Mejorando la Aplicación de Abstracción por Predicados a Especificaciones DynAlloy

Rodrigo Ariño[1], Renzo Degiovanni[1], Raul Fervari[2], Pablo Ponzio[13], and Nazareno Aguirre[13]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina
{rarinho, rdegiovanni, pponzio, naguirre}@dc.exa.unrc.edu.ar
[2] Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Córdoba, Argentina
fervari@famaf.unc.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. En este trabajo presentamos técnicas para mejorar la performance de *Abstracción por Predicados* en el contexto de análisis de especificaciones *DynAlloy*. Este trabajo extiende nuestro trabajo previo en la misma dirección, principalmente mediante un mecanismo de detección de inducción, que permite independizar, bajo ciertas condiciones, el tiempo de análisis del programa de la cota en la cantidad de iteraciones exigida por *DynAlloy*. Dado que el tiempo de análisis de programas *DynAlloy* depende exponencialmente de dicha cota, las ganancias obtenidas al aplicar esta optimización son notables. Sin embargo, la técnica no es aplicable en casos arbitrarios, sino sólo bajo ciertas condiciones que identificamos en este trabajo. Por otra parte, la técnica puede requerir intervención del usuario, a través de la introducción manual de predicados de abstracción. Las optimizaciones mencionadas fueron incorporadas a la implementación de nuestra herramienta, permitiéndonos evaluar las mismas en diversos casos de estudio que reportamos en este artículo. Estos casos de estudio corresponden a modelos *DynAlloy* provenientes de programas que operan sobre listas enlazadas.

1 Introducción

La *Abstracción por Predicados* [6] es una técnica que ha demostrado ser muy efectiva en el contexto de *Model Checking*, tanto como una forma de manejar el problema de la explosión de estados en modelos acotados como para permitir su aplicación a modelos de estados infinitos. Una ventaja destacada de esta técnica es que puede aplicarse de manera automatizada por herramientas de software.

En [1] presentamos una implementación de la técnica de abstracción por predicados aplicada a especificaciones escritas en el lenguaje *DynAlloy*, una extensión que provee la noción de estado y programa como transformador de éste al conocido lenguaje de especificaciones *Alloy*. La implementación mencionada, basada en [5], introduce algoritmo que, dados un modelo *DynAlloy* (extraido de

un programa Java) y un conjunto de predicados de entrada (obtenidos de las condiciones del programa), realiza una ejecución abstracta "bajo demanda" del programa respecto de los predicados, de forma completamente automática. El algoritmo además implementa una técnica de refinamiento de predicados (similar a [3]) para el caso en que aparezcan contraejemplos durante la ejecución abstracta. Uno de los aspectos más importantes del trabajo anterior es la idea de reuso de cálculos previos. Este punto fue clave para permitir que la implementación del algoritmo dado mejore el desempeño de Alloy Analyzer en los casos de estudio presentados en aquel artículo.

En este trabajo, introducimos nuevas ideas para mejorar el desempeño del algoritmo mencionado anteriormente, que surgieron de nuestra experiencia en la aplicación de Abstracción Por Predicados a modelos *DynAlloy*. La mejora más destacada, que denominamos detección de inducción, permite reutilizar partes del modelo abstracto que fueron calculadas anteriormente durante la ejecución abstracta, y bajo ciertas condiciones permite independizar el tiempo de análisis del programa de la cota en la cantidad de iteraciones dada por el usuario para el programa *DynAlloy* concreto. Sin embargo, y debido a limitaciones de la técnica, en muchos casos se requiere de la asistencia del usuario, mediante la provisión de predicados que hagan factible la aplicación de la técnica. La razón de esta limitación es que, en el caso general, estos predicados no pueden ser "descubiertos" por métodos como el presentado en [3].

Hemos adaptado nuestra herramienta de análisis para soportar esta nueva técnica, y describiremos aquí también esta adaptación. Discutiremos además alternativas en las estructuras de datos eficientes para mejorar el desempeño del módulo que almacena resultados previos durante la ejecución abstracta. Finalmente, reportaremos los resultados obtenidos en la aplicación de las técnicas introducidas en este trabajo a casos de estudio provenientes de programas Java que manipulan listas simplemente enlazadas, comparando su eficiencia con la del algoritmo original.

