Verificación de un sistema de votación usando SAT-Solving

Diego P. Dobniewski Gabriel Gasser Noblia

Director:
Lic. Juan Pablo Galeotti

e-mail: {ddobniew, gnoblia, jgaleotti}@dc.uba.ar

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

Abril 2010

Resumen

Diversos estudios exponen que el mantenimiento del software requiere una gran cantidad de recursos en comparación con el desarrollo del mismo. Además la experiencia sugiere que la detección temprana de errores en las aplicaciones reduce los costos futuros relacionados a la corrección de los mismos.

TACO es una herramienta para realizar verificación formal de programas, detectando de esta manera defectos en los mismos.

A lo largo del desarrollo de esta tesis introducimos el lenguaje de representación intermedia JDynAlloy. Modificamos TACO para que realice la traducción de JML a JDynAlloy y de JDynAlloy a DynAlloy.

Analizamos un caso de estudio con la nueva versión de la herramienta TACO y con la herramienta JMLFoge. Finalmente realizamos conclusiones sobre el comportamiento de TACO.

Agradecimientos

Mucha Gente merece nuestra gratitud por hacer posible esta tesis.

Primero a nuestro director Juan Pablo Galeotti por todos sus consejos, su ayuda, por alentarnos a trabajar fuerte en esta tesis y principalmente por su increíble dedicación.

A Greg Dennis y Kuat Yessenov del MIT por facilitarnos los códigos de KOA y por su ayuda al intentar reproducir el experimento de verificar KOA con JMLForge.

Diego Dobniewski

- A mi hermano Alejandro, quien me ayudó a elegir esta carrera.
- A mis viejos por darme la posibilidad de estudiar y alentarme a hacerlo.
- Al amor de mi vida Silvana por su infinita paciencia y colaboración.
- A mi princesita Morella por perdonarme los días que no pude jugar con ella.

Gabriel Gasser Noblia

- A mis padres, los pekos, por enseñarme que sin esfuerzo no puede lograrse nada, especialmente a mi madre quien se esforzó muchos para que yo pueda lograr esto.
- A mi princesa Melina, quien demostró ser una gran mujer en quien puedo apoyarme cada vez que lo necesite.
- A Mariano, mi hermano del alma, y Norma, por alentarme todo el tiempo para terminar esta tesis.
- A Silvana y More que permitieron que Diego pueda dedicar mucho del poco tiempo que teníamos a este trabajo.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Diversos estudios exponen que el mantenimiento del software requiere una gran cantidad de recursos en comparación con el desarrollo del mismo. Además la experiencia sugiere que la detección temprana de errores en las aplicaciones reduce los costos futuros relacionados a la corrección de los mismos [26].

En ciertos casos específicos, donde el software se considera crítico, los costos asociados a corregir dichos defectos pueden mesurarse en millones de dólares, como en el caso del proyecto Arianne 5 [25], o peor aún: poner en riesgo vidas humanas, como en el caso de Therac-25 [22].

Las técnicas de verificación de software prometen reducir drásticamente la cantidad de defectos en el software. El término 'verificación de software' se refiere a todos los mecanismos de verificar la correctitud de un programa con respecto a una especificación[1].

Actualmente existen dos enfoques; uno dinámico, el cual es bueno para la detección de errores (tests de unidad, tests de integración, tests funcionales, etc.) y el otro estático, bueno para proveer correctitud de programas (convenciones de código, detección de malas prácticas, verificación formal, etc.).

JML (ver sección 2.1) es una notación para especificar formalmente el comportamiento e interacciones de las clases y métodos Java. Desde el nacimiento de esta notación diversas herramientas fueron desarrolladas con el propósito de verificar los programas contra su especificación.

[1]Esta especificación puede ser explícita o implícita

JmlForge (ver sección 2.3) es una herramienta desarrollada por el Instituto de Tecnología de Massachusetts (MIT) que es capaz de realizar la verificación estática de código Java contra su especificación en lenguaje JML. Realiza análisis modular (es decir, en los llamados a métodos utiliza la especificación del método llamado) y utiliza la técnica de Ejecución Simbólica para traducir un programa Java anotado con JML a una fórmula lógica.

TACO es una herramienta desarrollada por el Relational Formal Methods Research Group del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires para realizar la verificación formal de programas Java contra su especificación en lenguaje JML. Realiza un análisis whole program (es decir, en los llamado a métodos utiliza la implementación del método llamado en vez de usar su especificación) y Weakest Precondition para transformar un programa a una fórmula lógica.

Ambas herramientas transforman los programas en una fórmula lógica llamada Condición de Verificación y utilizan SAT-Solving para validar o refutar dicha fórmula.

El propósito de este trabajo consiste en realizar la verificación de un software relativamente complejo y contrastar los resultados con los obtenidos por la herramienta JMLForge.

Para ello se eligió el sistema de votación KOA desarrollado por el gobierno holandés en 2003, el cuál ya había sido verificado usando ESC/Java2 [19] y JMLForge.

Dado que la herramienta TACO se encontraba en estado de prototipo, fue necesario volver a implementarla. Se mejoró su arquitectura interna y se extendió el soporte de los lenguajes Java y JML. Además se diseñó un lenguaje de representación intermedia (JDynAlloy) para facilitar la transformación de código Java anotado con JML al lenguaje DynAlloy [14].

1.1. Definiciones previas

Sat-solving: Un problema de decisión es una pregunta expresada en algún sistema formal cuya respuesta es si o no dependiendo del valor de los parámetros de entrada. El problema de satisfactibilidad (SAT) se enmarca dentro de los problemas de decisión, cuyas instancias son expresiones booleanas escritas con paréntesis, variables y los operadores lógicos *and*, *or* y *not*.

La pregunta que debe intentar responderse es si, dada la expresión booleana, existe una asignación de valores $True$ o $False$ a las variables (esto se conoce como valuación) de la expresión que hagan que dicha expresión sea verdadera.

El problema de la satisfactibilidad booleana se encuentra dentro los problemas NP-Completos, lo que en principio sugiere que el tiempo que demanda su resolución puede ser exponencial con respecto al tamaño de la entrada.

Precondición: Es un predicado que debe ser verdadero en el estado previo a la ejecución del método sobre el cual esta definido.

Poscondición: Es un predicado que debe ser verdadero en el estado final de la ejecución del método sobre el cual está definido.

Correctitud parcial El término correctitud se emplea para indicar que un algoritmo es correcto respecto a su especificación. Decimos que un algoritmo es parcialmente correcto si al cumplirse su precondición y, en caso de terminar, se cumple la poscondición. Decimos que el algoritmo es correcto si es parcialmente correcto y además podemos afirmar que el algoritmo termina.

Condición de Verificación Dado un programa con su respectiva fórmula de *precondición* y su fórmula de *poscondición*, llamamos Condición de Verificación (VC) [2] a una fórmula de lógica de primer orden que valida la correctitud parcial de dicho programa.

Loop unrolling Es una técnica que transforma los ciclos, que potencialmente pueden iterar una cantidad no acotada de veces, en sentencias equivalentes a las primeras n iteraciones. Debido a esto la cantidad de iteraciones

[2] VC por sus siglas en ingles: Verification Condition

generadas con *unrolling* pueden ser menor a la cantidad real de iteraciones que realizaría el ciclo [1].

```
1  while(i < max) {
2      f();
3      i++;
4  }
```

Código 1.1: Ciclo original

```
1  if(i < max) {
2      f();
3      i++;
4      if(i < max) {
5          f();
6          i++;
7          if(i < max) {
8              f();
9              i++;
10         }
11     }
12 }
```

Código 1.2: Loop unrolling, con hasta tres iteraciones

1.1.1. Lógica de Hoare

Triplas de Hoare Describen la relación del estado de un programa con respecto a la ejecución de sentencias del mismo y son usadas para razonar acerca de la correctitud de los programas.

Sean A y B predicados y s una sentencia de un programa, decimos que $\{A\}s\{B\}$ es una *Tripla de Hoare*. El predicado A es llamado *precondición*, y el predicado B es llamado *poscondición*.

La tripla $\{A\}s\{B\}$ representa que si se parte de un estado inicial que cumple el predicado *precondición*, al ejecutar s y si la ejecución de s finaliza, el estado luego de la ejecución cumplirá el predicado *poscondición*.

Dado un lenguaje sencillo que definiremos a continuación, utilizaremos lógica de hoare para asignarle semántica.

1. Asignación . . . x := y
2. Skip . . . skip
3. Secuencia . . . P;Q
4. Condicional . . . if B then P else Q endif

5. Ciclo . . . while B do P endwhile

Axioma Skip La sentencia *skip* no modifica el estado.

$$\overline{\{P\}skip\{P\}}$$

Axioma Asignación El estado luego de la ejecución de $x := E$ es equivalente al estado previo pero en donde todas las ocurrencias libres de x son reemplazadas por E.

$$\overline{\{P[E/x]\}x := E\{P\}}$$

Axioma Secuencia La ejecución secuencial de las reglas *s1* y *s2*, es equivalente a que al partir de un estado inicial que cumpla A ejecutar secuencialmente ambas reglas para luego arribar a un estado que cumpla C

$$\frac{\{A\}s1\{B\} \qquad \{B\}s2\{C\}}{\{A\}\ s1;s2\ \{C\}}$$

Axioma IF Si la condición G se cumple entonces *s1* será ejecutado, por lo que la precondición incluirá necesariamente a G. Análogamente para el caso en donde no se cumpla G con *s2*.

$$\frac{\{A \wedge G\}s1\{B\} \qquad \{A \wedge \neg\ G\}s2\{B\}}{\{A\}\ if\ G\ then\ s1\ else\ s2\ endif\ \{B\}}$$

Axioma Ciclo Partiendo de un estado que cumple A en cada iteración del ciclo puede pasar que G sea verdadero o que no lo sea. Si es verdadero se ejecutará el cuerpo del ciclo s y luego el estado resultante cumplirá A. Si G no es verdadero se saldrá del ciclo y el estado resultante cumplirá con B, ya que $(A \wedge \neg G) \Rightarrow B$.

$$\frac{\{A \wedge G\}s\{A\} \qquad (A \wedge \neg G) \Rightarrow B}{\{A\}\ while\ G\ do\ s\ endwhile\ \{B\}}$$

1.1.2. Weakest Precondition

Es una técnica para representar un programa como una fórmula lógica. Para ello se le asigna una semántica particular a un programa. Para más información acerca esta técnica consultar [14].

Idea intuitiva de Weakest Precondition Supongamos que queremos verificar la siguiente tripla de Hoare: $\{x \geq 4 \wedge x < 10 \wedge y < 2\}\ x := x + 1\ \{x \geq 5 \wedge y < 2\}$

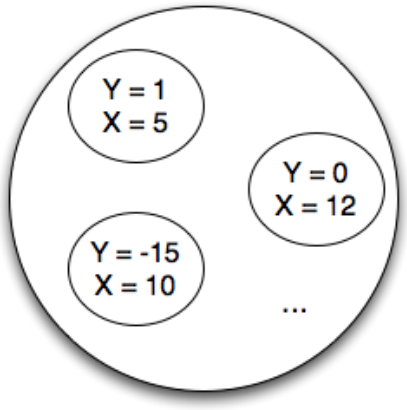

Figura 1.1: Conjunto de estados que cumplen $x \geq 5 \wedge y < 2$

Supongamos que tenemos un estado que cumple con "$x \geq 5 \wedge y < 2$" (ver figura 1.1).

La idea es lograr deducir el conjunto maximal de estados tal que si le aplico el programa *x:=x+1*, y éste termina, cumpla con "$x \geq 5 \wedge y < 2$" (ver Figura 1.2).

Tomo un conjuntos de estados W tal que si para todo estado $E \in W$ aplico el programa *x:=x+1*, obtengo un estado que cumple "$x \geq 5 \wedge y < 2$" (ver Figura 1.3).

Ahora bien, si todo estado que cumple "$x \geq 4 \wedge x < 10 \wedge y < 2$" está en W, entonces podríamos decir que empezando en un estado $E \in W$, ejecutando el programa *x:=x+1*, se llega a un estado que cumple con "$x \geq 5 \wedge y < 2$" (ver Figura 1.4).

Es decir habríamos probado $\{x \geq 4 \wedge x < 10 \wedge y < 2\}\ x := x + 1\ \{x \geq 5 \wedge y < 2\}$. Esto es por que la precondición deducida es más débil que la requerida, ya que $(x \geq 4 \wedge x < 10 \wedge y < 2) \rightarrow (x \geq 4 \wedge y < 2)$ pero no es cierto que $(x \geq 4 \wedge y < 2) \rightarrow (x \geq 4 \wedge x < 10 \wedge y < 2)$.

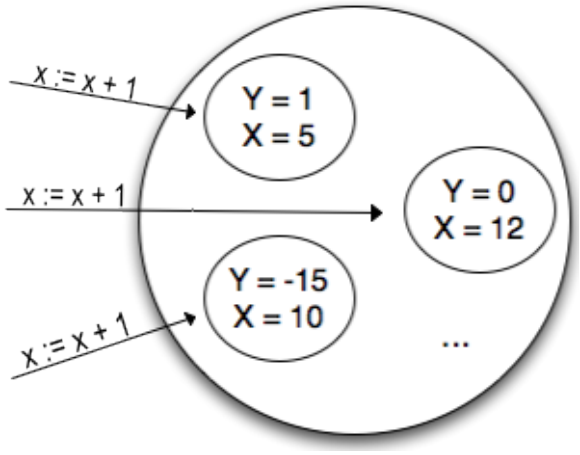

Figura 1.2: Conjunto de estados que estoy buscando

Definición de Weakest Precondition[11]: $WP(Program, Post)$ es una fórmula lógica que permite caracterizar al conjunto maximal de estados tal que si a un estado que cumple $WP(Program, Post)$ se le aplica el programa *Program*, y éste termina, el estado resultante cumpla la fórmula *Post*. $WP(Program, Post)$ se define de la siguiente manera.

- *WP(skip, B) = B*
- *WP(x:=E, B) = B[E/x]*

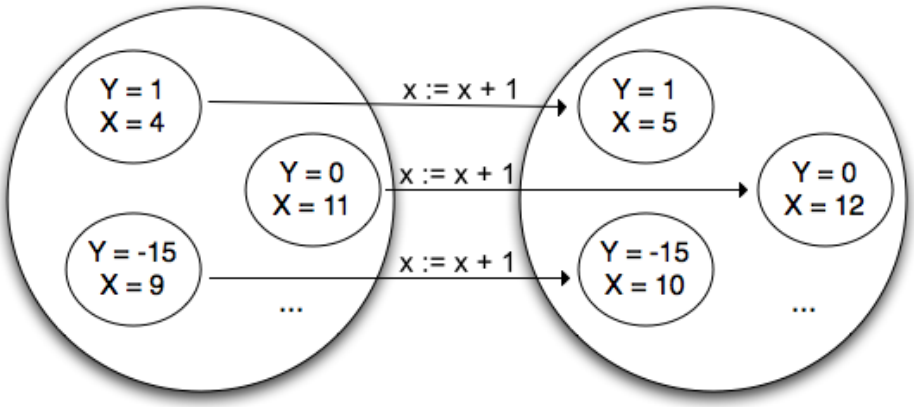

Figura 1.3: Conjunto W

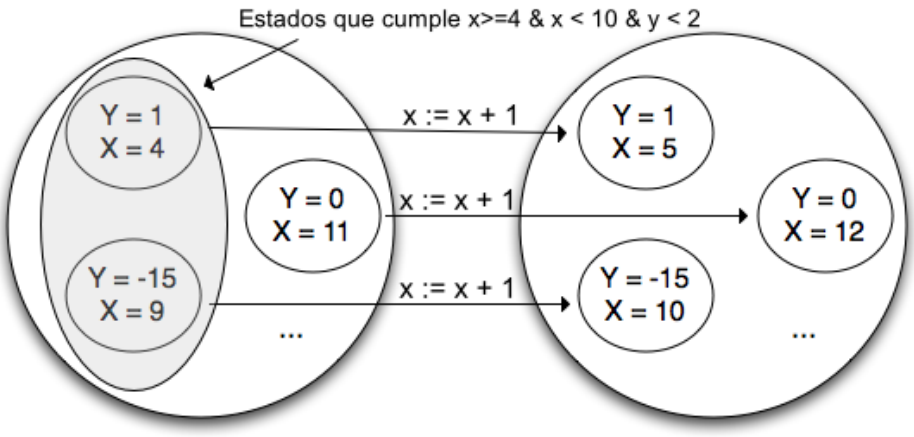

Figura 1.4: Conjuntos de estados que cumplen la precondición

- *WP(s1;s2, B) = WP(s1, WP(s2, B))*
- *WP(if E then s1 else s2 endif, B)* $= E \rightarrow$ *WP(s1,B)* $\wedge \neg E \rightarrow$ *WP(s2,B)*
- *WP(while E do s endwhile,B)* $=$ greater_lower_bound [3] $(\mathrm{WP}_k \mid k \geq 0)$

Donde WP_i: *WP(while E do s endwhile,B)* se define como:

$$\begin{cases} WP_0(S, WP_i()) = \neg E \rightarrow B \\ \\ WP_{i+1}(\text{while E do s endwhile}, B) = WP_i(\text{while E do s endwhile}, B) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad E \rightarrow WP(S, B) \end{cases}$$

Problemas prácticos Esta definición ideal posee problemas concretos que evitan que puedan ser usada en la práctica para calcular la WP de un programa. El cálculo del punto fijo exacto es extremadamente costoso en términos computacionales.

En la práctica se utilizan las técnicas conocidas como *Loop unrolling* y el uso de los invariantes en los ciclos para disminuir el costo computacional de analizar los ciclos. La técnica de los invariantes en los ciclos está fuera del alcance de esta tesis, pero es explicado en [3].

[3] greater_lower_bound es el ínfimo del conjunto

Por otro lado, *loop unrolling* elimina las sentencias *while*, simplificando el cálculo de la Weakest Precondition. Con ésta técnica se pueden reescribir programas sin utilizar ciclos. Todas las ejecuciones descriptas serán válidas por lo que, en el contexto de la verificación de programas, todas las violaciones encontradas serán correctas. Como punto en contra de ésta técnica tenemos que es posible que ciertas ejecuciones se "pierdan" por lo que algunas violaciones pueden no ser encontradas.

Extensiones al lenguaje Para los fines de la verificación de programas es necesario extender nuestro lenguaje con tres nuevas sentencias.

1. *assume* E: Se utiliza para filtrar ciertos caminos de ejecución. Sólo se considerarán los estado en los cuales la valuación del predicado E resulte verdadera. Se requiere en la verificación de la asignación, y para restringir los caminos de ejecución que no cumplan la precondición de un método dado.

2. *assert* E: Verifica que la valuación del predicado E sea verdadera. Para los estados en donde esto no sucede se considera que la verificación falla. Se requiere para verificar el cumplimiento de la poscondición de un método dado.

3. *havoc* x: Le asigna a la variable x un valor arbitrario. En el contexto de esta tesis se requiere en la verificación de ciclos [3].

Como así también debemos extender la semántica del lenguaje para incluir las nuevas sentencias [27].

