iteraciones, d) chequeos y e) elección no determinística (*choice*).

Al analizar una especificación DynAlloy se genera una especificación Alloy que la representa. Luego el análisis se reduce al de ésta última. Además de especificar cotas a las signaturas de la especificación, se indica una cota a la cantidad de iteraciones permitidas en

programa: *loop unrolling scope*. A modo de ejemplo, para comprender el mecanismo usado por DynAlloy para representar los cambios de estados, suponiendo que se desea verificar para un sistema con un estado s la especificación:

$$\{P(s)\}A(s); B(s)\{Q(s)\}$$

en donde se establece que para toda ejecución que comienza satisfaciendo P, luego de ejecutar la acción A seguida de la B, finalizan en un estado que satisface Q. Cada una de las acciones A y B se declara mediante sus respectivas precondiciones (P_A y P_B) y postcondiciones (Q_A y Q_B). DynAlloy genera como fórmula Alloy equivalente:

$$\forall s_0, s_1, s_2 \cdot P(s_0) \wedge P_A(s_0) \wedge Q_A(s_0, s_1) \wedge P_B(s_1) \wedge Q_B(s_1, s_2) \rightarrow Q(s_2)$$

El concepto general es que con cada potencial cambio de estado en el flujo de la especificación DynAlloy, se crean variables en la especificación Alloy para representar este cambio de estado. Notar que las variables de un modelo Alloy tienen un solo valor, mientras que las variables de DynAlloy varian potencialmente el valor que representar en cada transición del sistema que se está modelando.

La traducción de un programa DynAlloy puede generar una especificación Alloy relativamente compleja. Esto resulta principalmente de la necesidad de representar el cambio de estado. La especificación Alloy es incluso de mayor longitud a medida que se aumenta el *loop unrolling scope* ya que una especificación:

$$\{P(s)\}A(s)^*\{Q(s)\}$$

se traduce, para un *loop unrolling scope* n, a la fórmula:

$$\forall s_0, \ldots s_n \cdot P(s_0) \wedge P_A(s_0) \wedge Q_A(s_0, s_1) \wedge \ldots \wedge P_A(s_{n-1}) \wedge Q_A(s_{n-1}, s_n) \rightarrow Q(s_n)$$

DynAlloy es una alternativa, por ejemplo, para el estudio de programas Java [GF06] o arquitecturas dinámicas [BG08]. Resulta interesante analizar si el trabajo desarrollado en la sección anterior es de utilidad en el análisis de especificaciones DynAlloy.

En esta sección se utilizan como casos de estudio especificaciones DynAlloy que fueron generadas a partir de programas Java con anotaciones en JML [LLL+00] usando la herramienta TACO (*Translation of Annotated COde*) [FGRLP09]. Los programas estudiados también se presentan en [FGRLP09]. Una breve descripción de cada uno se lista a continuación.

LList-Cont Chequeo de pertenencia en listas simplemente enlazadas.

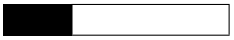
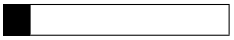
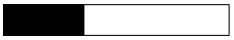
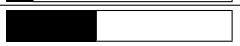
AList-Rem Eliminación de elementos en listas circulares doblemente enlazadas.

TrSet-Find Chequeo de pertenencia en *red-black trees* basado en `java.util`.

Avl-Find Chequeo de pertenencia en árboles AVL.

BH-Min Mínimo en colas de prioridades basadas en *binomial heaps*.

En el cuadro 20 se resumen los resultados obtenidos para estos experimentos comparando el *MiniSat prover* y la estrategia mij. En todos los casos se llegó a que la especificación era válida, salvo indicación de error. Al analizar una especificación DynAlloy, además de dar cotas para las signaturas de la misma manera que para una especificación Alloy, se debe indicar una cota para el *loop unrolling*, que limita la cantidad de iteraciones en un ciclo. Los casos de estudio de esta sección utilizan una cota de 6 para el *loop unrolling*. La columna de tamaño indica la cantidad de elementos en las signaturas Alloy.