El resto del artículo está organizado de la siguiente manera. En la próxima sección introduciremos los lenguajes *Alloy* y *Dynalloy*. En 3 repasaremos los puntos claves de nuestro trabajo previo. Más adelante, en la sección 4, presentaremos las optimizaciones mencionadas anteriormente. En 5 mostraremos los resultados de aplicar los diferentes algoritmos a diversos casos de estudio. Por último, discutiremos sobre la aplicabilidad de las técnicas presentadas, presentaremos algunas conclusiones y propuestas de trabajos futuros.

2 Los lenguajes Alloy y DynAlloy

Alloy es un lenguaje de especificación basado en lógica relacional, una extensión de lógica de primer orden diseñada para soportar operadores relacionales: imagen relacional, clausuras, traspuesta, etc. La sintaxis de *Alloy* es simple y semejante a la de un lenguaje orientado a objetos. El lenguaje fue diseñado con el objetivo de brindar soporte de análisis automático (Alloy Analyzer) que permite simular modelos y buscar contraejemplos de propiedades provistas por el usuario. El

análisis en *Alloy* se basa en la reducción a fórmulas proposicionales, y la utilización de un SAT-solver para verificar satisfactibilidad. *Alloy* es un lenguaje orientado a modelos, originalmente diseñado para especificar propiedades de estados de sistemas de software. En [7], Jackson propone especificar propiedades dinámicas en *Alloy* introduciendo manualmente la noción de traza de ejecución como parte de la especificación. Un problema con este enfoque es que la noción de traza de ejecución depende de las signaturas y operaciones de la especificación original, y debe ser definida de manera *ad-hoc* para cada modelo particular.

DynAlloy [4] es un lenguaje que ofrece una alternativa para el problema descripto anteriormente, mediante la incorporación de una sintaxis (y una semántica) para lidiar con ejecuciones, inspiradas en la lógica dinámica. *DynAlloy* extiende a *Alloy* con la noción de acción atómica, y agrega operaciones para componer acciones. Esto, sumado a la posibilidad de describir aserciones de correción parcial, dan lugar a una forma más sencilla de especificar propiedades dinámicas de los sistemas de software.

Por razones de espacio introduciremos los aspectos más relevantes del lenguaje *DynAlloy* mediante ejemplos. El estado de un sistema es especificado en *Alloy* y *DynAlloy* usando signaturas. Las signaturas definen conjuntos de elementos, y relaciones entre ellos. Como ejemplo, definimos la estructura de listas enlazadas de enteros:

```
one sig Null {}        sig Node {               sig List {
                         next: Node+Null,         head: Node+Null,
                         value: Int             }
                       }
```

En las definiciones anteriores, *Null* representa la referencia nula, frecuente en los lenguajes de programación, *Node* es el universo de los nodos que pueden pertenecer a la lista enlazada, *next* es una relación que asocia cada nodo con su sucesor y *value* asigna a cada nodo el valor entero almacenado en él. Finalmente, *List* caracteriza las listas enzadadas, donde *head* denota el nodo cabecera de una lista.

Las acciones atómicas en *DynAlloy* son usadas para describir operaciones sobre las signaturas del modelo. Éstas deben ser definidas en términos de sus correspondientes pre y postcondiciones. Por ejemplo, una operación que elimina el elemento a la cabeza de una lista puede ser definida de la siguiente manera:

```
action removeFirst[l: List]          {
  pre { l.head != NullValue }
  post { l'.head = l.head.next }     }
```

La operación *removeFirst* sólo se puede ejecutar si la cabeza de lista no es *Null*. Nótese la similitud entre las expresiones que utilizan . (como por ejemplo *l.head*) con la notación de los lenguajes orientados a objetos. La postcondición introduce una nueva variable *l'* para denotar el estado de *l* después de la ejecución de la acción. En este caso, el nodo que sucede a la cabeza de la lista se convierte en la nueva cabeza de la misma. De esta manera, podemos pensar la postcondición de una acción como una relación entre los estados previo y posterior a la ejecución de la acción.

