Hacia un entorno integrado para la verificación de contratos utilizando SAT solvers

Pablo Gabriel Bendersky
pbendersky@gmail.com
L.U.: 828/98

Directores:
Juan Pablo Galeotti
Diego Garverbetsky

Tesis de Licenciatura en Ciencias de la Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Febrero de 2010

Resumen

TACO es una herramienta de análisis automático de programas. En este trabajo se presentan modificaciones a TACO para mejorar su escalabilidad y usabilidad, y proveer una mejor herramienta de análisis.

Para realizar verificación de programas con ciclos, TACO utiliza *loop unrolling* como única técnica de resolución de los mismos. En este trabajo presentamos dos formas de utilizar invariantes de ciclo como alternativa al *loop unrolling* para mejorar la escalabilidad.
Para comprobar el impacto en la escalabilidad se realizaron una serie de experimentos, analizando diferentes programas con los diferentes tipos de análisis. A partir de los resultados experimentales pudimos comprobar las mejoras de performance al utilizar los nuevos tipos de análisis frente al análisis utilizando *loop unrolling*.

Finalmente, se presenta una herramienta de visualización de los contraejemplos encontrados por TACO, a fin de facilitar a los usuarios el análisis de los mismos para encontrar *bugs* en los programas que originaron dichos contraejemplos.

Índice general

Capítulo 1

Introducción

1.1. Análisis automático de programas

El análisis automático de programas está cada vez más presente en las actividades ligadas al desarrollo de software. Cada vez más, herramientas de automáticas se utilizan para complementar las estrategias tradicionales de QA [1, 2].

Dentro del análisis automático de programas, una de las ramas se dedica a la verificación de contratos. En la metodología *Design by Contract*[1] [3], la pre y postcondición de un método define un contrato entre dicho método y sus clientes.
Los clientes (llamador) deben asegurar la precondición del método, y pueden asumir la postcondición. De la misma forma, el método puede asumir la pre condición y debe asegurar la post condición.
La idea para realizar la verificación automática de contratos consiste en transformar el código en una fórmula, y luego probar (mediante herramientas automáticas) que dicha fórmula es correcta con respecto a su especificación. En general, esto significa que la fórmula que describe el programa y su especificación o contrato son consistentes.
Esta fórmula es llamada *verification condition* (*VC* de aquí en adelante).

1.2. TACO

TACO, o *Translation of Annotated Code* es una herramienta para analizar código Java anotado con JML.
TACO traduce programas Java + JML a una especificación Alloy, la cual

[1] Diseño por contratos

es verificable por la herramienta Alloy Analyzer. En la figura 1.1 (página 6) podemos ver los distintos lenguajes intermedios utilizados por TACO.

En las secciones siguientes veremos una introducción a estos lenguajes, que nos ayudará a comprender cómo se llega del programa Java + JML a la especificación Alloy a verificar.

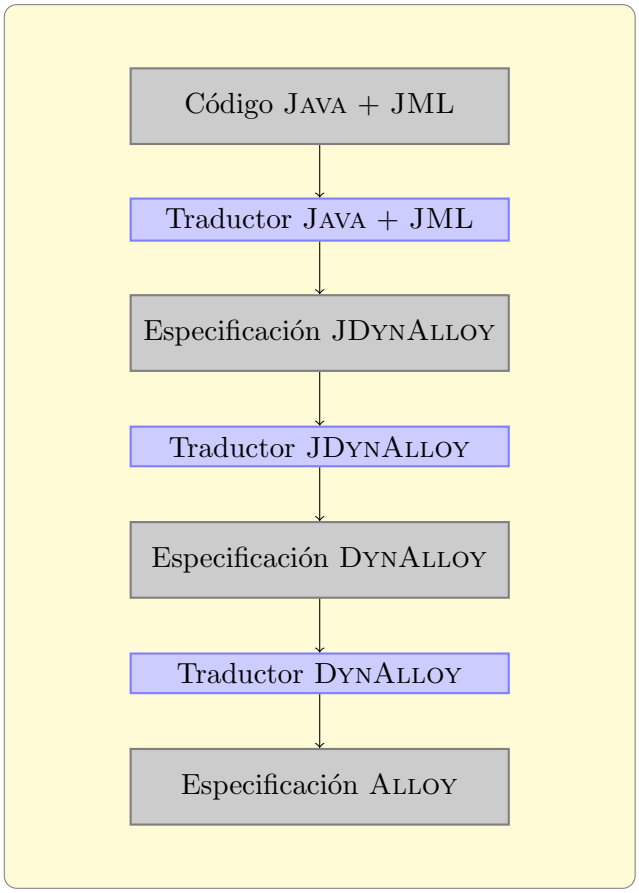


Figura 1.1: Lenguajes y traductores utilizados por TACO

1.3. Alloy

1.3.1. El lenguaje

Alloy es un lenguaje de especificación formal que permite describir propiedades estructurales [4].

Alloy provee una sintáxis declarativa, basada en conjuntos y fórmulas expresadas sobre los mismos, que permite definir los diferentes conjuntos presentes en la especificación, así como una forma de expresar restricciones entre los elementos y relaciones definidas. Su sintaxis provee las construcciones comúnmente encontradas en lenguajes orientados a objetos.

Una vez expresadas las restricciones, pueden escribirse también predicados y verificar que los mismos sean correctos. Esta verificación consiste en buscar algún *contraejemplo* que satisfaga las restricciones pero haga falso el predicado a evaluar.

El lenguaje sólo permite expresar relaciones estructurales en la especificación, así como restricciones y operaciones a aplicar sobre el mismo.

Alloy está basado en el lenguaje de especificación Z [5], del cual toma las partes esenciales para modelar objetos.

En el listado 1.1 (página 8) vemos una especificación Alloy (tomado de [6]), el cual define el comportamiento de una lista. Analicemos la especificación, para entender las distintas construcciones que provee el lenguaje.
En la línea 1 se define una *signature*. Una *signature*, en este caso Data, indica la existencia de un conjunto (de átomos) para los datos.
En la línea 3 se define la *signature* que representa a la lista enlazada. Esta *signature* consiste de dos atributos: un dato (val) y una relación con el siguiente elemento (next). El modificador `lone` indica que los atributos "val" y "next" relacionaran la lista con a lo sumo un elemento.

Alloy permite definir *signatures* que son subconjuntos de la *signature* "padre". En este caso, en la línea 8 definimos la *signature* Empty, que representa la lista vacía.

El predicado descripto en la línea 10 es el constructor de esta estructura, e indica como se construye la `List l'` a partir de los parámetros `d` y `l`.

En las líneas 14 y 18 se definen algunos *facts* (axiomas). Los *facts* deben ser válidos en toda instancia válida de la especificación. En este caso se define un axioma que indica que las listas son acíclicas y otro que indica que hay solo una instancia de tipo `Empty`.

Por último, en la linea 22 se define un `assert`, que es la propiedad que queremos verificar con la herramienta. En este caso, se está indicando que en toda lista, moviéndose a través de su relación `next` en algún momento se

```
sig Data { }

sig List {
    val : lone Data,
    next: lone List
}

sig Empty extends List { }

pred Cons(d : Data, l, l' : List) {
    l'.val = d and l'.next = l
}

fact AcyclicLists {
    all l : List | l !in l.(^next)
}

fact OneEmpty {
    one Empty
}

assert ToEmpty {
    all l: List | l != Empty implies Empty in l.^next
}
```

Listado 1.1: Ejemplo de especificación ALLOY

debe llegar a `Empty`, es decir, la lista tiene fin.

1.3.2. Análisis automático de especificaciones

Uno de los objetivos del diseño del lenguaje ALLOY, fue que todas las especificaciones pudieran ser verificadas de manera automática.

Para esto, ALLOY provee una herramienta llamada ALLOY ANALYZER que permite realizar verificaciones sobre las especificaciones.

Existen dos tipos de análisis soportados por ALLOY (como puede verse en [4]):

1. *Simulación* donde se verifica la consistencia de un predicado a través de la búsqueda de una instancia.

2. *Chequeo* donde se intenta encontrar un contraejemplo a un `assert`.

La *simulación* es realizada a través del comando `run`. Este comando verifica la consistencia de un predicado ALLOY, generando instancias de la

especificación que lo satisfagan. En caso de poder generar instancias, es un indicador de que el predicado es consistente.

En el caso del *chequeo*, lo que se busca es verificar si un `assert` es verdadero o no. Para ello, el Alloy Analyzer buscará instancias de la especificación que no satisfagan ese `assert`.

Alloy está diseñado para buscar instancias dentro de un *scope* finito. Es por ello que tanto los comandos `run` como `check` deben ser acompañados del *scope* a utilizar en la búsqueda. Por ejemplo:

```
check ToEmpty for 5 List
```

En este caso, se está especificando que las instancias donde buscar un contraejemplo del `assert` pueden tener hasta cinco elementos de tipo `List`. En caso de no especificarse el *scope*, Alloy utilizará un valor por defecto de tres átomos de cada signatura declarada en la especificación. La necesidad de utilizar un *scope* finito radica en que el verificador Alloy utiliza un SAT Solver para encontrar contraejemplos del `assert`. La fórmula que se presenta al SAT Solver es generada a partir del *scope* de análisis especificado.

1.3.3. Funcionamiento del verificador

El funcionamiento del Alloy Analyzer se encuentra descripto en [7]. La verificación de una fórmula consta de cinco pasos:

1. Se realizan manipulaciones sobre la fórmula relacional original (conversión a Forma Negada Normal y skolemización), y se obtiene una fórmula relacional equivalente.

2. La fórmula se traduce, para el scope elegido, en una fórmula proposicional, de tal forma que la fórmula proposicional tiene un modelo sólo si la fórmula relacional tiene un modelo para el *scope* elegido.

3. Se transforma la fórmula a la Forma Normal Conjuntiva (CNF), el formato habitual usado por los SAT Solvers.

4. Se presenta la fórmula al SAT Solver.

5. Si el SAT Solver encuentra una valuación que haga satisfacible la fórmula proposicional, entonces se reconstruye un contraejemplo de la especificación relacional.

La búsqueda de contraejemplos queda sujeta a encontrar valuaciones de una fórmula SAT utilizando un SAT Solver. Para ello, el SAT Solver arma un árbol de búsqueda con todas las valuaciones posibles de la fórmula SAT,

y prueba el valor de verdad de cada una de ellas. Este problema es NP-Completo en función de la cantidad de variables de la fórmula CNF, por lo cual esta búsqueda es una operación costosa.

1.4. DynAlloy

DynAlloy es una extensión al lenguaje de especificación Alloy que permite incorporar cambios de estado a Alloy [8]. La semántica de DynAlloy está basada en la lógica dinámica [9], haciendo posible expresar *acciones* atómicas (y acciones más complejas a partir de estas) que modifican el estado.

Las acciones atómicas en DynAlloy están definidas a través de su pre y post condición, las cuales están representadas como fórmulas Alloy.

```
1 action Head[l : List, d : Data] {
2     pre { l != Empty }
3     post { d' = l.val }
4 }
5
6 action Tail[l : List] {
7     pre { l != Empty }
8     post { l' = l.next }
9 }
```

Listado 1.2: Ejemplo de especificación DynAlloy

En el listado 1.2 podemos ver como se puede extender la especificación Alloy presentada en el listado 1.1 a través de acciones. En este caso se definen las acciones `Head` y `Tail`, a través de sus pre y post condiciones. Las variables primadas en las post condiciones denotan el valor de las dichas variables *después* de la ejecución de las acciones.

DynAlloy permite además expresar aserciones de manera similar a una tripla de Hoare (introducidas en [10]). Las triplas se expresan de la forma $\{P\}Q\{R\}$, donde P es la pre condición, Q el programa, y R la post condición. En nuestro ejemplo, una tripla podría ser:

$\{l\ !=\ Empty\}$
program Tail[l]
$\{l = Empty\}$

Las especificaciones DynAlloy conservan la característica de Alloy de ser verificables de manera automática. Para ello, la herramienta Traduc-

tor DynAlloy traduce una especificación DynAlloy en una Alloy, la cual puede ser verificada usando el Alloy Analyzer. Esta traducción tiene la característica de que si existe un contraejemplo de la especificación DynAlloy original, existirá también un contraejemplo en la especificación Alloy resultante. Finalmente, los *asserts* expresables en DynAlloy son verficables por la herramienta de la misma manera que lo son los *asserts* de Alloy.

1.4.1. Iteraciones en DynAlloy

Además de las acciones atómicas que vimos anteriormente, DynAlloy permite expresar acciones por composición, como podemos ver en la figura 1.2.

$$
\begin{array}{rcll}
act & ::= & p\{pre(\overline{x})\}\{post(\overline{x})\} & \text{“atomic action”} \\
 & | & formula? & \text{“test”} \\
 & | & act + act & \text{“non-deterministic choice”} \\
 & | & act;act & \text{“sequential composition”} \\
 & | & act^{*} & \text{“iteration”}
\end{array}
$$

Figura 1.2: Gramática para acciones DynAlloy

La traducción de la acción *iteration*, donde se desea obtener una especificación equivalente a ejecutar de manera secuencial cero o más veces la acción provista, debiera ser:

$act = (skip) + (act) + (act; act) + (act; act; act) + ...$

qué es una fórmula no finita, y por lo tanto no expresable en Alloy.

Para resolver esto, y permitir la correcta traducción de la acción *iteration* a una fórmula Alloy finita, el usuario provee un número de iteraciones, y el traductor genera una especificación Alloy utilizando ese número de invocaciones a la acción. De esta forma se logra finitizar las acciones de tipo *iteration* y lograr una especificación Alloy que sea verificable por el Alloy Analyzer.

En la tabla 1.1 (página 12) podemos ver un ejemplo de traducción del `repeat` de DynAlloy a su expresión Alloy correspondiente para tres *loop unrolls*, pasando por una especificación intermedia donde se resuelve el `repeat` como una composición secuencial del cuerpo del ciclo el número de *loop unrolls* provisto.

Esta técnica la denominamos *loop unrolling* por su similitud con la técnica de optimización de compiladores [11].

El uso de *loop unrolling* requiere que el usuario elija el número de unrolls al momento de realizar la traducción, obteniendo una especificación Alloy que sólo puede ser utilizada para buscar trazas de ejecución de la longitud determinada por el número de *loop unrolls* en las iteraciones del programa.

También podemos observar que a medida que la cantidad de *unrolls* seleccionada por el usuario crece, la especificación Alloy se hace más extensa (y por lo tanto la fórmula CNF generada en la verificación también crece).

```
program addThree[x:Int] {
    repeat {
        addOne[x]
    }
}
```

Especificación DynAlloy original

```
program addThree[x:Int] {
    addOne[x];
    addOne[x];
    addOne[x]
}
```

Especificación DynAlloy realizando *loop unrolling*

```
pred addThree[x_0: Int, x_1: Int,
  x_2: Int, x_3: Int] {
  (addOne[x_0,x_1]
    or (
      TruePred[]
      and
      (x_0=x_1)
    )
  ) and (
    addOne[x_1,x_2]
    or (
      TruePred[]
      and
      (x_1=x_2)
    )
  ) and (
    addOne[x_2,x_3]
    or (
      TruePred[]
      and
      (x_2=x_3)
    )
  )
}
```

Especificación Alloy

Tabla 1.1: Traducción de iteraciones DynAlloy

1.5. JML

JML (Java Modelling Language) es un lenguaje de especificación de comportamiento para Java. Dentro de las anotaciones provistas por JML, un subconjunto de ellas puede utilizarse para trabajar con *Design by Contract* [12]. *DBC* es un método para desarrollar programas donde se permite especificar un "contrato" entre una clase o método y sus clientes.

Esto significa que el cliente debe garantizar el cumplimiento de ciertas condiciones al momento de invocar un método, y en respuesta el método invocado garantiza que otras características se cumplan luego de la invocación.

En el listado 1.3 (página 14) podemos ver como se define el contrato de un método, a través de las anotaciones `ensures` y `requires`, en las líneas 1 a 9 del listado.

El lenguaje JML provee un cuantificador universal y uno existencial (`forall` y `exists`) los cuales permiten escribir expresiones más complejas.

JML permite además expresar invariantes de ciclo [13], utilizando la anotación `@loop_invariant`. En el mismo ejemplo (listado 1.3) podemos ver en las líneas 15 a 19 como se anota el invariante de un ciclo utilizando JML.

1.5.1. Utilización de JML en TACO

En TACO, JML es utilizado para expresar los contratos de los métodos, los cuales son traducidos por el Traductor JDynAlloy a expresiones JDynAlloy que expresan la pre y post condición de los métodos a verificar.

```
1  public class LinearSearch extends Object {
2      /*@
3        @ requires
4        @    element >= 0;
5        @ ensures
6        @    \result < a.length;
7        @ ensures
8        @    (\result >= 0 &&
9        @     \result < a.length ==> a[\result] == element);
10       @*/
11     public static int search(int[] a, int element) {
12         int retValue;
13         int i;
14         retValue = -1;
15         i = 0;
16         /*@
17           @ loop_invariant
18           @  i >= 0 && i <= a.length &&
19           @  (\forall int j; j >= 0 && j < i; a[j] != element);
20           @*/
21         while (i < a.length && a[i] != element) {
22             i = i + 1;
23         }
24
25         if (i < a.length) {
26             retValue = i;
27         }
28
29         return retValue;
30     }
31 }
```

Listado 1.3: Pre y post condiciones en JML

1.6. Traducción de programas Java + JML

Para llegar a una especificación DynAlloy a partir de un programa Java anotado con JML se utiliza el Traductor Java y el Traductor JDynAlloy.

La forma de traducir programas Java a DynAlloy fue presentada en [6]. En esta sección repasaremos algunos conceptos utilizados en esta traducción.

1.6.1. Traducción de clases

Por cada clase presente en Java, la especificación DynAlloy resultante tendrá una *signature* correspondiente. Por ejemplo, para cualquier programa Java se crearán al menos las siguientes *signatures* Alloy:

```
1 one sig null {}
2 sig Object {}
3 sig List extends Object {}
4 sig Throwable extends Object {}
5 sig Exception extends Throwable {}
6 sig RuntimeException extends Exception {}
7 sig NullPointerException extends RuntimeException {}
8 sig SystemArray extends Object {}
```

Listado 1.4: Ejemplo de traducción Java a DynAlloy

En la línea 1 vemos la definición de la *signature* para `null`, y en las líneas 2 a 7 la definición de *signatures* para algunas clases estándar de Java. En la línea 8, vemos la definición de SystemArray, que se utiliza para los arreglos de Java.

Las variables de instancia se traducen a relaciones binarias en la especificación DynAlloy:

val : List → one null + Val

next : List → one null + List

Un caso particular son los arreglos de Java, donde también se traduce el contenido de los mismos como una relación binaria entre la *signature* SystemArray y una secuencia con los datos que el arreglo almacena.

Además, la traducción genera algunas acciones DynAlloy atómicas necesarias para representar la creación de nuevos objetos y el uso de las clases estándar.

1.6.2. Traducción de anotaciones JML

Por cada método que contenga anotaciones JML `@requires` o `@ensures`, la traducción generará predicados ALLOY para representarlos. Estos predicados son utilizados como pre y postcondiciones en las *assertions* generadas. Esta traducción también fue presentada en [6].