Axioma Assume

$$\overline{\{E \rightarrow B\}\ assume\ E\{B\}}$$

Axioma Assert

$$\overline{\{E \wedge B\}\ assert\ E\{B\}}$$

Axioma Havoc

$$\overline{\{\forall\ x\ |B\}\ \ havoc\ x\ \{B\}}$$

Finalmente debemos extender la definición de WP, agregando:

1. WP(havoc x, B) = $\forall$ x |B
2. WP(assume E, B) = E $\rightarrow$ B
3. WP(assert E, B) = E $\wedge$ B

Para un programa dado, la VC puede ser generada utilizando la WP, partiendo de la *poscondición* del programa:

Sea P' la reescritura sin ciclos de P, definimos VC como:

$$\text{VC}(P,\textit{poscondición}) = \textit{precondición} \rightarrow \text{WP}(P',\textit{poscondición})$$

Ejemplo de VC Construiremos la VC_{swap} utilizando WP para verificar el código en 1.3.

```
1  REQUIRES X≥0 ∧ Y≥0
2  ENSURES X=Y_old ∧ Y=X_old
3
4  swap(int X, int Y)
5      L1. int T := X;
6      L2. X := Y;
7      L3. Y := T;
```

Código 1.3: Ejemplo WP

Queremos ver cuál es el conjunto de estados iniciales, tal que al ejecutar el programa, y si éste finaliza, el estado final cumpla la poscondición.

$$WP(L1; L2; L3) = WP(L1, WP(L2, WP(L3, X = Y_{old} \wedge Y = X_{old}))) \tag{1.1}$$

X_{old} e Y_{old} es una forma de referirse al valor inicial de los parámetros X e Y. Llamemos a dichos valores X_0 e Y_0 respectivamente. Sin pérdida de generalidad podemos asumir como parte de la precondición X=X_0 e Y=Y_0, que no significa otra cosa que X e Y en el estado inicial poseen su valor inicial. Luego tenemos que:

$$WP(L1; L2; L3) = WP(L1, WP(L2, WP(L3, X = Y_0 \wedge Y = X_0 \wedge Y = X_0))) \tag{1.2}$$

resolvamos L3 primero, ya que es la ultima sentencia en la secuencia.

$$WP(Y := T, X = Y_0 \wedge Y = X_0) = (X = Y_0 \wedge Y = X_0)[T/Y] \equiv X = Y_0 \wedge T = X_0 \quad (1.3)$$

Ahora tenemos que:

$$WP(L1; L2; L3, X = Y_0 \wedge T = X_0) \equiv WP(L1, WP(L2, X = Y_0 \wedge T = X_0)). \quad (1.4)$$

por lo que debemos resolver $WP(L2, X = Y_0 \wedge T = X_0))$.

$$WP(X := Y, X = Y_0 \wedge T = X_0)) = (X = Y_0 \wedge T = X_0)[Y/X] \equiv Y = Y_0 \wedge T = X_0. \quad (1.5)$$

Ahora bien como:

$$WP(L1, WP(L2, X = Y_0 \wedge T = X_0)) \equiv WP(L1, Y = Y_0 \wedge T = X_0) \quad (1.6)$$

$$WP(T := X, Y = Y_0 \wedge T = X_0) = (Y = Y_0 \wedge T = X_0)[X/T] \equiv Y = Y_0 \wedge X = X_0 \quad (1.7)$$

definimos VC_{swap} como:

$$VC_{swap} \equiv precondición \rightarrow WP \quad (1.8)$$

entonces finalmente obtenemos:

$$VC_{swap} \equiv (X \geq 0 \wedge Y \geq 0 \wedge Y = Y_0 \wedge X = X_0) \rightarrow (Y = Y_0 \wedge X = X_0) \quad (1.9)$$

Como puede verse en la ecuación (1.9), VC_{swap} es teorema, por lo que el programa es correcto.

Esta técnica, sin embargo, no se encuentra exenta de problemas. El tamaño de la VC puede ser exponencial con respecto al tamaño del programa [13].

En el código 1.4 se muestra un ejemplo de código con una sentencia *if*. Su correspondiente VC puede verse en la ecuación 1.10.

$$\begin{aligned} VC \equiv & (X + Y > 0 \wedge X = X_0 \wedge Y = Y_0) \rightarrow \\ & ((X > Y \rightarrow Y + 1 > Y) \wedge (X \leq Y \rightarrow X + 1 > X)) \end{aligned} \quad (1.10)$$

```
1  REQUIRES x + y > 0;
2  ENSURES x > y;
3  example(int x,int y) {
4      if (x>y) {
5          x := y;
6      } else {
7          y := x;
8      }
9      x := x + 1;
10 }
```

Código 1.4: Ejemplo WP con bifurcación

Los lenguajes de programación modernos incluyen sentencias para el manejo de excepciones. En el fondo estas sentencias esconden la generación de código no estructurado. La generación de las condiciones de verificación para ese código no estructurado es una dificultad que debe ser solucionada.

En la subsección 1.4 veremos una alternativa al método de WP para la generación de la VC.

1.2. Lenguaje Alloy

Alloy[17] es un lenguaje relacional de especificación formal. Se dice que es relacional porque su sistema de tipos y expresiones está basado en relaciones. La sintaxis de Alloy permite expresar tipos abstractos de datos utilizando signaturas, así como también es posible agregar restricciones sobre dichas signaturas. Ciertas funcionalidades de Alloy se asemejan a características de los lenguajes orientados a objetos.

```
1  sig Data { }
2
3  sig List {
4      val : lone Data,
5      next: lone List
6  }
7
8  one sig Empty extends List {}
9  sig TwoList extends List { val2: Data }
10
11 assert ToEmpty {
12     all l: List | l != Empty implies Empty in l.^next
13 }
```

Código 1.5: Ejemplo DynAlloy

En el código 1.5 se define una signatura *List* en la línea 3. La signatura *List* posee un field llamado *val* (línea 4) y otro llamado *next* (línea 5). En Alloy los fields son funciones que mapean una instancia, o átomo, de una signatura con una instancia del tipo del field. El field *val* relaciona un nodo de la lista con su valor, un átomo del conjunto *Data*. La keyword *lone* en la definición del field indica que la función es parcial. El field *next* permite relacionar un nodo de la lista con el siguiente. El símbolo ^ (línea 12) representa en Alloy la clausura transitiva no reflexiva.

Las signaturas *Empty* (línea 8) y *TwoList* (línea 9) definen subconjuntos disjuntos de *List*. La keyword *one* (línea 8) declara que *Empty* es un conjunto *singleton*; es decir, sólo existirá un átomo dentro de dicho conjunto. En la línea 11 se define la aserción *ToEmpty* que será válida si toda lista es vacía o finaliza con una lista vacía. *Alloy Analyzer* permite realizar la verificación automática de dicha aserción dentro de un *scope* acotado.

Por ejemplo el siguiente comando verificará que la aserción sea verdadera utilizando todas las combinaciones posibles de conjuntos (signaturas) que posean hasta 5 elementos:

```
check ToEmpty for 5
```

Si el *Alloy Analyzer* encuentra un contraejemplo lo presenta ya sea en forma de texto o gráficamente, e indica que la aserción no es válida.

1.3. Lenguaje DynAlloy

DynAlloy es una extensión de Alloy que permite definir acciones atómicas que modifican el estado. Además permite utilizar esas acciones para construir acciones más complejas. Las acciones atómicas son definidas en términos de su precondición y poscondición [14]. En el código 1.6 se puede ver un ejemplo de definición de una acción atómica, mientras que en el cuadro 1.1 se muestra la sintaxis que se puede utilizar para construir acciones complejas.

```
1 action Head[l : List, d : Data] {
2     pre { l != Empty }
3     post { d' = l.val }
4 }
```

Código 1.6: Acción atómica

DynAlloy toma el sistema de tipos, expresiones y fórmulas de Alloy. DynAlloy incorpora la habilidad de especificar y chequear automáticamente

aserciones de correctitud parcial, para lo cual incorpora la keyword *assertCorrectness*. Para hacerlo transforma los programas en fórmulas utilizando la técnica *weakest precondition* vista en la sección 1.1.2.

act	::=	
	a[$param_1$, ..., $param_n$]	atomic action call
	| assume formula	test
	| if formula {act} else {act}	choise
	| act;act	sequential composition
	| repeat {act}	iteration
	| call p[$param_1$, ..., $param_n$]	program call

Cuadro 1.1: Acciones DynAlloy

1.4. Ejecución simbólica

La Ejecución Simbólica o Symbolic Execution (SE) originalmente fue ideada para realizar testeo de programas [18], pero en la actualidad se utiliza también para generar la VC de un programa, como puede verse en [9].

Normalmente al realizar una ejecución de un programa debemos asignarle ciertos valores a sus argumentos. SE propone utilizar símbolos que reemplacen a dichos valores concretos y realizar la ejecución simbólica, tal como si se tratase de una ejecución ordinaria.

```
1 int sum(int x,int y) {
2     int r = x + y;
3     return r;
4 }
```

Código 1.7: Ejecución concreta

Tomemos el ejemplo que figura en el Código 1.7, y realicemos el siguiente llamado: **sum**(i_0,i_1). Nótese que hemos utilizado símbolos que representan enteros, en vez de literales enteros. El resultado intuitivo del llamado es la expresión $i_0 + i_1$. Ahora podremos remplazar a esos símbolos por valores concretos si lo deseamos.

Por ejemplo, si $i_0 = 4$ y $i_1 = 2$, tendríamos que el valor de retorno habría sido $\mathbf{4 + 2 = 6}$. Es claro que la invocación con SE representa un conjunto

de llamadas concretas. Y esa es exactamente la motivación de su utilización en verificación de programas.

Antes de utilizar esta técnica es necesario eliminar los ciclos con la técnica de loop unrolling (sección 1.1), de la misma manera que sucedía con Weakest Precondition.

Además es necesario modificar el programa de la siguiente manera; antes de la primera sentencia se debe agregar una sentencia **assume precondición**. Así mismo se debe agregar como última instrucción del programa una sentencia **assert poscondición**.

El estado de una ejecución simbólica en cada punto del programa se define como una tupla $\langle \mathcal{P}c, \mathcal{E} \rangle$. El estado simbólico en un punto del programa representa todos los posibles estados concretos del programa en ese punto.

Estado de una ejecución simbólica es la tupla $\langle \mathcal{P}c, \mathcal{E} \rangle$, donde:

1. $\mathcal{P}c$ es llamado restricción del camino o *Path Condition*. Es un conjunto de fórmulas que representa la valuación de los condicionales que se han recorrido hasta el punto actual del programa.

2. $\mathcal{E}$ es llamado el Entorno Simbólico. Es el mapeo entre los nombre de variables del programa y expresiones del lenguaje.

1.4.1. Árboles de ejecución

Los árboles de ejecución muestran los posibles caminos de ejecución de un programa. Cada uno de los nodos representa un conjunto de posibles estados del programa. Cada uno de los ejes representa a una sentencia del lenguaje. Las sentencias **IF** generan dos caminos, uno para cuando la condición sea verdadera y otro para cuando sea falsa. Los nodos hojas de los árboles de ejecución son llamados estados finales, ya que son los estados en donde el programa detendrá su ejecución.

1.4.2. Condición de Verificación usando Symbolic Execution

Para utilizar SE en el contexto de la verificación de programas el objetivo es encontrar los estados simbólicos finales $\langle \mathcal{P}c_{i_f}, \mathcal{E}_{i_f} \rangle$, que son los estados en los puntos finales de la ejecución.

La VC se produce realizando la conjunción de todos $\mathcal{P}c_{i_f}$

$$VC = \mathcal{P}c_{1_f} \wedge \ldots \wedge \mathcal{P}c_{n_f} \tag{1.11}$$

Ejemplo de SE Aplicaremos SE al ejemplo swap (código 1.3) visto en la subsección 1.1.2.

Definiremos el estado inicial $\langle \mathcal{P}c, \mathcal{E} \rangle$ con las siguientes consideraciones:

1. $\mathcal{P}c$ es igual a **true**
2. El binding inicial $\mathcal{E}$ debe relacionar a los parámetros con sus valores iniciales. Además agregamos dos nuevas variables X_{old} y Y_{old} que estarán respectivamente ligadas con los valores iniciales de X e Y.

Por lo tanto el estado inicial es:

$$\langle \{true\}, \{X = X_0, Y = Y_0, X_{old} = X_0, Y_{old} = Y_0\} \rangle.$$

Como asumimos la precondición y afirmamos la poscondición debemos agregar una instrucción, tal como lo mostramos en el código 1.8:

1. **assume** X$\geq$0 $\wedge$ Y$\geq$0 como primera instrucción.
2. **assert** X=Y_{old} $\wedge$ Y=X_{old} como última instrucción
3. Reemplazar todas las sentencias
   - **if (E) {s1} else {s2}**, por
   - **if (E) {assume E; s1} else {assume ¬E; s2}**

Más adelante, en el código 1.9 veremos un ejemplo de SE con instrucciones *if*.

La figura 1.5 muestra el árbol de ejecución del código 1.8, donde puede verse el estado inicial y que cada eje representa una sentencia del programa.

El siguiente paso es asignarle un estado a cada nodo del árbol, partiendo del estado inicial y ejecutando simbólicamente cada sentencia siguiendo las reglas que se detallan a continuación:

```
1  REQUIRES X≥0 ∧ Y≥0
2  ENSURES X=Y_old ∧ Y=X_old
3
4  swap(int X, int Y)
5       L1. assume X≥0 ∧ Y≥0;
6       L2. int T := X;
7       L3. X := Y;
8       L4. Y := T;
9       L5. assert X=Y_old ∧ Y=X_old;
```

Código 1.8: Swap con assume y assert

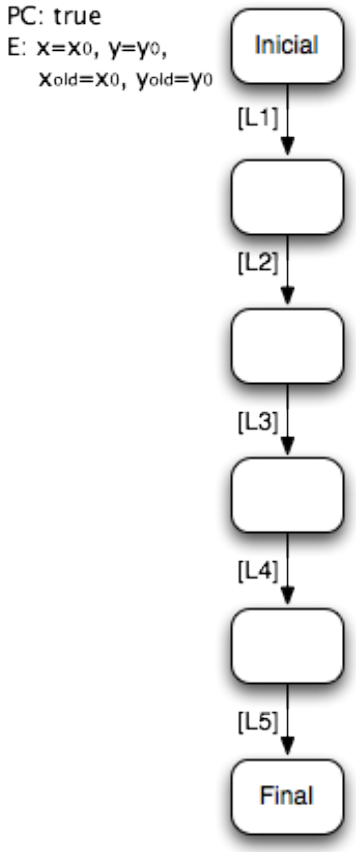

Figura 1.5: Árbol de ejecución Swap inicial

1. Sentencia **x := y** se le asigna a la variable **x** el valor de la variable **y** en ese estado. No se modifica $\mathcal{P}c$.

2. Sentencia **assume E** se remplaza $\mathcal{P}c$ por $\mathcal{P}c \wedge E$. No se modifica $\mathcal{E}$.

3. Sentencia **assert E** se remplaza $\mathcal{P}c$ por $\mathcal{P}c \rightarrow E$. No se modifica $\mathcal{E}$.

En la figura 1.6 podemos ver el árbol de ejecución de Swap con todos los estados. Si tomamos el único nodo hoja, podremos ver que el único estado final es:

$$\begin{aligned}\langle \mathcal{P}c_{1f}, \mathcal{E}_{1f} \rangle = &\langle \{(X_0 > 0 \wedge Y_0 > 0) \rightarrow (Y_0 = Y_0 \wedge X_0 = X_0)\}, \\ &X = Y_0, Y = X_0, X_{old} = X_0, Y_{old} = Y_0, T_0 = X_0\}\rangle\end{aligned}$$

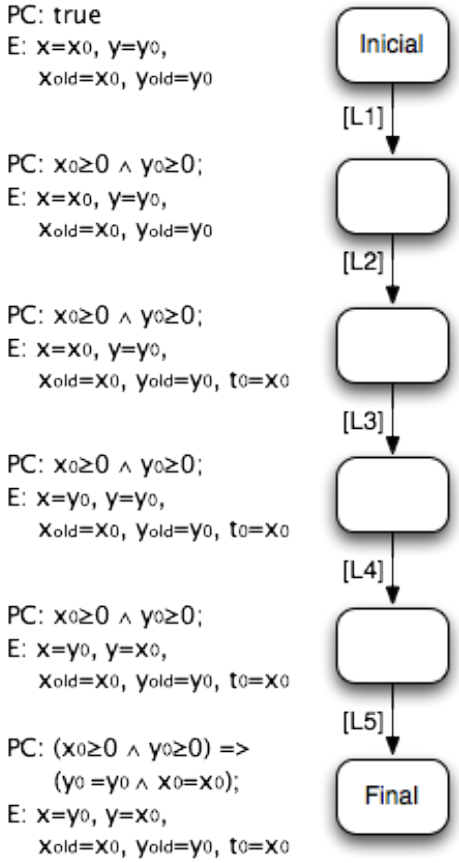

Figura 1.6: Árbol de ejecución Swap final

Luego podremos generar la VC como indicamos previamente:

$$VC \equiv \mathcal{P}c_{1_f} \equiv (X_0 > 0 \wedge Y_0 > 0) \rightarrow (Y_0 = Y_0 \wedge X_0 = X_0) \tag{1.12}$$

Si recordamos la VC obtenida con el método WP, en la ecuación (1.9), podremos apreciar la similitud entre ambas (ver cuadro 1.2).

	VC
WP	$(X \geq 0 \wedge Y \geq 0 \wedge Y = Y_0 \wedge X = X_0) \rightarrow (Y_0 = Y_0 \wedge X_0 = X_0)$
SE	$(X_0 > 0 \wedge Y_0 > 0) \rightarrow (Y_0 = Y_0 \wedge X_0 = X_0)$

Cuadro 1.2: Comparación de VC con WP y SE.

Generalizando, podemos afirmar que existe una equivalencia lógica entre la VC obtenida por ambos métodos. Esto es razonable ya que ambos métodos generan una fórmula que será verdadera si y sólo si el programa cumple su especificación. Es decir que ambas fórmulas necesariamente deben ser equivalentes.

Por otro lado ambos métodos, WP y SE, comparten el problema del crecimiento exponencial del tamaño de la fórmula con respecto al tamaño del programa. En SE las sentencias **if** requieren generar una nueva bifurcación en el árbol de ejecución con lo que finalmente la VC generada será de mayor longitud.

Ejemplo de SE con bifurcación En este ejemplo mostraremos como se genera un árbol de ejecución para un programa que contenga sentencias **IF**.

Para ellos analizaremos el programa que se puede ver en el código 1.9, el cual primero debe ser transformado en el código que figura en 1.10 para luego generar la correspondiente VC.