Caso	Tamaño	MiniSat ■	mij □	MiniSat vs. mij	% cláusulas
LList-Cont	5	4 s	1 s		50 %
	7	4 s	4 s		51 %
	10	8 s	18 s		51 %
	12	9 s	36 s		51 %
	15	12 s	1.5 m		49 %
	17	24 s	3.1 m		49 %
	20	39 s	6.3 m		45 %
AList-Rem	5	5 s	4 s		58 %
	7	7.8 s	14 s		60 %
	10	2.1 m	3.4 m		63 %
	12	Error (30 m)	**21 m**		63 %
TrSet-Find	5	2.9 m	19 s		72 %
	7	14 m	3.5 m		73 %
Avl-Find	5	5.3 s	11 s		69 %
	7	39 s	4.9 m		72 %
BH-Min	5	5 s	7 s		71 %
	7	1.4 m	42 s		74 %
	10	Error (2 h)	**2.1 h**		78 %

Cuadro 20: Especificaciones DynAlloy analizadas con *MiniSat prover* y la estrategia mij.

En la mayoría de los casos con MiniSat se obtuvieron mejores tiempos. Un hecho positivo es que, como se muestra en la columna "% cláusulas", solo es necesario trabajar con una parte de la fórmula. Entre un 49 % y un 78 %. Esta reducción permitió superar en unos pocos casos las instancias con las que podía lidiar (AList-Rem y BH-Min). Es notable la mejora en los casos de TrSet-Find. Para los casos AList-Rem y BH-Min que fueron resueltos por ambas estrategias, los tiempos con similares. Para los restantes LList-Cont y Avl-Find, si bien ambas técnicas pudieron resolver las mismas instancias, MiniSat obtuvo una evidente ventaja.

En el caso LList-Cont, sorprende cómo disminuye la proporción de cláusulas necesarias para analizar la especificación a medida que aumenta el tamaño de las signaturas.

5. Conclusiones

Así como en los trabajos citados en la sección 6, hay espacio para mejorar la escalabilidad del análisis de especificaciones Alloy. En el presente trabajo se vieron distintas mejoras que se pueden aplicar de forma independiente al *sat solver* elegido y sobre todo sin entrada adicional por parte del usuario. De esta manera se contribuye al conjunto de heurísticas y estrategias para atacar un problema tan clásico como útil.

Se reconoce que las mejoras logradas no son esenciales en sí, dado los tiempos requeridos para obtener una diferencia sustancial con respecto a MiniSat, a excepción de que este último termine en error. Como se planteó en su momento el interés radicaba en aumentar la escalabilidad y eso fue conseguido, sobre todo en el caso de estudio de la sección 4.3. Sin aprovechar aspectos semánticos de la especificación ni alterar el *sat solver* subyacente es difícil lograr una disminución notable en el tiempo requerido. Aunque sí, como se logró, aprovechar mejor los recursos.

El desempeño obtenido en los experimentos de la sección 4.4 exhibe que se está lejos de haber conseguido una estrategia genérica que convenga aplicar siempre, pero que en los casos donde MiniSat falla, es posible obtener el resultado buscado con alguna de las estrategias de resolución incremental.

Unas de las cualidades que resalta es el poco tiempo necesario para alcanzar la última iteración en comparación con lo que lleva todo el proceso. Ésto es algo que puede aprovecharse como se menciona en la sección 7.

6. Trabajo relacionado

En [UK08] se presenta Kato, una herramienta para intentar aumentar la escalabilidad del Alloy Analyzer mediante la priorización de cláusulas al analizar la especificación. Sin entrar en detalles técnicos, la especificación Alloy se divide sintácticamente en dos partes o *slices*. La primera debería corresponder a la especificación más importante, quizás estructural de los modelos. Mientras que la segunda debería contener las restricciones sobre el resto del modelo que están fuertemente gobernadas por los valores concretos que se pueden fijar en la primera parte.

El criterio de separación son las variables relacionales. Por ejemplo, para una especificación de árboles binarios de búsqueda con las relaciones root , left , right , parent, size , key, las últimas dos van a estar restringidas una vez fijadas las anteriores. El paso inicial es armar una especificación base que descarte las restricciones que mencionan este último tipo de variables relacionales. Luego se analiza la especificación base, con esperanza de que sea más fácil que la original. Al encontrarse un contraejemplo, se busca extenderlo a la especificación original y se descarta en caso de que sea espúreo, buscando un siguiente contraejemplo en la especificación base.

Ambas propuestas, la de Kato y la del actual trabajo, presentan formas de trabajar de manera incremental con el modelo. El criterio de separación de Kato puede ser muy rígido para estructuras con fuerte cohesión entre las distintas relaciones, como se indica en

el trabajo para los *red-black trees*.