1.6.3. Soporte de excepciones

Una parte del lenguaje JAVA que se traduce, es su soporte de excepciones. Para lograr esto, el Traductor JAVA agrega a la traducción de cada método un parámetro adicional llamado `throw` el cual será modificado si el método arroja una excepción.
Por ejemplo, la signatura de la traducción del método search presentado en el listado 1.3 (página 14) es la siguiente:

```
1 program LinearSearch::search[
2   var throw:Throwable+null,
3   var return:Int,
4   var list:SystemArray,
5   var element:Int]
```

La variable `throw` se inicializa en `null` al comienzo del método, y en caso de haber una excepción la variable es actualizada.
Finalmente, en la traducción de la postcondición a verificar, se agrega la cláusula:

throw' = null

lo que hace que exhiba un contraejemplo en caso de que el programa arroje una excepción.

1.6.4. Limitaciones

Si bien la traducción de JAVA a DYNALLOY es muy completa, algunos elementos del lenguaje no son traducidos. Algunas de las limitaciones actuales son:

- No se soporta *Reflection*
- El código JAVA debe estar normalizado siguiendo ciertas reglas:
  - No se soporta la construcción `for`, la cual debe reemplazarse por `while`.
  - No se soportan algunos operadores pre y postfijos, como `++`.

- No se permite declarar e inicializar variables en una misma instrucción.

1.7. Aportes de este trabajo

En este trabajo se presentan los siguientes aportes:

- Una alternativa a la verificación usando *loop unrolling* basada en el teorema del invariante. Este tipo de verificación permite trabajar con ciclos sin las limitaciones del *loop unrolling*, utilizando invariantes de ciclo. Además, permite verificar que los invariantes de ciclo satisfagan las hipótesis del teorema del invariante (capítulo 2, sección 2.4).
- Una segunda alternativa a la verificación usando *loop unrolling*, atomización de ciclos (usando invariante). Este tipo de verificación permite trabajar con invariantes de ciclo asumiendo que los mismos satisfacen las hipótesis del teorema del invariante. Este tipo de verificación presenta ventajas en cuanto a *performance* y escalabilidad (capítulo 2, sección 2.5).
- Un análisis empírico donde se comparan diferentes métricas al realizar verificaciones usando las tres técnicas de verificación (*loop unrolling*, teorema del invariante y atomización de ciclos) (capítulo 2, sección 2.8).
- Una herramienta de visualización de contraejemplos, la cual nos permite ver los contraejemplos encontrados por la herramienta sobre la especificación DynAlloy(capítulo 3).
- Un tutorial de uso, donde se muestra con un ejemplo el uso de las nuevas verificaciones y del visualizador para encontrar errores en un programa Java(capítulo 4).

1.8. Trabajo Relacionado

A continuación mencionamos otras herramientas que utilizan métodos similares a TACO para realizar análisis automático de programas.

- En [14] se presenta AAL, un lenguaje de anotaciones para Java basado en Alloy. Este lenguaje es similar a JML, y en el trabajo se presenta una traducción a Alloy similar a la que realiza TACO.

- ESC/Java2 [15] realiza chequeos estáticos sobre programas JAVA anotados con JML.
- Forge [16] realiza verificación de código JAVA utilizando ALLOY como lenguaje para expresar la *VC*, de manera similar a TACO. A diferencia de TACO el lenguaje intermedio que utiliza es FIR (Forge Intermediate Representation), el cual puede verse como una versión imperativa de ALLOY. DYNALLOY, por el contrario, no es imperativo, sino declarativo, al igual que ALLOY.
- Spec# [17] es una extensión al lenguaje C# similar a JML. Las especificaciones Spec# pueden ser verificadas utilizando Boogie[18].

Capítulo 2

Verificación de programas con ciclos

2.1. Motivación

La verificación de programas con ciclos es en general compleja. Entre otras cosas porque no se puede saber a priori cuantas veces será ejercitado el cuerpo del ciclo sino hasta que el programa es efectivamente ejecutado.

En TACO, los programas escritos en JAVA + JML son traducidos a diferentes lenguajes intermedios hasta llegar a una especificación ALLOY (ver figura 1.1 en la página 6). Dado que ALLOY es un lenguaje de modelado usado para verificar especificaciones, y no de programación, el mismo no permite expresar ciclos.

Como los ciclos son una parte fundamental de cualquier programa JAVA, el Traductor JDYNALLOY y el Traductor DYNALLOY utilizan el mecanismo descripto en la sección 1.4.1 (página 11) para convertir esos ciclos en especificaciones ALLOY verificables por la herramienta.

Muchas herramientas de análisis utilizan los invariantes de ciclo (ya sean provistos por el usuario o inferidos automáticamente) para verificar la corrección del programa.

El objetivo de esta parte del trabajo es extender TACO de modo que, usando los invariantes de ciclo presentes en la especificación JML, genere *VC*s más compactas y verificables por el ALLOY ANALYZER (el backend de TACO).

2.2. Limitaciones del *loop unrolling*

TACO expresa la *verification condition* utilizando el lenguaje ALLOY. La resolución de ciclos para esta traducción consiste en realizar *loop unrolling* [8].

Si bien el *loop unrolling* se realiza al traducir de DYNALLOY a ALLOY podemos analizarlo con código JAVA anotado con JML para ejemplificarlo.

Código Java + JML - Ciclo	**Código Java + JML - Loop Unroll**

`i = 0;`
`n = 3;`
`while (i < n) {`
`  doSomething(i);`
`  i++;`
`}`
`// @assert i > 0;`
	`i = 0;`
	`n = 3;`
	`if (i < n) {`
	`  doSomething(i);`
	`  i++;`
	`  if (i < n) {`
	`  doSomething(i);`
	`  i++;`
	`  if (i < n) {`
	`  doSomething(i);`
	`  i++;`
	`  }`
	`  }`
	`}`
	`// @assume !(i < n);`
	`// @assert i > 0;`

Tabla 2.1: Ejemplo de loop unrolling en JAVA.

Como podemos ver en la tabla 2.1 (página 20), el *loop unroll* consiste en copiar el cuerpo del ciclo una cantidad fija de veces, guardado con un `if`, para garantizar que no se ejecute el código si la guarda del ciclo deja de ser verdadera, y un `assume` al final para garantizar que la guarda del ciclo no sea verdadera a la salida.

Si bien la técnica está inspirada en una optimización utilizada en compiladores [11], en el caso de DYNALLOY el *loop unrolling* no se utiliza como una técnica de optimización, sino para acotar los posibles trazas de ejecución que se verificarán.

Esta solución permite al usuario seleccionar la longitud de las trazas a analizar, pero acarrea algunas limitaciones:

- Se analiza sólo un subconjunto de las posibles trazas de ejecución del

programa, determinado por el número de *loop unrolls* elegido por el usuario.

- El número de iteraciones debe ser elegido *antes* de la verificación, como parte del proceso de traducción.
- Al incrementar el *scope* de análisis, muchas veces es necesario cambiar el número de *loop unrolls* acompañando este incremento. Por ejemplo, si el programa recorre un arreglo de JAVA, al subir el *scope* debemos subir el número de *loop unrolls* para que las trazas de ejecución a verificar reflejen el cambio del *scope*.
- El tiempo de verificación depende del *scope* elegido y de la longitud de las trazas de ejecución. A mayor *scope* y/o mayor longitud de las trazas, mayor es el tiempo requerido para la verificación.
- El tamaño de la especificación ALLOY generada usando unrolls crece exponencialmente en el caso de ciclos anidados.

De estas limitaciones, el hecho de analizar un subconjunto de las posibles trazas de ejecución puede hacer que no se encuentren *bugs* presentes en el programa por estar realizando el análisis con un número bajo de *loop unrolls*. Supongamos el programa presentado en el listado 2.1 (página 22), adaptado de un ejemplo de JMLForge [19].

Al verificar este programa con un *scope* de secuencias de longitud 5 y 1 *loop unroll* no se encuentran errores. Sin embargo, al utilizar 2 *loop unrolls* la herramienta encuentra un error en el programa. La causa del error puede verse en la condición del `for` de la línea 36. Cuando el elemento buscado está pasando la mitad de la lista, el elemento retornado es el anterior al elemento buscado, y de ahí el contraejemplo que la herramienta encuentra.

La alternativa que se realizó en este trabajo de tesis consiste en extender el soporte de JML presente en TACO, de manera que pueda aprovechar los invariantes de ciclo -en caso de que sean provistos- y hacer que los mismos sean parte de las sucesivas traducciones. El objetivo final es poder obtener una especificación ALLOY que utilice los invariantes de ciclo, y así lograr una especificación más compacta e independiente de la cantidad de *unrolls* seleccionada por el usuario.

```
1  public class LinkList {
2      //@ model instance non_null JMLObjectSequence seq;
3
4      Value head, tail;
5      int size;
6      /*@ invariant (head == null && tail == null && size == 0) ||
7        @   (head.prev == null && tail.next == null &&
8        @    \reach(head, Value, next).int_size() == size &&
9        @    \reach(head, Value, next).has(tail) &&
10       @    (\forall Value v; \reach(head, Value, next).has(v) ;
11       @             v.next != null ==> v.next.prev == v));
12       @*/
13     /*@ represents seq \such_that
14       @  (size == seq.int_size()) &&
15       @  (head == null ==> seq.isEmpty()) &&
16       @  (head != null ==> (head == seq.get(0) &&
17       @            tail == seq.get(size - 1))) &&
18       @  (\forall int i ; i >= 0 && i < size - 1;
19       @        ((Value)seq.get(i)).next == seq.get(i + 1));
20       @*/
21     /*@ requires index >= 0 && index < seq.int_size();
22       @ ensures \result == seq.get(index);
23       @*/
24      /*@ pure @*/ Value get(int index) {
25         // optimize for common cases
26         if (index == 0) return head;
27         if (index == size - 1) return tail;
28         Value value;
29         if (index <= (size >> 1)) {   // if index is in front half of list,
30             value = head;             // search from the beginning
31             for (int i = 0; i < index; i++) {
32                 value = value.next;
33             }
34         } else {                      // if index is in back half of list,
35             value = tail;             // search from the end
36             for (int i = size; i > index; i--) {
37                 value = value.prev;
38             }
39         }
40         return value;
41     }
42
43     /*@ nullable_by_default @*/
44     public static class Value {
45         Value next, prev;
46     }
47 }
```

Listado 2.1: Programa con un error no detectable para loop unroll bajo

2.3. Teorema del invariante

En las secciones siguientes utilizaremos invariantes de ciclo para generar la *VC*. A continuación, se muestra el teorema del invariante [10], el cual es utilizado como fundamento teórico para lo que sigue a continuación.

Tomemos el siguiente programa:

```
while (B) {
    cuerpo
}
```

un ciclo con guarda B, precondición P_c y postcondición Q_c.
Sea I un predicado booleano. Si valen:

1. $P_c \rightarrow I$
2. $(I \wedge \neg B) \rightarrow Q_c$
3. la ejecución del cuerpo del ciclo preserva I

entonces para cualquier valor de las variables del programa que haga verdadera P_c, si el ciclo termina, será verdadera la postcondición Q_c y podemos decir que el ciclo es correcto.

2.4. Verificación usando el teorema del invariante

2.4.1. Introducción

La generación de la *VC* utilizando invariantes de ciclo en lugar de *loop unroll* requiere que el programador provea de los invariantes de ciclo al código Java. Al ser invariantes no inferidos por alguna herramienta, sino escritos por el programador, siempre existe la posibilidad de que el invariante de ciclo provisto no cumpla las hipótesis del teorema del invariante, generando nuevos errores durante la verificación.

Una de las formas de mitigar la aparición de nuevos errores, es proveer al programador una forma de verificar los invariantes de ciclo que escribe. Este mecanismo está presente en otras herramientas similares, como por ejemplo Boogie [18].

En esta parte del trabajo de tesis, veremos como se extiende TACO para proveerlo de una generación de *VC* similar a la encontrada en Boogie.

2.4.2. Traducción utilizando el teorema del invariante

Para proveer a DynAlloy de invariantes traducibles a Alloy se extendió el soporte de JML presente en el Traductor Java.
JML provee sintáxis para anotar los ciclos en Java. Además, los cuantificadores `forall` y `exists` de JML tienen una igual semántica a la de los operadores `all` y `some` de Alloy.

Dado que los invariantes de ciclo están expresados en JML, la traducción de los mismos es realizada por el Traductor Java.

En la tabla 2.2 podemos ver la traducción de código Java anotado con JML a JDynAlloy utilizando los invariantes, y verificando que se satisfaga el teorema del invariante.

Código Java	**Especificación JDynAlloy**

`/*@ loop_invariant INV @*/`
`while (B) {`
`  C(M)`
`}`
	`assert INV`
	`havoc M`
	`assume INV`
	`if B`
	`  C`
	`  assert INV`
	`endif`
	`assume ¬B`

Tabla 2.2: Traducción de Código Java para verificar invariantes.

A continuación, veremos que probando la corrección de la especificación JDynAlloy resultante, se prueban las hipótesis del teorema del invariante.

Si recordamos el teorema del invariante, las condiciones que este impone para garantizar que el ciclo sea correcto son:

1. $P_c \rightarrow I$
2. $(I \wedge \neg B) \rightarrow Q_c$
3. $\{I \wedge B\}\ C\ \{I\}$

En el listado 2.2 (página 25) podemos ver cómo las hipótesis del teorema del invariante son representadas en la especificación JDynAlloy.

```
1 assert INV    // 1. El invariante debe valer al comienzo del ciclo.
2 havoc M
3 assume INV    // 3. El cuerpo del ciclo debe preservar el invariante.
4 if B          // (lineas 3 a 7)
5    C          //
6    assert INV //
7 endif
8 assume ¬B    // 2. Debe valer el invariante pero no la guarda.
```

Listado 2.2: Verificación del teorema del invariante

La especificación Alloy resultante de la traducción utilizando *loop unrolling* varía según el número de *unrolls* elegido por el usuario. Esto se debe a que al traducir los ciclos, el Traductor DynAlloy generará copias del cuerpo del ciclo según el número de *loop unrolls* utilizado. Por este motivo, una especificación Alloy traducida utilizando *loop unrolling* solo podrá ser utilizado para generar trazas de ejecución que ejercitan cada ciclo a lo sumo la cantidad de unrolls seleccionada por el usuario.

Por el contrario, como el invariante es una fórmula lógica que involucra algunas de las variables y parámetros del método traducido, la misma será traducida a una fórmula Alloy utilizando esas variables, y no necesitará generar variables para representar instantes intermedios de la ejecución. Esto hace que una misma especificación Alloy pueda luego utilizarse para buscar trazas de ejecución que no están restringidas por una máxima cantidad de ejercicios del ciclo, sino que dependerán exclusivamente del *scope* de análisis.

2.4.3. Principales ventajas y desventajas

Este tipo de análisis presenta algunas ventajas y desventajas respecto al análisis utilizando *loop unrolling*. A continuación mencionamos algunas de las ventajas:

- El análisis no depende del número de *loop unrolls* sino sólo del *scope*.
- Las especificaciones Alloy resultantes son mucho más compactas, dado que no crecen en función del número de *unrolls*, haciendo que la herramienta sea más escalable.
- La mejora en la escalabilidad hace que se aborten menos verificaciones, ampliando la cantidad de programas que se pueden verificar.
- El análisis verifica las hipótesis del teorema del invariante.

La principal desventaja que se encuentra al utilizar este tipo de análisis, es que el invariante de ciclo debe ser provisto por el usuario. En el futuro, podrían integrarse herramientas de inferencia de invariantes como Daikon [20] o Houdini [21].

2.5. Atomización de ciclos (usando el invariante)

2.5.1. Atomización en DynAlloy

En el trabajo [22] se presenta el concepto de *program atomization*[1] en el contexto del análisis de una especificación DynAlloy. Dicho *paper* presenta una forma de *atomizar* la verificación de una especificación DynAlloy aprovechando el resultado de una análisis previo de una de las partes que componen esta acción.

En términos generales (extraído de [22]), si se desea verificar la tripla:

$$\begin{gathered}\{\alpha\}\\ P\\ \{\beta\}\end{gathered}$$

donde el programa P tiene un subprograma (o subtérmino P_s). Si ya se realizó la verificación de:

$$\begin{gathered}\{\alpha_s\}\\ P_s\\ \{\beta_s\}\end{gathered}$$

y no se encontraron contraejemplos para un número k de *loop unrolls*.

Dadas estas condiciones, podemos escribir una nueva acción atómica descripta por:

$$\begin{gathered}\{\alpha_s\}\\ a_{Ps}\\ \{\beta_s\}\end{gathered}$$

y reemplazar en el programa original P todas las ocurrencias de P_s por a_{Ps}, obteniendo una programa alternativo, P_v. Si analizamos P_v para el mismo número de *loop unrolls* k y no encontramos contraejemplos, estará garantizado (en [22] puede verse el teorema y su demostración) que tampoco se encontrarán contraejemplos en P para el mismo número de *loop unrolls* k.

[1] Atomización de programas

Inspirados en la idea de atomización de acciones, presentaremos a continuación una posible *atomización* de un ciclo Java anotado con JML. Es importante notar que a diferencia de la atomización presentada en [22], la atomización que presentaremos no depende de una verificación realizada previamente, sino que se basa en un invariante de ciclo provisto por el usuario para usarlo como posible *atomización* del ciclo.

2.5.2. Atomización de ciclos Java

En la tabla 2.3 podemos ver este tipo de traducción de un ciclo Java anotado con JML a código JDynAlloy.

Código Java+ JML	**Especificación JDynAlloy**

`/*@ loop_invariant INV @*/`
`while (B) {`
`  C(M)`
`}`
	`havoc M`
	`assume INV ∧ ¬B`

Tabla 2.3: Traducción de Java con invariante a JDynAlloy

Este tipo de análisis se realiza bajo la suposición de que el el invariante de ciclo cumple las hipótesis del teorema del invariante. Esta es una suposición muy fuerte, y es aquí donde dependemos de que el usuario haya provisto un "buen" invariante de ciclo, es decir, que satisfaga las hipótesis del teorema del invariante.

En caso de que no se tenga la certeza de que el invariante de ciclo es válido, puede usarse el análisis presentado en la sección anterior, donde se verifica la corrección del mismo.

Suponiendo que se cumplen las hipótesis del teorema del invariante, podemos asumir que `INV` es verdadero a la entrada y a la salida del ciclo, donde además la guarda del ciclo (`B`) pasa a ser falsa. Además, sabemos que el cuerpo del ciclo, representado por `C(M)` en el código Java (donde `M` es el conjunto de variables modificadas por el cuerpo del ciclo `C`), modificará a lo sumo sus variables durante la ejecución.

2.5.3. Principales ventajas y desventajas

Respecto al análisis presentado en la sección anterior, este tipo de análisis utilizando atomización de ciclos usando invariantes, tiene la ventaja de

que la *VC* resultante es más sencilla que las anteriores, ya que se omite la traducción del cuerpo del ciclo.

La principal desventaja de este tipo de análisis es que es un análisis *unsafe*. Entendemos por *unsafe* que los análisis realizados pueden no encontrar contraejemplos en un programa incorrecto. Por ejemplo, si la ejecución del cuerpo del ciclo no preserva el invariante de ciclo, esto no será detectado por este tipo de análisis.

Otra desventaja es que pueden aparecer *contraejemplos espurios*, es decir contraejemplos sobre programas que no tienen errores. Estos errores pueden provenir de invariantes de ciclo subespecificados, por ejemplo un invariante de ciclo siempre verdadero ($INV = true$), el cual satisface las hipótesis del teorema del invariante, pero puede hacer que falle la postcondición, encontrando así un contraejemplo espurio, dado que es producto de un invariante mal especificado y no de un error en el programa original.