DynAlloy provee tres operadores que permiten componer acciones atómicas: composición secuencial (;), elección no determinista (+) e iteración (*). Podemos formar programas *DynAlloy* combinando estos tres operadores, acciones atómicas y tests (aserciones). Los programas pueden anotarse con pre y post-condiciones, para especificar la propiedad deseada. Por ejemplo:

```
assertCorrectness removeAll[l:List]   {
 pre = { }
 prg = {([l.head != NullValue]?;removeFirst(l))*;[l.head = NullValue]?}
 post = { l'.head = NullValue }       }
```

La aserción *removeAll* puede ser pensada como la especificación correspondiente al programa `while (head(l) != NullValue) do removeFirst(l)`. Su post-condición indica que cuando el programa termina la lista es vacía (y debe ser verificada). Dada una especificación *DynAlloy*, el *DynAlloy Translator* construye automaticamente un modelo *Alloy* que nos permite simular el programa usando *Alloy Analyzer*. Dado que el análisis se basa en una reducción a satisfactibilidad booleana, y el lenguaje es de primer orden, es necesario proveer cotas para el número máximo de iteraciones y el número de elementos para las signaturas (no necesariamente las mismas cotas) para poder llevar a cabo la traducción. El usuario es quien debe proveer dichas cotas. Las ejecuciones del programa que violan la especificación son exhibidas al usuario como contraejemplos por el analizador. En caso de no encontrar tales contraejemplos, se garantiza la validez de la propiedad verificada *en universos acotados por las cotas provistas por el usuario*, pero no necesariamente su validez absoluta, ya que es posible que existan contraejemplos de la propiedad de mayor tamaño que el determinado por las cotas. Sin embargo, este mecanismo ha demostrado ser sumamente útil, para identificar errores u otros problemas en especificaciones (e.g., inconsistencias, etc.).

3 Abstracción por Predicados para DynAlloy

La idea central detrás de la *Interpretación Abstracta* [2] consiste en interpretar las computaciones de programas en dominios más simples (abstractos), que contienen menos información sobre las computaciones pero son más fáciles de construir y explorar. Su principal ventaja es que, si las abstracciones se construyen de manera conservativa, es posible verificar propiedades del modelo concreto analizando sólo el espacio de estados abstracto. Esto permite, en principio, tratar el problema de la explosión de estados en el caso de programas sobre dominios finitos (como en nuestro caso), e incluso tratar programas que manipulan datos potencialmente infinitos.

Abstracción por Predicados es una técnica introducida en [6] como una forma de automatizar la construcción de modelos abstractos, a partir un conjuto $P = \{\varphi_1, ..., \varphi_l\}$ de predicados sobre el espacio de estados del programa. Basándonos en este trabajo desarrollamos un algoritmo de Abstracción por Predicados para el lenguaje de especificaciones *DynAlloy* [5]. El espacio de estados concreto (S) está compuesto por las signaturas que definen los estados de la especificación

DynAlloy. Dado un conjunto P de predicados de abstracción, el espacio de estados abstracto (S^A) consiste del conjunto de monomios[4] sobre las variables booleanas $B_1, ..., B_l$. Las funciones de abstracción ($\alpha : \wp(S) \rightarrow S^A$) y concretización ($\gamma : S^A \rightarrow \wp(S)$) se definen como se muestra a continuación:

- $\alpha(\psi(s)) = (\bigwedge\{B_i | \forall\, s : \psi(s) \Rightarrow \varphi_i(s)\}) \wedge (\bigwedge\{\neg B_i | \forall\, s : \psi(s) \Rightarrow \neg\varphi_i(s)\})$
- $\gamma(s^A) = s^A[B_1/\varphi_1, ..., B_l/\varphi_l]$

Así, dado un predicado ψ sobre el espacio de estados concreto (esto es, un conjunto de estados concretos), podemos construir automáticamente el estado abstracto correspondiente verificando, para cada $p \in P$, si todos los estados concretos que satisfacen ψ satisfacen φ_i, $\neg\varphi_i(s)$ o ninguno de los anteriores. En este trabajo utilizamos Alloy Analyzer para realizar estos chequeos. Las cotas requeridas por el analizador deben ser provistas por el usuario. En cambio, para concretizar un estado abstracto podemos simplemente reemplazar sintácticamente cada B_i por el φ_i correspondiente.