Generaremos ahora la VC. Como puede verse en la figura 1.7, tenemos dos estados finales distintos. Por lo que la VC queda representada por:

$$VC \equiv \mathcal{P}c_{1_f} \wedge \mathcal{P}c_{2_f} \tag{1.13}$$

```
1  REQUIRES x + y > 0;
2  ENSURES x > y;
3  example(int x,int y) {
4      if (x>y) {
5          x := y;
6      } else {
7          y := x;
8      }
9      x := x + 1;
10 }
```

Código 1.9: Ejemplo SE con bifurcación

```
1  REQUIRES x + y > 0;
2  ENSURES x > y;
3  example(int x,int y) {
4      L1. assume x + y > 0;
5      L2. if (x>y) {
6      L3.     assume x>y;
7      L4.     x := y;
8          } else {
9      L5.     assume x≤y;
10     L6.     y := x;
11         }
12     L7. x := x + 1;
13     L8. assert x > y;
14 }
```

Código 1.10: Ejemplo SE con bifurcación con **assume** y con **assert**

reemplazando obtenemos:

$$\begin{aligned} VC \equiv \quad & ((X_0 + Y_0 > 0 \wedge X_0 > Y_0) \rightarrow (Y_0 + 1 > Y_0)) \wedge \\ & ((X_0 + Y_0 > 0 \wedge X_0 \leq Y_0) \rightarrow (X_0 + 1 > X_0)) \end{aligned}$$

Si recordamos la VC obtenida con el método WP, en la ecuación 1.10, podremos apreciar la similitud entre ambas (ver cuadro 1.3).

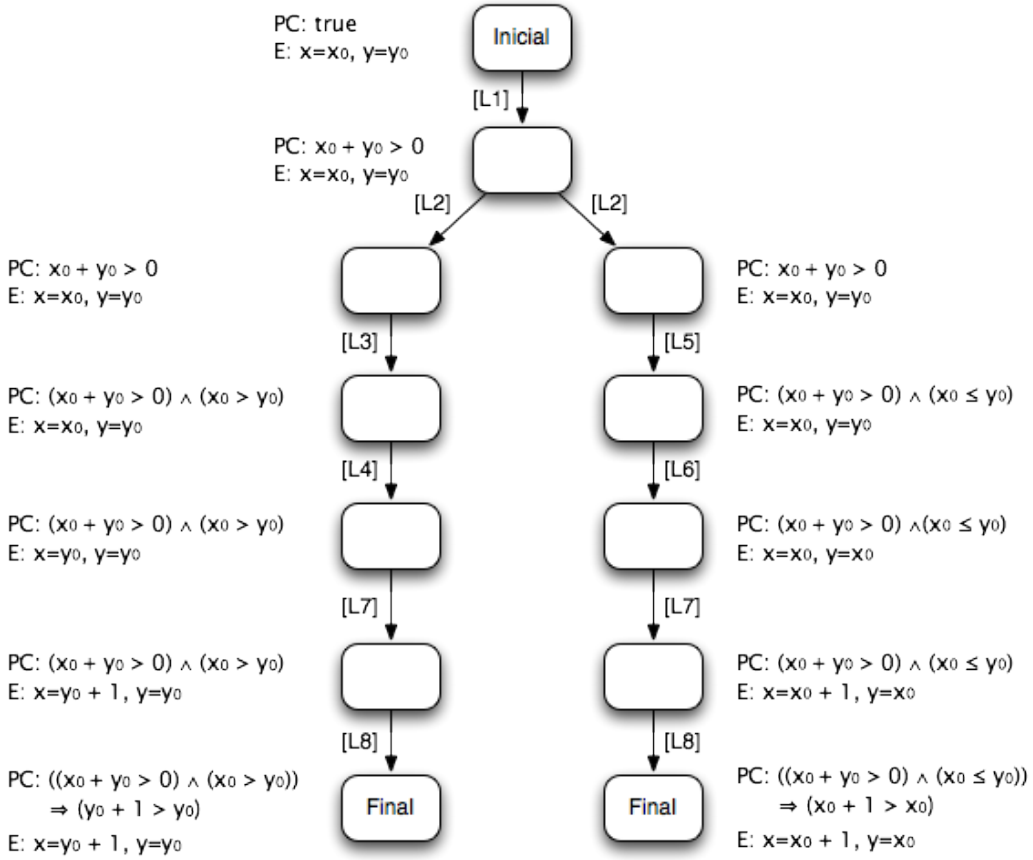

Figura 1.7: Árbol de ejecución con bifurcación

	VC
WP	$(X + Y > 0 \wedge X = X_0 \wedge Y = Y_0) \rightarrow$ $((X > Y \rightarrow Y + 1 > Y) \wedge (X \leq Y \rightarrow X + 1 > X))$
SE	$((X_0 + Y_0 > 0 \wedge X_0 > Y_0) \rightarrow (Y_0 + 1 > Y_0)) \wedge$ $((X_0 + Y_0 > 0 \wedge X_0 \leq Y_0) \rightarrow (X_0 + 1 > X_0))$

Cuadro 1.3: Comparación de VC con WP y SE con bifurcación.

Capítulo 2

Sistema de votación KOA

En 2003 el parlamento holandés decidió construir un sistema de votación basado en Internet para ciudadanos holandeses residentes en el exterior. De allí surge el sistema llamado "Kiezen op Afstand" (KOA[1]).

Hacia fines de 2003 se realizó una revisión externa de los requerimientos y el diseño del sistema en la que participó el Dr. Bart Jacobs perteneciente al grupo Security of Systems (SoS) de la Universidad de Nijmegen. Una de las recomendaciones que realizó el panel fue que el sistema no debería ser diseñado, implementado y testeado por la misma empresa.

Como resultado de estas recomendaciones el gobierno decidió la creación de un subsistema de recuento de votos a ser desarrollado de manera aislada por una tercera parte. El grupo SoS propuso que este subsistema sea formalmente verificado usando las herramientas JML [5] para anotar el código (precondiciones, poscondiciones, invariantes, etc.) y ESC/Java2 [19] para verificar la implementación de los métodos contra su correspondientes anotaciones.

En la figura 2.2 puede verse un diagrama de secuencia que ejemplifica el modo de uso del subsistema de recuento de votos. Como puede apreciarse en dicho diagrama, la clase *CandidateList* es el punto de entrada para la creación del resto de las instancias (KiesKring, Candidate, etc.). Es decir que, *CandidateList* es una suerte de factory del resto de los objetos del subsistema.

[1] "Kiezen op Afstand" se traduce literalmente del Holandés como Votación Remota

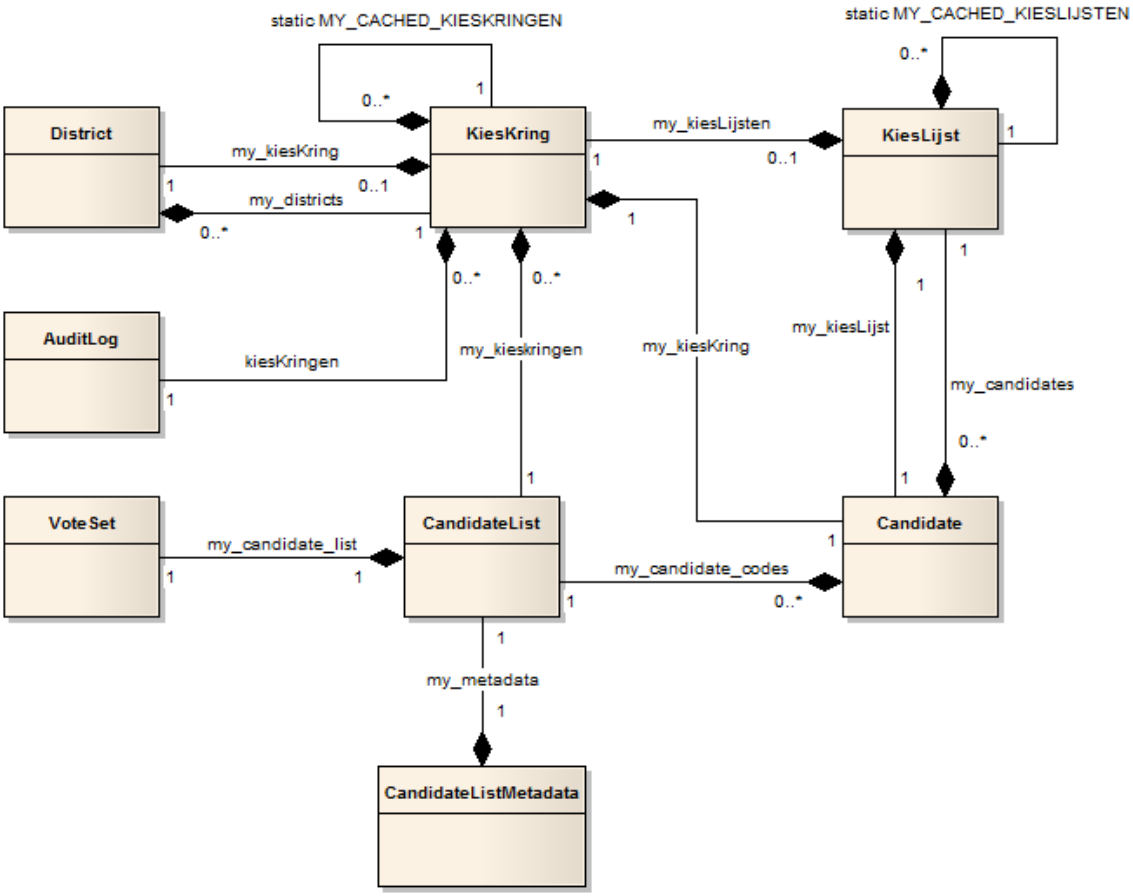

Figura 2.1: Diagrama de clase del sistema KOA

Además en la figura 2.1 puede observarse un diagrama de clase de dicho subsistema.

Para facilitar la interpretación de dicha figura, aclararemos la terminología empleada. La clase *District* representa un distrito dentro de la ciudad, los cuales pueden pertenecer a varios distritos electorales (representados por la clase *KiesKring*). Cada distrito electoral cuenta con una lista de candidatos (representada por la clase *KiesLijst*) y cada candidato está representado en la clase *Candidate*.

Como mencionamos anteriormente, la clase *CandidateList* es el punto de entrada de la aplicación y por ende contiene la información sobre todos los distritos y candidatos, y además la compone una instancia de *CandidateList-*

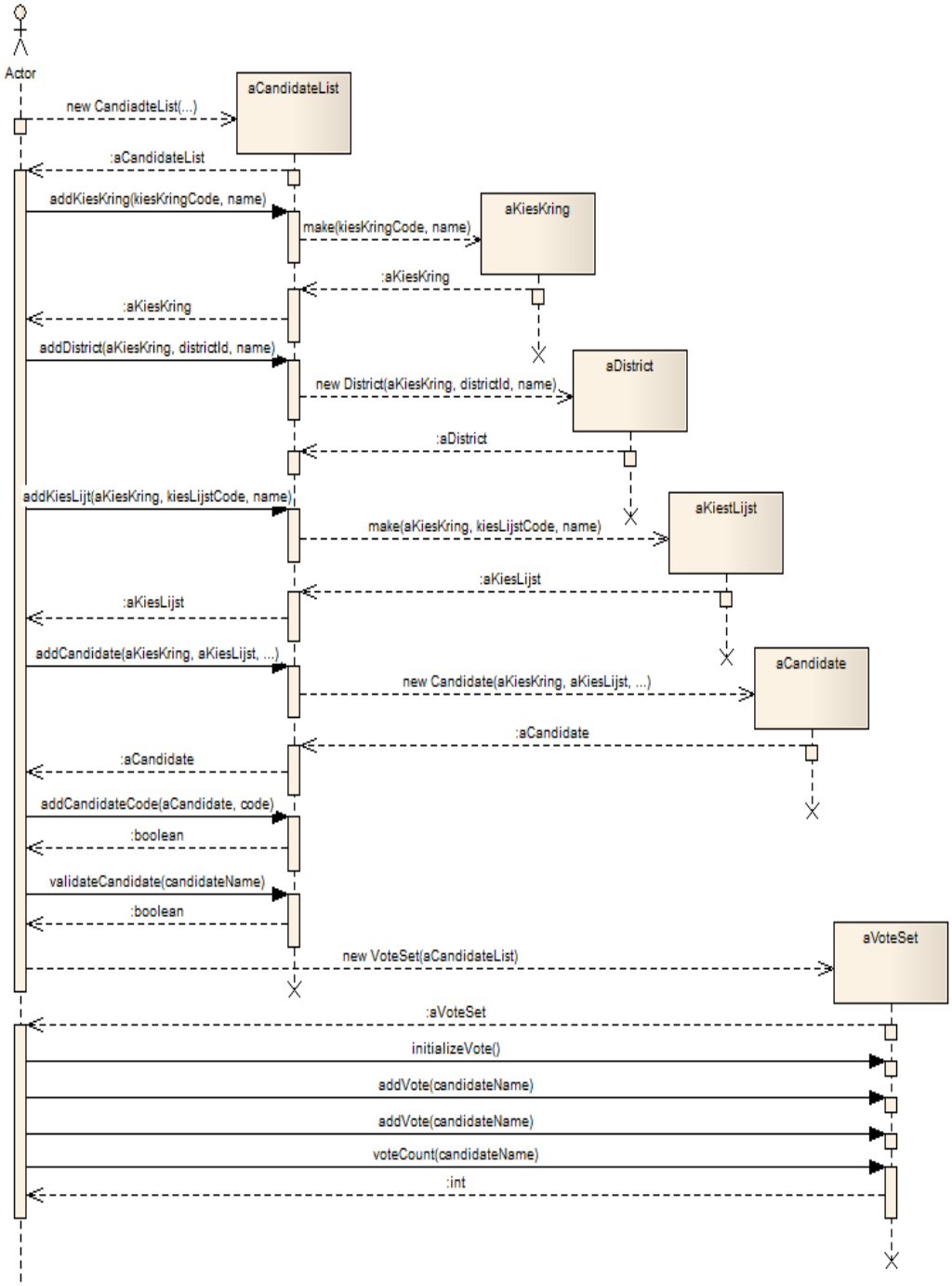

Figura 2.2: Diagrama de secuencia del uso del sistema KOA

Metadata que contiene metadatos relacionados a dicha clase.

Finalmente, la clase *VoteSet* es la encargada de realizar el manejo del proceso de elección (iniciar la votación, agregar votos a cada candidato, realizar el recuento de votos, etc.) y la clase *AuditLog* es simplemente un registro de auditoría.

2.1. ¿Que es JML?

JML [20] es una notación para especificar formalmente el comportamiento e interacciones de las clases y métodos Java. Esto significa que se utiliza para describir los detalles de cómo una clase o interfaz Java interactúa con clientes.

JML utiliza la sintaxis de las expresiones Java, y además agrega un conjunto propio de expresiones con el propósito de lograr que sea más fácil de leer y usar que los lenguajes de especificación que usan notación matemática [21].

Cuando existen las especificaciones JML en los módulos de Java (clases e interfaces), es posible contrastar el código escrito contra sus especificaciones. Existen numerosas herramientas disponibles para hacer estos chequeos, incluidas el verificador de aserciones JML en tiempo de ejecución (jmlrac) y el Verificador Extendido Estático (ESC/Java2) [5]. El uso de estas herramientas permite la detección de discrepancias entre la especificación y el comportamiento del código.

Algunos de los aspectos más sobresalientes de este lenguaje es la posibilidad de definir la *precondición* y *poscondición* de un método.

Además, permite definir *invariantes de clase* (deben ser válidos antes y después de la ejecución de un método, ya sea un método de clase o uno de instancia), *invariantes de instancia* (similares a los invariantes de clase, pero solo deben ser válidos en los métodos de instancia), *invariantes de ciclos* (debe ser válido antes y después de cada iteración del ciclo sobre el cual se aplica).

Un concepto muy similar al de *invariantes* son los *constraint* que, mientras que lo invariantes deben se válidos tanto en la *precondición* como en la *poscondición*, son condiciones que deben ser válidas solamente en la poscondición del método. Al igual que con los *invariantes*, las *constraint* puede tener un alcance de clase o de instancia.

2.1.1. Ejemplo de anotaciones JML

El código 2.1 muestra un simple ejemplo del uso de la sintaxis de JML. A continuación explicaremos algunas de las características de JML usadas:

```
1  /*@
2    @ requires (\forall int j;
3    @                 j >= 0 && j < this.goals.length-1;
4    @                 this.goals[j] <= this.goals[j+1]);
5    @
6    @ modifies this.goals;
7    @
8    @ ensures this.goals.length = \old(this.goals.length) + 1;
9    @ ensures (\forall int j;
10   @                 j >= 0 && j < this.goals.length-1;
11   @                 this.goals[j] <= this.goals[j+1]);
12   @*/
13 public void sortedInsert(/*@ non_null @*/Integer value) {
14     boolean inserted = false;
15     int j = 0;
16     Integer[] modifiedArray = new Integer[this.goals.length+1];
17     //@ loop_invariant (\forall int j;
18     //@                  j >= 0 && j < modifiedArray.length-1;
19     //@                  modifiedArray[j] == null ||
20     //@                  (modifiedArray[j+1] != null ==>
21     //@                  modifiedArray[j] <= modifiedArray[j+1])
22     for (int x = 0; x < this.goals.length; x++) {
23         if (this.goals[x] <= value || inserted) {
24             modifiedArray[j] = this.goals[x];
25         } else {
26             modifiedArray[j] = value;
27             modifiedArray[j+1] = this.goals[x];
28             inserted = true;
29             j++;
30         }
31         j++;
32     }
33     if (!inserted) {
34         modifiedArray[modifiedArray.length-1] = value;
35     }
36     this.goals = modifiedArray;
37 }
```

Código 2.1: Ejemplo de notaciones JML

- La palabra reservada **requires** (línea 2 a 4) impone como precondición que el arreglo *this.goals* debe estar ordenado.
- La palabra reservada **modifies** (línea 6) impone lo que denominamos un frame constraint, indicando que el único campo de la instancia que puede ser modificado durante la ejecución del método es *goals*.

- La palabra reservada **ensures** (líneas 8 a 11) impone la poscondición que debe cumplirse al finalizar la ejecución del método.

- La palabra reservada **non_null** (línea 13) establece como precondición que el parámetro *value* no puede ser *null*.

- La construcción **\forall** Xx**;** $P(x)$**;** $Q(x)$ (líneas 2 y 9) indica que para todas las instancias x del tipo X, si cumplen con $P(x)$ entonces deben cumplir con $Q(x)$.

- La construcción **\old(**x**)** (líneas 8) indica que al momento de evaluar el valor de la variable x, debe tomarse el valor que ésta tenía en el pre-estado.

2.2. ¿Qué es ESC/Java2?

La herramienta ESC/Java2 [19] es una extensión de ESC/Java [12] que implementa la traducción del lenguaje JAVA anotado con especificación JML a una lógica que luego se verifica usando un SMT-Solver. Originalmente desarrollado por "Digital Equipment Corporation's Systems Research Center" (DEC/SRC) es capaz de realizar automática, formal y estáticamente la verificación de especificaciones JML contra código Java. El código fuente más la especificación en JML se traducen a condiciones de verificación (VC), éstas se envían a un demostrador de teoremas, el cual verifica la validez de las VCs, y en caso de encontrarlos genera contraejemplos con los potenciales errores.

Debido a diversos factores la herramienta ESC/Java dejó de ser desarrollada y los códigos fuentes no estaban disponibles. Luego de 2 años y gracias al incremento de popularidad en la utilización de JML los desarrolladores liberaron los códigos fuentes de ESC/Java, lo que impulsó la creación del proyecto ESC/Java2. Los principales objetivos para esta nueva versión eran:

1. Hacer que el código sea consistente con la versión actual (en ese momento la versión disponible era la 1.4) de Java;

2. Parsear completamente la versión actual de JML;

3. Verificar la mayor cantidad de anotaciones del lenguaje JML posible;

4. Empaquetar la herramienta de manera que sea fácil su aplicación en diversos entornos.

Del mismo modo que muchas otras herramientas de verificación de software, ESC/JAVA2 realiza una primera traducción a un lenguaje de representación intermedia. Este lenguaje es una representación simple del *Guarded Command Language (GCL)* [11]. Dicha representación intermedia es luego procesada por un generador de VCs el cual genera un VC por cada *guarded command*. Finalmente estas VCs son enviadas a un demostrador de teoremas (Simplify [10]) el cual realiza la verificación.