Estas desventajas son muy importante en este tipo de análisis, por lo que antes de usarse debiera realizarse al menos un análisis usando la verificación usando el teorema del invariante presentada anteriormente para mitigar los riesgos de trabajar con una atomización que no represente apropiadamente al ciclo original.

2.6. Implementación de assume, havoc y assert

2.6.1. Introducción

En la traducción a JDynAlloy se introducen tres nuevas sentencias: `assume`, `havoc` y `assert` que son utilizadas para las verificaciones antes presentadas.

Como parte de este trabajo de tesis, se extendió el lenguaje JDynAlloy para agregar estas tres sentencias, y traducirlas en todos los niveles hasta obtener una especificación Alloy.

A continuación analizaremos como cada uno de estas operaciones se traduce a DynAlloy, poniendo particular énfasis en la traducción de `assert`, ya que -como veremos- su traducción a Alloy no es trivial.

2.6.2. Assume

La traducción del `assume` es inmediata. El lenguaje que DynAlloy permite expresar predicados lógicos, y las expresiones que podemos encontrar en un `assume` son traducibles a un predicado. Además, la semántica del `assume` es igual a la del *test action* provisto por DynAlloy. En la tabla 2.4

(página 29) podemos ver la traducción de un `assume` expresado en JAVA a DYNALLOY y luego a ALLOY.

```
public static void assume(int i) {
        //@ assume i == 0;
        i = 20;
}
```

Código JAVA

```
pred AssumeAssertCondition0[i:univ]{
   equ[i,0]
}

program AssumeAssert_assume_0[i:Int] {
   assume AssumeAssertCondition0[i];
   i:=20
}
```

Especificación DYNALLOY

```
pred AssumeAssertCondition0[i:univ]{
   equ[i,0]
}

pred AssumeAssert_assume_0[i_0: Int,i_1: Int] {
  AssumeAssertCondition0[i_0]
  and
  (i_1=20)
}
```

Especificación ALLOY

Tabla 2.4: Traducción de assume

2.6.3. Havoc

La sentencia `havoc` obtiene un nuevo valor para una variable. En DYNALLOY, esto es equivalente a tener una acción que no impone precondiciones ni post-condiciones, como podemos ver en el listado 2.3 (página 30).

Esta acción, una vez invocada, genera un nuevo *instante* de la variable v llamado v', sin ninguna restricción sobre su valor.

```
1 pred havocVarPost[u:univ]{
2 }
3 action havocVariable[v:univ] {
4    pre { }
5    post { havocVarPost[v'] }
6 }
```

Listado 2.3: Implementación de sentencia havoc en DynAlloy

Si bien a nivel JDynAlloy la sentencia que se utiliza es siempre la misma, a nivel DynAlloy se deben hacer diferencias según si se trata del `havoc` de una variable, un campo o el contenido de un arreglo. Para permitir el `havoc` de todas ellas, se proveen diferentes *acciones*, y el traductor se ocupa de utilizar la que corresponda de acuerdo al tipo de la variable.

Detección de variables

En los dos tipos de verificaciones presentados, se realiza un `havoc M`, donde `M` es el conjunto de variables modificadas por el cuerpo del ciclo.
Para generar los `havoc` en DynAlloy es necesario encontrar ese conjunto de variables `M` y traducirlo a las sentencias `havoc` correspondientes, tanto para variables como para accesos a un arreglo.

Para detectar las variables modificadas dentro de un ciclo, se implementó un *visitor* [23] sobre el AST[2] de JDynAlloy. Dicho *visitor* detecta las expresiones JDynAlloy correspondientes a modificaciones de variables o arreglos, y guarda dichas expresiones para luego poder generar el `havoc` de cada una de ellas.
En el caso de modificaciones a arreglos, se distinguen dos casos: la modificación de una referencia a un arreglo, la cual es tratada como la modificación de cualquier variable, y la modificación del contenido de un arreglo, la cual es traducida de manera diferente.

2.6.4. Assert

De todas las traducciones, la traducción del `assert` resultó ser la más compleja.

En los lenguajes imperativos, al encontrarse con un `assert` inválido, lo habitual es que la ejecución del programa se interrumpa.

[2]AST: Abstract Syntax Tree, Árbol de sintáxis abstracto

Sin embargo, lo que la herramienta puede verificar son aserciones de correctitud parcial (triplas de Hoare) de la siguiente forma:

$$\{P\}B\{Q\}$$

En este caso el verificador buscará valores de las variables que hagan falsa esta fórmula, en cuyo caso la postcondición Q no será satisfecha. En el caso de que el cuerpo del programa contenga un `assert`, el mismo estará codificado en la traducción de B. Si el `assert` falla, hará que la verificación falle, encontrando así un contraejemplo, pero no interrumpiendo la ejecución de la verificación.

Además, en caso de que un `assert` falle, querremos que este estado se propague en el *call stack*, de forma tal que todos los métodos finalicen fuera de su flujo normal de ejecución.

Esta propagación es similar a la existente en el manejo de excepciones Java, las cuales ya son soportadas por la herramienta (ver sección 1.6.3 en página 16). Por este motivo, la forma en que se decidió manejar los `assert` es a través de una "pseudo-excepción". Esta "pseudo-excepción" no proviene de la traducción de una excepción Java como el resto, sino que aparece como parte de la traducción de JDynAlloy a DynAlloy.

Utilizar una pseudo-excepción nos permite reutilizar toda la lógica ya presente en las herramientas para manejar las excepciones de Java, lo cual garantiza que la excepción se propagará a través de las llamadas a métodos.

Para ilustrar la traducción de `assert` veamos un ejemplo. Supongamos que se tiene el código Java + JML mostrado en el listado 2.4 (página 32).

La traducción a JDynAlloy de ese código genera un `assert`, como podemos ver en el listado 2.5 (página 32).

Finalmente, la traducción a DynAlloy genera el código que podemos ver en el listado 2.6 (página 33), donde se comentan las partes relevantes al *assert*.

```
/*@
  @ ensures i == 1;
  @*/
public static void assertDemo(int i) {
    i = 2;
    //@ assert i == 0;
    i = 0;
    i = i + 1;
}
```

Listado 2.4: Ejemplo de código Java + JML utilizando assert

```
program AssumeAssert::assertDemo[var throw:Throwable+null,
                                 var i:Int]
Specification
{
   ensures { i = 1 }
}
Implementation
{
   {
      throw:=null;
      i:=2;
      assert i = 0;
      i:=0;
      i:=i + 1;
   }
}
```

Listado 2.5: Código JDynAlloy producido a partir de Java con asserts

```
1  program AssumeAssert_assertDemo_0[throw:Throwable+null, i:Int]
2  var [
3    assertionFailure:boolean       // Variable local que indica si
4  ]{                               // falló algún assert
5    assertionFailure:=false;       // Inicialización. Asserts válidos.
6    throw:=null;
7    i:=2;
8    if not i = 0 {                 // Test de la condición del assert
9      assertionFailure:=true       // Si falló, se asigna la variable.
10   };
11   i:=0;                          // Continúa el programa
12   i:=i + 1;
13   if assertionFailure = true {
14     throw:=AssertionFailure      // Si hubo asserts, se tira la
15   }                              // excepción.
16 }
```

Listado 2.6: Traducción a DynAlloy de un programa con asserts

2.7. Limitaciones

La generación de *VC* utilizando invariantes de ciclo requiere que los ciclos del programa sean anotados en JML.

En la traducción realizada por Traductor JDynAlloy, sin embargo, no toda la sintáxis de JML es soportada.

En esta sección analizaremos algunas limitaciones actuales de la herramienta, y como las mismas fueron subsanadas en los programas usados para los experimentos.

2.7.1. Acceso al pre-estado de las variables

JML permite referenciar el pre-estado de las variables, a través del operador `\old`.

Tomemos por ejemplo el programa Upsort, utilizado como parte de los ejemplos. Dentro del invariante de ciclo, una cláusula indica que el arreglo preserva sus elementos. Esto puede expresarse en JML utilizando el operador `\old` de la siguiente manera:

```
1 /*@
2   @ loop_invariant
3   @   (\forall int i; i >= 0 && i < n;
4   @     (\exists int j; j >= 0 && j < n && a[i] == \old(a[j]))
5   @   );
6   @*/
7   while (...)
```

Listado 2.7: Uso de operador old en JML

DynAlloy solo permite trabajar con el postestado de las variables en las postcondiciones de las acciones, y por lo tanto sólo se permite utilizar el operador `\old` en las anotaciones `@ensures` de JML.

Esta limitación no demostró ser problemática, ya que en los experimentos se pudo trabajar con una variable auxiliar que represente el pre-estado, como se puede verse en el listado 2.8 (página 35).

En el ejemplo puede verse como la variable `prea` almacena el pre-estado de `a`, de tal forma de no depender del operador `\old` de JML para anotar el método. Esta solución requiere al usuario agregar restricciones (a manera de precondiciones adicionales) que relacionan la variable y su pre-estado, de tal forma que se respete la semántica del operador `\old` y que la relación entre las variables permita representar correctamente las expresiones JML ya sea para pre y post-condiciones o invariantes de ciclo.

```
/*@
  @ loop_invariant
  @    (\forall int i; i >= 0 && i < n;
  @        (\exists int j; j >= 0 && j < n && a[i] == prea[j])
  @    );
  @ requires a.length == n && prea.length;
  @*/
  while (...)
}
```

Listado 2.8: Alternativa al uso de old con variable adicional

Si bien en este trabajo se modificó el código original para agregar una variable que almacena el pre-estado, esto podría ser generado automáticamente por la herramienta, y de esta forma dar soporte al operador `\old` de JML.

2.7.2. Invocación de métodos puros en anotaciones

JML permite marcar ciertos métodos de una clase como *puros* [12], a través de la anotación `@pure`. Esta anotación es utilizada para identificar métodos que no tienen efectos colaterales, como ser un *getter* de JAVA[3], o un método que realiza cálculos auxiliares. Una vez que un método es marcado como puro, el mismo puede ser invocado desde expresiones JML.

TACO no soporta actualmente la invocación de métodos desde expresiones JML, lo que dificulta expresar algunas pre y post-condiciones en JML.

En los experimentos realizados en este trabajo, esto no resultó ser una complicación mayor. En los casos donde los invariantes dependían de funciones auxiliares (por ejemplo contar los elementos repetidos en el invariante de Upsort), se modificó el contrato de los métodos (en este caso, requerir arreglos sin elementos repetidos), de forma tal de no necesitar invocaciones a métodos *puros*.

[3]Los *getters* no necesariamente están libres de efectos colaterales. En esos casos, no debieran ser anotados como métodos puros.

2.8. Experimentación y evaluación

2.8.1. Metodología

Para comprobar la escalabilidad de la herramienta, medimos la performance de la verificación de programas JAVA anotados con JML.

Dado que nos interesa ver si la escalabilidad de la herramienta mejora utilizando la verificación usando el teorema del invariante y la verificación usando atomización de ciclos frente al análisis con *loop unrolling*, realizamos verificaciones de programas utilizando los tres tipos verificación para diferentes *scopes* de análisis.

Lo que nos interesa comparar es como impactan los cambios en el *scope* en los tiempos de verificación.

Además de los tiempos, veremos como evolucionan ciertas métricas de las fórmulas CNF que genera la herramienta, las cuales pueden dar una pauta de como crece el costo de la verificación.

Conjunto de experimentos

Para la experimentación se armó un conjunto de programas (ver sección 2.9 en página 38) con ciclos, los cuales fueron anotados con invariantes de ciclo.

Con el objetivo de que la comparación sea justa entre las tres variantes de análisis, utilizaremos programas que son correctos, (es decir, programas donde la herramienta no encuentra contraejemplos). Además, los programas no están sobre ni subespecificados, para evitar verificaciones triviales.

Como se puede ver en la figura 2.1 (página 37), los árboles de búsqueda generados por el SAT Solver para los diferentes tipos de verificación tendrán tamaños distintos. Esto se debe principalmente a que (como se expuso en la sección 2.2) para diferente número de *unrolls* el Traductor DYNALLOY genera diferentes especificaciones ALLOY, las cuales tienen diferente número de variables a ser evaluadas durante la verificación. Lo que nos interesa comparar en los experimentos, es la relación existente entre el tamaño de estos árboles. Para asegurarnos que el SAT Solver recorra los árboles por completo, realizamos experimentos donde no se encuentran contraejemplos, lo que garantiza que el SAT Solver debe barrer todo el árbol de búsqueda de cada tipo de análisis.

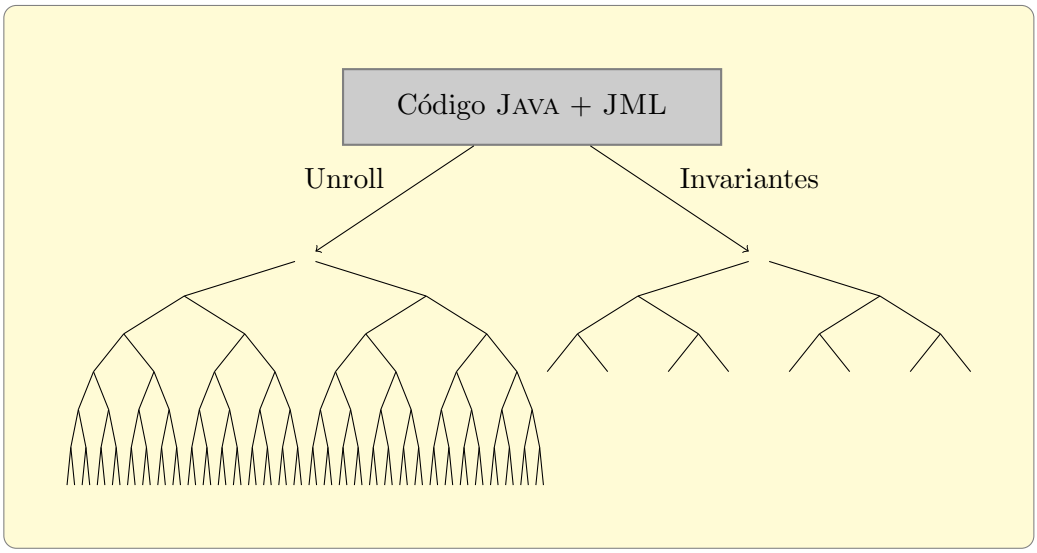


Figura 2.1: Comparación de árboles de búsqueda

Scope, longitud de secuencias y unrolling

Muchos de los experimentos realizados com parte del conjunto de experimentos, son programas que operan sobre arreglos de Java. Los arreglos en Java terminan siendo representados con secuencias Alloy (utilizando `seq`).

Al traducir a Alloy utilizando loop unrolling, la herramienta requiere especificar la cantidad de unrolls a realizar durante la traducción. Para estos experimentos, se usó un número de unrolls igual a la longitud máxima de la secuencia especificada en el scope de análisis.
Esto garantiza que al verificarse secuencias de longitud n los programas que se evaluen sean aquellos donde se realizan *hasta* n unrolls.

Además de la longitud de las secuencias, Alloy maneja como parte de su scope la variable de *bitwidth*. El *bitwidth* indica cuantos bits tendrán los números enteros, y por lo tanto cuantos números enteros estarán disponibles durante la verificación.

Alloy impone una relación entre el *bitwidth* y la longitud máxima de las secuencias, donde el *bitwidth* debe crecer acompañando a la longitud máxima de las secuencias (si se desea una longitud de secuencia de n, el *bitwidth* debe ser de $(\log_2 n) + 1$).

Para realizar los experimentos decidimos utilizar el mínimo *bitwidth* ne-

cesario para la longitud de secuencias elegida.

Ejecución de los experimentos

Para cada programa se realizaron la verificación con *loop unrolling*, la verificación usando el teorema del invariante, y la verificación usando atomización de ciclos, comenzando con un scope de secuencias / número de *unrolls* de 3. Las pruebas fueron ejecutadas por aproximadamente doce horas por experimento, y se toma como referencia el máximo scope / número de *unrolls* alcanzado en ese lapso de tiempo.

2.9. Descripción de los casos de estudio

A continuación se presentan los programas utilizados para realizar los experimentos.

ArrayCopy

Descripción: El programa realiza una copia de un arreglo en otro.

Complejidad: $O(n)$

ArrayMakeNegative

Descripción: El programa recibe como parámetros de entrada dos arreglos, a y b. A la salida devuelve una copia del arreglo a, pero donde los elementos que pertenecen a b son ahora negativos.

Complejidad: $O(n * m)$

ArrayMerge

Descripción: El programa recibe dos arreglos, a y b ordenados, y devuelve un arreglo ordenado producto de realizar un *merge* de los dos arreglos.

Complejidad: $O(n + m)$

ArrayReverse

Descripción: El programa devuelve un arreglo con los *invertido* del original. El primer elemento es el último, y así sucesivamente.

Complejidad: $O(n)$

BubbleSortArray

Descripción: Ordena un arreglo utilizando el algoritmo Bubble Sort.

Complejidad: $O(n^2)$

LinearSearch

Descripción: El programa recibe un arreglo de enteros *list* y un elemento *element*. Devuelve la posición de *element* en *list*, o -1 si no se encuentra el elemento.

Complejidad: $O(n)$

MCD

Descripción: Calcula el MCD (Máximo Común Divisor) entre dos números enteros.

Complejidad: $O(log\ n)$

SubArrayFind

Descripción: Busca un subarreglo dentro de otro, y retornando la posición inicial del subarreglo, o -1 si no se encuentra.

Complejidad: $O(n * m)$

Upsort

Descripción: Ordena un arreglo utilizando el algoritmo Upsort.

Complejidad: $O(n^2)$

2.10. Resultados y análisis

A continuación se presentan gráficos individuales de la ejecución de cada uno de los programas.
Para cada programa se realizaron las tres tipos de verificaciones, para diferentes *scopes* y número de *loop unrolls*.