Utilizando α y γ definimos el estado inicial abstracto (I^A) y las transiciones abstractas ($SP^A(\tau_i, s^A)$) de la manera siguiente:

1. $I^A = \alpha(preProg)$
2. $SP^A(\tau_i, s^A) = \alpha(SP(\tau_i, \gamma(s^A)))$

Esto es, I^A se obtiene aplicando directamente α a la precondición del programa *DynAlloy* a verificar. Por otra parte, dados una acción τ_i y un estado abstracto s^A, definimos el resultado de la ejecución abstracta de τ_i en s^A como la abstracción del conjunto de estados que se obtienen de aplicar τ_i a todos los estados en la concretización de s^A. En 2, SP denota al transformador de predicados *strongest postcondition*[5] (postcondición más fuerte).

El algoritmo [1] construye las abstracciones bajo demanda, esto es, visita el grafo de control del programa utilizando el recorrido *Primero en Profundidad* (*Depth First Search*), abstrayendo primero el estado inicial, y aplicando las transiciones abstractas en el orden mencionado. Las iteraciones se ejecutan hasta un máximo de veces provisto por el usuario. Si deseamos computar una abstracción conservativa, tenemos que incluir la postcondición del programa *DynAlloy* (propiedad concreta a probar) entre los predicados de abstracción (llamémosla φ_p).

Es fácil ver que el procedimiento descrito anteriormente cubre al menos todas las posibles ejecuciones concretas (y posiblemente muchas más) hasta el límite dado para las iteraciones. De este modo, si todas las ejecuciones abstractas finalizan en monomios con B_p en forma positiva, entonces podemos asegurar que el programa concreto satisface la propiedad (hasta la cota dada para los ciclos). Si este no es el caso, el algoritmo se detiene cuando encuentra la primera traza

[4] Un monomio es una conjunción de variables proposicionales y sus negaciones, donde cada variable aparece a lo sumo una vez. La constante booleana $false$ también es considerada un monomio.

[5] Definido como: $SP(\tau_i, \psi) = \{s' \mid \forall\, s : S \mid \psi(s) \wedge pre_{\tau_i}(s) \wedge post_{\tau_i}(s, s')\}$.

que no satisfaga el requerimiento anterior. Estas trazas se llaman contraejemplos abstractos, y pueden ser provocadas por un fallo en el modelo concreto, o porque la abstracción es demasiado gruesa -esto es, se requieren predicados adicionales para poder verificar la propiedad. Si al concretizar un contraejemplo abstracto se obtiene una ejecución real que viola la postcondición del programa, entonces se muestra al usuario el contraejemplo y el procedimiento termina. En otro caso, estamos ante la presencia de un contraejemplo espurio: un contraejemplo abstracto que no tiene una contraparte real, producido por la pérdida de información durante el proceso de abstracción. Los contraejemplos espurios deben eliminarse si se quiere continuar con el proceso de verificación en abstracto. Para ello utilizamos una adaptación del método presentado en [3], que por problemas de espacio no será discutido aquí. En nuestro trabajo anterior expusimos las ideas básicas de la técnica [1].

Para concluir, queremos destacar dos características importantes del algoritmo. En primer lugar, el algoritmo no construye el espacio abstracto de estados completo, como lo hace la técnica propuesta en [6], ya que esto fue -en los experimentos que realizamos- demasiado caro computacionalmente. En cambio, como se discutió anteriormente, el modelo abstracto se construye bajo demanda. Otro aspecto importante de nuestro algoritmo es que partes de las abstracciones computadas pueden ser reusadas. Como las abstracciones son construidas utilizando Alloy Analyzer, que se basa en un procedimiento de reducción a satisfactibilidad booleana (problema que a su vez es NP-completo), reusar tanta información como sea posible es fundamental para mejorar el tiempo de ejecución. La idea detrás del reuso es que, si ya construimos previamente un estado abstacto s'^A, obtenido a partir de la aplicación de la acción τ_i al estado s^A, entonces las variables booleanas que aparecen en s'^A van a aparecer de la misma manera en cualquier estado que resulte de la aplicación de τ_i a cualquier estado abstracto consistente s''^A más fuerte que s_A (esto es $s''^A \implies s_A$). Esto se debe a la forma en que computamos las abstracciones (verificando implicaciones lógicas). Por lo tanto, el algoritmo almacena los resultados de las ejecuciones de transiciones abstractas, y reusa esta información siempre que sea posible en futuras aplicaciones de las mismas transiciones.