2.2.1. Verificación de KOA usando ESC/Java2

Los desarrolladores dividieron el sistema KOA en 3 subsistemas: *File I/O*, *Graphical I/O* y estructuras de datos y algoritmos *Core*. Debido al desafío que planteaba tener que especificar el sistema completo en el lapso de tiempo disponible, se decidió que solamente el subsistema *Core* iba a ser completamente especificado y verificado usando ESC/Java2.

	File I/O	Graphical I/O	Core
Clases	8	13	8
Métodos	154	200	83
NCSS	837	1599	395
Specs	446	172	529
Specs:NCSS	1:2	1:10	5:4

Cuadro 2.1: Resumen del sistema KOA

El cuadro 2.1 muestra un resumen del tamaño (en número de clases y métodos), la complejidad (tamaño del código sin comentarios, o NCSS[2]) y cobertura de los 3 subsistemas.

Debido a que no era factible realizar la verificación del 100 % del sistema dentro del tiempo previsto, además de la verificación con ESC/JAVA2, se

[2]NCSS viene de "Non-Comment Source Statements" que significa, sentencias de código que no son comentarios

generaron 8000 casos de test usando la herramienta *jmlunit*[3] [6]. Estos tests cubrían el 100 % del *Core* del sistema y no reportaron error alguno en dicho subsistema.

2.3. ¿Que es JmlForge?

JmlForge es una herramienta desarrollada por el Instituto de Tecnología de Massachusetts (MIT) que es capaz de realizar la verificación estática y modular de código Java contra su especificación en lenguaje JML.

Para alcanzar la mayor modularidad posible el código Java+JML es primero traducido a una representación intermedia de *Forge* [9] (FIR[4]). FIR es un lenguaje de especificación relacional, capaz de representar sentencias imperativas, especificaciones declarativas y abstracciones relacionales usando una gramática sencilla. Con el objetivo de facilitar el análisis modular FIR usa sentencias de especificación: cuando se encuentra un llamado a un procedimiento, este es reemplazado por la instanciación de su especificación.

El siguiente paso de la herramienta es la traducción del código procedural a lógica relacional. Para ello se usa la técnica de ejecución simbólica (sección 1.4).

Además del método y su especificación el usuario de *Forge* debe proveer límites al análisis:

- La cantidad de loop unrolling que deben realizarse.
- Un límite al tamaño de los enteros, expresado en su tamaño en bits.
- La cantidad máxima de instancias de cada clase que existirán en el heap de ejecución.

La lógica relacional generada más los límites impuestos para la verificación son enviados a *KodKod*[5], el cual luego lo traduce a un problema de satisfactibilidad que luego es resuelto por un SAT-Solver. Si se encuentra

[3]jmlunit es una herramienta que genera automáticamente clases JUnit para las especificaciones JML

[4]FIR es el acrónimo de Forge Intermediate Representation

[5]KodKod es un eficiente SAT-Solver para lógica de primer orden con relaciones, clausura transitiva y modelos parciales

una solución, *KodKod* mapea ésta a una instancia de la fórmula en lógica relacional, el cual es luego mapeado por *Forge* a una traza de contraejemplo del método FIR original.

La traducción de JML a FIR no maneja todo el lenguaje JML y además introduce algunas optimizaciones que pueden generar contraejemplos falsos o no reportarlos cuando debería. Algunas de esas imprecisiones en la traducción son: carencia de soporte para aritmética de números reales, E/S, inicialización estática, reflexión, representación incompleta de Strings.

En la sección 6.1.1 mostraremos los resultados obtenidos en la realización de la verificación del subsistema de recuento de votos de KOA publicado en [8].

Capítulo 3

Lenguaje JDynAlloy

Debido a la complejidad de tener que lidiar con lenguajes de programación de alto nivel (JAVA, .NET, etc.) muchas herramientas de verificación de software traducen los programas en lenguaje de alto nivel a un lenguaje de representación intermedia que facilite el análisis. Por ejemplo, ESC/Java2 [19] traduce código JAVA a una variante de *Guarded Command Language (GCL)*[1] [11]; Boogie [2] traduce el bytecode de .NET a BoogiePL [7]; JMLForge traduce a FIR. Estas representaciones intermedias facilitan la transformación y la optimización del código, y simplifican la futura traducción a condiciones de verificación.

JDynAlloy es un lenguaje de especificación relacional, ya que sus tipos de datos son relaciones (este enfoque puede apreciarse en [16]), creado con el objetivo de disponer de una representación intermedia entre DynAlloy y los lenguajes de alto nivel que permitiese reducir la complejidad de traducir y analizar código de alto nivel. Al igual que muchos lenguajes de alto nivel, en JDynAlloy pueden representarse ciclos, condicionales, asignaciones, llamados a métodos, expresiones, etc. Además agrega la posibilidad de incorporar invariantes de clase, invariantes de instancia, precondiciones y poscondiciones usando lógica de primer orden.

Este capítulo incluye una descripción de la estructura y la semántica de JDynAlloy.

[1] GCL es un lenguaje definido por Dijkstra que permite asignar un predicado transformador a cada comando de un lenguaje imperativo.

3.1. Expresiones y fórmulas de JDynAlloy

Las expresiones y fórmulas que se utilizan en JDynAlloy son las mismas que define DynAlloy [14].

Una diferencia entre las expresiones JDynAlloy y las de JAVA es que las expresiones en el lenguaje JDynAlloy no producen efectos colaterales, es decir, que ninguna expresión por sí misma modifica el estado de un programa.

Otra diferencia es que JAVA utiliza evaluación *lazy*[2] para las expresiones de JAVA, mientras que JDynAlloy utiliza evaluación de expresiones *eager*[3].

3.2. Semántica de JDynAlloy

El lenguaje de especificación JDynAlloy provee un marco de trabajo para realizar verificación estática de programas. En esta sección se verá la interpretación que se debe realizar de los distintos artefactos que componen el lenguaje (en el apéndice A puede verse la gramática del lenguaje JDynAlloy).

3.2.1. Módulos

JDynAlloy provee la capacidad de expresar tipos abstractos de datos a través de los módulos. Cada módulo provee también una signatura Alloy [17]. Esta signatura describe la representación formal del tipo abstracto en Alloy. Un módulo puede tener *fields*, los mismos representan variables miembros del módulo.

```
1 module A
2 sig A extends java_lang_Object {} {}
3 field f1: A->one(Int) {}
```

Código 3.1: Módulo JDynAlloy

En el código 3.1 podemos ver la declaración del módulo A, el cual hereda de *java_lang_Object*, que es un módulo *built-in*[4]. En la línea 3 se encuentra declarado el *field f1* que es de tipo función de A a *Int* (recordemos que el sistema de tipos esta tomado de Alloy).

[2] es la técnica de retrasar la computación hasta que el resultado sea requerido.

[3] es la técnica de evaluar las expresiones para llevarlas a un valor lo antes posible.

[4] Los módulos *built-in* se explican en la sección 3.2.9

3.2.2. Invariantes

El invariante de un módulo JDynAlloy permite distinguir entre las instancias del tipo que están bien formadas y las que no.

```
1 module Persona
2 sig Persona extends java_lang_Object {} {}
3 field padre: Persona->one(Persona) {}
4 object_invariant not ( thiz.padre = thiz)
```

Código 3.2: Ejemplo de Invariante

En el código 3.2 se declara el módulo *Persona*, que posee un field *padre*. El invariante de la línea 4 restringe las instancias válidas del módulo *Persona* a aquellas en donde el padre de una *Persona* no sea sí mismo.

Los *class invariants* se diferencian de los *object invariants* porque los *object invariants* se deben mantener para todos los programas que utilicen instancias del tipo que posee el invariante, mientras que los *class invariants* se deben mantener para absolutamente todos los programas.

La semántica de *class invariant* presenta un problema para el análisis modular, ya que es necesario mantener el *class invariant* de todos los módulos en la verificación de cada programa, contradiciendo la lógica del análisis modular. Sin embargo los *class invariants* son parte del subconjunto de JML utilizado por el caso de estudio (sección 2), y por lo tanto deben ser soportados por JDynAlloy.

3.2.3. Constraint

La referencia de JML explica qué *constraint* puede utilizarse para limitar la forma en que los valores cambian con el tiempo.

```
1  module Casa
2  sig Casa extends java_lang_Object {} {}
3  field aEstrenar: Casa->one(boolean) {}
4  object_constraint aEstrenar=false implies aEstrenar'=false
5  program Casa::Constructor[]
6  Implementation {
7      aEstrenar := true;
8  }
9
10 program Casa::unPrograma[...]
11 Specification { ... }
12 Implementation { ... }
```

Código 3.3: Ejemplo de *Constraint*

Tomemos el ejemplo del código 3.3. Una casa al instanciarse es a estrenar. Sin embargo si en algún momento *aEstrenar* toma el valor *false*, el programa *unPrograma* de *Casa* no podrá volver a setearlo en *true*, ya que todo programa de *Casa* deberá asegurar el predicado del *object_constraint* de la línea 4.

3.2.4. Represents

La keyword *represents* brinda una manera de relacionar los valores de los fields inherentes al tipo abstracto de dato con aquellos que son introducidos para facilitar la especificación sobre el tipo de datos y sus operaciones.

```
1  module Node
2  sig Node extends java_lang_Object {} {}
3  field next: LinkedList->one(Node+null) {}
4
5  module LinkedList
6  sig LinkedList extends java_lang_Object {} {}
7  field head: LinkedList->one(Node+null) {}
8  field isEmpty: LinkedList->one(boolean) {}
9  represents thiz.isEmpty such that
10     (isEmpty implies head = null)
11     (head = null implies isEmpty)
12 object_invariant thiz.isEmpty = false
```

Código 3.4: Ejemplo de *Represents*

El comportamiento de *represents* está ilustrado en el código 3.4. En él puede apreciarse la especificación de una lista encadenada que nunca puede ser vacía. De esta manera se utiliza el field *isEmpty* en la línea 8 para facilitar la especificación del invariante.

Notemos en las líneas 3 y 7 la introducción de la keyword *null*. La expresión *null* debe ser interpretada como una constante disjunta del resto de los tipos, es decir que si desea que un campo pueda tomar el valor *null* es necesario indicarlo explícitamente como se muestra en el ejemplo.

3.2.5. Especificación del método

La especificación de un método es el contrato que indica las condiciones que deben reunirse al invocar un método, y las condiciones que éste debe asegurar luego de su ejecución.

Para la especificación se utilizan tres keyword cuya semántica es similar a la que poseen en JML (subsección 2.1.1): *requires*, *modifies* y *ensures*.

La especificación consiste de casos de especificación que especifican distintos escenarios. Algún escenario puede especificar el camino normal en caso de no producirse error. Otro escenario, por ejemplo, puede especificar que si se cumplen determinadas condiciones indeseables, se debe lanzar una excepción.

```
1  module AList
2  sig AList extends java_lang_Object {} {}
3  field size: LinkedList->one(Int) {}
4  ...
5  program AList::get[var thiz:AList,var throw:java_lang_Throwable+null,var
6  return:java_lang_Object+null,index:Int]
7  Specification
8  {
9      SpecCase #0 {
10         requires {
11             lt[index,size]
12         }
13         ensures {
14             throw = null and
15             ...
16         }
17     }
18     SpecCase #1 {
19         requires {
20             gt[index,size]
21         }
22         ensures {
23             instanceOf[throw, java_lang_IndexOutOfBoundsException]
24         }
25     }
26 }
27 ...
28 Implementation
29 ...
```

Código 3.5: Ejemplo de especificación del programa

En el ejemplo del código 3.5 encontramos tres nuevos predicados built-in de JDynAlloy.

- *lt* (línea 11) es el predicado *menor*. Indica si el entero de la izquierda es menor al de la derecha
- *gt* (línea 20) es el predicado *mayor*. Indica si el entero de la izquierda es mayor al de la derecha
- *instanceOf* (línea 23) es un predicado que toma una instancia de un

módulo y un tipo de datos. El predicado será verdadero si y solo si la instancia pertenece al tipo.

En el ejemplo 3.5 tenemos especificados dos casos. En la línea 9 comienza el primer caso, que representa el camino normal de ejecución. Este caso especifica lo que asegura el programa *AList::get* en caso de que el parámetro *index* cumpla con la precondición; es decir, que sea menor o igual al índice del máximo elemento. Lo que asegura el programa en ese caso fue omitido casi en su totalidad, sin embargo se puede apreciar, en la línea 14, que el programa en ese caso asegura que no haya error. El resto de la especificación de este caso fue omitida ya que dada su complejidad no contribuye a la explicación. Sin embargo informalmente podemos afirmar que la especificación debe predicar sobre *return* y asegurarse que su valor sea igual al valor del index-ésimo elemento de la lista.

Dentro de la especificación de un programa se puede especificar un *frame constraint*, el cual indica cuáles son los fields que pueden ser modificados durante la ejecución del programa. Esto se logra utilizando la keyword *modifies*. En el código 3.6 se muestra un programa del módulo *LinkedList* (código 3.4). Este programa agrega un nodo al inicio de la lista. Para la implementación del programa se necesita modificar el field *head* de la instancia de *thiz*. Se puede ver en la línea 9 cómo la especificación declara que dicho field puede ser modificado por la implementación del programa.

```
1  program LinkedList::AddToFront[
2  var thiz:LinkedList,
3  var throw:java_lang_Throwable+null,
4  var return:java_lang_Object+null,index:Int,
5  var o:Node+null]
6  Specification
7  {
8      SpecCase #0 {
9          modifies { thiz.head }
10         modifies { o.next }
11         ensures {
12             throw = null and
13             thiz.head' = o and
14             o.next' = thiz.head
15         }
16     }
17 }
18 ...
19 Implementation
20 ...
```

Código 3.6: Ejemplo de especificación del programa

Es posible declarar que el programa no tendrá restricción alguna para modificar el estado. Para ello se utiliza la keyword *EVERYTHING*. Se puede ver un ejemplo en el código 3.7.

```
1  module Node sig Node extends java_lang_Object {} {}
2  field next: LinkedList->one(Node+null) {}
3
4  module Persona
5  sig Persona extends java_lang_Object {} {}
6  field nombre: LinkedList->one(java_lang_String+null) {}
7  field apellido: LinkedList->one(java_lang_String+null) {}
8  field direccion: LinkedList->one(java_lang_String+null) {}
9  field telefono: LinkedList->one(java_lang_String+null) {}
10
11 program Persona::reset[
12 var thiz:Persona,
13 var throw:java_lang_Throwable+null]
14 Specification
15 {
16     SpecCase #0 {
17         modifies { EVERYTHING }
18         ensures {
19             throw = null and
20             thiz.nombre = null and
21             thiz.apellido = null and
22             thiz.direccion = null and
23             thiz.telefono = null
24         }
25     }
26 }
```

Código 3.7: Ejemplo de *modifies EVERYTHING*

Si no se especificara el *modifies* en el contrato de un programa, entonces el programa no debería realizar ninguna modificación al estado.

3.2.6. Sentencias JDynAlloy

Explicaremos a continuación las distintas sentencias que posee el lenguaje JDynAlloy.

Declaración de variables: La declaración de variables introduce una nueva variable del tipo especificado.

```
var flag:boolean;
```

Assert: En caso de no cumplirse la condición, interrumpe la ejecución del programa. La interrupción del programa se logra lanzando la pseudo-excepción *AssertionFailure* [3].

```
assert i > 0;
```

Assume: Asume como cierta la fórmula recibida como parámetro [3].

```
assume x>y;
```

Havoc: La sentencia obtiene un nuevo valor arbitrario para la variable indicada [3].

```
havoc flag;
```

Skip: Esta sentencia es ignorada.

```
skip;
```

CreateObject: En el *heap* de JDynAlloy se registran las instancias de los objetos ya asignados [16]. Esta sentencia le asigna a la variable parámetro una instancia del tipo especificado que no se encuentra en el *heap* y luego la registra en el mismo.

```
var s:java_lang_String;
createObject<java_lang_String>[s];
```

Asignación: Le asigna a la expresión de la derecha, la expresión de la izquierda [16].

```
i := i + 1;
```

Program call: Invoca el programa especificado con los argumentos provistos. El ejemplo a continuación invoca el programa *Cronometro::reiniciar*

```
var o:Cronometro;
call reiniciar[o,throw];
```

Bloque: Permite ejecutar secuencialmente múltiples sentencias.

```
{
    var t:Int;
    t := x;
    x := y;
    y := t;
};
```

If: Ejecución condicional dependiendo del valor de la condición.

```
if lt[x,y] {
    x := y;
} else {
    y := x;
};
```

While: Ejecuta el código del cuerpo de la sentencia, mientras la condición sea verdadera.

```
while lt[x,10] {
   x := x + 1;
};
```

3.2.7. Constructores

La diferencia entre los constructores y los programas ordinarios es que la instancia de *thiz* que recibe un constructor es una instancia fresca, es decir que no se encontraba en el *heap*. El código 3.8 muestra el constructor del módulo LinkList del código 3.4.

```
1  program AList::Constructor[
2      var thiz:LinkList,
3      var throw:java_lang_Throwable+null]
4  Specification
5  {
6      SpecCase #0 {
7          requires {
8              true
9          }
10         ensures {
11             throw = null and head = null
12         }
13     }
14 }
15 Implementation
16 {
17     throw := null;
18     head := null;
19 }
```

Código 3.8: Ejemplo de declaración de un constructor

3.2.8. Llamado a métodos en especificaciones

JDynAlloy permite utilizar llamados a métodos dentro de las especificaciones. Este comportamiento es análogo al de JML. Para representarlo JDynAlloy extiende las fórmulas de DynAlloy con la fórmula *callSpec*.

La sintaxis es `callSpec aProgram [ param`$_1$ `, ..., param`$_n$ `]`. Donde *aProgram* es el nombre del programa.

Un ejemplo puede verse en el código 3.9. En la línea 31 encontramos el predicado *callSpec*:

```
callSpec sum[throw, cs_ret_sum, 1, 2]
```

Para razonar acerca de dicha fórmula, modelemos el significado de la especificación como: "precondición del programa *sum* implica poscondición del programa *sum*"

```
(gt[left,0] and gt[right,0]) implies
((throw'=null) and return = left + right)
```

Ahora reemplacemos los parámetros declarados (*throw*, *return*, *left*, *right*) por los parámetros actuales (*throw*, *cs_ret_sum*, *1*, *2*)

```
1  program ClaseA::sum[
2    var throw:java_lang_Throwable+null,
3    var return:Int,
4    var left:Int,
5    var right:Int]
6  Specification {
7      SpecCase #0 {
8          requires {
9              gt[left,0] and gt[right,0]
10         }
11         ensures {
12             (throw'=null) and
13             return = left + right
14         }
15     }
16 }
17 Implementation
18 ...
19
20 program ClaseA::returnThree[
21   var thiz:ClaseA,
22   var throw:java_lang_Throwable+null,
23   var return:Int,
24   var left:Int,
25   var right:Int]
26 Specification {
27     SpecCase #0 {
28         ensures {
29             (throw'=null) and
30             (some cs_ret_sum:Int | {
31                 (callSpec sum[throw, cs_ret_sum, 1, 2]) and
32                 (return = cs_ret_sum))
33             }
34         }
35     }
36 }
```

Código 3.9: Ejemplo de callSpec

```
(gt[1,0] and gt[2,0]) implies ((throw'=null) and return = 1 + 2)
```

JDynAlloy reemplaza a todos los *callSpec* por fórmulas válidas DynAlloy siguiendo razonamientos similares a éste, como puede verse en la sección 5.