1. Verificación utilizando *loop unrolling*.
2. Verificación utilizando Teorema del Invariante.

3. Verificación utilizando Atomización de ciclos.

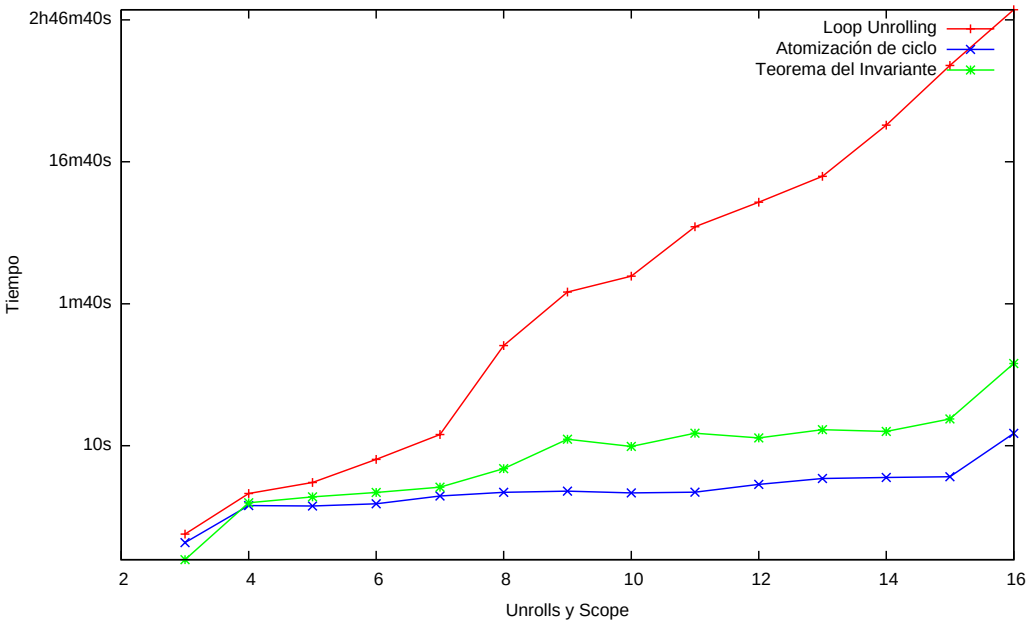


Figura 2.2: ArrayCopy - Comparación de tiempos

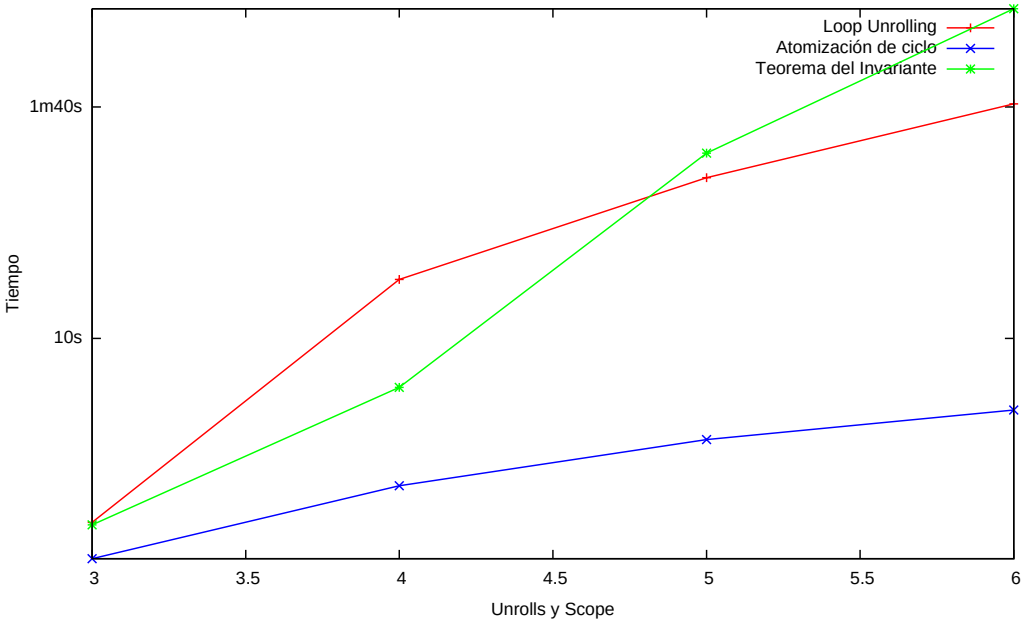


Figura 2.3: ArrayMakeNegative - Comparación de tiempos

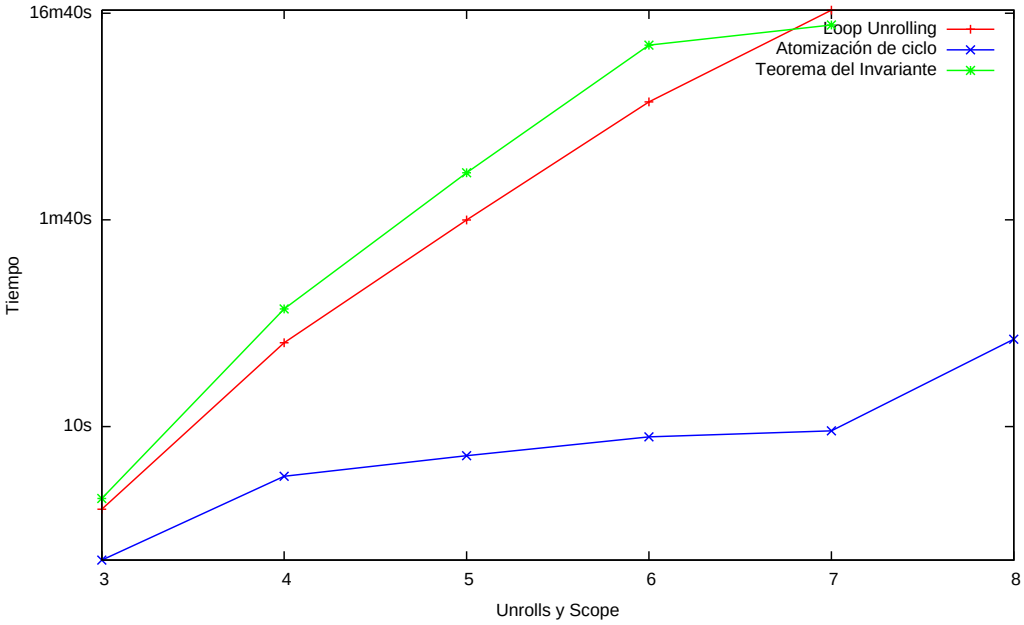


Figura 2.4: ArrayMerge - Comparación de timepos

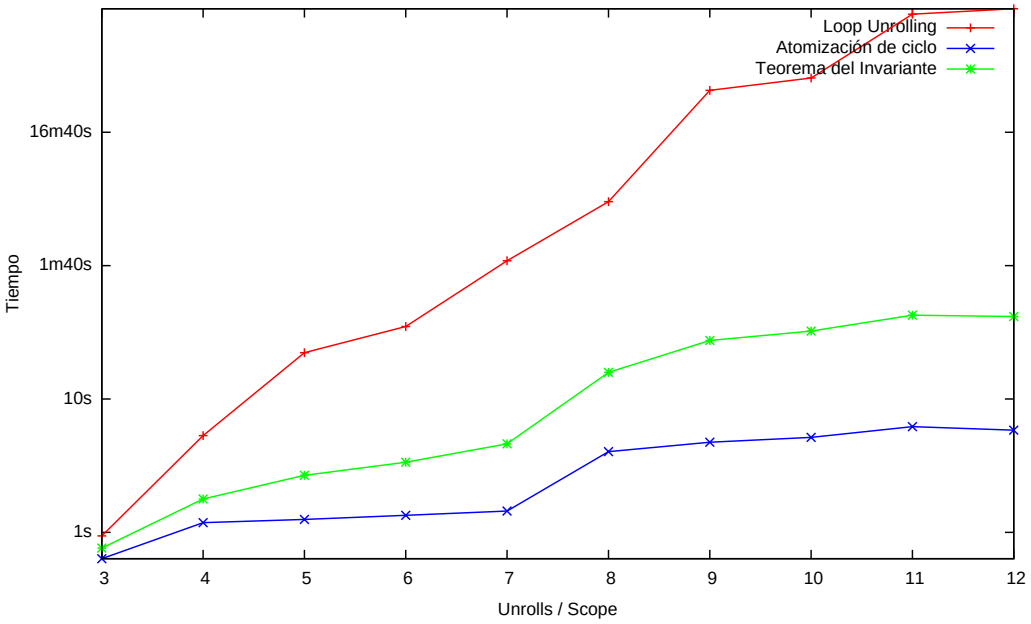


Figura 2.5: ArrayReverse - Comparación de tiempos

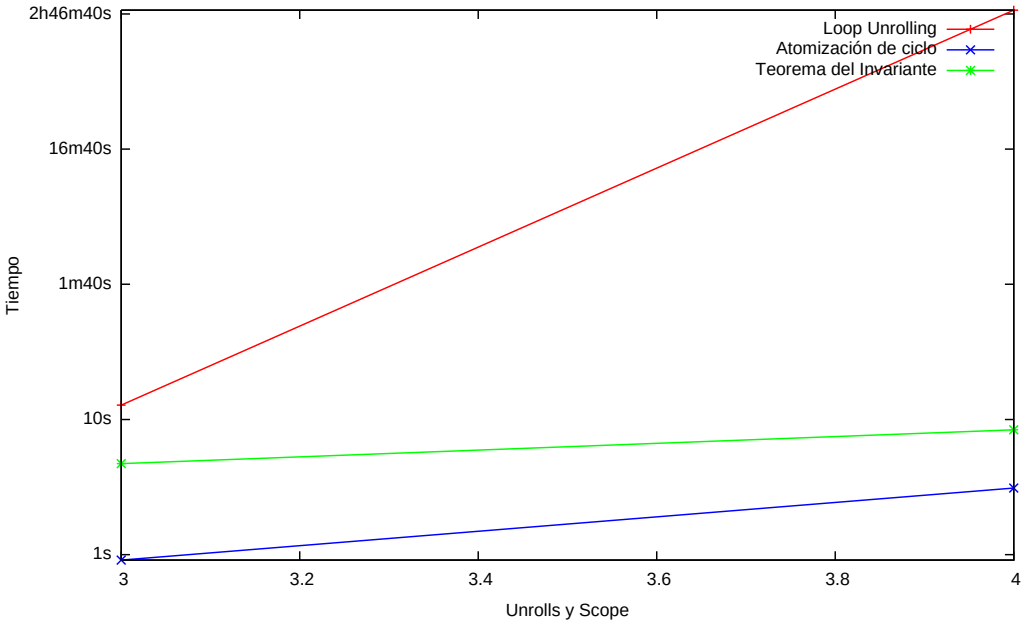


Figura 2.6: BubbleSortArray - Comparación de tiempos

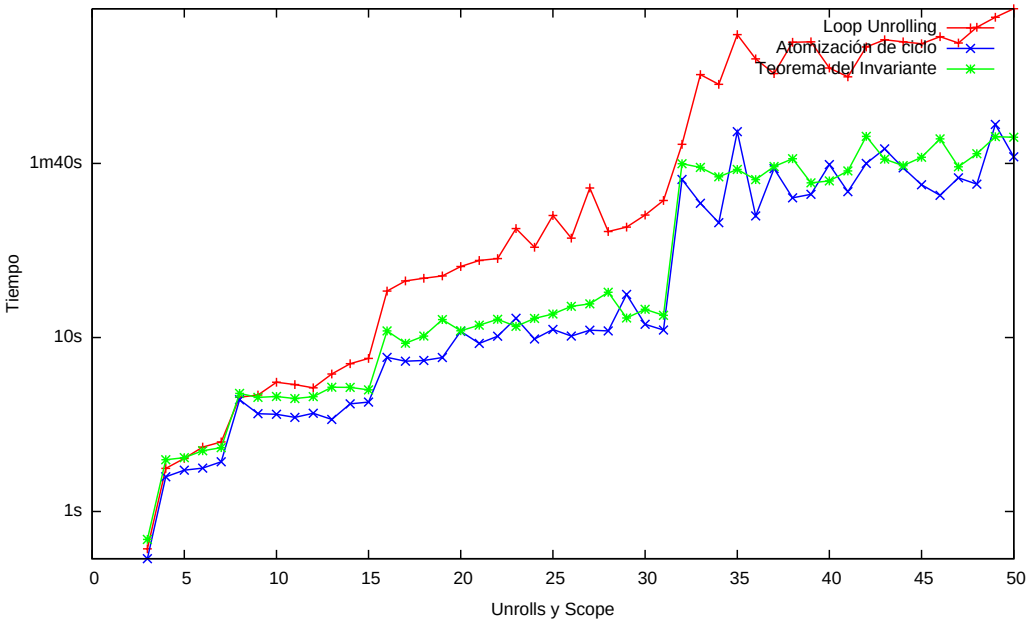


Figura 2.7: LinearSearch - Comparación de tiempos

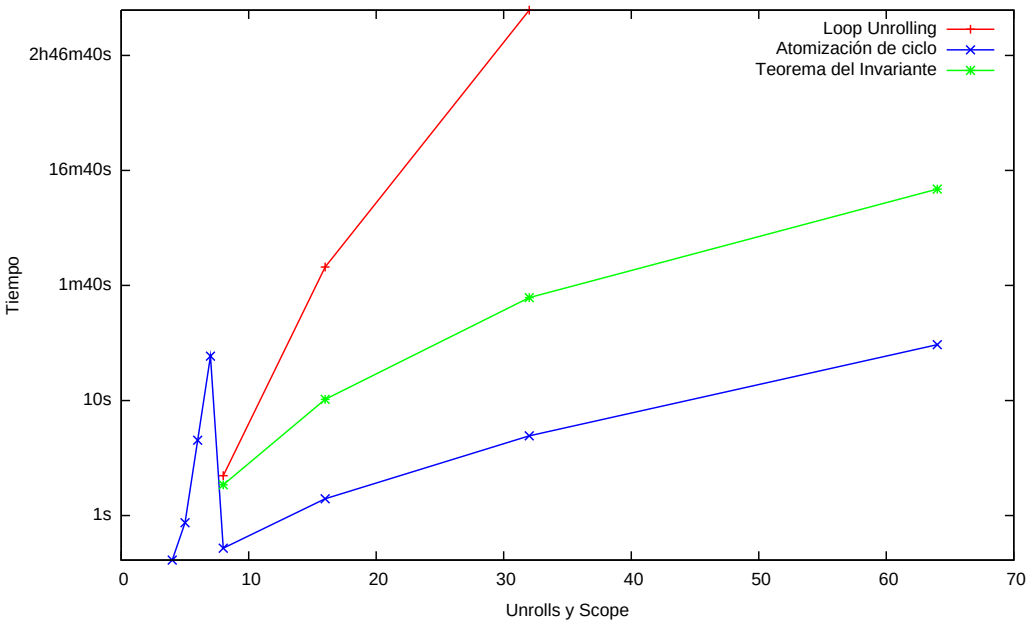


Figura 2.8: MCD - Comparación de tiempos

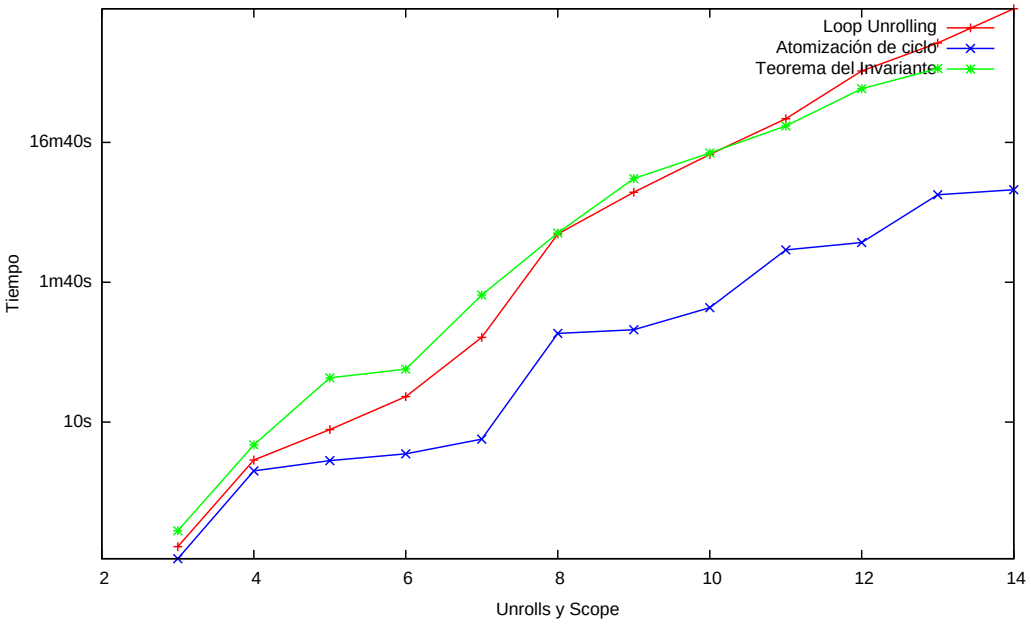


Figura 2.9: SubArrayFind - Comparación de tiempos

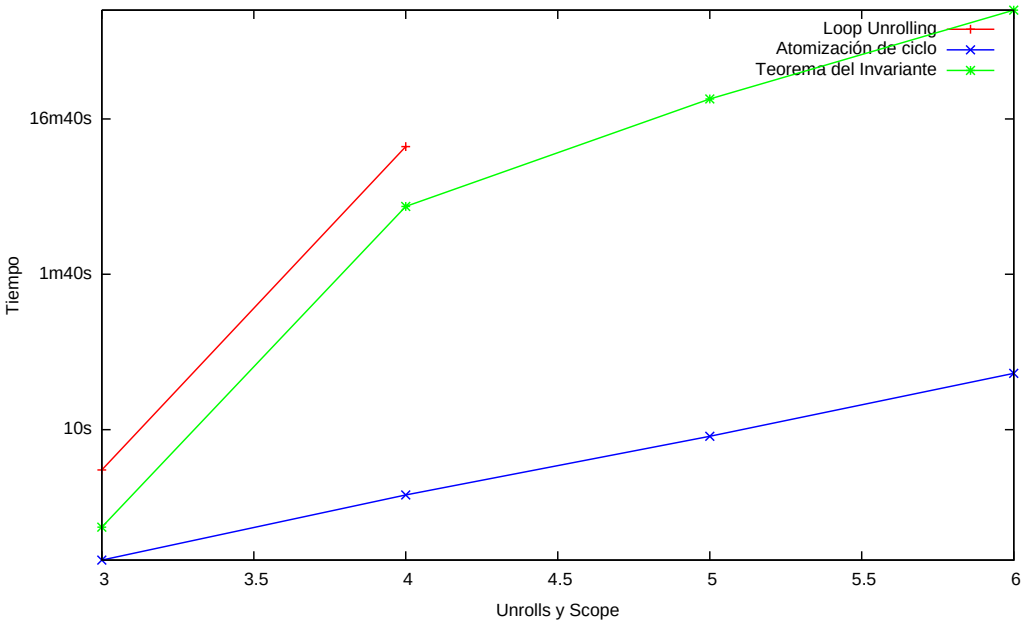


Figura 2.10: Upsort - Comparación de tiempos

En las figuras 2.11 (página 46) a 2.14 (página 48) podemos ver un resumen de los experimentos realizados, comparando la verificación usando *loop unroll* con la verificación usando atomización de ciclos.

En la figura 2.11 (página 46), en el eje X se grafica la cantidad de *loop unrolls* y *scope* utilizado, mientras que en el eje Y se puede ver el coeficiente resultante de dividir el tiempo requerido por la verificación utilizando *loop unrolling* sobre el tiempo requerido por la verificación utilizando atomización de ciclo.

Las figuras 2.12 (página 47) a 2.14 (página 48) se grafica nuevamente en el eje X la cantidad de *loop unrolls* y *scope* utilizado, mientras que en el eje Y se consigna el coeficiente entre diferentes métricas obtenidas del SAT Solver utilizado por el Alloy Analyzer y que miden el tamaño de la fórmula CNF sobre la cual se realiza el análisis.

Las figuras 2.15 (página 48) a 2.18 (página 50) grafican los mismos datos, pero esta vez comparando la verificación usando *loop unroll* con la verificación usando el teorema del invariante.

Como puede verse en los gráficos comparativos, en los programas que tienen complejidad $O(n^2)$ (BubbleSort, Upsort, ArrayReverse, SubArrayFind) puede verse un incremento mucho mayor en el tiempo requerido para ejecutar utilizando *loop unrooll*. En estos casos, el tiempo en que se ejecutaron los experimentos, solo permitió verificar el programa para scopes bajos.

Luego se puede ver otro tipo de crecimiento de los tiempos en los programas de complejidad $O(n)$ (ArrayMakeNegative, LinearSearch), mucho menos pronunciado que en los de complejidad $O(n^2)$, debido a que el impacto del *loop unroll* en el tamaño de la especificación Alloy generada es mucho menor.

Finalmente, puede verse un caso donde el impacto de utilizar invariantes de ciclo en lugar de *loop unroll* no es tan significativo (LinearSearch).