4 Optimizaciones al Algoritmo de Abstracción por Predicados DynAlloy

Detección de Inducción. Estas es la técnica más importante implementada en este trabajo, y se basa en la siguiente observación. Al ejecutar el algoritmo discutido en la sección anterior muchos de los estados abstractos aparecen más de una vez. Sean $t = s_0^A, s_1^A, s_2^A, s_3^A, .., s_k^A, .., s_n^A$ una traza que fue evaluada previamente en la ejecución abstracta y $t' = s_0^A, s_1'^A, s_2'^A, s_3'^A, .., s_k^A$ la traza actualmente en consideración por el procedimiento, tales que t' difiere de t a partir de algún estado anterior a s_k^A. Como todos los sucesores abstractos ($suc(s_k^A)$) de s_k^A fueron calculados cuando el algoritmo trató la traza t (porque el recorrido es en profundidad), no es necesario computar nuevamente $suc(s_k^A)$ durante el tratamiento de t'. Esto es, no puede suceder que aparezcan estados abstractos "nuevos" que

lleven a una violación de la propiedad a partir de s_k^A en t'. Para entender mejor esta idea veamos un ejemplo simple. Consideremos un modelo de conjuntos, con sus respectivas acciones *add* y *del*, que agregan y eliminan elementos respectivamente, y el siguiente programa anotado:

```
assertCorrectness programExample [s: set Elem]  {
  pre = { empty[s] }
  program = { ((add[s];del[s]) + skip)* }
  post = { empty[s'] }                          }
```

Los predicados `empty`, `one` y `two`, que indican que un conjunto tiene cero, uno o dos elementos, respetivamente, son los predicados de abstracción a utilizar. El número de trazas posibles es 2^n, donde n es la cantidad de iteraciones a considerar. Intuitivamente, podemos ver que ejecutar `add[s];del[s]` en abstracto tiene el mismo efecto que la acción `skip`. Por lo tanto, no importa que rama se elija en una ejecución de la clausura, ya que indefectiblemente se llegará al estado inicial abstracto. Este hecho se puede observar claramente en la Figura 1.

Para la implementación de esta técnica utilizamos un grafo que almacena los estados computados durante la ejecución abstracta, junto con la posición de programa (program counter) a la que corresponden. Previo a la ejecución abstracta etiquetamos cada acción del programa *DynAlloy* con su posición correspondiente. Por razones de espacio no discutiremos este procedimiento aquí (para más detalles, el lector puede consultar [9]). Así, el algoritmo nuevo es una modificación del anterior que utiliza un grafo como estructura de datos, e implementa la técnica presentada arriba. El cambio más importante entre ambos algoritmos se produce luego de ejecutar una acción en el recorrido en profundidad del programa. En este punto, el algoritmo nuevo realiza una búsqueda en el grafo por un estado abstracto idéntico al obtenido (nótese que para que esto suceda los program counters deben ser iguales). Si la búsqueda da un resultado positivo, podemos aplicar la técnica de detección de inducción y evitar calcular los sucesores abstractos del estado corriente. En otro caso, el estado nuevo se agrega al grafo, y la ejecución continúa normalmente, de acuerdo al algoritmo discutido en la sección anterior.

Como veremos más adelante, en los resultados experimentales, detección de inducción en conjunción con los predicados adecuados permite que a partir

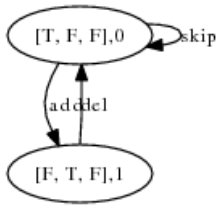

Fig. 1: Grafo resultante de la ejecución de `programExample`.

de cierta cantidad de iteraciones no sea necesario computar estados abstractos nuevos. Esto implica que el tiempo de construcción del modelo abstracto no crece al incrementar la cota en la cantidad de iteraciones.