3.2.9. Módulos Built-In

Ciertos módulos han sido incorporados al lenguaje JDynAlloy para facilitar la traducción de programas en JAVA. En la tabla 3.1 se muestra una lista exhaustiva de módulos *built-in* de JDynAlloy.

java_lang_Object	java_lang_Byte
java_lang_Integer	java_lang_String
java_lang_Boolean	java_lang_Class
java_util_Date	java_util_Iterator
java_util_Map	java_util_HashMap
java_util_SortedMap	java_util_TreeMap
java_util_Set	java_util_HashSet
java_util_List	java_lang_Throwable java_lang_Exception
java_lang_RuntimeException	java_lang_ClassCastException
java_lang_IllegalArgumentException	java_lang_IndexOutOfBoundsException
java_util_NoSuchElementException	java_lang_NullPointerException
java_lang_SystemArray	org_jmlspecs_models_JMLObjectSequence
org_jmlspecs_models_JMLObjectSet	

Cuadro 3.1: Módulos built-in de JDynAlloy

3.2.10. Arreglos y colecciones

Arreglos: para soportar la representación de los arreglos de Java se incorporó el tipo *java_lang_SystemArray*. En el código 3.10 encontramos su representación. En la línea 3 podemos observar el field *Object_Array* que es de tipo secuencia. En esa secuencia será almacenada la información de los elementos del arreglo.

Los predicados built-in *updateArrayPost*, *arrayAccess* y *arrayLength* son utilizados para manipular la clase SystemArray.

```
1 module java_lang_SystemArray
2 sig java_lang_SystemArray extends java_lang_Object {} {}
3 field Object_Array:(java_lang_SystemArray)->(seq univ) {}
4
5 program java_lang_SystemArray::Constructor[
6   var thiz:java_lang_SystemArray,
7   var throw:java_lang_Throwable+null,
8   var length:Int]
9 ...
```

Código 3.10: Representación de arrays

JMLObjectSequence: es la contrapartida de la clase homónima defina por JML para facilitar la escritura de contratos. En el código 3.11 encontramos la descripción del módulo JDynAlloy.

Al escribir este módulo hemos implementado una técnica particular con el objetivo de optimizar la representación de esta secuencia. La técnica consiste en hacer que JDynAlloy evite la creación de instancias de *JMLObjectSequence* y utilice directamente en su lugar una secuencia nativa de alloy. La manipulación de dicha secuencia se realiza utilizando los predicados built-in *fun_list_size*, *fun_list_get* y *fun_list_empty*. De esta manera evitamos el overhead causado por envolver dicha secuencia con el único propósito de manipularla.

```
1  module org_jmlspecs_models_JMLObjectSequence
2  sig org_jmlspecs_models_JMLObjectSequence {} {}
3
4  program org_jmlspecs_models_JMLObjectSequence::get[
5    var thiz:org_jmlspecs_models_JMLObjectSequence,
6    var throw:java_lang_Throwable+null,
7    var return:univ,
8    var index:Int]
9  ...
10
11 program org_jmlspecs_models_JMLObjectSequence::int_size[
12   var thiz:org_jmlspecs_models_JMLObjectSequence,
13   var throw:java_lang_Throwable+null,
14   var return:Int]
15 ...
16
17 program org_jmlspecs_models_JMLObjectSequence::isEmpty[
18   var thiz:org_jmlspecs_models_JMLObjectSequence,
19   var throw:java_lang_Throwable+null,
20   var return:boolean]
21 ...
```

Código 3.11: Representación de JMLObjectSequence

JMLObjectSet es un caso similar al anterior. En el código 3.12 puede verse el módulo. La aplicación de la misma técnica se utiliza nuevamente para optimizar la representación de *JMLObjectSet*. La manipulación del conjunto alloy subyacente se logra por medio de los predicados built-in *fun_set_size* y *fun_set_contains*.

3.2.11. Herencia de módulos

Un módulo puede heredar de otro utilizando la palabra clave *extends* en la declaración de su signatura. Cuando un módulo extiende a otro hereda sus fields y sus programas.

```
1  module org_jmlspecs_models_JMLObjectSet
2  sig org_jmlspecs_models_JMLObjectSet {} {}
3  program org_jmlspecs_models_JMLObjectSet::has[
4    var thiz:org_jmlspecs_models_JMLObjectSet,
5    var throw:java_lang_Throwable+null,
6    var return:boolean,
7    var e:java_lang_Object+null]
8  ...
9
10 program org_jmlspecs_models_JMLObjectSet::int_size[
11   var thiz:org_jmlspecs_models_JMLObjectSet,
12   var throw:java_lang_Throwable+null,
13   var return:Int]
14 ...
```

Código 3.12: Representación de JMLObjectSet

Veamos un ejemplo en el código 3.13. La keyword *abstract* (línea 2) implica que no se permitirán instancias de tipo A. Desde el punto de vista de la lógica relacional lo que sucede es que A será un conjunto compuesto exclusivamente por instancias B y C. Por otro lado B y C serán conjuntos disjuntos.

```
1  module A
2  abstract sig A  {} {}
3
4  module B
5  sig B extends A {} {}
6
7  module C
8  sig C extends A {} {}
```

Código 3.13: Herencia en JDynAlloy

Capítulo 4

Traducción del lenguaje JML a JDynAlloy

La traducción del lenguaje JML al lenguaje Alloy atraviesa una serie de traducciones a lenguajes intermedios que facilitan este proceso. En la figura 4.1 se muestran todas las etapas de transformación y la salida producida por cada uno de ellas.

En esta sección explicaremos todas las etapas requeridas hasta obtener la especificación JDynAlloy que luego será traducida al lenguaje DynAlloy.

4.1. El simplificador JML

La etapa de simplificación toma como entrada código JML y genera como salida un código JML simplificado. El objetivo de dichas simplificaciones radica en reducir la cantidad de estructuras que deberán ser manejadas en la etapa de traducción de JML a JDynAlloy.

Dentro de las principales modificaciones que se realizan podemos mencionar la inicialización de las variables miembros, las cuales son trasladadas en cada constructor de la clase. En caso de que no exista una declaración explícita de al menos un constructor, el simplificador creará explícitamente uno por defecto y colocará allí la inicialización de las variables miembros.

Otra modificación consiste en la reducción de la cantidad de estructuras

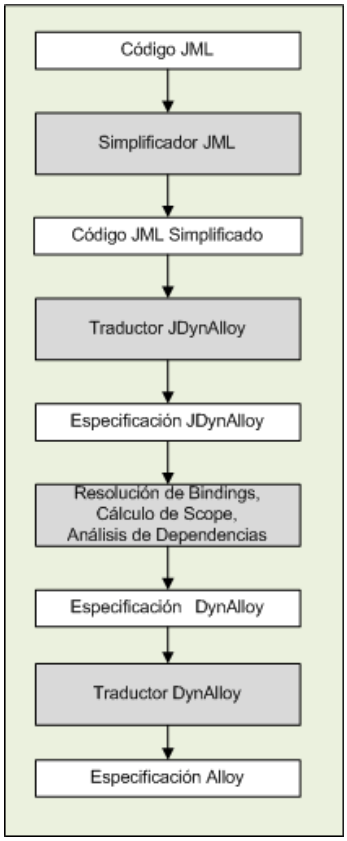

Figura 4.1: Etapas de transformación del código JML hasta código Alloy

de ciclos, transformándolos a la estructura de un *while*. En el cuadro 4.1 puede verse un ejemplo de esta transformación.

Código sin simplificar	Código simplificado

`int x = 0;`
`for (int i=0; i<3; i++) {`
`  x += i;`
`}`
	`int x = 0;`
	`int i = 0;`
	`while (i<3) {`
	`  x += i;`
	`  i++;`
	`}`

Cuadro 4.1: Simplificación de un ciclo For a un While

En cuanto a las modificaciones realizadas sobre las anotaciones JML podemos resaltar la simplificación del llamado a métodos. La idea es agregar un existencial que cuantifique una única variable por cada llamado, luego se asigna el resultado del método a su correspondiente variable cuantificada. Este procedimiento es necesario ya que en JAVA los llamados a métodos son expresiones, mientras que en JDynAlloy son predicados (callSpec).

En el código 4.1 mostramos un método que en su *poscondición* indica que el resultado de dicho método depende del resultado de la llamada de dos métodos (getAnInt y parseInt). La simplificación de este método puede apreciarse en el código 4.2.

```
1     //@ ensures \result <==>
2     //@         \old(getAnInt(Integer.parseInt(aString)) == 2);
3     public boolean nestedCalls(String aString) {
4         this.anInt = this.anInt - 1;
5         return (getAnInt(Integer.parseInt(aString)) == 2);
6     }
```

Código 4.1: Anotaciones JML sin simplificar

En el puede verse que por cada llamado a método se creó un cuantificador existencial y se asigna el resultado de la llamada a la función a su correspondiente variable cuantificada. Finalmente, utiliza la variable *cs_return_getAnInt*, que contiene el valor de retorno del llamado al método *getAnInt* para realizar la comparación final.

```
1   /*@
2     @ ensures
3     @  (\exists int cs_ret_parseInt; ;
4     @   (\exists int cs_ret_getAnInt; ;
5     @    cs_ret_parseInt==\old(Integer.parseInt(aString)) &&
6     @    cs_ret_getAnInt==\old(this.getAnInt(cs_ret_parseInt)) &&
7     @    \result <==> \old(cs_ret_getAnInt == 2)));
8     @*/
9   public boolean nestedCalls(java.lang.String aString) {
10    int es_var__6;
11    int es_var__8;
12
13    this.anInt = this.anInt - 1;
14    es_var__6 = java.lang.Integer.parseInt(aString);
15    es_var__8 = this.getAnInt(es_var__6);
16
17    return (es_var__8==2);
18  }
```

Código 4.2: Anotaciones JML simplificadas

En cuanto a las simplificaciones a las expresiones del lenguaje, podemos destacar el tratamiento a los condicionales. Dado que la evaluación de los condicionales en JAVA se realiza de manera lazy es preciso simular dicho comportamiento. Para ello se realiza una construcción con sentencias *if* anidadas que computan el resultado del condicional en una variable booleana.

Los cuadros 4.2 y 4.3 muestran la transformación que se realiza al código de una estructura *if* para el caso de la conjunción y de la disyunción respectivamente.

Código sin simplificar	Código simplificado

`if (A && B){`
`  hacer algo...`
`}`
	`boolean computedConditional;`
	`if (A) {`
	`  if (B) {`
	`  computedConditional = true;`
	`  } else {`
	`  computedConditional = false;`
	`  }`
	`} else {`
	`  computedConditional = false;`
	`}`
	`if (computedConditional){`
	`  hacer algo...`
	`}`

Cuadro 4.2: Simplificación de la conjunción

Dentro de JDynAlloy los literales de String se representan usando el módulo built-in java_langString, cuya representación interna consiste de dos enteros, uno que representa un identificador para el String y otro que representa el tamaño. Para representar dicho identificador se optó por usar el *hashCode* de la clase String en JAVA. Vale aclarar que la técnica de utilizar un entero para modelar las instancias de String se aplica también a los siguiente módulos de JDynAlloy: `java_lang_Integer`, `java_util_Date`, `java_lang_Byte` y `java_lang_Character`.

Para acompañar esta representación, es necesario que el simplificador modifique los literales por un llamado a un método que se encarga de la creación del String de JDynAlloy. De esta manera la sentencia:

Código sin simplificar	Código simplificado

`if (A || B){`
`  hacer algo...`
`}`
	`boolean computedConditional;`
	`if (A) {`
	`  computedConditional = true;`
	`} else {`
	`  if (B) {`
	`  computedConditional = true;`
	`  } else {`
	`  computedConditional = false;`
	`  }`
	`}`
	`if (computedConditional){`
	`  hacer algo...`
	`}`

Cuadro 4.3: Simplificación de la disyunción.

```
String aString = "unString";
```

se transforma a:

```
String aString = String.buildInstance("unString", 8);
```

Vale la pena aclarar que esta representación de Strings en JDynAlloy puede ocasionar el reporte de violaciones falsas o el no reporte de violaciones existentes. Ello se debe a que, si bien el *hashCode* de un String es único, cuando se verifica un programa es necesario restringir el *scope* de los números enteros y ello puede llevar a que dos String distintos en JML obtengan el mismo identificar en JDynAlloy (ver explicación de cálculo de *scope* en la sección 5.4).

Finalmente, con el objetivo de evitar la colisión de nombres entre variables (ya que JDynAlloy no lo soporta), el simplificador realiza un renombre de las variables miembros de las clases y también de los parámetros de los métodos.

4.2. Traducción al lenguaje JDynAlloy

En esta sección abordaremos el proceso de transformación de código JML al lenguaje de especificación JDynAlloy. Quedará claro que, gracias a que ciertas estructuras de JDynAlloy emulan estructuras propias de lenguajes orientados a objetos, la transformación de un lenguaje orientado a objetos a JDynAlloy es bastante directa.

Por cada clase JAVA, se define un módulo en JDynAlloy que contendrá la definición de sus variables miembros representada por los *fields*; finalmente cada método será representado por un *program* en JDynAlloy. De esta forma, una declaración de clase:

```
public class ClaseA {
    private int field1;
    ...
    private int fieldn;

    public void metodo1(){
        ...
    }

    public int metodo2(int param){
        ...
    }
}
```

se traduce a JDynAlloy como:

```
module ClaseA
sig ClaseA extends java_lang_Object {}
field field1:(ClaseA)->one(Int) {}
...
field field2:(ClaseA)->one(Int) {}

program ClaseA::metodo1[var thiz:ClaseA,
                        var throw:java_lang_Throwable+null]
    Specification { ... }
    Implementation{ ... }

program ClaseA::metodo2[var thiz:ClaseA,
                        var throw:java_lang_Throwable+null,
                        var return:Int,
                        var param:Int]
    Specification { ... }
    Implementation{ ... }
```

A continuación explicaremos cómo se realiza la traducción de ciertas estructuras particulares de JAVA que no tiene un mapeo directo en JDynAlloy.

Dentro de estas podemos mencionar la construcción *try*{} *catch*{}, la cual se resuelve usando una serie de condicionales anidados.

Por ejemplo, el siguiente fragmento de código:

```
...
try{
    stringArray[2] = "unString";
} catch (IndexOutOfBoundsException e) {
    hacer algo...
} finally {
    hacer algo mas...
}
...
```

se traduce a:

```
...
stringArray[2] = "unString";
if not (throw=null) {
    if instanceOf[throw, java_lang_IndexOutOfBoundsException+null] {
        var e:java_lang_IndexOutOfBoundsException+null;
        e:=throw;
        hacer algo...
    } else {
        skip;
    };
} else {
    skip;
};
hacer algo mas...
...
```

El uso de condicionales también se aplica para simular la posibilidad que brinda JAVA de realizar cortes de control en el código mediante el uso de la keyword *return* o *throw*; es decir, dentro de un método un programador puede introducir múltiples puntos de retorno, lo que produce que ciertas parte de código del método no sean ejecutados.

Para lograr este comportamiento, el traductor incorpora una variable booleana a cada programa, la cual indicará si se ejecutó alguna instrucción que produzca un corte de la ejecución del método (ya sea mediante un *return* o una instrucción *throw*). Dicha variable se utiliza para encapsular la ejecución de cada línea de código dentro de un condicional que verifique si se llegó o no a una instrucción de corte de control.

Por ejemplo el código:

```
if (intArray.length == 0) {
    return -1;
}
int i = intArray[2];
i++;
return i;
```

se traduce a:

```
throw:=null;
var exit_stmt_reached:boolean;
exit_stmt_reached:=false;
var es_var__23:Int;
if exit_stmt_reached=false {
    if equ[arrayLength[Object_Array,intArray], 0] {
        if exit_stmt_reached=false {
            return:=negate[1];
            exit_stmt_reached:=true;
        };
    };
};
if exit_stmt_reached=false {
    es_var__23:=arrayAccess[Object_Array,intArray,2];
};
var var_7_i:Int;
if exit_stmt_reached=false {
    var_7_i:=es_var__23;
};
if exit_stmt_reached=false {
    var_7_i:=add[var_7_i,1];
};
if exit_stmt_reached=false {
    return:=var_7_i;
    exit_stmt_reached:=true;
};
```

Como puede observarse, cada sentencia está condicionada por la expresión *(exit_stmt_reached=false)*, que solo será válida si el programa alcanza una instrucción *return* o *throw*.

Con respecto a las anotaciones con el lenguaje JML, en el cuadro 4.4 podemos ver el listado de keywords[1] soportadas. La explicación de cada uno de estos keywords puede consultarse en [20].

Como mencionamos anteriormente, la traducción de JML a JDynAlloy es bastante directa, por lo que la mayoría de estas construcciones tiene un mapeo uno a uno. Por ejemplo las keywords *assert*, *assume*, *invariant*, *requires*, *ensure*, etc. tiene su contraparte directa en el lenguaje JDynAlloy; es decir, una instrucción *requires* en JML se traduce directamente a una instrucción

[1] palabras reservadas por el lenguaje

Keyword de JML implementadas		
assert	in	reach
assignable	invariant	represents
assume	mapsInto	requires
constraint	model	result
ensures	non_null	signals
ensures_redundantly	nullable	signals_only
exceptional_behavior	nullable_by_default	static Invariant
exists	normal_behavior	static Constraint
forall	old	such_that
implies_that	pure	sum

Cuadro 4.4: Keyword de JML implementadas

requires en JDynAlloy.

Para algunos casos hay que encontrar mecanismos para reproducir el comportamiento. Un ejemplo de ésto es la palabra reservada *non_null*, la cual previene que una variable pueda tomar el valor *null*.

Para lograr dicho efecto, el traductor agrega la cláusula $not(variable = null)$ a la precondición o poscondición del método o al invariante de la clase, dependiendo si la restricción está asignada a los parámetros de un método, al valor de retorno del mismo o a una variable miembro de la clase.

Un caso particular surge con la palabra reservada *assignable* la cual crea un *frame constraint* indicando qué variables pueden ser modificadas dentro de un llamado a un método. A continuación se muestra un listado de posibles usos:

1. assignable $\backslash nothing$
2. assignable $\backslash everything$
3. assignable $field_1, \ldots, field_n$
4. assignable objectState
5. assignable $field_1.*$

Los casos 1 a 3 tiene un mapeo directo contra JDynAlloy, pero en los casos 4 y 5 son especiales ya que simplemente se trata de mecanismos para abreviar la escritura. En el punto 4 se usa un *datagroup*, que es un conjunto de locaciones. Para este caso el traductor analiza todas las variables contenidas dentro de dicho conjunto y crea una lista por extensión similar al mostrado en el punto 3. Lo mismo ocurre con el punto 5, con la diferencia que lo que se analiza son todas las variables miembros de la clase representada por *field1*.