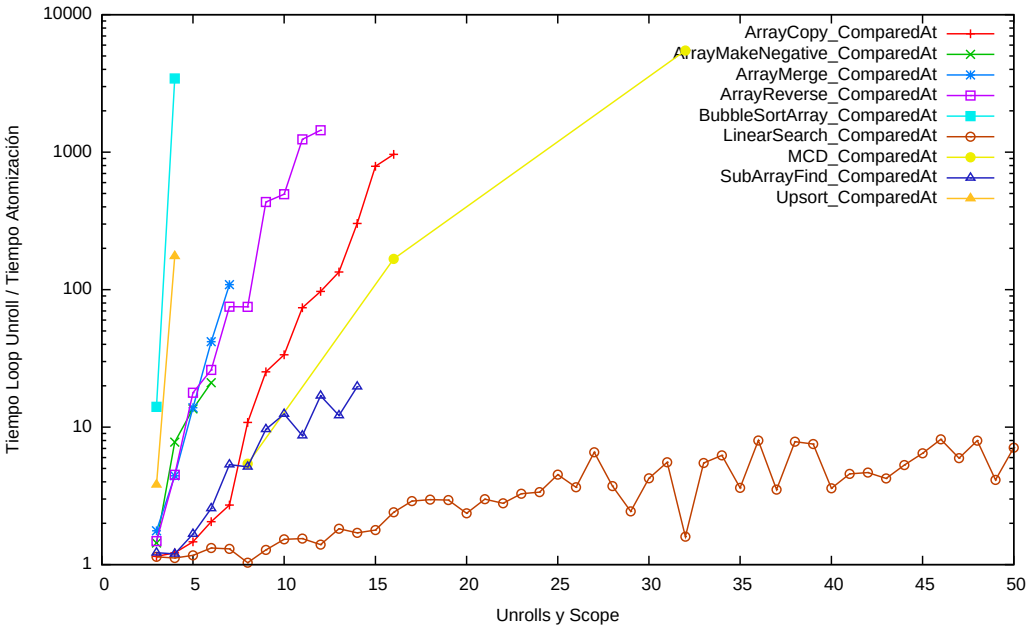


Figura 2.11: Resumen comparativo - Atomización vs Unroll - Tiempo

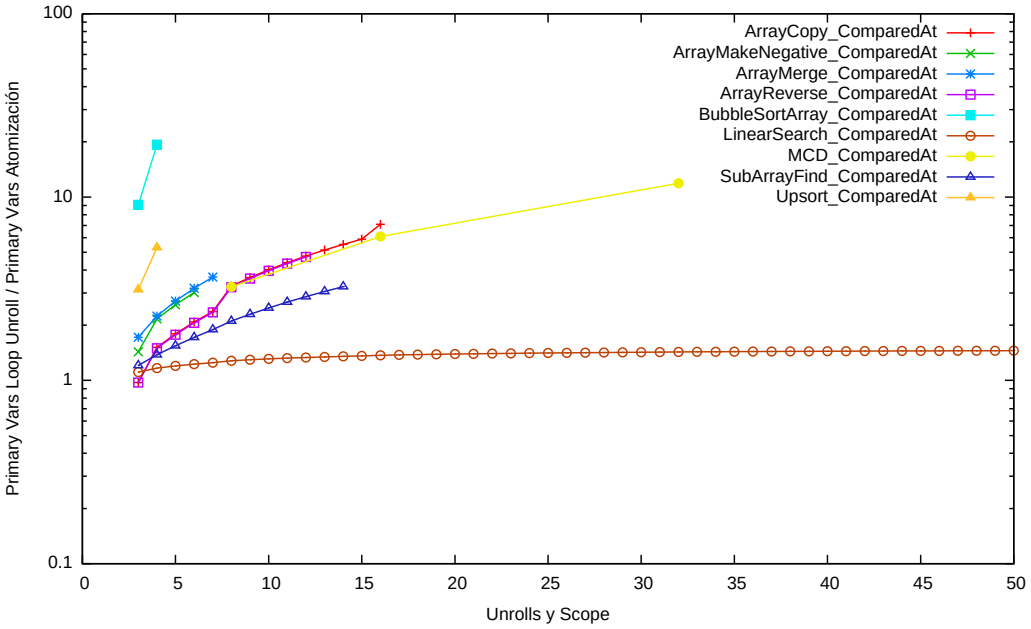


Figura 2.12: Resumen comparativo - Atomización vs Unroll - Primary Vars

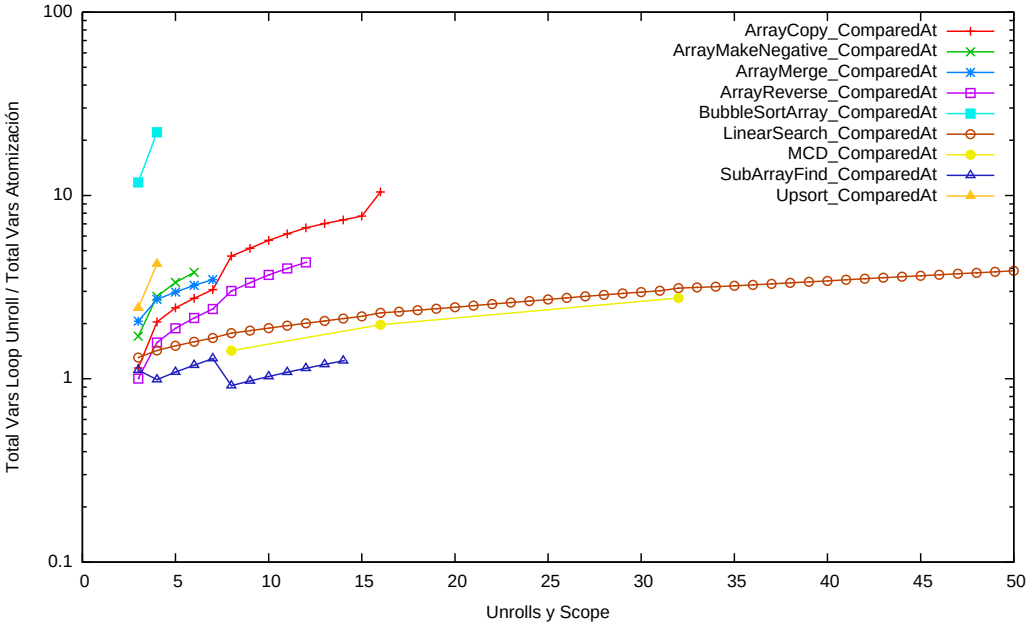


Figura 2.13: Resumen comparativo - Atomización vs Unroll - Total Vars

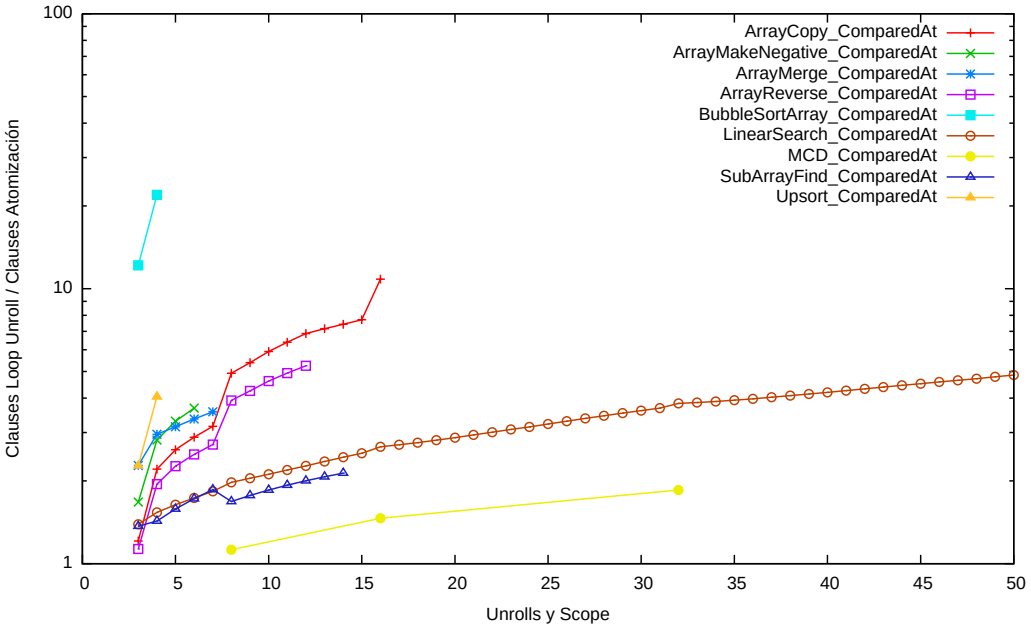


Figura 2.14: Resumen comparativo - Atomización vs Unroll - Clauses

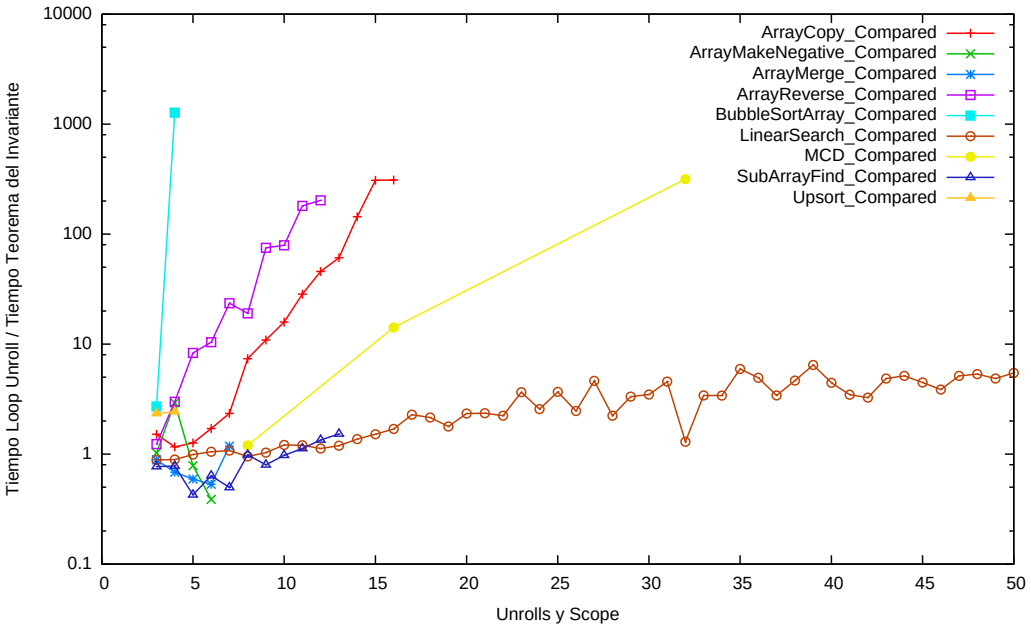


Figura 2.15: Resumen comparativo - Teorema del Invariante vs Unroll - Tiempo

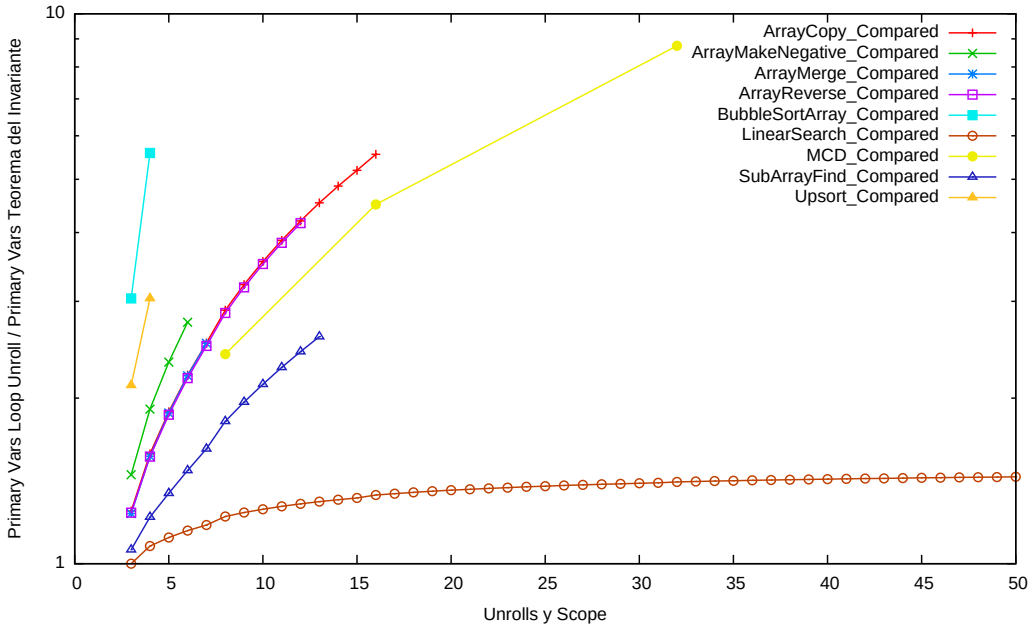


Figura 2.16: Resumen comparativo - Teorema del Invariante vs Unroll - Primary Vars

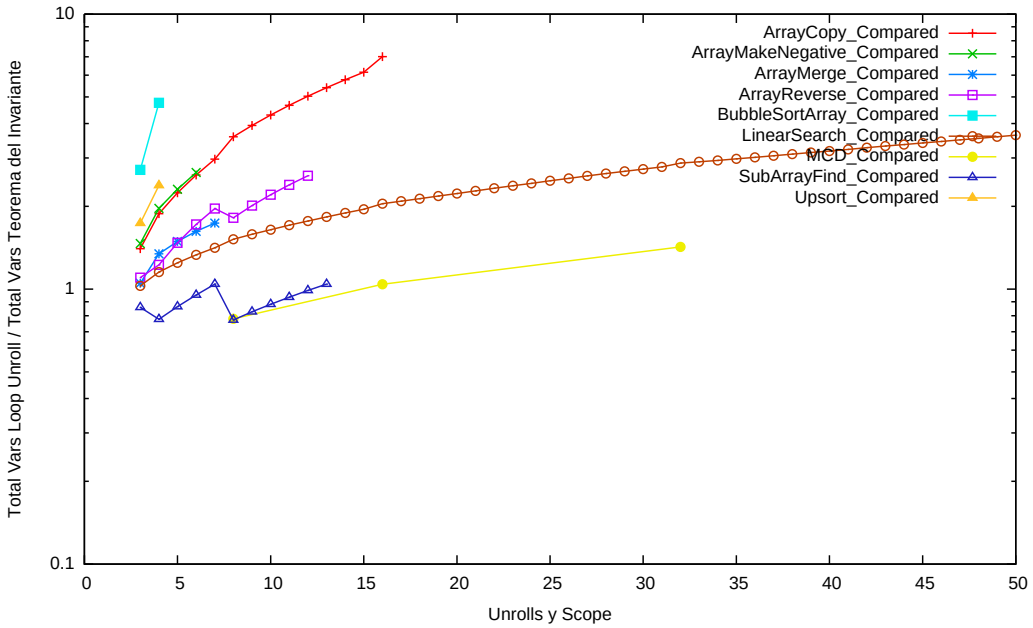


Figura 2.17: Resumen comparativo - Teorema del Invariante vs Unroll - Total Vars

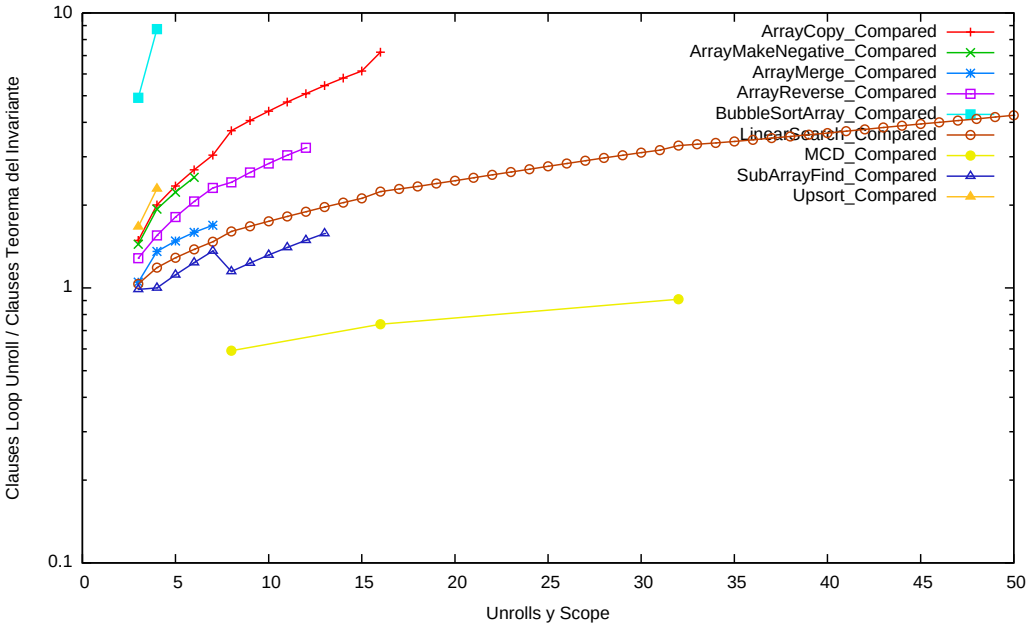


Figura 2.18: Resumen comparativo - Teorema del Invariante vs Unroll - Clauses

Del análisis de estos resultados, se nota lo siguiente:

- En todos los casos, como se esperaba, la versión utilizando *loop unrolling* demora mucho más en recorrer todas las trazas de ejecución que la versión usando atomización ciclos.
- Los programas donde la verificación requiere modificar el contenido de arreglos (secuencias ALLOY) son los que más demoran en verificarse (ver figuras 2.2, 2.4, 2.5). Esto se debe a la representación utilizada en la *VC* de los arreglos de JAVA.
- Para un número de unrolls más grande, no necesariamente el tiempo requerido para la verificación es mayor. Esto podemos verlo en las figuras 2.8 o 2.7, donde se puede apreciar que en algunos casos los tiempos bajan al subir el número de unrolls.
- Los tiempos de verificación de programas de complejidad $O(n^2)$ utilizando *loop unrolling* crecen de manera exponencial, haciendo que en esos casos usar el invariante de ciclo genere una diferencia en los tiempos más significativa.
- Del gráfico 2.12 (página 47) se desprende que cada programa tiene una cota máxima de cuanto puede mejorar su performance utilizando atomización de ciclos respecto a *loop unrolling*.
- En algunos casos, la versión utilizando *loop unrolling* es mejor que la versión utilizando el teorema del invariante (ver figuras 2.3 y 2.4). En estos casos el impacto de traducir el cuerpo del ciclo es alto, y por lo tanto la verificación del invariante resulta más costosa.

Capítulo 3

Visualización de contraejemplos

3.1. Motivación

Como ya mencionamos anteriormente, TACO funciona traduciendo de código Java anotado con JML a lenguajes intermedios cada vez más cercanos en su semántica al lenguaje utilizado por la herramienta de análisis, Alloy. Esto lo podemos ver en la figura 1.1 (página 6).

Una vez obtenido la especificación Alloy se puede utilizar esta herramienta para verificar la corrección del programa. Si se encuentra un contraejemplo en la especificación Alloy, significa que se encontró una violación en la especificación JML del programa Java original.

Una vez hallado un contraejemplo Alloy, la herramienta provee un visualizador de contraejemplos y un evaluador de expresiones que permiten al programador localizar el origen del mismo.