Reutilización de cálculos previos. Otra modificación importante que permite mejorar la eficiencia de la herramienta es la optimización del módulo que almacena cálculos previos. La nueva implementación se basa en la representación de un retículo por medio de tablas hash, junto con algoritmos eficientes para minimizar los tiempos de búsqueda e inserción en la estructura. Las inserciones se realizan en el reticulado según la cantidad de variables definidas, esto es, un monomio m será insertado en un nivel inferior a otro m' si en m' aparecen más variables que en m. Por limitaciones de espacio no podemos dar detalles de este procedimiento aquí.

5 Casos de estudio

En esta sección presentaremos algunos resultados experimentales que muestran la eficiencia de las técnicas desarrolladas. Los casos de estudio son programas *DynAlloy* que se obtienen (mediante una traducción automática) de programas Java que manipulan listas simplemente encadenadas. En todos los casos se utilizaron como conjunto de predicados inicial la precondición, postcondición y los tests del programa. Se tuvieron en cuenta los tiempos que *Alloy Analyzer* (M1) requirió para verificar la propiedad en el modelo concreto, comparándolos con los tiempos obtenidos con nuestra herramienta anterior *(abstracción bajo demanda)* (M2) y nuestro algoritmo actual (M3). Los experimentos se realizaron en una PC con procesador Intel Core 2 Duo de 2Ghz, con 2Gb de memoria RAM y Sistema Operativo GNU/Linux 2.6. La versión de Alloy Analyzer utilizada es la 4.1.8. Las ejecuciones que tardaron más de una hora fueron interrumpidas. Los resultados se muestran en la tabla 2.

(C1) La eliminación preserva la aciclicidad de las listas. Como primer caso de estudio consideramos un programa que dada una lista acíclica elimina los nodos que contienen valores en un conjunto dado como parámetro. La propiedad que se desea verificar sobre este programa es que la lista resultante preserva la aciclicidad. La propiedad se escribió en forma de invariante de ciclo, y el refinador de abstracciones tuvo que descubrir dos nuevos predicados para poder verificarla.

(C2) Asignación de valores a una lista según un flag. El segundo caso de estudio corresponde a la especificación de un programa *DynAlloy* que, partiendo de una lista vacía y un flag booleano, crea una lista (realizando sucesivas inserciones a la cola) asignando valores a los nodos de la misma dependiendo del valor del flag. Si el flag es true, se insertan a la lista una cierta cantidad de nodos con el valor uno, en otro caso se insertan nodos con el valor dos. Al finalizar, se agrega al final de la lista un nodo con el valor tres. La propiedad a verificar indica que la lista se construyó correctamente, es decir, si el flag es true entonces la lista contiene

lurs	M1	M2	M3
15	17m42s	57s	34s
16	28m 31s	1m 19s	34s
17	OoM	1m 52s	34s
20	OoM	10m 33s	34s
23	OoM	20m 56s	34s
24	OoM	TO	34s

(a) C1

lurs	M1	M2	M3
10	1m 20s	18s	17s
15	15m 30s	18s	17s
25	TO	20s	17s
50	TO	21s	17s
100	TO	24s	17s
1000	TO	OF	17s

(b) C2

lurs	M1	M2	M3
10	2m 10s	1m 34s	1m 15s
12	45m 20s	2m 32s	1m 15s
13	TO	4m 42s	1m 15s
14	TO	11m 53s	1m 15s
15	TO	32m 50s	1m 15s
16	TO	TO	1m 15s

(c) C3

Fig. 2: Comparación de los tiempos entre los diferentes métodos. *lurs*: loop unrools, TO: Timeout, OoM: Out of Memory, OF: Overflow.

varios unos seguidos de un tres, si es false los elementos son todos dos y al final un tres. Para este ejemplo la cantidad de predicados a descubrir fue demasiado grande (dependiente de la cantidad de iteraciones a ejecutar el ciclo), por lo que el análisis para cotas grandes siempre superaba el tiempo máximo permitido. Por lo tanto se realizó un análisis del modelo y se observó la necesidad de introducir un invariante de ciclo que asegura que todos los elementos insertados son o bien uno o bien dos de acuerdo al valor del flag. Dado este invariante, la herramienta fue capaz de aplicar la técnica detección de inducción, con ganancias significativas en el tiempo de ejecución del modelo. Estos tiempos son los que se muestran en la tabla 2.