La anotación *pure* indica que el método al cual está asignada no deberá modificar ningún valor. Esto lo representamos agregando la instrucción *assignable \nothing* como *frame constraint* del método.

Un punto interesante del lenguaje JML es que no sólo permite especificar el comportamiento normal de un método, sino que también permite especificar comportamientos excepcionales. Esto se logra usando las keywords *normal_behavior* y *exceptional_behavior* respectivamente.

Un comportamiento excepcional indica que una excepción debe ser lanzada si se cumple la precondición. Teniendo en cuenta esto, el código 4.3 se traduce al código 4.4

```
1  /*@
2    @   normal_behavior
3    @     requires index >= 0;
4    @     ensures \result == aList.get(index);
5    @ also
6    @   exceptional_behavior
7    @     requires index < 0;
8    @     signals_only java.lang.IndexOutOfBoundsException;
9    @*/
10   public Integer getElement(int index, /*@non_null@*/ List aList) {
11     return (Integer) aList.get(index);
12   }
```

Código 4.3: Ejemplo de uso de las keywords normal_behavior y exceptional_behavior

Como puede verse se generaron dos casos de especificación, uno para el comportamiento normal y otro para el excepcional. Además, y como indica en la especificación, en caso de cumplirse la precondición del caso excepcional la única excepción posible es una instancia de *IndexOutOfBoundsException*, en caso contrario la verificación debería fallar.

```
program ClaseA::getElement[
  var thiz:ClaseA,
  var throw:java_lang_Throwable+null,
  var return:java_lang_Integer+null,
  var index:Int,
  var aList:java_util_List+null]
Specification {
    SpecCase #0 {
       requires { gte[index,0] }
       requires { not (aList=null) }
       ensures {
          (throw'=null) and
          (some cs_ret_get_3:java_lang_Object+null | {
              callSpec get[aList',throw',cs_ret_get_3,index]) and
              equ[return', cs_ret_get_3]
            }
          )
       }
    }
    SpecCase #1 {
       requires { lt[index,0] }
       requires { not (aList=null) }
       ensures {
          not (throw'=null) and
          instanceOf[throw', java_lang_IndexOutOfBoundsException+null]
       }
    }
}
Implementation {
    throw:=null;
    var exit_stmt_reached:boolean;
    exit_stmt_reached:=false;
    var es_var__11:java_lang_Object+null;
    var es_var__12:java_lang_Integer+null;
    if exit_stmt_reached=false{
       call get[aList,throw,es_var__11,index];
    };
    if exit_stmt_reached=false {
       es_var__12:=((java_lang_Integer+null) & (es_var__11));
    };
    if exit_stmt_reached=false {
        return:=es_var__12;
        exit_stmt_reached:=true;
    };
}
```

Código 4.4: traducción de las keywords normal_behavior y exceptional_behavior

Al analizar el caso de estudio surgió la necesidad de lidiar con literales enteros más grandes de los que se podía analizar utilizando JDynAlloy. El problema surge por que JDynAlloy solo permite analizar números enteros en un rango acotado por el parámetro *bitwidth*. Si por ejemplo dicho parámetro

fuese igual a 3, los enteros estarían limitados al siguiente conjunto

$$\{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$$

El hecho de que en el caso de estudio existieran literales como *100000* requirió que se tome alguna decisión de cómo manejar estos casos. La decisión que se tomó consiste en utilizar la función matemática *módulo* para limitar el tamaño de las constantes. Esta transformación puede inducir a la generación de falsos contraejemplos, pero sin embargo es inevitable alguna acción, dada la naturaleza acotada del análisis que realiza JDynAlloy. Para mitigar los efectos de la transformación se decidió que además de aplicar la función *módulo*, que siempre devuelve un número positivo, se conserve el signo del literal transformado. En el código 4.5 se puede ver el pseudo-código de la función *preventBitwidthOverflow* que se encarga de dicha transformación.

```
Algorithm preventBitwidthOverflow
Input
Literal_value : Int
Bitwidth : Int

Output
Return: Int

Implementation
Var IntSetCardinality : Int
Var BoundedValueWithoutSign : Int

IntSetCardinality := pow(2, Bitwidth)
BoundedValueWithoutSign := abs(literalValue mod (IntSetCardinality / 2))
If (Literal_value>0)
    Return := BoundedValueWithoutSign
Else
    Return := -BoundedValueWithoutSign
End If
```

Código 4.5: función preventBitwidthOverflow

Al finalizar esta etapa se obtiene como salida una especificación JDynAlloy, sobre la cual se ejecutan una serie de procesos, como la resolución de bindings, el cálculo del *scope*, entre otras, para luego comenzar con la traducción al lenguaje de especificación DynAlloy.

Capítulo 5

Traducción al lenguaje DynAlloy

En esta sección abordaremos el proceso de transformación entre los lenguajes de especificación JDynAlloy y DynAlloy. Se expondrá el hecho de que estos lenguajes poseen varias correspondencias directas, simplificando así el proceso de traducción.

5.1. Programas e invocaciones

La relación entre una invocación a un programa (o a su especificación) y su definición se conoce como *binding*. Cuando en un módulo existen múltiples programas con el mismo nombre pero distintos parámetros se dice que el programa esta *sobrecargado*. Se puede ver un ejemplo en el código 5.1.

El proceso de resolución de los *bindings* toma en cuenta dicha particularidad de los programas sobrecargados. JDynAlloy genera sus *bindings* utilizando reglas similares a las de Java [24]. La principal diferencia es que Java tiene en cuenta la visibilidad de los métodos (private, public, package), que es un concepto que no tiene contrapartida en JDynAlloy.

El hecho de que en JDynAlloy varios métodos puedan tener el mismo nombre debe ser manejado por el traductor, ya que DynAlloy no lo permite.

La solución consiste en renombrar los programas, para que no se repitan los nombres, y modificar las invocaciones para reflejar dicho renombre.

```
1  program C::f[
2       var thiz:C
3       var throw:java_lang_Throwable+null,
4       var return:Int,
5       var i:Int]
6  Specification
7  ...
8  Implementation
9  ...
10
11 program C::f[
12      var thiz:C,
13      var throw:java_lang_Throwable+null,
14      var return:Int,
15      var o:java_lang_Object]
16 Specification
17 ...
18 Implementation
19 ...
```

Código 5.1: Ejemplo de un programa sobrecargado

5.1.1. Métodos virtuales

En JDynAlloy un módulo puede sobreescribir un programa declarado por su *módulo padre* solo sí el programa original está marcado como *virtual*.

El código 5.2 muestra un ejemplo de dicha situación. En la traducción a DynAlloy se reescribirá la definición del método virtual. El objetivo es simular al *dispatching* en tiempo de ejecución de los lenguajes orientados a objetos. El procedimiento se encuentra explicado con más detalle en [23]. Podemos ver cómo el código 5.2 se traduce a DynAlloy en el pseudo-código 5.3.

Se puede ver en la líneas 10 a la 15 (del código 5.3) que fue agregado el código del dispatcher que invocará al método concreto correspondiente al tipo concreto de la instancia *thiz*.

5.2. Resolución del callSpec

El *callSpec* es la única fórmula de JDynAlloy que no posee su contrapartida en DynAlloy. Como finalmente las expresiones deben transformarse en expresiones DynAlloy es necesario reemplazar las fórmulas *callSpec* por

```
1  module A
2  sig A extends java_lang_Object {}
3
4  virtual program A::f[
5       var thiz:A,
6       var throw:java_lang_Throwable+null,
7       var return:Int]
8  Specification { ... }
9  Implementation {
10     return := 1;
11 }
12
13 module B
14 sig B extends A {}
15
16 program B::f[
17      var thiz:B,
18      var throw:java_lang_Throwable+null,
19      var return:Int]
20 Specification { ... }
21 Implementation {
22     return := 2;
23 }
24
25 module C
26 sig C extends A {}
27
28 program C::f[
29      var thiz:C,
30      var throw:java_lang_Throwable+null,
31      var return:Int]
32 Specification { ... }
33 Implementation {
34     return := 3;
35 }
```

Código 5.2: Ejemplo de un programa sobrecargado

código DynAlloy equivalente.

Consideremos el programa *sum* declarado en el código 5.4. Para llamar a este método deberíamos utilizar la siguiente sintaxis:

```
callSpec IntegerUtils_sum[throw, sum_return, x, 2]
```

Dicha invocación es equivalente a la fórmula del código 5.5. Por lo tanto al realizar la traducción de JDynAlloy a DynAlloy reemplazaremos el llamado por la fórmula que se aprecia en el código 5.5.

```
1  module A
2  sig A extends java_lang_Object {}
3
4  program A_f_0    [
5       var thiz:A,
6       var throw:java_lang_Throwable+null,
7       var return:Int]
8  Specification { ... }
9  Implementation {
10      if thiz es de tipo B {
11          call B_f_0[thiz, throw, return]
12      }
13      if thiz es de tipo C {
14          call C_f_0[thiz, throw, return]
15      }
16 }
17
18 module B
19 sig B extends java_lang_Object {}
20
21 program B_f_0[
22      var thiz:B,
23      var throw:java_lang_Throwable+null,
24      var return:Int]
25 Specification { ... }
26 Implementation { ... }
27
28 module C
29 sig C extends java_lang_Object {}
30
31 program C_f_0[
32      var thiz:C,
33      var throw:java_lang_Throwable+null,
34      var return:Int]
35 Specification { ... }
36 Implementation { ... }
```

Código 5.3: Ejemplo de dispatch

5.3. Resolución de modifies

Para verificar las restricciones impuestas por la cláusula *modifies* se agregan ciertas fórmulas a la poscondición del método analizado. Dependiendo de la expresión afectada por la cláusula *modifies* se agregaran distintas fórmulas.

El código 5.6 nos muestra el caso en el que el *modifies* afecta a un field y en el código 5.7 se muestra la fórmula agregada durante la resolución de *modifies*, a la poscondición para reflejar dicha restricción.

El caso en el que *modifies* afecta al field de un field se encuentra representado en código 5.8, el código adicional generado para dar soporte a esta

```
program IntegerUtils::sum[
      var throw:java_lang_Throwable+null,
      var return:Int,
      var left:Int,
      var right:Int]
Specification {
      SpecCase #0 {
         requires { geq[left,0] and geq[right,0]) }
         ensures { throw'=null and return'=sum[left, right] }
      }
      SpecCase #1 {
         requires { lt[left,0] or lt[right,0] }
         ensures { throw'=java_lang_IllegalArgumentExceptionLit }
      }
 }
```

Código 5.4: Programa a ser llamado dentro de una especificación

```
(geq[x,0] and geq[2,0]) implies (throw'=null and sum_return'=sum[x, 2]) and
(x<0 or 2<0) implies (throw'=java_lang_IllegalArgumentExceptionLit)
```

Código 5.5: Código DynAlloy equivalente

```
module A
sig A extends java_lang_Object {} {}
field x:(A)->one(Int) {}
field y:(A)->one(Int) {}
field z:(A)->one(Int) {}

program A::f[
  var thiz:A,
  var throw:java_lang_Throwable+null,
  var return:boolean,
  var o:java_lang_Object+null]
Specification {
    SpecCase #0 {
        modifies { thiz.x }
    }
}
```

Código 5.6: Modifies afectando un field

restricción se encuentra en el código 5.9.

Es posible también especificar una cláusula *modifies* que evite que se modifique el contenido de un arreglo. El código 5.10 presenta un ejemplo de dicha situación, a su vez el código 5.11 representa la poscondición que será agregada para mantener la restricción producto de dichos *modifies*.

```
ensures {
    all mod_q_1:A | {
        not (mod_q_1.x=mod_q_1.x')
        implies mod_q_1=thiz'
    } and
    (thiz.y=thiz.y') and
    (thiz.z=thiz.z')
}
```

Código 5.7: Restricción adicional por *modifies*

```
module B
sig B  {} {}
field s:(B)->one(Int) {}

sig A extends java_lang_Object {} {}
field b:(A)->one(B+null) {}
field c:(A)->one(B+null) {}
field d:(A)->one(boolean) {}

program A::f[
  var thiz:A,
  var throw:java_lang_Throwable+null,
  var return:boolean,
  var o:java_lang_Object+null]
Specification
{
    SpecCase #0 {
        modifies { thiz.b.s }
    }
}
...
```

Código 5.8: Modifies afectando un field de un field

```
ensures {
    all mod_q_1:B | {
        not (mod_q_1.s=mod_q_1.s')
        implies (mod_q_1=thiz'.b')
    }
    and thiz.b=thiz.b' and
    (thiz.c=thiz.c')  and
    (thiz.d=thiz.d')
}
```

Código 5.9: Restricción adicional por *modifies*

Cabe aclarar que, aunque JDynAlloy permite expresar la cláusula *mo-*

difies sobre una posición concreta del arreglo, la implementación actual no realiza una traducción fiel, ya que la restricción resultante evita la modificación de todas y cada una de las posiciones del arreglo. La motivación de esta limitación es emular el comportamiento de JMLForge para poder realizar una comparación más fiel sobre los aspectos clave de las herramientas JMLForge y TACO.

```
1  module A
2  sig A extends java_lang_Object {} {}
3  field l:(A)->one(java_lang_SystemArray+null) {}
4  field m:(A)->one(int) {}
5  field n:(A)->one(int) {}
6
7  program A::f[
8    var thiz:A,
9    var throw:java_lang_Throwable+null,
10   var return:boolean,
11   var index:Int]
12 Specification
13 {
14     SpecCase #0 {
15         modifies {
16             arrayAccess[Object_Array,thiz.ar,l,index]
17         }
18     }
19 }
20 ...
```

Código 5.10: Modifies afectando un arreglo

```
1  ensures {
2      all mod_array_q_0:java_lang_SystemArray | {
3          not mod_array_q_0.Object_Array'=mod_array_q_0.Object_Array
4          implies mod_array_q_0=thiz.l
5       }
6       and thiz.l=thiz.l' and
7      (thiz.m=thiz.m') and
8      (thiz.n=thiz.n')
9  }
```

Código 5.11: Restricción adicional por *modifies*

Cuando *modifies* se utiliza en conjunción con el keyword *EVERYTHING* no es necesario agregar ninguna restricción a la poscondición del programa, ya que justamente esta combinación se utiliza para indicar que el programa no tiene restricciones a la hora de modificar el estado.

El caso opuesto es cuando no se especifica cláusula *modifies* alguna para el programa, como se muestra en el código 5.12. En ese caso se asume que el programa no debe modificar el estado. En el código 5.13 se explícita la resolución de este caso.

```
1  module A
2  sig A extends java_lang_Object {} {}
3  field x:(A)->one(Int) {}
4  field y:(A)->one(Int) {}
5  field z:(A)->one(Int) {}
6
7  program A::f[
8    var thiz:A,
9    var throw:java_lang_Throwable+null,
10   var return:boolean,
11   var index:Int]
12 Specification
13 {
14     ensures {throw = null}
15 }
16 ...
```

Código 5.12: No se especifica modifies

```
1  ensures {
2     ensures {
3        (thiz.y=thiz.y') and
4        (thiz.z=thiz.z') and
5        (thiz.x=thiz.x')
6     }
7  }
```

Código 5.13: Restricción adicional por *modifies*

5.4. Cálculo de Scope

JDynAlloy hereda de Alloy el mecanismo que permite limitar el *scope* o cantidad máxima de instancias que compondrán los contraejemplos encontrados. El límite se establece indicando el máximo de instancias por tipo.

En el código 5.14 están definidos los tipos A, B y C. Supongamos que tenemos las instancias concretas A_0, A_1, B_0, B_1, C_0 y C_1. En la figura 5.1 se ve cada tipo representado como un conjunto y a cada instancia concreta

```
1 module A
2 sig A  {} {}
3
4 module B
5 sig B extends A {} {}
6
7 module C
8 sig C extends A {} {}
```

Código 5.14: Ejemplo de jerarquía JDynAlloy

representada como un punto. Se puede apreciar que tanto el conjunto B como el C poseen dos elementos, mientras que el conjunto A posee seis (todas las instancias están dentro del conjunto A).

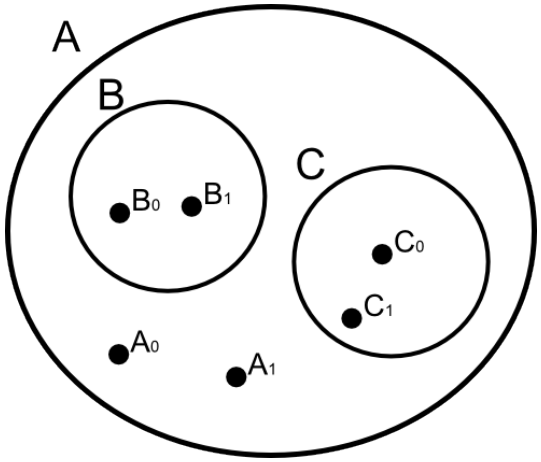

Figura 5.1: Scope estilo Alloy.

Como A es el conjunto con más elementos y como su cantidad de elementos es seis, entonces JDynAlloy necesita un scope igual o mayor que seis para considerar dicho escenario.

Por otro lado JMLForge utiliza una representación de la herencia de tipos distinta a la vista en la sección 3.2.11. En la figura 5.2 se puede apreciar la manera en que JMLForge interpreta la semántica de *scope*, ya que en JMLForge el conjunto A sólo contiene las instancias que pertenecen estrictamente[1] al tipo A. Esto es porque los conjuntos que representan los tipos son todos disjuntos entre sí. Para establecer si la instancia concreta C_0 es instancia del

[1] X_0 es estrictamente del tipo X si está declarada como de tipo X

tipo A, se debe verificar la siguiente condición:

$$C_0 \; in \; (A \cup B \cup C)$$

Del lado derecho del *in* se debe realizar la unión de cada uno de los conjuntos que representan a los tipos que descienden de A.

Como el conjunto con más elementos tiene dos elementos JMLForge necesita de scope igual o mayor que dos para considerar el escenario de la figura 5.2.

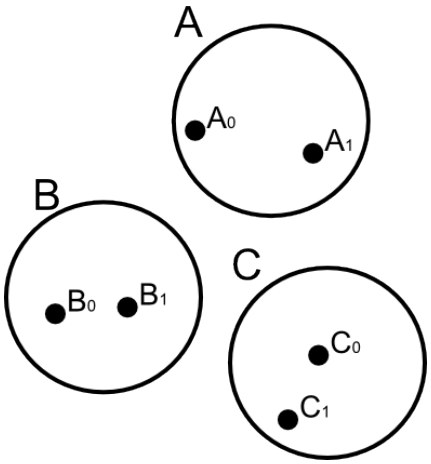

Figura 5.2: Scope estilo JMLForge.

Para poder realizar una comparación entre JDynAlloy y JMLForge es necesario utilizar la misma semántica del *scope*. Por lo que se decidió incorporar un mecanismo para permitir a JDynAlloy emular la semántica del *scope* de JMLForge.