El problema que se encuentra al utilizar TACO es que a medida que el código Java original es traducido a lenguajes más cercanos en su sintáxis y semántica a Alloy hasta llegar a una especificación en este último lenguaje (ver figura 1.1 en página 6), la especificación es cada vez más compleja. Además, para poder entender la especificación resultante, es necesario tener cierto conocimiento de como funcionan los traductores, para poder entender las equivalencias entre los diferentes lenguajes utilizados.

Más aún, como la semántica de Alloy es de átomos y relaciones, el visualizador y evaluador provistos facilita analizar y trabajar con estas especificaciones. Por otro lado, el resto de los lenguajes usados (Java, JDynAlloy y DynAlloy) son lenguajes imperativos (Java) o lenguajes que permiten expresar acciones y programas (JDynAlloy, DynAlloy) donde la visua-

lización esperable es la de una traza de ejecución, pudiendo analizar la evaluación de los valores de las variables a medida que progresa el programa.

En las figuras 3.2 (página 54) y 3.3 (página 55) podemos ver el visualizador provisto por el Alloy Analyzer mostrando un contraejemplo encontrado sobre la ejecución del programa Java LinearSearch. Como se puede ver, resulta dificil interpretar la información presentada por el visualizador con el objetivo de encontrar el error en el programa original.

El objetivo de esta parte del trabajo es extender la herramienta para visualizar los contraejemplos sobre la especificación DynAlloy, presentando al usuario una interfaz que permita analizar la traza de ejecución proveniente del contraejemplo encontrado durante el análisis.

En la figura 3.1 podemos ver como se integra el Visualizador DynAlloy con el resto de la herramienta.

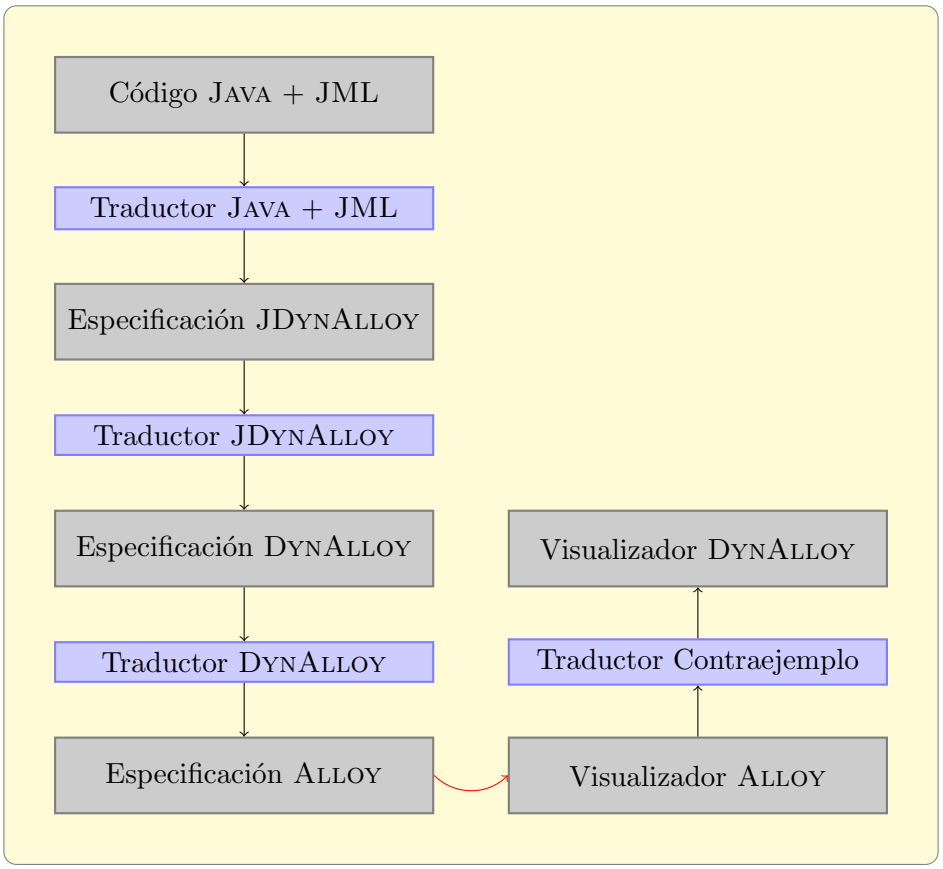


Figura 3.1: Visualización de contraejemplos en DynAlloy

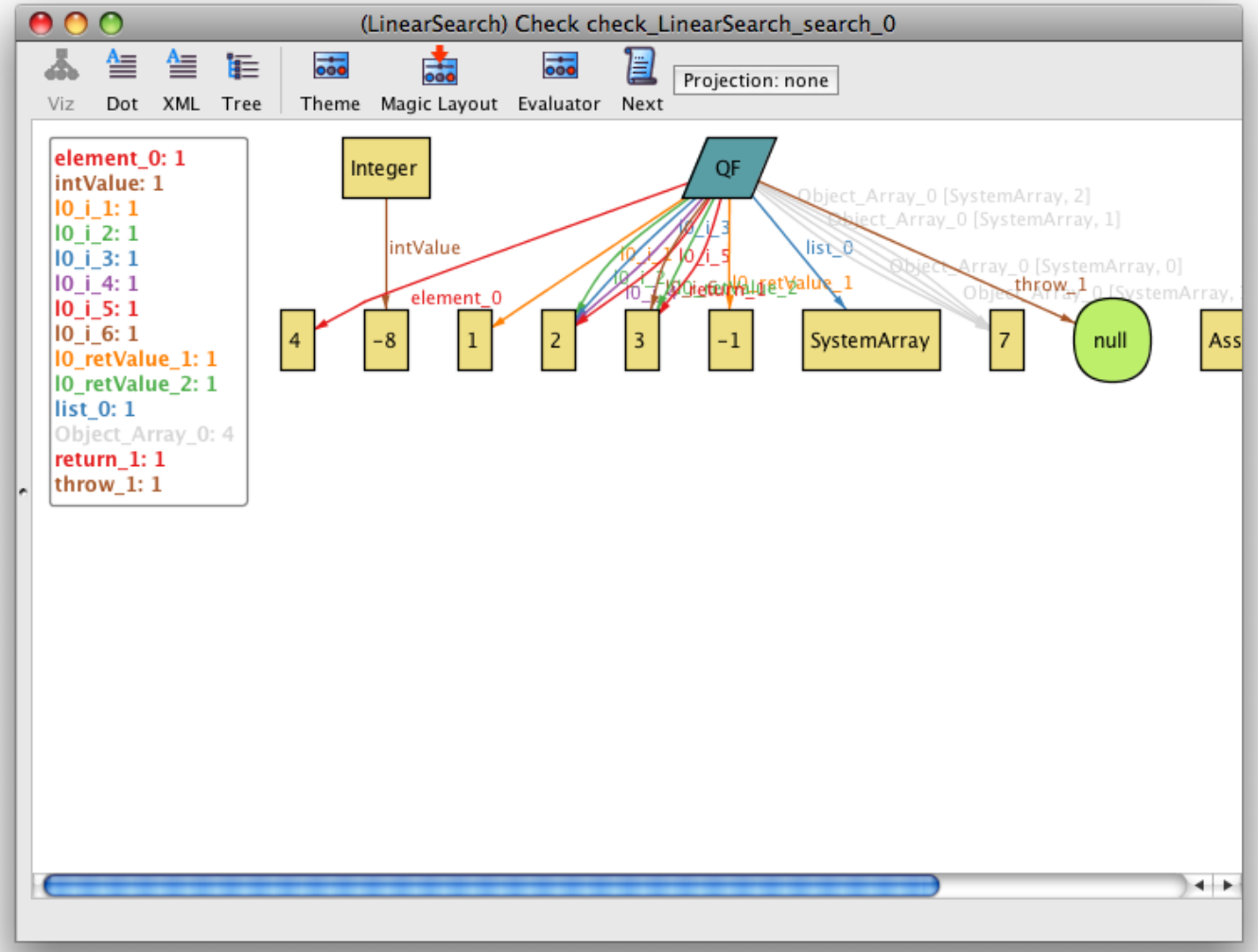


Figura 3.2: Visualizador ALLOY en un contraejemplo de LinearSearch

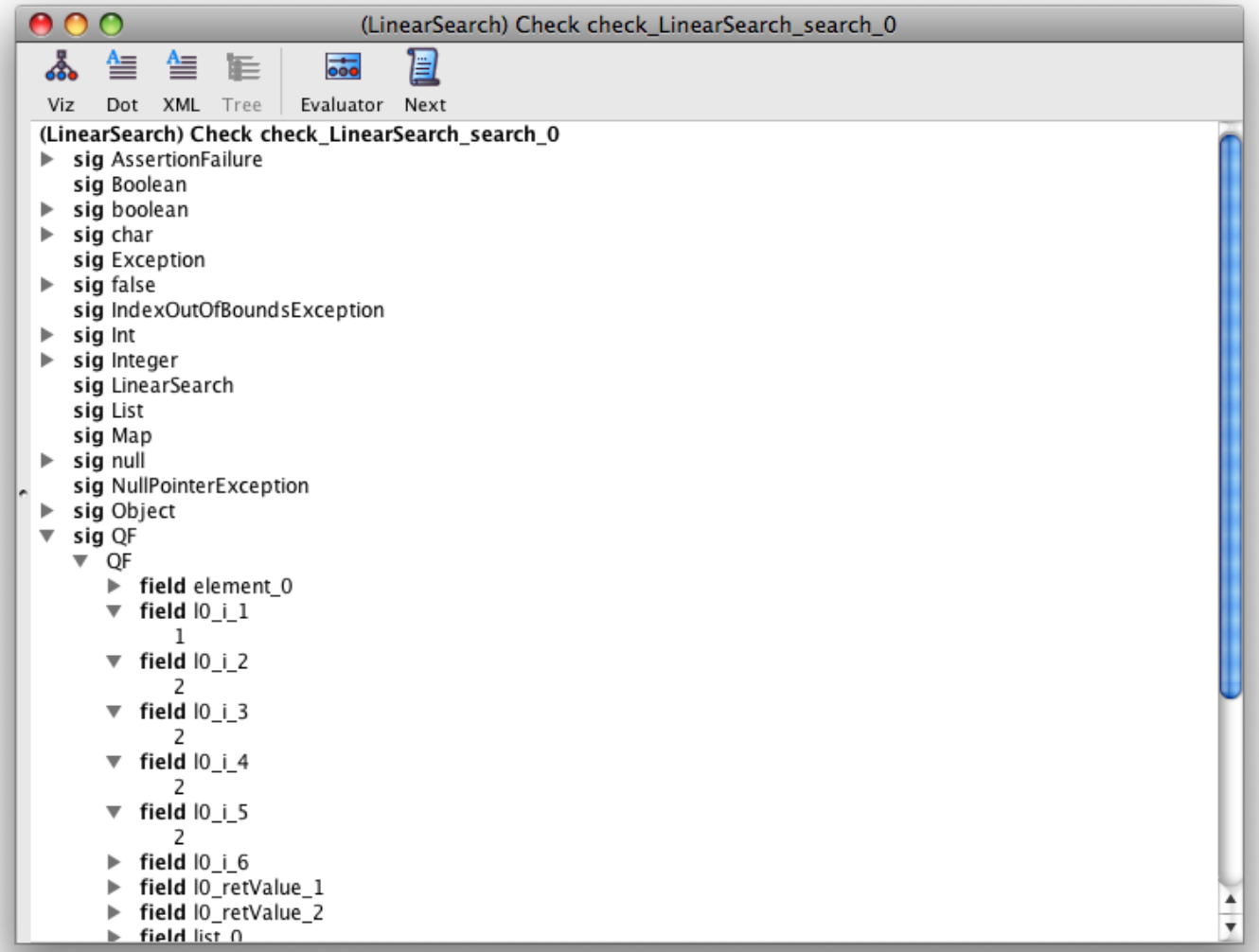


Figura 3.3: Visualizador ALLOY en un contraejemplo de LinearSearch (tree)

3.2. Implementación

3.2.1. Correlación DynAlloy a Alloy

El Traductor JDynAlloy y el Traductor DynAlloy funcionan leyendo el archivo original, convirtiendo el mismo a un AST para luego (utilizando el patrón de diseño *visitor*) convertirlo a un AST de otro lenguaje.
Finalmente, el nuevo AST es escrito a un archivo obteniendo la representación traducida del programa o especificación original.

Para poder visualizar los contraejemplos, que serán expresados en Alloy por el verificador, necesitamos poder convertir *expresiones* Alloy en *programas* DynAlloy. La forma de obtener este mapeo es guardando, durante la traducción, una equivalencia entre el programa DynAlloy origen y la expresión Alloy destino.
Si este mapeo es bidireccional, podremos utilizarlo de manera inversa para, a partir de una expresión Alloy del contraejemplo, obtener el código DynAlloy que la generó.

3.2.2. Evaluación de contraejemplo

Una vez que la herramienta encuentra un contraejemplo en Alloy nos interesa traducir el mismo a una traza de ejecución DynAlloy para poder visualizarlo.

Para esto, generamos un *visitor* sobre la fórmula Alloy resultante. En cada nodo, el *visitor* evalúa el valor de verdad de la expresión Alloy, y en caso de ser verdadera obtiene el programa DynAlloy que la generó. Este programa es entonces agregado a la traza de ejecución resultante.

3.2.3. Ejemplo de correlación y evaluación

Para ejemplificar la correlación DynAlloy a Alloy y la evaluación de la misma, tomemos la especificación DynAlloy que se presenta en el listado 3.1 (página 57).

Esta especificación tiene tres *acciones* y un programa. `addOne` incrementa una variable en 1, `addTwo` incrementa una variable en 2 y finalmente `addThree` invoca a `addOne` o `addTwo` según el valor del parámetro recibido. El programa invoca la acción `addThree` con un valor de cero (según lo especifica su precondición).
En el *assert* tenemos una condición que no se cumplirá, de manera tal de obtener un contraejemplo en la verificación.

```
action addOne[i:Int] {
    pre { True }
    post { i' = i + 1 }
}

action addTwo[i:Int] {
    pre { True }
    post { i' = i + 2 }
}

program addThree[x:Int] {
    if x = 1 {
        addTwo[x]
    } else {
        addOne[x]
    }
}

assertCorrectness myAssertion[k:Int] {
  pre { equ[k,0] }
  program {
        call addThree[k]
  }
  post { k' = 2 }
}
check myAssertion
```

Listado 3.1: Especificación DynAlloy para suma de números

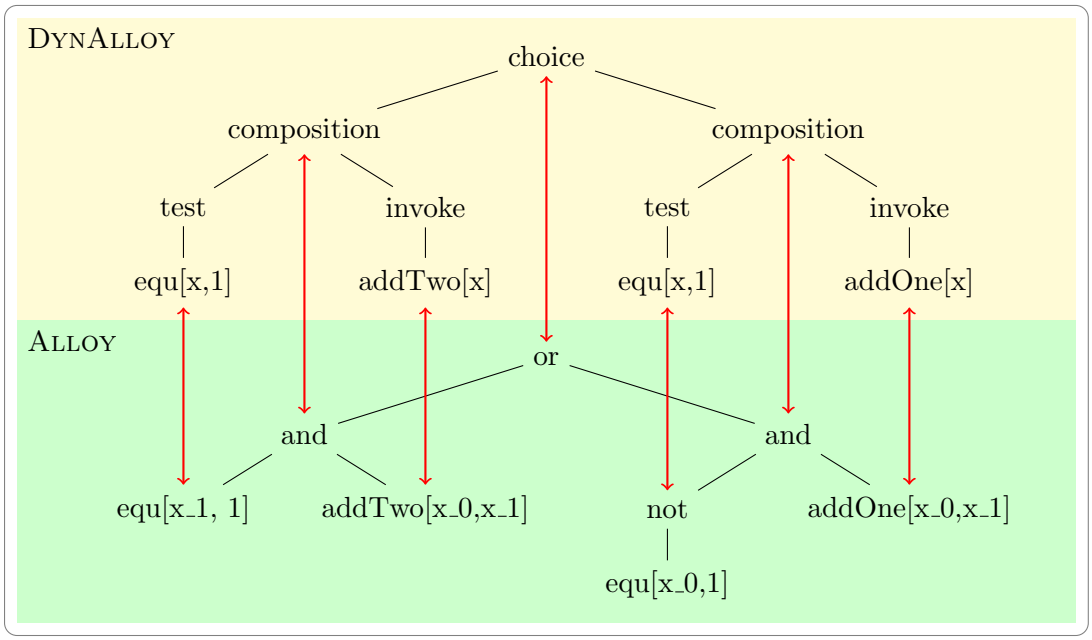
En el listado 3.2 vemos la especificación resultante de traducir utilizando el Traductor DynAlloy.

Finalmente, en la figura 3.4(a) (página 59) podemos ver la correlación entre el AST DynAlloy y el AST Alloy para las especificaciones aquí presentadas. En la segunda parte de esa figura 3.4(b) (página 59) vemos la evaluación realizada sobre el AST de Alloy una vez obtenido el contraejemplo. En verde se indican los nodos donde la evaluación fue verdadera, mientras que en rojo se indican los nodos donde fue falsa. El número a la izquierda del nodo indica el orden en que los mismos fueron visitados.

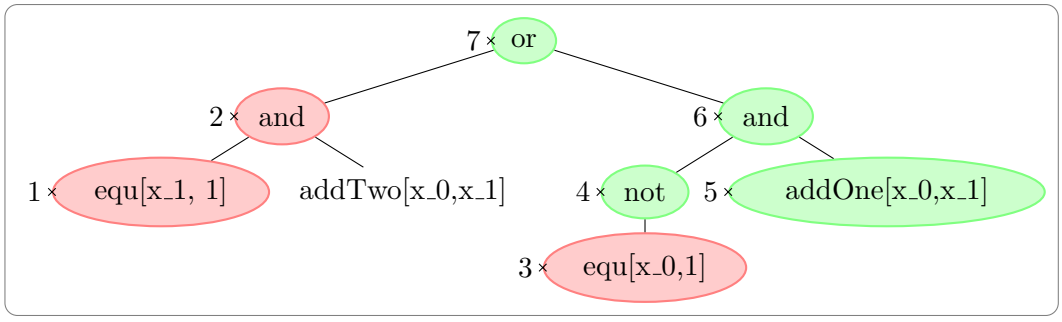
Tomando los nodos donde la evaluación dio verdadera, en el orden que fueron visitados, podemos buscar su correlación DynAlloy y de esa forma obtener la traza de ejecución del programa DynAlloy que generó el contraejemplo.

```
pred addOne[i_0: Int,i_1: Int]{
  True
  and
  i_1 = i_0 + 1
}

pred addTwo[i_0: Int,i_1: Int]{
  True
  and
  i_1 = i_0 + 2
}

pred addThree[x_0: Int, x_1: Int] {
  (
    x_0  = 1
    and
    addTwo[x_0,x_1]
  ) or (
    (x_0 != 1)
    and
    addOne[x_0,x_1]
  )
}
assert myAssertion{
all k_0 : Int, k_1 : Int,
    k_2 : Int, k_3 : Int | {
    (
      k_0 = 0 and addThree[k_0,k_1,k_2,k_3]
    )
    implies k_3 = 2
  }
}
```

Listado 3.2: Especificación ALLOY resultante



(a) Correlación entre ASTs de DynAlloy y Alloy



(b) Evaluación del AST de Alloy

Figura 3.4: Correlación entre un AST DynAlloy y un AST Alloy

3.2.4. Ejemplo de uso

Tomando la especificación presentada en el listado 3.3 (página 61) veremos a continuación como se un contraejemplo encontrado por TACO utilizando el nuevo visualizador, y -a manera de comparación- como se ve el mismo contraejemplo utilizando el visualizador ALLOY.