La técnica presentada en [UK08] requiere un *sat solver* que permita enumerar soluciones, si bien son varios, no todos lo hacen. También se hace una manipulación sintáctica de la especificación similar a la realizada en sección 3 a fín de aprovechar mejor el procedimiento para analizar la especificación.

A diferencia de la técnica presentada en este trabajo, Kato requiere una entrada adicional: el criterio de separación. Para facilitar dicha tarea, se realiza una exploración de los mejores candidatos para dejar al usuario elegir alguna con cierta noción del impacto en base a experimientos en dominios reducidos.

Por último, dado que ambas técnicas trabajan en niveles distintos de abstracción, en principio, son combinables aunque no es evidente que deba haber mejora.

En [DVT07] se presenta Miniatur, donde se implementan varias ideas para mejorar el análisis de programas escritos en Java. El trabajo aporta a) una traducción de programas y especificación al lenguaje Alloy y b) una codificación alternativa de enteros y arreglos.

La traducción incorpora lógica para incluir en la fórmula sólo aquellas secciones del programa relevantes a la especificación que se espera verificar. De hacer una traducción sin tener en cuenta la propiedad a verificar, se produce una especificación Alloy más compleja de lo necesario. Un efecto similar se logra en el presente trabajo, tanto en la minimización de modelos (sección 3) como en resolución incremental (sección 4) al incluir solo las restricciones necesarias. Al igual que en el caso de Kato, la herramienta Miniatur sería combinable, aunque no es evidente que deba haber mejora con respecto a este aporte.

La codificación alternativa de enteros y arreglos está motivada en un mejor aprovechamiento de las cotas del análisis para poder representar una mayor cantidad de enteros y arreglos con más elementos. Este aporte es tan ortogonal que es probable que muchas especificaciones Alloy resulten beneficiadas por incorporarlo dado que, lidiar con enteros suele traer ciertas incomodidas por ser tan reducida la cantidad de elementos representados.

7. Trabajo futuro

Con el objetivo de saber cuán preciso es el conjunto insatisfacible de cláusulas obtenido al finalizar el proceso de resolución incremental, sería deseable compararlo con el *unsat core* para aquellos casos que pueden ser resueltos por el MiniSat prover. Una forma de medir el índice de aciertos y de errores se utiliza en [DFG+10].

A fin de lograr una buena aproximación a un *unsat core* a partir de las cláusulas anteriores es muy probable que se deba realizar una minimización de éstas para lograr un conjunto insatisfacible minimal. Algunas técnicas de minimización se presentan en [TCJ08].

En los resultados se aprecia que las últimas, y sobre todo la última, son las iteraciones que más demoran. Si el usuario, dado el tiempo insumido decide finalizar el proceso en cierta iteración, se estaría en presencia de un *posible* conjunto insatisfacible de cláusulas. Esta situación es similar a la motivación de [DFG+10]. Sería deseable estudiar cómo se comporta en ejecuciones parciales el procedimiento y ver si es posible ofrecer información al usuario para aumentar su confianza más allá de la intuición por el tiempo transcurrido.

Hasta el momento no se estudió el efecto de escoger un conjunto de cláusulas iniciales no trivial. Sin entrar en detalles semánticos de cada especificación, el utilizar como cláusulas iniciales aquellas que provienen de la aserción puede orientar de forma positiva la sucesiva generación de contraejemplos a lo largo de las iteraciones.

Otra potencial mejora que queda como trabajo futuro es un refinamiento en la etapa NuevasCláusulas. Las últimas estrategias utilizan el valor asignado a cada una de las variables proposicionales para detectar las cláusulas no satisfechas. En muchos casos, especialmente al comienzo, se está utilizando el valor asignado a variables que ni siquiera aparecen en la fórmula CNF. Como no aparecen, se sabe que se podrían alterar dicho valor sin afectar la satisfacción con respecto a la fórmula CNF. Con esta motivación se puede hacer una incorporación de nuevas cláusulas más conservadora y que al mismo tiempo contemple varias valuaciones por iteración. El mecanismo sería usar una valuación v que incluya sólo aquellas variables que aparecen en la fórmula CNF g. Esta valuación representaría a varias que satisfacen la fórmula g. En la etapa de NuevasCláusulas al usar la valuación (que resulta incompleta para evaluar f) es posible que no se determine, para algunas cláusulas, si es satisfecha o no. Por este motivo, para garantizar la correctitud y el progreso del algoritmo va a ser necesario no realizar esta técnica en cada iteración.