El mecanismo consiste en dos etapas. En la primera etapa se calcula cuál es el *scope* que se requiere por cada tipo de dato. Tomemos por ejemplo la jerarquía de la figura 5.3 y utilicemos el procedimiento con un valor de *scope* igual a dos.

Por otro lado, ya que B y C son nodos hojas y representan tipos concretos, se debe asignar *scope* igual a dos a cada uno, como se puede apreciar en la figura 5.4.

El *scope* resultante para el tipo A será de dos (la cantidad de instancias estrictas de A) más la suma del *scope* de sus hijos, en este caso cuatro, como

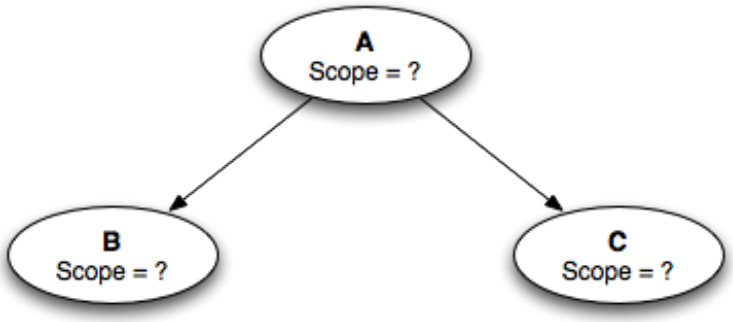

Figura 5.3: Jerarquía de ejemplo para el cálculo de scope.

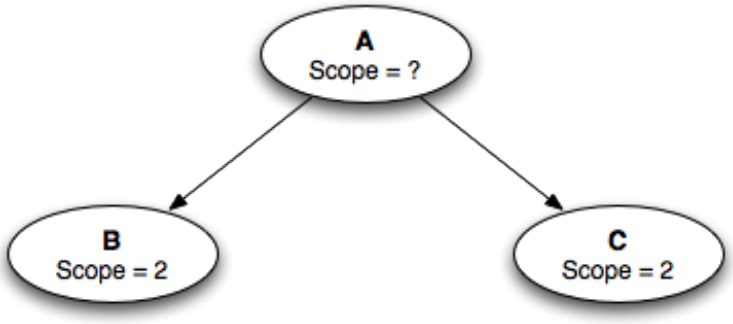

Figura 5.4: El *scope* fue asignado a las hojas

se muestra en la figura 5.5.

De haberse declarado a A como abstracto el *scope* para éste nodo habría sido cuatro.

La segunda y última etapa del mecanismo consiste en generar literales que representan las distintas instancias que pueden existir de cada tipo concreto. Dichas literales se pueden ver en el código 5.15.

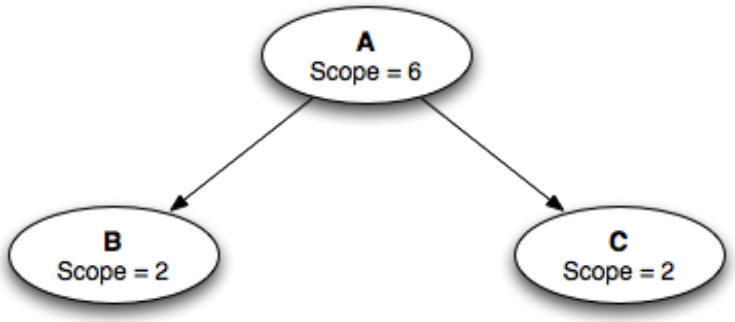

Figura 5.5: Los nodos intermedios son resueltos usando el scope de sus hijos

```
1  one sig A_1 extends A {} {}
2  one sig A_2 extends A {} {}
3  one sig B_1 extends B {} {}
4  one sig B_2 extends B {} {}
5  one sig C_1 extends C {} {}
6  one sig C_2 extends C {} {}
```

Código 5.15: Constantes generadas de acuerdo al *scope*

5.5. Relevance Analysis

Hemos visto en la sección 3.2.2 como los invariantes son una herramienta valiosa a la hora de especificar un módulo. Sin embargo verificar los invariantes de todas las instancias consideradas al momento de realizar una verificación puede resultar en una *VC* extensa y, posiblemente, que no sea factible la verificación de dicha fórmula en un período de tiempo razonable.

Por este motivo fue incorporado a JDynAlloy un análisis de qué módulos son requeridos [4] a la hora de verificar un programa en particular. Lamentablemente, este análisis es imperfecto y existe la posibilidad de que no detecte módulos que efectivamente sean requeridos para el análisis deseado. La consecuencia de esto sería que a las instancias de dichos módulos no se les exigiría cumplir con su invariante, por ende no estarían necesariamente bien formadas, lo que podría inducir a JDynAlloy a reportar contraejemplos inválidos o a no reportar contraejemplos existentes.

El algoritmo descripto a continuación fue presentado en [28]. El mismo se basa en encontrar la escena asociada con el programa que se desea analizar.

La escena es una tupla ⟨M,F,P⟩ donde M es un conjunto de módulos, F es un conjunto de Fields y P es un conjunto de programas. La escena representa los módulos, fields y programas que son relevantes en el análisis de dicho programa. El pseudo código 5.16 muestra el algoritmo principal, los pseudo códigos 5.17 y 5.18 muestran algoritmos auxiliares.

El algoritmo calcula la escena siguiendo los siguientes criterios descriptos en [28]:

1. Todo módulo en M tiene a su súper módulo en M (código 5.16 líneas 33 a la 35).

2. El tipo de cada field en F está en M (código 5.16 línea 30).

3. El tipo que declara un programa en P y los tipos de cada uno de los argumentos de dicho programa están en M (código 5.16 líneas 16 a la 18).

4. Para todo programa en P la escena de su especificación está incluida en la escena de P (código 5.16 líneas 19 a la 22).

5. La escena de todo field especificado mediante la cláusula *represents* en F está contenida en la escena (código 5.16 líneas 36 a la 39).

6. La escena de cada invariante de cada módulo en M está contenida en la escena (código 5.16 líneas 40 a la 42).

7. La escena de cualquier expresión que contiene una invocación a método incluye la escena de ese método. (código 5.18 líneas 13 a la 35).

5.6. Traducción de las sentencias JDynAlloy

Muchas sentencias de JDynAlloy poseen una traducción directa a DynAlloy, otras requieren acciones auxiliares de DynAlloy para su traducción. A continuación veremos el detalle de la traducción de cada una de ellas.

Declaración de variables: una declaración de variable en JDynAlloy se traduce directamente a una declaración de variable en DynAlloy.

```
1  Algorithm RelevanceAnalysis
2  Input
3  Program_to_check: Program
4  Modules: Set of Module
5
6  Output
7  Return: Set of Modules
8  Implementation
9  Var Scene: scene
10 Var Marked: Set of Program+Module+Field
11
12 scene = <{ Program_to_check }, {}, {}>
13 marked = {}
14 While ∃ X / X ∈ scene ∧ (X ∉ marked) marked.add( X )
15     If X is Program
16         For each P ∈ X.Parameters
17             scene.add( p.Type)
18         End For each
19         For each C ∈ X.SpecCase
20             AnalyzeFormula( scene, C.Precondition )
21             AnalyzeFormula( scene, C.Postcondition )
22         End For each
23         If X = Program_to_check
24             For each s ∈ x.Body
25                 AnalizeStatement( scene, s )
26             End For each
27         End If
28     End If
29     If X is Field
30         scene.add( x.Type )
31     End If
32     If X is Module
33         If (X.Super ≠ null)
34             Scene.add( X.Super.Type )
35         End If
36         For each J ∈ X.Represents
37             Scene.add( J.ReferencedField.Type )
38             AnalyzeFormula(scene, J.Predicate )
39         End For each
40         For each i ∈ X.Invariant
41             AnalyzeFormula(scene, I.Predicate )
42         End For each
43 End While
```

Código 5.16: Algoritmo RelevanceAnalysis

Assert: en el código 5.19 presentamos un ejemplo del uso de la instrucción *assert* en JDynAlloy, el cual luego de se traducido a DynAlloy queda como se muestra en el código 5.20.

Para más detalles de esta traducción se puede consultar [3].

```
1  Algorithm AnalyzeFormula
2  Input/ Output
3  scene:Scene
4
5  Input
6  p: Formula
7
8  Implementation
9  For each c ∈ MethodCall in P
10     scene.Add( c.Method )
11 End For each
12 For each f ∈ FieldExpression in P
13     Scene.Add( f.Prefix.Type )
14     scene.Add( f.Field )
15 End For each
16 For each v ∈ Quanitified Variable in P
17     scene.Add( v.Type )
18 End For each
```

Código 5.17: Algoritmo AnalyzeFormula

```
1  Algorithm AnalyzeStatement
2  Input/ Output
3  scene:Scene
4  Input
5  S: Statement
6
7  Implementation
8  If S is VariableDeclaration
9      scene.Add( S.Type )
10 Else If S is CreateObject
11     scene.Add( S.Type )
12 Else
13 For each c ∈ MethodCall in S
14     scene.Add( c.Method )
15 End For each
16 For each f ∈ FieldExpression in S
17     Scene.Add( f.Prefix.Type )
18     scene.Add( f.Field )
19 End For each
20 For each v ∈ Quanitified Variable in S
21     scene.Add( v.Type )
22 End For each
23 End If
```

Código 5.18: Algoritmo AnalyzeStatement

Assume: la traducción a DynAlloy es directa ya que DynAlloy posee la sentencia *assume* con la misma semántica [3].

```
1  program A::f[
2      var thiz:A,
3      var throw:java_lang_Throwable+null,
4      var return:boolean,
5      var i:Int]
6  {
7     assert gt[i,0];
8  }
```

Código 5.19: Ejemplo de uso de la instrucción assert.

```
1  program A_f_0[
2      thiz:A,
3      throw:java_lang_Throwable+null,
4      return:boolean,
5      i:Int,
6      ...
7  ] var [
8    assertionFailure:boolean,
9  ...
10 ]{
11    assertionFailure:=false;
12    throw:=null;
13    if gt[i,0]  {
14       skip
15    } else {
16       assertionFailure:=true
17    }
18    if equ[assertionFailure,true]  {
19       throw:=AssertionFailure
20    } else {
21       skip
22    }
23 }
```

Código 5.20: Traducción de una instrucción assert.

Havoc: para la traducción de ésta sentencia se utilizan acciones DynAlloy provistas para este fin, la acción que se utilizará en cada traducción depende del tipo de la expresión parámetro. Para más información referirse a [3].

Skip: la traducción a DynAlloy es directa ya que DynAlloy posee la sentencia *skip* con la misma semántica.

CreateObject: la traducción de ésta sentencia es explicada con detalle en [16]. El *heap* de JDynAlloy se representa con un conjunto de elementos llamado *usedObjects*. A ese conjunto pertenecen los objetos que ya han sido

asignados. La acción de DynAlloy *getUnusedObject* es creada en la traducción con el objetivo de obtener un objeto *fresco*, es decir un objeto que aún no se encuentre en el *heap*, y agregar dicho objeto al *heap*. Se puede ver la acción *getUnusedObject* en el código 5.21. A continuación un ejemplo que ilustra de dicha traducción.

```
createObject<A>[obj];
```

se traduce al siguiente código:

```
getUnusedObject[obj,usedObjects];
assume instanceOf[obj,A];
```

Asignación: la traducción de esta sentencia es explicada con detalle en [16]. La traducción de esta sentencia varía su comportamiento dependiendo del tipo de expresión que se encuentra a la izquierda del operador asignación. En el código 5.22 se muestra un ejemplo en JDynAlloy de tres casos de asignación que son explicados a continuación. Mientras que en el codigo 5.23 se muestra la traducción a DynAlloy de esos mismos casos.

Cuando del lado izquierdo de la asignación aparece una variable, se traduce directamente al operador de asignación de DynAlloy. Un ejemplo se

```
1  pred getUnusedObjectPost[
2    usedObjects1:set java_lang_Object,
3    usedObjects0:set java_lang_Object,
4    n1: java_lang_Object+null
5  ]{
6    n1 !in usedObjects0
7    usedObjects1 = usedObjects0 + (n1)
8  }
9
10 action getUnusedObject[
11   n:java_lang_Object+null,
12   usedObjects:set java_lang_Object
13 ]{
14     pre {
15        TruePred[]
16     }
17     post {
18        getUnusedObjectPost[usedObjects',usedObjects,n']
19     }
20 }
```

Código 5.21: Acción getUnusedObject

muestra en la línea 7 del código 5.22, que se traduce como se muestra en la línea 10 del código 5.23.

El caso de actualizar el valor de un arreglo se traduce utilizando una acción DynAlloy cuya declaración se puede ver en el código 5.24. En la línea 8 del código 5.22 se muestra un ejemplo de actualización de un arreglo, que se traduce como se muestra en la línea 11 del código 5.23.

Un field en DynAlloy es una función entre el tipo del módulo y el tipo del field contenido en dicho módulo. Por lo tanto, para actualizar el valor de un field es necesario modificar dicha función utilizando el operador *relational overriding*. Se muestra en la línea 9 del código 5.22 un ejemplo de dicha actualización, que se traduce como se muestra en la línea 12 del código 5.23.

```
1  program A::f[A,
2      var throw:java_lang_Throwable+null,
3      var x:Int,
4      var y:Int,
5      var a:java_lang_SystemArray]
6  {
7      x:=y;
8      arrayAccess[Object_Array,a,0]:=2;
9      thiz.field1:=2;
10 }
```

Código 5.22: Asignación JDynAlloy

```
1  program A_f_0[
2    thiz:ar_edu_dynjml4alloy_bug_ifexpression_A,
3    throw:java_lang_Throwable+null,
4    var x:Int,
5    var y:Int,
6    var a:java_lang_SystemArray]
7  ...
8  ] var [
9  ] {
10     x:=y
11     updateArray[Object_Array,a,0,2]
12     field1:=(field1)++((thiz)->(2))
13 }
```

Código 5.23: Traducción de asignación a DynAlloy

Program call: La traducción a DynAlloy es directa ya que DynAlloy posee la sentencia *call* con la misma semántica.

```
1  pred updateArrayPost[
2    Object_Array1: java_lang_SystemArray -> (seq univ),
3    Object_Array0: java_lang_SystemArray -> (seq univ),
4    array: java_lang_SystemArray+null,
5    index:Int,
6    elem:univ
7  ]{
8    Object_Array1 = Object_Array0 ++
9    (array->(array.Object_Array0++(index->elem)))
10 }
11
12 action updateArray[
13   Object_Array:java_lang_SystemArray->(seq univ),
14   array:java_lang_SystemArray+null,
15   index:Int,
16   elem:univ
17 ]{
18     pre {
19       TruePred[]
20     }
21     post {
22       updateArrayPost[Object_Array',
23                       Object_Array,
24                       array,
25                       index,
26                       elem]
27     }
28 }
```

Código 5.24: Acción updateArray

Bloque: La traducción a DynAlloy es directa ya que DynAlloy posee la sentencia *sequential composition* con la misma semántica.

If: La traducción a DynAlloy es directa ya que DynAlloy posee la sentencia *if* con la misma semántica.

While: La traducción de esta sentencia utiliza la sentencia *repeat* de DynAlloy. Para detalles sobre las distintas formas de traducir la sentencia *while* a DynAlloy referirse a [3]. A continuación un ejemplo básico de dicha traducción:

```
while lt[i, 10] {
    i:=add[i,1];
};
```

se traduce al siguiente código:

```
repeat {
   assume gt[i, 0];
   i:=add[i,1]
};
assume not ( gt[i,0] );
```

5.7. Generación de assertCorrectness

Para realizar la verificación del programa se genera en DynAlloy un *assertCorrectness* como se muestra en el código 5.25.

```
1
2  assertCorrectness check_A_f_0[...]{
3      pre={
4          precondition_A_f_0[...]
5      }
6      program={
7          call A_f_0[...]
8      }
9      post={
10         postcondition_A_f_0[...]
11     }
12 }
```

Código 5.25: Ejemplo de assertCorrectness para el programa A::f

El predicado precondición (línea 4) será construido como la conjunción de los siguientes elementos:

1. La precondición del programa que se desea analizar.
2. Los *class invariants* de todos los módulos.
3. Los *object invariants* de los módulos relevantes para el análisis del programa.

El programa DynAlloy *A_f_0* es la traducción del programa JDynAlloy a DynAlloy. Dicha traducción consiste en la traducción de las sentencias que lo componen. En la aserción se invoca a *A_f_0* (línea 7).

El predicado poscondición (línea 10) será generado realizando la conjunción de los siguientes elementos:

1. La poscondición del programa que se desea analizar.

2. Los *class invariants* de todos los módulos.

3. Los *object invariants* de los módulos relevantes para el análisis del programa.

4. Los *constrains* del módulo analizado.

Capítulo 6

Experimentación

En esta sección mostraremos los resultados obtenidos al realizar la verificación del subsistema de recuento de votos de KOA. Como mencionamos anteriormente, este análisis estará limitado a sólo 8 clases que forman el core de la funcionalidad.

Salvo que se indique lo contrario, todas las experimentaciones realizadas fueron corridas en una computadora con procesador Intel®Core™2 Quad Q8200 a 2.34GHZ, 4GB de RAM, corriendo el sistema operativo Windows Vista (64-bit). Debido a que las herramientas son single-threaded, no se toma ventaja de los múltiples procesadores.

Dentro de la clases verificadas algunos métodos no poseen especificación explícita, por lo que no fueron analizados en ninguna de las clases. Dichos métodos son: *toString*, *equals*, *hashCode*, *compareTo*.

6.1. Verificación de KOA usando JMLForge

Con el objetivo de lograr una comparación de ambas herramientas se intentó reproducir la verificación del subsistema de recuento de votos de KOA usando la herramienta JMLForge [8].

6.1.1. Verificación de KOA usando JmlForge realizada por el MIT

En esta sección mostraremos los resultados obtenidos al verificar las 8 clases que forman el core de la funcionalidad del subsistema de recuento de votos de KOA, publicados en [8].

La experimentación se realizó en una computadora Mac Pro con un procesador Dual-Core™Intel®Xeon de 3GHz y 4.5GB de RAM, corriendo el sistema operativo Mac OS X 10.4.11.

Clases	**Métodos**	**sloc**	**slocc**	**dslocc**	**dslocc/métodos**	**violaciones**	**scope medio**	**t. medio(seg)**
AuditLog	90	286	1237	1617	18.0	1	5.0	2.3
CandidateListMetadata	10	72	246	643	64.3	1	5.0	33.6
Candidate	12	103	363	1013	84.4	1	5.0	59.3
KiesKring	15	119	482	1272	84.8	5	5.0	249.7
District	6	53	163	543	90.5	0	5.0	18.5
KiesLijst	12	111	367	1432	119.3	4	5.0	104.6
CanidateList	13	130	493	1746	134.3	3	4.5	1416.8
VoteSet	11	113	400	2688	244.4	4	3.7	1783.9
Suma o promedio	169	987	3751	10954	64.8	19	4.8	262.7

Cuadro 6.1: Resumen del análisis estático de cada clase

El cuadro 6.1 muestra un resumen de los resultados obtenidos. La columna *sloc* indica el número de líneas de código de cada clase; *slocc* incluye código y comentarios también. *dslocc* suma al valor de *slocc* la cantidad de líneas de comentarios de las clases de la cual dependen directamente. La columna *dslocc/métodos* es una aproximación de la complejidad de realizar el análisis modular de un método dentro de la clase. La columna *Violaciones* indica la cantidad de errores encontrados en cada clase, ya sea en el código dentro del

método o en la especificación del mismo. finalmente *scope medio* y *tiempo medio (seg)* indican el límite medio de la cantidad de objetos a existir en el heap en cualquier ejecución y el tiempo medio de ejecución de los métodos dentro de la clase, respectivamente.