En la figura 3.5 (página 62) podemos ver el Visualizador DYNALLOY mostrando un contraejemplo encontrado por TACO.

El visualizador cuenta con tres paneles principales:

- Sobre la izquierda, un editor con la especificación DYNALLOY que se está analizando.
- En el panel superior derecho, se puede acceder tanto a la salida de DYNALLOY como de ALLOY (tab de *Output*) como a los *watches*.Ver 3.6(a) (página 63).
- En el panel inferior derecho, se puede ver la traza de ejecución del contraejemplo. La misma puede navegarse sobre el árbol, así que usando los botones de *Step Into*, *Step Over* y *Step Return*, los cuales siguen las convenciones de los depuradores de código tradicionales. Ver 3.6(b) (página 63).

Para comparar, en las figuras 3.7 (página 64) y 3.8 (página 65) podemos ver como se ve el mismo contraejemplo usando el visualizador ALLOY, el cual hasta este trabajo era la única alternativa para analizar los contraejemplos obtenidos con la herramienta. Como se puede ver, el visualizador está orientado a la semántica de ALLOY, poniendo énfasis en permitir el análisis sencillo de objetos y relaciones entre los mismos.

```
module traceability

one sig null {}

pred isNull[u:univ] {
  u=null
}

pred isNotNull[u:univ] {
  u!=null
}

pred TruePred[] {}

sig List {}

sig Node {}

program goLast[thiz: List,
               head: List->one(Node+null),
               next: Node->one(Node+null)]
var [curr: Node+null]
{
   curr := thiz.head;
   repeat {
      assume isNotNull[curr];
      curr := curr.next
   };
   assume isNull[curr]
}

assertCorrectness assertGoLast[thiz: List,
                               head: List->one(Node+null),
                               next: Node->one(Node+null)] {
  pre = { isNotNull[thiz] }
  program = { call goLast[thiz,head,next] }
  post = { isNull[thiz] }
}

check assertGoLast
```

Listado 3.3: Ejemplo DynAlloy - Lista

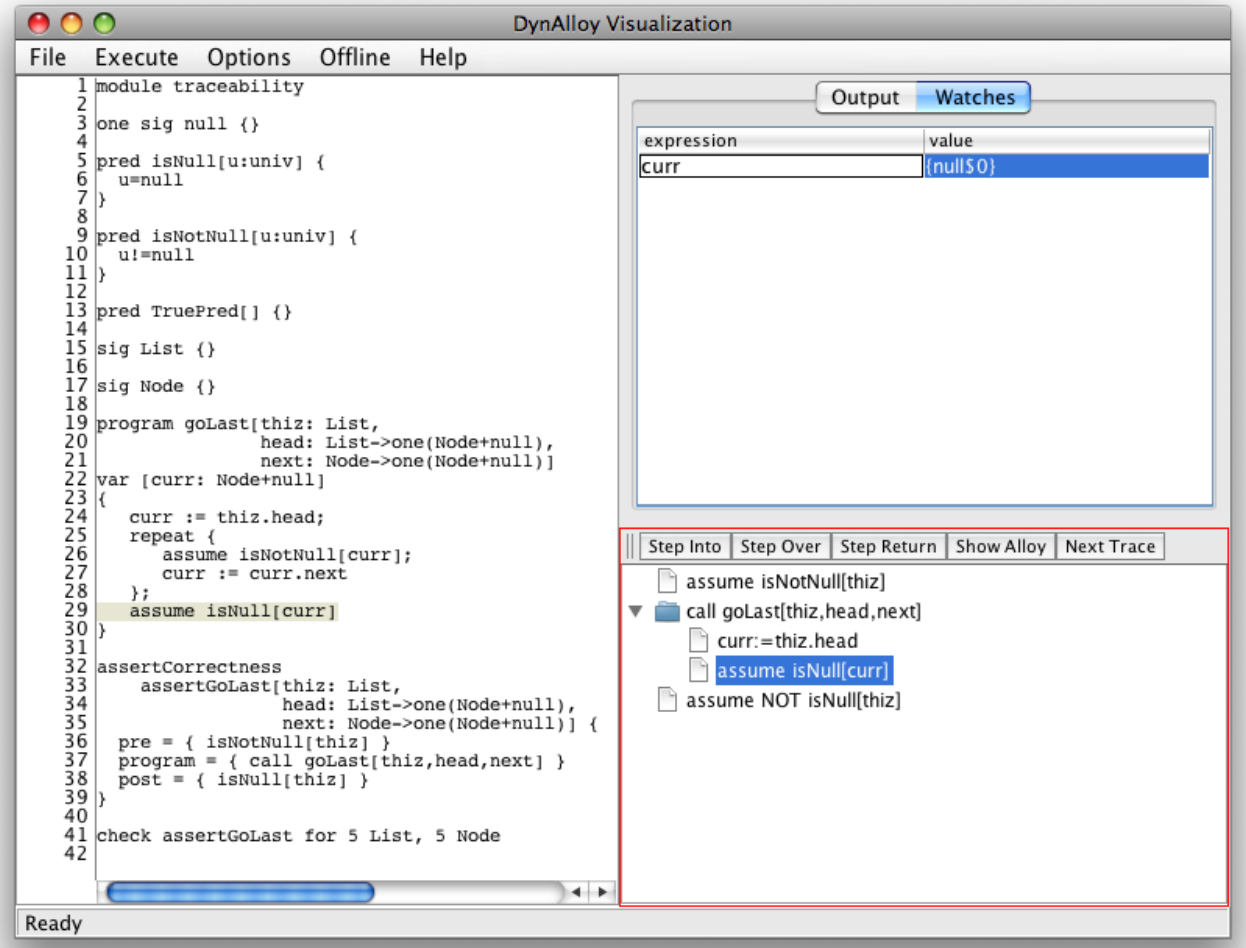
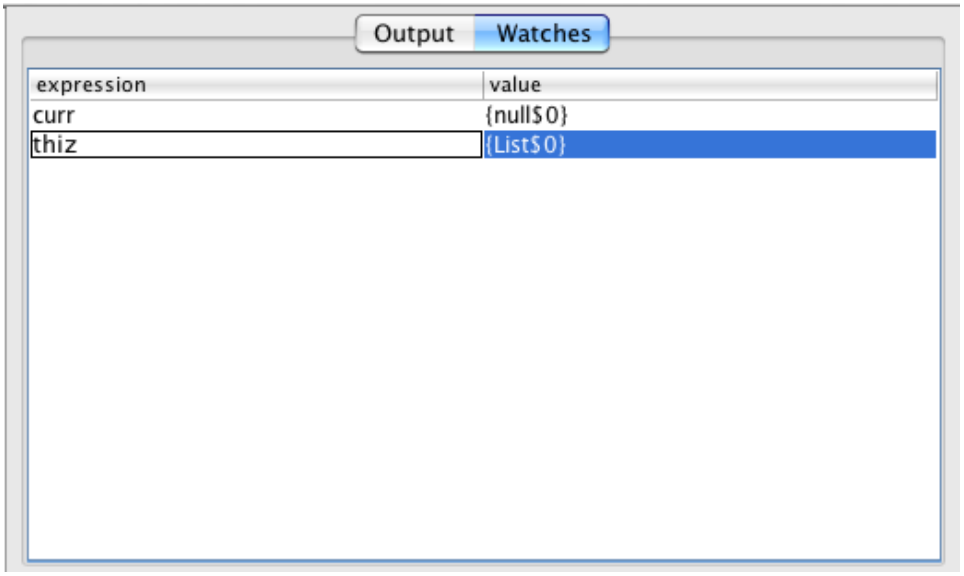
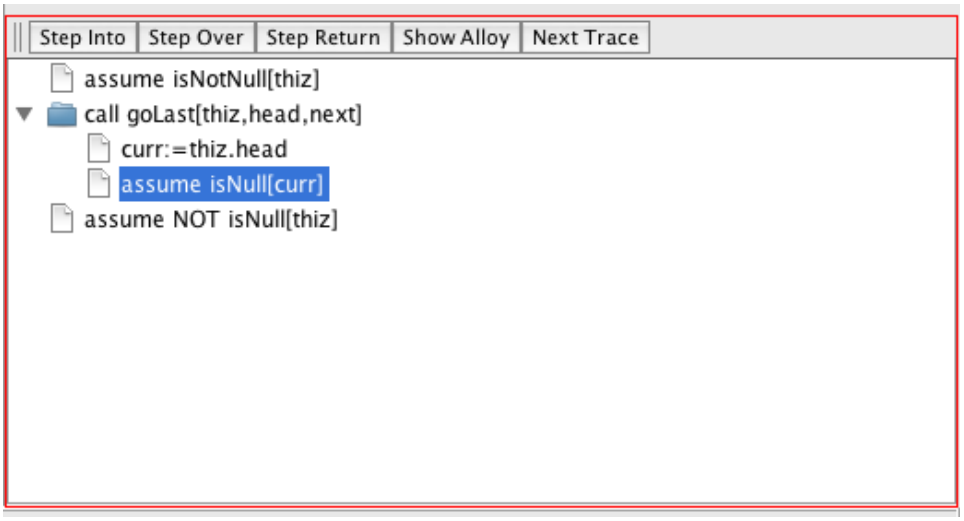


Figura 3.5: Visualizador DynAlloy



(a) Detalle de los watches



(b) Detalle de la traza de ejecución

Figura 3.6: Detalle del Visualizador DynAlloy

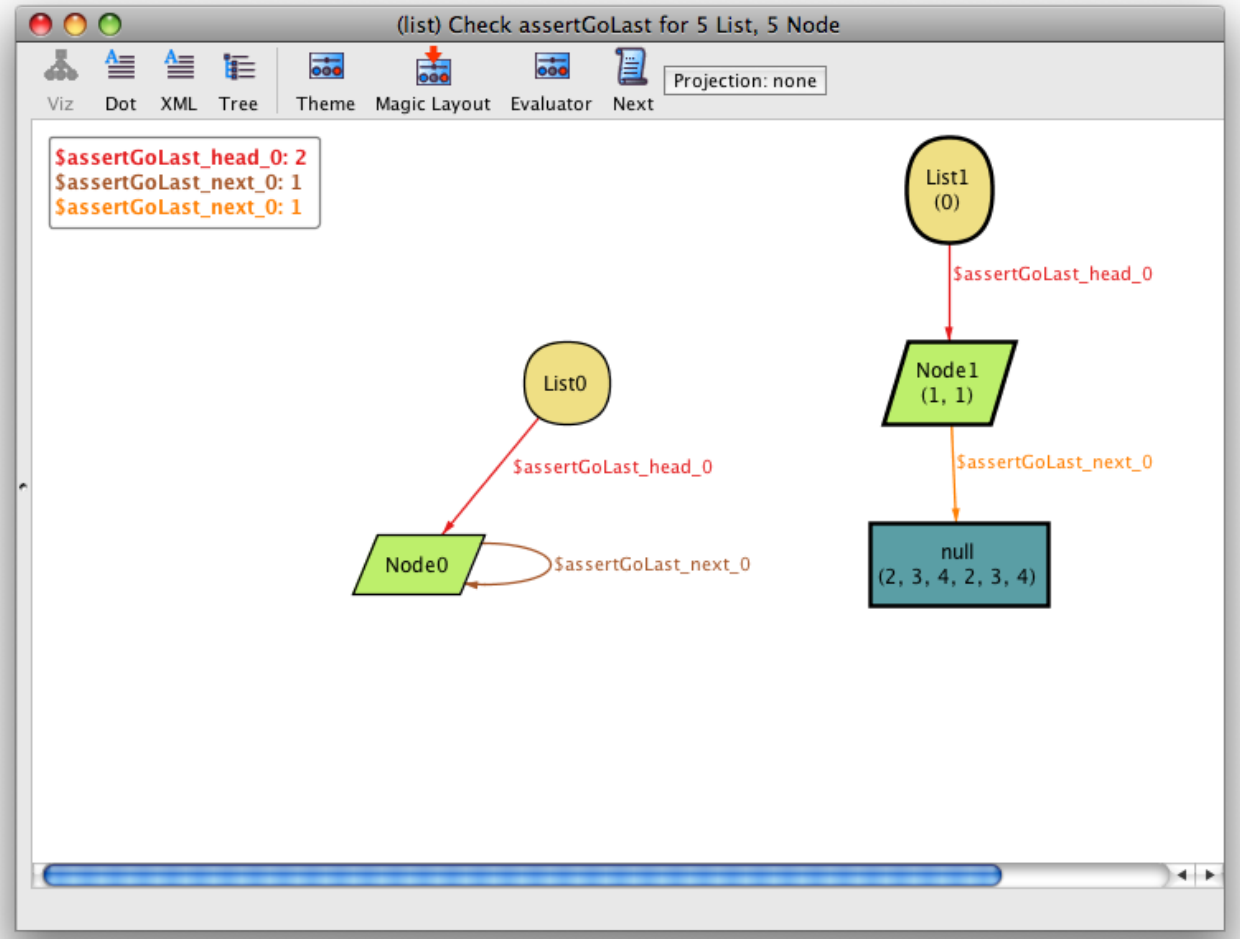


Figura 3.7: Visualizador Alloy

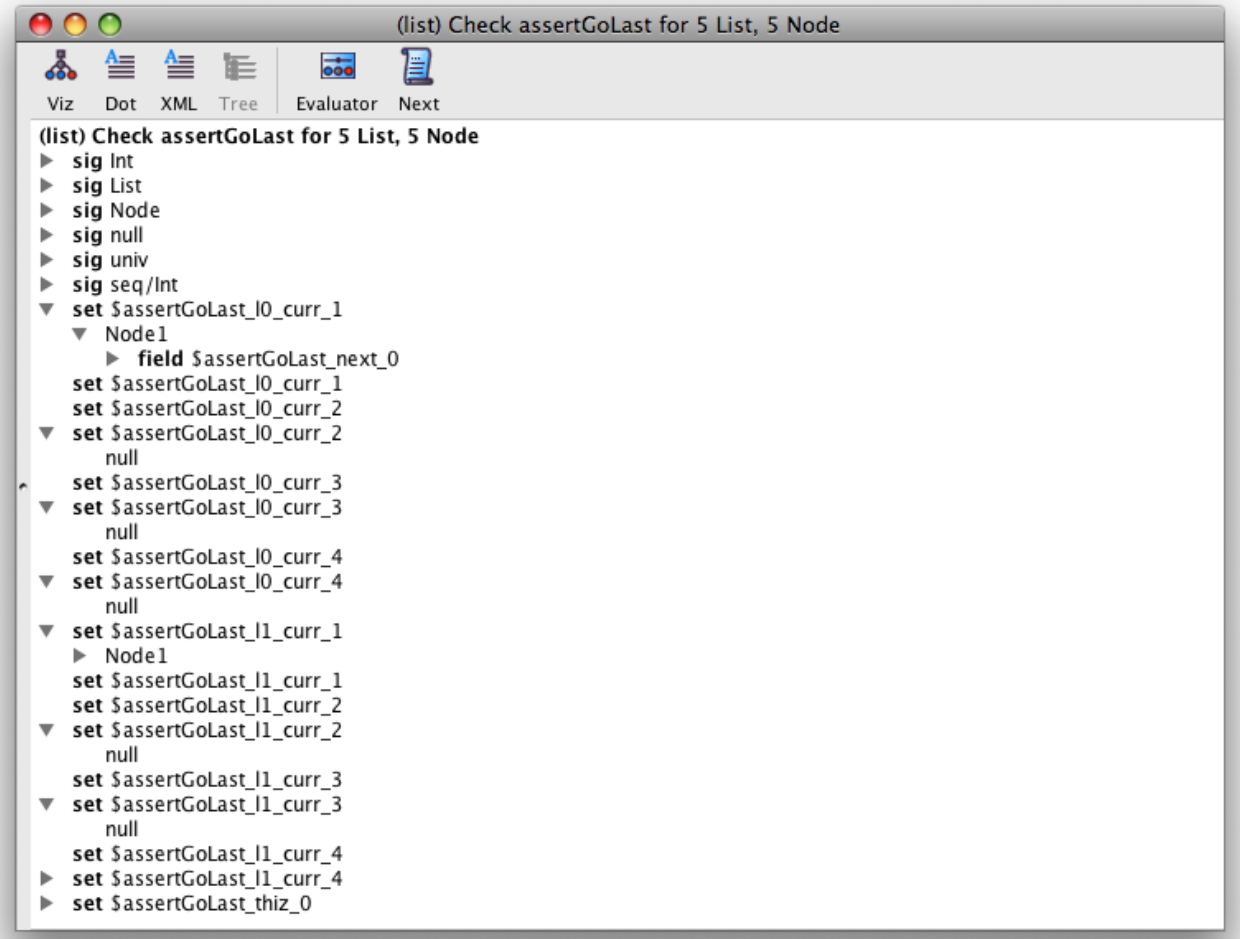


Figura 3.8: Visualizador Alloy - Vista Tree

Capítulo 4

Tutorial de uso

Introducción

En este capítulo se mostrarán todas las mejoras realizadas a TACO a través de un ejemplo de uso. El ejemplo abarcará el uso del visualizador de contraejemplos, así como la generación de *VC* utilizando invariantes de ciclo.

Programa inicial

Para este tutorial, utilizaremos el programa LinearSearch, que fue utilizado como parte del conjunto de experimentos.

En el listado 4.1 (página 67) podemos ver el código JAVA original, con las anotaciones JML para representar el contrato del método.

El método `search` recibe como parámetros un arreglo enteros (`list`) y un entero a buscar (`element`). El contrato especifica que el método retorna -1 si `element` no es encontrado, o la posición dentro del arreglo en caso contrario.

Primer análisis del programa

Realizamos el primer análisis del programa utilizando TACO, para el *scope* y número de *unrolls* por defecto de tres. Esto generará las especificaciones JDYNALLOY y DYNALLOY correspondientes a este programa.

En el listado 4.2 (página 68) podemos ver la parte más relevante de la especificación DYNALLOY, correspondiente al método `search` del programa original.

Al ejecutar el programa con el analizador, encontramos que el mismo encuentra un contraejemplo.

```
1  package samples;
2
3  /**
4   * @j2daType
5   *
6   */
7  public class LinearSearch extends Object {
8
9      /**
10      * @j2daMethod
11      *
12      */
13     /*@
14       @ ensures
15       @    \result < list.length;
16       @ ensures
17       @    (\result >= 0 && \result < list.length ==>
18       @           list[\result] == element);
19       @*/
20     public static int search(int[] list, int element) {
21         int retValue;
22         int i;
23         retValue = -1;
24         i = 1;
25         while (i < list.length - 1 && list[i] != element) {
26             i = i + 1;
27         }
28
29         if (i < list.length) {
30             retValue = i;
31         }
32
33         return retValue;
34     }
35 }
```

Listado 4.1: Programa LinearSearch

```
program LinearSearch_search_0[
  throw:Throwable+null,
  return:Int,
  list:SystemArray,
  element:Int
] var [
  retValue:Int,
  i:Int
]{
   throw:=null;
   skip;
   skip;
   retValue:=negate[1];
   i:=0;
   repeat {
      assume LinearSearchCondition0[Object_Array,element,i,list];
      i:=add[i,1]
   };
   assume LinearSearchCondition1[Object_Array,element,i,list];
   if LinearSearchCondition2[Object_Array,i,list] {
      retValue:=i
   };
   return:=retValue
}
```

Listado 4.2: Especificación DynAlloy de LinearSearch

Utilizando el visualizador de contraejemplos, podemos analizar la traza de ejecución producto del contraejemplo, para poder encontrar el error en el programa original.