(C3) División de una lista en dos. El último caso también tiene que ver con un problemas bastante sencillo sobre listas. Se divide una lista dada en dos nuevas según el valor de sus elementos: si el elemento es un uno se lo asigna a una lista, y si es un dos a la otra. La propiedad a verificar indica que la primera lista contiene todos uno y la segunda todos dos.

6 Discusión

Nuestra experiencia con *Abstracción por Predicados* aplicada a especificaciones *DynAlloy* muestra que en el caso en que no se disponga de invariantes de ciclos adecuados para utilizar como predicados de abstracción la técnica de detección de inducción no es aplicable, y todas las trazas abstractas (de longitud acotada) son potenciales contraejemplos espurios (véase C2 en la sección 5). Utilizando las técnicas actuales de refinamiento de abstracciones, en el peor caso sólo se elimina una traza abstracta a la vez, introduciendo un predicado por cada una de ellas. Como la cantidad de trazas abstractas depende exponencialmente de la cota dada por el usuario para las iteraciones del programa, la cantidad de predicados de abstracción a introducir también es exponencial en dicha cota. Dado que la complejidad del modelo abstracto crece exponencialmente con la cantidad de predicados, en estos casos la aplicación de abstracción puede llegar a ser más ineficiente que análizar el modelo en concreto, sobre todo para cotas

relativamente grandes. Por otro lado, se observó que las ganancias al aplicar abstracción, y sobre todo detección de inducción, son muy importantes en casos en los que la propiedad a verificar y el modelo tienen una forma particular, o cuando el usuario provee los predicados correctos obtenidos mediante la inspección minuciosa del código. La aplicación de detección de inducción en estos casos (cuando la propiedad a verificar es válida) permite que el tiempo requerido para calcular la abstracción y realizar la verificación se mantenga constante a partir de una cierta cota (generalmente pequeña) para las iteraciones.

Es importante destacar que el trabajo reportado en [5] fue el primer intento en aplicar Abstracción por Predicados a programas que operan sobre estructuras de datos simples y propiedades especificadas en una lógica de primer orden relacional. En cambio, hay una gran cantidad de investigadores que han utilizado Abstracción por Predicados exitosamente verificando hardware, protocolos de comunicación y drivers de dispositivos. Por otra parte, existen diversas técnicas que permiten tratar con programas que manipulan estructuras de datos, entre las cuales destacamos Shape Analysis y Separation Logic. Utilizarlas en combinación con Abstracción por Predicados es una posibilidad que pensamos explorar en el futuro. Otro hilo de investigación interesante que surge del trabajo actual es el desarrollo de técnicas de descubrimiento de predicados más adecuadas a nuestro problema particular, que permitan, por ejemplo, eliminar varios contraejemplos espurios a la vez.

References

1. R. Ariño, R. Degiovanni, R. Fervari, P. Ponzio and N. Aguirre. *Towards Scaling Up DynAlloy Analysis using Predicate Abstraction*, en XV Congreso Argentino de Ciencias de la Computación (CACIC 2009). Universidad Nacional de Jujuy. Argentina. Octubre de 2009.
2. P. Cousot, R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.*, in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977. ACM Press.
3. S. Das and D. Dill, *Counterexample Based Predicate Discovery in Predicate Abstraction*, in International Conference on Formal Methods in Computer Aided Design FMCAD 2002, USA, LNCS, Springer, 2002.
4. M. Frias, J.P. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: upgrading alloy with actions*, in International Conference on Software Engineering ICSE 2005, St. Louis, USA. ACM Press, 2005.
5. N. Aguirre, M. Frias, P. Ponzio, B. Cardiff, J.P. Galeotti and G. Regis, *Towards Abstraction for DynAlloy Specifications*, in Proceedings of the Tenth International Conference on Formal Engineering Methods (ICFEM), LNCS, Springer, 2008.
6. S. Graf and H. Saïdi, *Construction of abstract state graphs with PVS*, in Computer Aided Verification CAV 1999, Haifa, Israel, LNCS 1254, Springer, 1997.
7. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
8. E. Clarke, A. Biere, R. Raimi, Y. Zhu *Bounded Model Checking Using Satisfiability Solving.*
9. E. Clarke, O. Grumberg and D. Peled. *Model Checking.* The MIT Press, 2000.