La herramienta desarrollada informa estadísticas sobre el tiempo insumido en cada iteración. Éstas se podrían aprovechar para crear indicadores que señalen la posibilidad de haber alcanzado la última iteración, aunque no se haya finalizado. Dejando al usuario optar por detener el análisis, quedándose con un resultado tentativo, a fin de reducir dramáticamente el tiempo total empleado.

La detección de posible última iteración podría utilizarse para disparar otras técnicas de análisis a partir del conjunto de cláusulas actual. Por ejemplo técnicas como ParAlloy [RGPF10] en donde se aplica una resolución en paralelo. ParAlloy se vería beneficiado por la disminución de variables primarias, hecho que se logra en algunos casos de estudio. Se menciona la mejora solo ante la eliminación de variables primarias por detalles técnicos del funcionamiento de ParAlloy: éste utiliza representaciones intermedias de la fórmula a analizar en donde no se han introducido las variables secundarias.

Referencias

[ADKM02] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". Technical report, IN POPL '02: Proceedings of the 29Th Acm Symposium on the Principles of Programming Languages, 2002.

[BG08] Antonio Bucchiarone and Juan Pablo Galeotti. Dynamic software architectures verification using dynalloy. In *GT-VMT 2008 Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques*, march 2008.

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

[Cha08] F. Change. Alloy analyzer 4.0, 2008.

[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.

[DFG+10] Nicolás D'Ippolito, Marcelo F. Frias, Juan P. Galeotti, Esteban Lanzarotti, and Sergio Mera. Alloy+hotcore: A fast approximation to unsat core. In *ASM*, pages 160–173, 2010.

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

[DVT07] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a sat solver. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 195–204, New York, NY, USA, 2007. ACM.

[EMCJ99] Doron A. Peled Edmund M. Clarke Jr., Orna Grumberg. *Model Checking*. The MIT Press, 1999.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FGRLP09] Marcelo Frias, Juan Pablo Galeotti, Nicolás Rosner, and Carlos G. López Pombo. Efficient sat-based analysis of annotated o-o programs by automated computation of tight bounds for class fields. 2009.

[FLPB+05] Marcelo F. Frias, Carlos G. López Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S. E. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.

[GF06] Juan Pablo Galeotti and Marcelo Frias. Dynalloy as a formal method for the analysis of java programs. In K. Sacha, editor, *Software Engineering Techniques: Design for Quality*, volume 227, pages 249–260. IFIP International Federation for Information Processing, Springer, 2006.

[GKS08] Roman Gershman, Maya Koifman, and Ofer Strichman. An approach for extracting a small unsatisfiable core. *Form. Methods Syst. Des.*, 33(1-3):1–27, 2008.

[GRLPF10] Juan Pablo Galeotti, Nicolas Rosner, Carlos Lopez Pombo, and Marcelo Frias. Analysis of invariants for efficient bounded verification. In *2010 International Symposium on Software Testing and Analysis*. ACM Sigsoft, 2010.

[IEE04] IEEE. Swebok: Guide to the software engineering body of knowledge, 2004.

[Jac02] Daniel Jackson. Case stude 1: Railway safety, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[JS04] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In *SAT (Selected Papers*, pages 183–198, 2004.

[LLL+00] Gary T. Leavens, K. Rustan M. Leino, Gary T. Leavens, Clyde Ruby, Clyde Ruby, K. Rustan, K. Rustan, M. Leino, M. Leino, Erik Poll, Erik Poll, Bart Jacobs, and Bart Jacobs. Jml: notations and tools supporting detailed design in java, 2000.

[RGPF10] Nicolás Rosner, Juan P. Galeotti, Carlos López Pombo, and Marcelo F. Frias. Paralloy: Towards a framework for efficient parallel analysis of alloy models. In *ASM*, pages 396–397, 2010.

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

[Sör08] Niklas Sörensson. Effective sat solving. 2008.

[SSJ+03] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. *Automated Software Engineering, International Conference on*, 0:94, 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] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pages 632–647. Wiley, 2007.

[Tse68] G. S. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[UK08] Engin Uzuncaova and Sarfraz Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 310–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Zav06] Pamela Zave. Compositional binding in network domains. In *FM*, pages 332–347, 2006.