En la figura 4.1 (página 72) podemos ver este análisis.

Lo que podemos observar es que el valor buscado (variable `element`) es 6, mientras que el arreglo donde se está buscando es `[5, -6, 5]`. A pesar de que el elemento no es parte del arreglo, la traza de ejecución está pasando por la línea 610 de la especificación DynAlloy, como podemos ver en la figura.

El único caso donde el programa debiera entrar a ese `if` es si se encontró el elemento en el arreglo, pero en este caso la variable `i` vale 2, indicando que el arreglo no fue revisado completamente.

Si revisamos la guarda del `while` original, podemos ver el error:

```
25        while (i < list.length - 1 && list[i] != element) {
```

En lugar de estar recorriendo el arreglo completo, se está omitiendo el último elemento del mismo.

La corrección es trivial, y el `while` corregido queda:

```
25        while (i < list.length && list[i] != element) {
```

Ejecutando el análisis nuevamente corroboramos que no se encuentran contraejemplos.

Incorporación del invariante de ciclo

Ahora que el programa original funciona, y podemos verificarlo para diferentes valores de *loop unroll* (fijado al momento de la traducción), lo que deseamos hacer es analizar el mismo programa para una cantidad no indicada de ejercicios del ciclo.

El primer paso es incorporar el invariante al archivo Java, utilizando la anotación `@loop_invariant` de JML.

A continuación vemos como se incorpora el invariante de ciclo al programa original:

```
25  /*@
26    @ loop_invariant
27    @     i > 0 && i <= list.length &&
28    @     (\forall int j; j >= 0 && j < i; list[j] != element);
29    @*/
30  while (i < list.length && list[i] != element) {
```

Verificación del invariante de ciclo

Lo que nos interesa ahora es verificar que el invariante de ciclo escrito sea correcto, es decir que cumpla el teorema del invariante.

Para ello, podemos usar la generación de *VC* que verifica la validez del mismo, y luego realizar un análisis en busca de contraejemplos.

En el listado 4.3 (página 71) podemos ver la parte relevante de la especificación DynAlloy donde se realiza la verificación del invariante de ciclo de acuerdo a la traducción presentada en la tabla 2.2 (página 24).

Al ejecutar la verificación, como podemos ver en la figura 4.2 (página 74), encontramos nuevamente un contraejemplo. Al revisarlo usando el visualizador, notamos que uno de los `asserts` no es verdadero. En particular, se trata del `assert INV` que verifica que el invariante es válido a la entrada del ciclo.

De aquí podemos concluir que el invariante del ciclo que fue escrito, no respeta el teorema del invariante, y debemos arreglarlo.

Revisando el invariante de ciclo:

```
25 /*@
26   @ loop_invariant
27   @     i > 0 && i <= list.length &&
28   @     (\forall int j; j >= 0 && j < i; list[j] != element);
29   @*/
30 while (i < list.length && list[i] != element) {
```

vemos el error cometido. En la entrada del ciclo $i == 0$ y por lo tanto no valdrá la condición $i > 0$, la cual debiera ser $i >= 0$.

Corregimos el invariante de ciclo de tal forma de escribir:

```
25 /*@
26   @ loop_invariant
27   @     i >= 0 && i <= list.length &&
28   @     (\forall int j; j >= 0 && j < i; list[j] != element);
29   @*/
30 while (i < list.length && list[i] != element) {
```

Con esta modificación, el análisis no encuentra contraejemplos, por lo que podemos concluir que el invariante de ciclo es válido dentro del scope de análisis utilizado.

```
program LinearSearch_search_0[
  throw:Throwable+null,
  return:Int,
  list:SystemArray,
  element:Int
] var [
  assertionFailure:boolean,
  retValue:Int,
  i:Int
]{
   assertionFailure:=false;
   throw:=null;
   retValue:=negate[1];
   i:=0;
   if LinearSearchCondition0[Object_Array,element,i,list] {
      skip
   } else {
      assertionFailure:=true
   };
   havocVariable[i];
   assume LinearSearchCondition0[Object_Array,element,i,list];
   if LinearSearchCondition1[Object_Array,element,i,list] {
      i:=add[i,1];
      if LinearSearchCondition0[Object_Array,element,i,list] {
         skip
      } else {
         assertionFailure:=true
      }
   };
   assume LinearSearchCondition2[Object_Array,element,i,list];
   if LinearSearchCondition3[Object_Array,i,list] {
      retValue:=i
   };
   return:=retValue;
   if equ[assertionFailure,true] {
      throw:=AssertionFailure
   }
}
```

Listado 4.3: Especificación DynAlloy con verificación de invariante

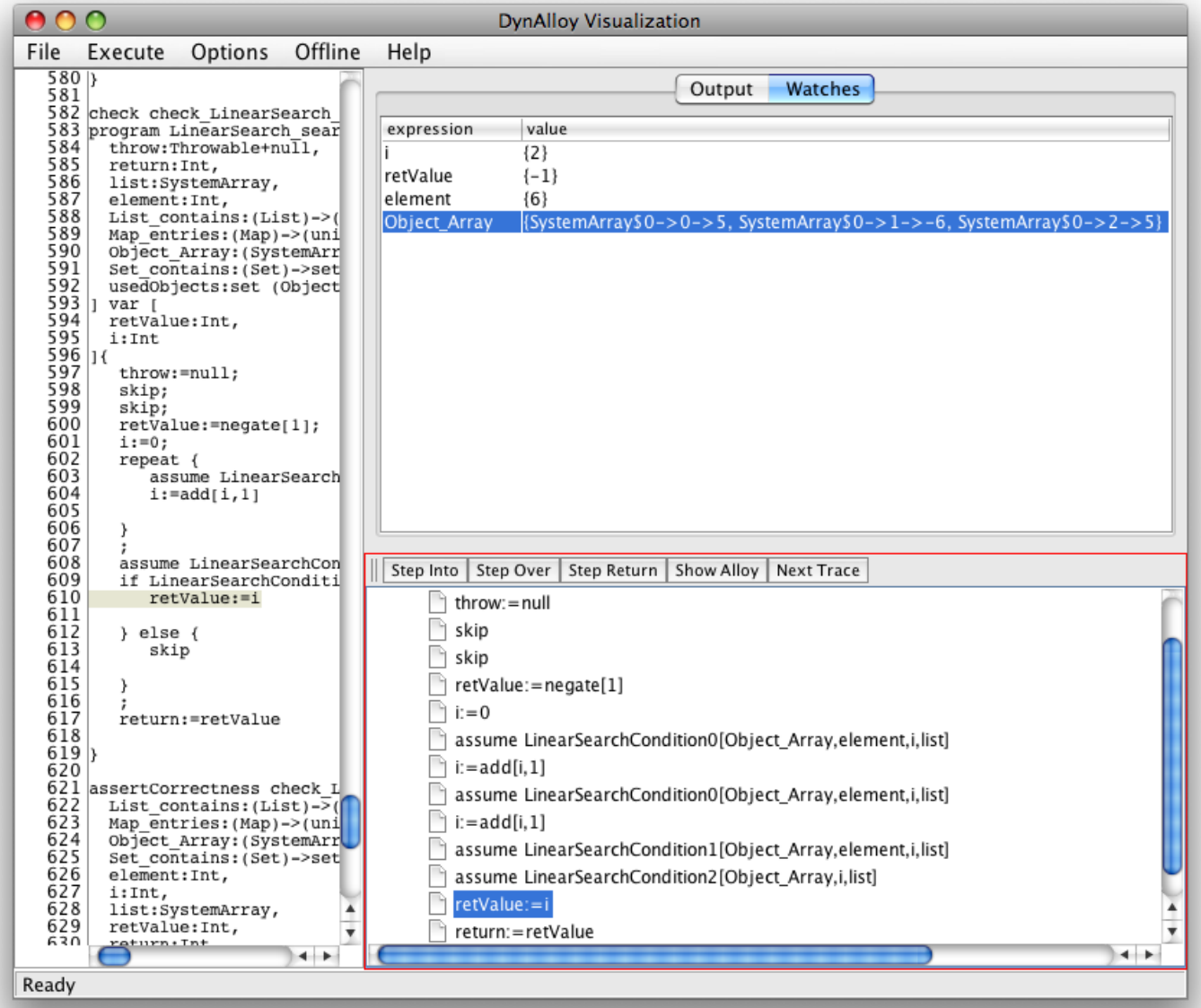


Figura 4.1: Primer análisis con el Visualizador DynAlloy

VC usando solo el invariante de ciclo

Ahora que tenemos mayor certeza de que el invariante de ciclo es válido, podemos realizar un análisis usando solo el invariante de ciclo, y evitar así el costo de verificar las hipótesis del teorema del invariante. Este análisis debiera resultar en una CNF más pequeña, y en una mejora de performance.

Como se prevee, esta nueva verificación no encuentra contraejemplos. A continuación, se consignan algunas variables que arroja la herramienta y que miden el tamaño de la fórmula CNF para los dos tipos de análisis usando invariantes de ciclo:

	Con verificación	Sin verificación
Variables	4478	3864
Variables Principales	391	365
Cláusulas	11480	9370

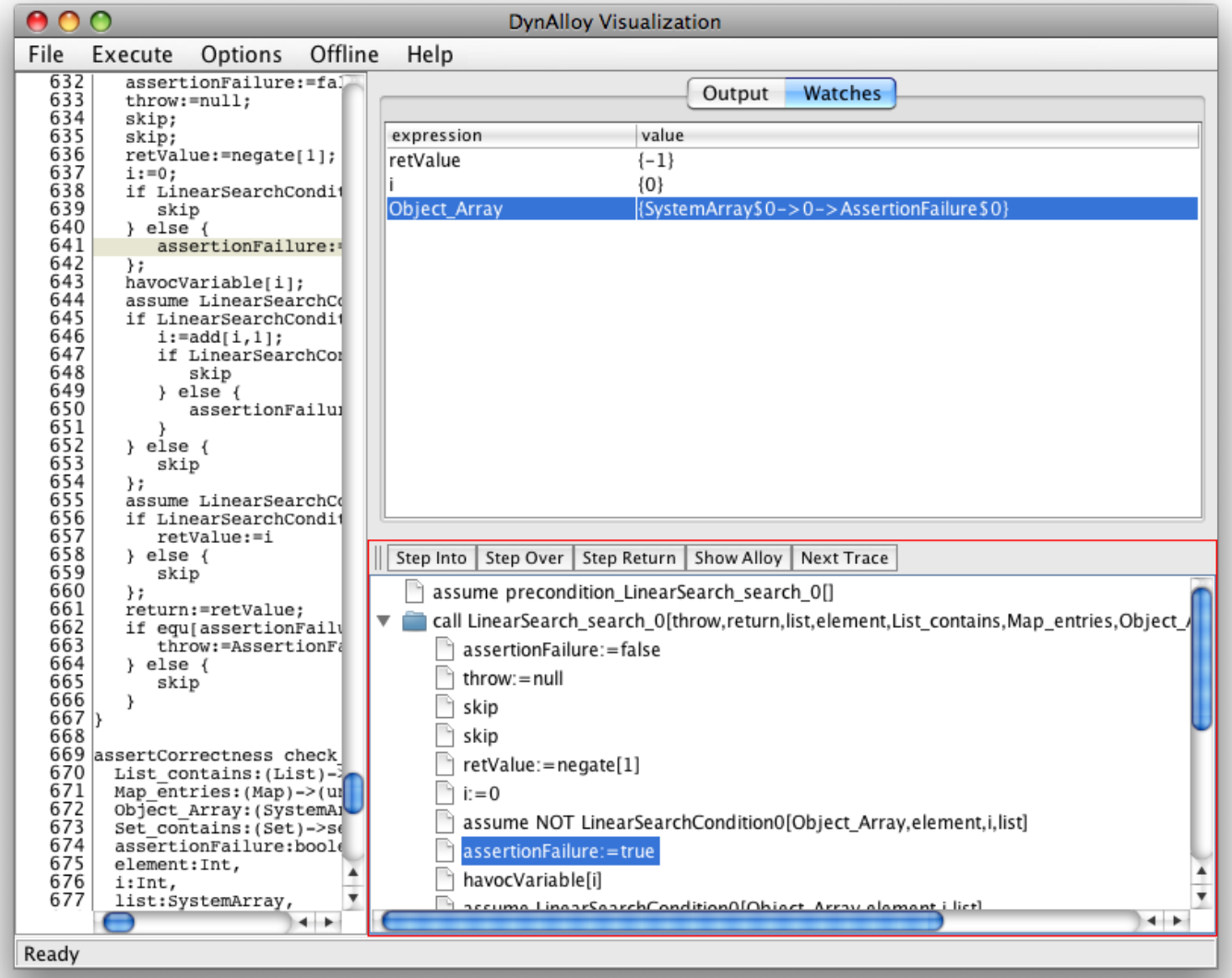


Figura 4.2: Análisis del invariante de ciclo con el visualizador DynAlloy

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

TACO es una herramienta valiosa para encontrar errores en programas JAVA, a través de la traducción de los mismos a una especificación ALLOY y su posterior verificación.

Una de las limitaciones que se identificaron en la herramienta, fue el uso de *loop unrolling* como única técnica para analizar programas JAVA con ciclos.

En el capítulo 2 se presentaron dos alternativas de generación de *VC* para hacer de TACO una herramienta más escalable.

En la primera, se genera una *VC* que verifica la validez del teorema del invariante, haciendo de TACO una herramienta que no solo pueda buscar errores en los programas, sino que pueda encontrar fallas en la especificación de invariantes de ciclo.

Luego, se introdujo una segunda alternativa de generación de *VC* que utiliza solamente el invariante de ciclo, haciendo que la especificación ALLOY resultante no dependa del cuerpo del ciclo. Esta variante de análisis mejora la *performance* (como se pudo ver en la experimentación), pero presenta la importante desventaja de ser *unsafe*, haciendo que en algunos casos donde el programa tiene errores, no se encuentren contraejemplos.

En el capítulo 3 se presenta una herramienta para visualizar los contraejemplos producidos durante el análisis sobre la especificación DYNALLOY.

Sin este visualizador, la interpretación de los contraejemplos obtenidos requiere que el usuario tenga un conocimiento muy profundo de como el Traductor DYNALLOY traduce los programas, y la correlación de los nombres

de variable que genera. Además, el análisis de los contraejemplos utilizando el visualizador y evaluador provisto por Alloy no se adapta al análisis de una traza de ejecución, sino al análisis de un modelo de conjuntos y relaciones.

El visualizador de contraejemplos en DynAlloy contribuye a mejorar la usabilidad de TACO, haciendo más fácil de usar la herramienta para usuarios nuevos, y no requiriendo conocer el lenguaje Alloy ni su evaluador de expresiones.

5.2. Trabajo futuro

A partir de este trabajo, se identifican algunas posibles mejoras de la herramienta, que pueden ser tenidas en cuenta para trabajos futuros.

- Extender el visualizador al lenguaje Java. Aplicando la misma idea con la que se llevó a cabo el visualizador DynAlloy, se podría llegar a visualizar los contraejemplos sobre el programa Java original, simplificando aún más el uso de la herramienta.
- Incorporar más soporte de JML. Como se identificó en la sección 2.7, se encontraron dos limitaciones importantes en el soporte de JML. El soporte de invocación de métodos *puros* y el soporte de pre-estado. Ambas limitaciones pueden ser revisadas en el futuro.
- Representación de arreglos Java en Alloy. Dentro de las pruebas realizadas, notamos que aquellos programas que modifican un arreglo son los de peor performance. Esto se debe a la representación que se hace en Alloy de los arreglos en Java. Se podría trabajar en buscar una representación alternativa, que no tenga tanto impacto en la performance.
- Incorporación de inferencia de invariantes. Teniendo el soporte de análisis utilizando invariantes de ciclo, se puede incorporar una herramienta de inferencia de invariantes, para tratar de automatizar aún más el análisis.

Agradecimientos

Este trabajo, la carrera y tantas otras cosas no hubieran sido posibles sin la ayuda de mucha gente. Quiero agradecer:

- a Juan Pablo y Diego, por haberme dirigido a lo largo de este trabajo, aportando ideas, comentarios y críticas para poder llegar a buen puerto.
- a mis viejos, Eduardo y Violeta, por todo lo que hacen e hicieron por mí, y por todo el interés que ponen en mis cosas.
- a mis hermanos, Ariel y Lila, por todo.
- a Kari, por todo el amor, el apoyo y la paciencia que me dio en los últimos 10 años.
- a mis abuelos, Samuel, Ana, Moishe y Minda, por todos los buenos recuerdos.
- a mis amigos de siempre: Axel, Fabian, Ale y Martita, quienes siempre están a pesar de la distancia.
- a mis amigos de la facu, sin los cuales no hubiera podido disfrutar tanto de la cursada, en especial a Juan, Froma, Lito, Herny, Javier y Roxana.
- a Javier, por su ayuda en el trabajo para que pueda terminar la tesis.

Índice de tablas

Índice de figuras

Índice de listados

Bibliografía

[1] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, p. 8, ACM, 2007.

[2] T. Ball, B. Cook, V. Levin, and S. Rajamani, "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," *Lecture notes in computer science*, vol. 2999, pp. 1–20, 2004.

[3] B. Meyer, "Applying 'design by contract'," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[4] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[5] J. Spivey, *The Z notation: a reference manual*. 1992.

[6] J. Galeotti and M. Frias, "DynAlloy as a formal method for the analysis of Java programs," *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING-PUBLICATIONS-IFIP*, vol. 227, p. 249, 2006.

[7] D. Jackson, "Automating first-order relational logic," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 130–139, 2000.

[8] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre, "DynAlloy: upgrading Alloy with actions," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, (New York, NY, USA), pp. 442–451, ACM, 2005.

[9] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic logic*. The MIT Press, 2000.

[10] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[11] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[12] G. Leavens and Y. Cheon, "Design by Contract with JML," *Draft, available from jmlspecs. org*, vol. 1, p. 4, 2005.

[13] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. M üller, J. Kiniry, P. Chalin, and D. Zimmerman, "JML reference manual," *Draft), April 2003*.

[14] S. Khurshid, D. Marinov, and D. Jackson, "An analyzable annotation language," *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 231–245, 2002.

[15] P. Chalin, J. Kiniry, G. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," *Lecture Notes in Computer Science*, vol. 4111, p. 342, 2006.

[16] G. Dennis, F. Chang, and D. Jackson, "Modular verification of code with SAT," in *Proceedings of the 2006 international symposium on Software testing and analysis*, p. 120, ACM, 2006.

[17] M. Barnett, K. Leino, and W. Schulte, "The Spec# programming system: An overview," *Lecture Notes in Computer Science*, vol. 3362, pp. 49–69, 2005.

[18] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. Leino, "Boogie: A modular reusable verifier for object-oriented programs," *Lecture Notes in Computer Science*, vol. 4111, p. 364, 2006.

[19] "Forge - Bounded Program Verification." `http://sdg.csail.mit.edu/forge/`.

[20] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2007.

[21] S. Lahiri, S. Qadeer, J. Galeotti, J. Voung, and T. Wies, "Intra-module inference," in *Proceedings of the 21st International Conference on Computer Aided Verification*, p. 508, Springer, 2009.

[22] M. Frias, L. Pombo, G. Carlos, J. Galeotti, and N. Aguirre, "Efficient analysis of dynalloy specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 1, p. 4, 2007.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, MA: Addison-Wesley, January 1995.