Como puede observarse en el cuadro 6.1 se encontraron errores en un total de 19 métodos de los 169 analizados. Dichos errores no habían podido ser encontrados usando las herramientas *jmlunit* y *ESC/Java2*. El cuadro 6.2 muestra un detalle de los métodos con errores y sus respectivos límites utilizados para la verificación.

Clases	**Métodos**	**Limites de verificación**
AuditLog	getCurrentTimeStamp	2 / 1 / 1
Candidate	init	2 / 3 / 1
CandidateList	init	2 / 3 / 1
CandidateList	addDistrict	2 / 3 / 1
CandidateList	addKiesLijst	2 / 3 / 1
CandidateListMetada	init	1 / 3 / 1
KiesKring	init	2 / 3 / 1
KiesKring	addDistrict	1 / 3 / 1
KiesKring	addKiesLijst	2 / 3 / 1
KiesKring	clean	2 / 3 / 3
KiesKring	make	2 / 3 / 1
KiesLijst	addCandidate	2 / 3 / 1
KiesLijst	clear	1 / 3 / 3
KiesLijst	compareTo	2 / 3 / 1
KiesLijst	make	2 / 3 / 1
VoteSet	addVote(int)	2 / 3 / 1
VoteSet	addVote(String)	1 / 3 / 1
VoteSet	validateKiesKringNumber	2 / 3 / 1
VoteSet	validateRedundantInfo	2 / 3 / 1

Cuadro 6.2: Violaciones de la especificación y límites (scope/tamaño de enteros/ unrollings) necesarios para la detección del error

6.1.2. Reproducción de la Verificación realizada en el MIT

Al reproducir el experimento de verificación del subsistema de KOA realizado en el MIT, encontramos una mayor cantidad de métodos con violaciones, y algunos de los métodos reportados no pudieron ser verificados debido a inconvenientes con la herramienta *JMLForge* (ver cuadro 6.3). A pesar de ello pudimos reproducir el 97.7 % de la experimentación, por lo que consideramos que la falta de evaluación de éstos métodos no afectan los resultados obtenidos.

En el cuadro 6.3 figura la lista de métodos que no pudieron ser analizados debido a que la herramienta *JMLForge* abortó la verificación arrojando una excepción. La última columna de dicha tabla indica si el método reportaba violación en la verificación realizada por la gente de MIT.

En los métodos en los que no pudimos reproducir la verificación, se tomarán los resultados publicados en [8].

Clases	**Método**	**Error**	**Reporta Violación?**
CandidateList	totaPartyCount	NullPointerException	NO
CandidateList	addCandidate	NullPointerException	NO
CandidateList	clear	NullPointerException	NO
KiesKring	clear	IllegalArgumentException	SI
KiesLijst	clear	IllegalArgumentException	SI
VoteSet	resetVotes	NullPointerException	NO

Cuadro 6.3: Métodos que no pudieron ser verificados con JMLForge debido a errores en la herramienta.

Para este experimento se estableció un tiempo máximo de verificación (timeout) de 30 minutos. Para los métodos que superaron este umbral se procedió a reducir progresivamente el scope hasta lograr una verificación exitosa dentro de dicho umbral.

En el cuadro 6.4 puede observarse un resumen del experimento. La columna 'Violaciones' muestra la cantidad total de inconsistencias entre el método

Clases	Violaciones	Scope medio	T. medio(seg)
AuditLog	2	5.0	7.3
CandidateListMetadata	1	5.0	37.0
Candidate	1	5.0	69.1
KiesKring	5	5.0	43.4
District	0	5.0	24.6
KiesLijst	3	5.0	60.6
CandidateList	4	4.5	116.7
VoteSet	4	3.7	109.2
Suma o *promedio*	**20**	*4.8*	*58.9*

Cuadro 6.4: Resultado de la reproducción de la verificación de KOA con JML-Forge.

y su especificación encontradas por cada clase. Para obtener los valores de las 2 columnas siguientes se tomó el promedio de scope y tiempo de cada método de una clase, para luego calcular la media para dicha clase.

Si comparamos la verificación realizada en [8] (ver cuadro 6.1) con la reproducción de dicha verificación realizada por nosotros (ver cuadro 6.5) podemos ver que la segunda arrojó mayor cantidad de violaciones. Las diferencias están resaltadas en letra **negrita**.

6.2. Verificación de KOA usando TACO

A continuación mostraremos los resultados obtenidos de la verificación del subsistema de recuento de votos de KOA usando la herramienta TACO. Para ello nuevamente se estableció un timeout de 30 minutos y se redujo el scope de los métodos cuya verificación superó dicho umbral.

La progresiva reducción de scope en la verificación de un método puede producir que la verificación sea inválida debido a que dicho scope no alcanza para encontrar al menos una traza de ejecución del programa que haga cumplir la precondición del método. Por ese motivo, previa a la ejecución de la verificación, se ejecutó una simulación del método a ser verificado.

Clases	Métodos	Limites de verificación	Rep. por el MIT
AuditLog	**addKiesKring**	**2 / 3 / 1**	**No**
AuditLog	getCurrentTimeStamp	2 / 1 / 1	
Candidate	init	2 / 3 / 1	
CandidateList	init	2 / 3 / 1	
CandidateList	addDistrict	2 / 3 / 1	
CandidateList	**addKiesKring**	**2 / 3 / 1**	**No**
CandidateList	addKiesLijst	2 / 3 / 1	
CandidateListMetada	init	1 / 3 / 1	
KiesKring	init	2 / 3 / 1	
KiesKring	addDistrict	1 / 3 / 1	
KiesKring	addKiesLijst	2 / 3 / 1	
KiesKring	clear	2 / 3 / 3	
KiesKring	make	2 / 3 / 1	
KiesLijst	addCandidate	2 / 3 / 1	
KiesLijst	clear	1 / 3 / 3	
KiesLijst	make	2 / 3 / 1	
VoteSet	addVote(int)	2 / 3 / 1	
VoteSet	addVote(String)	1 / 3 / 1	
VoteSet	validateKiesKringNumber	2 / 3 / 1	
VoteSet	validateRedundantInfo	2 / 3 / 1	

Cuadro 6.5: Violaciones de la especificación y límites (scope/tamaño de enteros/ unrollings) necesarios para la detección del error con JMLForge.

Se define la simulación de un método a la verificación de la existencia de al menos una traza del método cuyo pre-estado haga válida su precondición.

De esta manera nos aseguramos de no incurrir en falsos negativos triviales producidos por la falta de trazas que cumplan la precondición.

Para realizar el experimento se utilizó la semántica de scope que usa *JMLForge* (ver sección 5.4) y además se utilizó el análisis de relevant classes (ver sección 5.5).

En el cuadro 6.6 podemos ver los resultados obtenidos en el análisis. En total se encontraron 17 métodos que violaban su especificación.

Clases	Violaciones	Scope medio	T. medio(seg)
AuditLog	2	4.9	68.2
CandidateListMetadata	0	5.0	93.5
Candidate	1	4.9	75.8
KiesKring	5	4.9	109.4
District	0	5.0	44.8
KiesLijst	2	4.7	258.4
CanidateList	6	3.6	959.5
VoteSet	1	3	499.7
Suma o *promedio*	**17**	*4.5*	*263.7*

Cuadro 6.6: Resultado de la reproducción de la verificación de KOA con TACO. Estos valores no incluyen los métodos que no pudieron ser analizados debido a que dieron timeout.

Inicialmente se realizó la verificación con un scope de 5 instancias de cada tipo, con un bitwidth de 4[1] para los enteros y una cantidad de loop unrolling de 3. La mayoría de los métodos terminaron rápido, pero algunos otros, sobre todo los que realizan llamadas a métodos dentro de su implementación, superaron el umbral de 30' establecido como timeout. Para dichos métodos redujimos el scope y/o el bitwidth hasta que el análisis pueda ser completado dentro del tiempo establecido.

De un total del 174 métodos analizados hay 3 que no pudieron cumplir un análisis exitoso dentro de dicho lapso de tiempo, incluso reduciendo el valor de scope hasta el mínimo posible[2], por lo que no fueron incluidos en los cálculos de scope y tiempo medio. Dichos métodos son:

- VoteSet.validateRedundantInfo
- VoteSet.validateKiesKringNumber
- VoteSet.addVote(String)

[1]bitwidth de 4 representa a los enteros en el rango -8 a 7.

[2]El valor mínimo posible es el entero mas pequeño que permita encontrar una simulación para el método.

Para éstos tres métodos se realizó una verificación sin límites de tiempo, pero el SAT-Solver abortó la ejecución arrojando un excepción producida por una explosión en el tamaño de la VC generada.

Clases	Métodos	Limites de verificación	Rep. por JMLForge
AuditLog	addKiesKring	2 / 3 / 1	
AuditLog	**clear**	**2 / 3 / 1**	**No**
Candidate	Constructor	2 / 3 / 1	
CandidateList	Constructor	2 / 3 / 1	
CandidateList	addDistrict	2 / 3 / 1	
CandidateList	**addCandidate**	**2 / 3 / 1**	**No**
CandidateList	**addCandidateCode**	**2 / 3 / 1**	**No**
CandidateList	addKiesKring	2 / 3 / 1	
CandidateList	addKiesLijst	3 / 3 / 1	
KiesKring	Constructor	2 / 3 / 1	
KiesKring	addDistrict	2 / 3 / 1	
KiesKring	addKiesLijst	3 / 3 / 1	
KiesKring	clear	3 / 3 / 3	
KiesKring	make	2 / 3 / 1	
KiesLijst	addCandidate	2 / 3 / 1	
KiesLijst	clear	2 / 3 / 3	
VoteSet	addVote(int)	2 / 3 / 1	

Cuadro 6.7: Violaciones de la especificación y límites (scope/tamaño de enteros/ unrollings) necesarios para la detección del error con TACO

El cuadro 6.7 muestra específicamente los métodos que tienen violaciones de sus contratos, como así también el scope mínimo necesario para que dichos métodos encuentren el contraejemplo.

Los métodos resaltados en letra **negrita** reportaron violación en el análisis con TACO, pero no así en el análisis con *JMLForge*. Dichas violaciones fueron analizadas para descartar que sean falsos positivos (ver sección 6.2.1).

En el cuadro 6.8 se muestra una comparación entre la cantidad de violaciones encontradas por *JMLForge* contra las encontradas por TACO. Por

Clases	Violaciones (JMLForge)	Violaciones (TACO)
AuditLog	2	2
CandidateListMetadata	1	0
Candidate	1	1
KiesKring	5	5
District	0	0
KiesLijst	3	2
CanidateList	4	6
VoteSet	4	1
Suma	20	17

Cuadro 6.8: Comparación de la cantidad de violaciones encontradas por TACO contra las encontradas por JMLForge

otro lado, en el cuadro 6.9 podemos ver en detalle cuales métodos reportaron violación en *JMLForge*, cuales en TACO y si dicha violación es real o no; es decir, si no es falso positivo.

Método	Violación (JMLForge)	Violación (TACO)	Real?
AuditLog.clear	No	Si	Si
AuditLog.getCurrentTimestamp	Si	No	No
CandidateListMetadata.Constructor	Si	No	Si
KiesLijst.make	Si	No	Si
CandidateList.addCandidate	No	Si	Si
CandidateList.addCandidateCode	No	Si	Si
VoteSet.validateRedundantInfo	Si	Timeout	Si
VoteSet.validateKiesKringNumber	Si	Timeout	Si
VoteSet.addVote(String)	Si	Timeout	Si

Cuadro 6.9: Comparación entre las violaciones encontradas por TACO contra las encontradas por JMLForge

Dentro de las diferencia en las violaciones encontradas en ambos análisis podemos mencionar las siguientes:

- En la clase *AuditLog* en el método *clear* encontramos una violación que no fue encontrada por *JMLForge* (ver sección 6.2.1) y en el método *getCurrentTimeStamp JMLForge* reporta una violación que no fue encontrada por TACO (ver sección 6.2.1).

- En la clase *CandidateListMetadata JMLForge* reporta una violación en el constructor de la clase que no fue encontrada por TACO (ver sección 6.2.1)

- En la clase *KiesLijst* la diferencia se produce por el método *make* el cual arroja una excepción en tiempo de ejecución no soportado por TACO (ver sección 6.2.1).

- En la clase *CandidateList* encontramos 2 violaciones, una en el método *addCandidate* y la otra en el método *addCandidateCode*, ambas violaciones no fueron reportada por *JMLForge* (ver sección 6.2.1 y 6.2.1).

- En la clase *VoteSet* la diferencia radica en que 3 de los métodos reportados con violaciones por la herramienta *JMLForge* no pudieron ser analizados por TACO debido a que dichos métodos superaron el umbral del tiempo máximo de verificación.

En el gráfico de la figura 6.1 se muestra una comparación entre los límites de scope empleados por *JMLForge* contra los usados por TACO.

Podemos ver que en las últimas 2 clases (*CandidateList* y *VoteSet*) hay una diferencia más marcada. Luego de analizar cada una de estas clases nos dimos cuenta que la implementación de la mayoría de sus métodos realiza varias invocaciones a otros métodos, lo cual, teniendo en cuenta que TACO realiza un análisis de whole program, incrementa en gran medida la cantidad de código a ser analizado.

Esta diferencia puede verse claramente en la comparación de los tiempos medios de verificación que se muestra en la figura 6.2.

6.2.1. Diferencias de resultados obtenidos entre las verificaciones de KOA

En esta sección analizaremos las diferencias de los resultados obtenidos entre las verificaciones realizados a KOA con la herramienta *JMLForge* y la herramienta TACO.

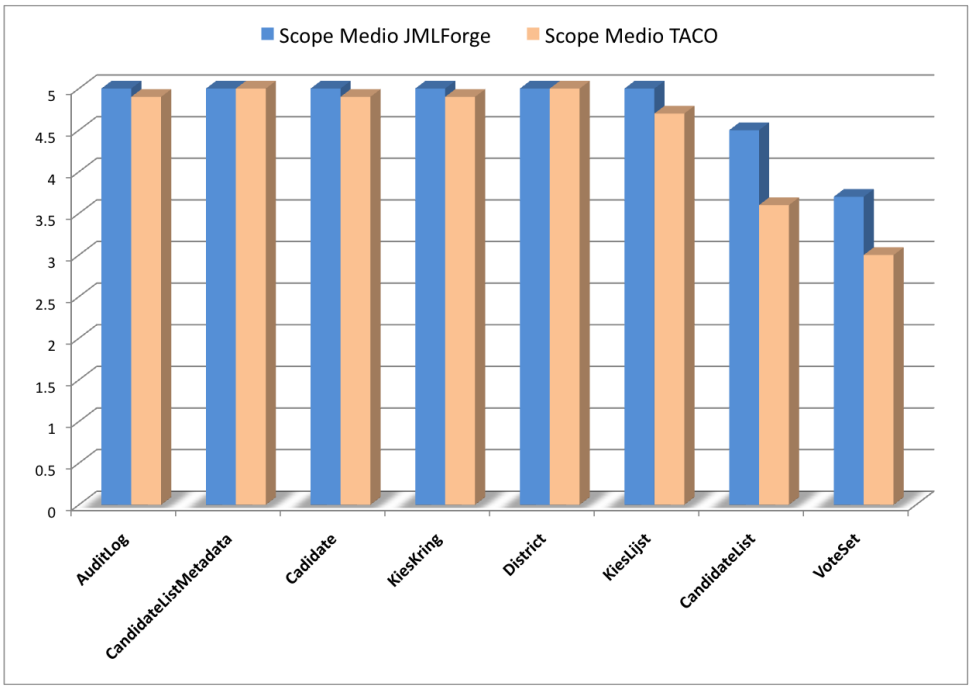

Figura 6.1: Comparación del scope medio usado por JMLForge contra el usado por TACO

Método AuditLog.clear

Este método presenta un problema de especificación. Un fragmento de la definición de AuditLog se puede ver en el código 6.1. El field *importVotesSuccess* es de tipo *boolean* (línea 3). Existe un *constraint* (línea 4) que especifica que no es posible que un método modifique el valor de dicho field una vez que el mismo tome el valor verdadero. Por otra parte, el método *clear* (línea 13) se asegura de setear el valor de dicho field en falso (líneas 10 y 15). Si a una instancia de *AuditLog*, cuyo field *importVotesSuccess* sea verdadero, se le aplica el método *clear*, se obtendrá una violación al *constraint* mencionado.

Método AuditLog.getCurrentTimestamp

El análisis realizado con *JMLForge* reporta que este método tiene un contraejemplo, sin embargo, el análisis realizado con TACO indica que no posee contraejemplo.

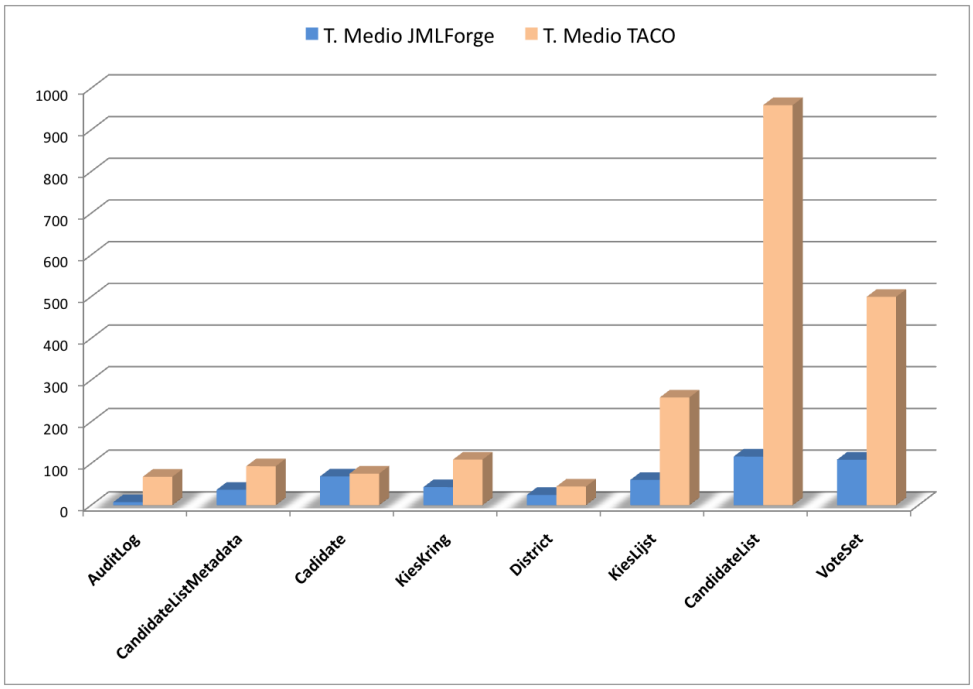

Figura 6.2: Comparación del tiempo medio arrojado por JMLForge contra el arrojado por TACO

```
1  public class AuditLog {
2    ...
3    /*@ spec_public @*/ private static boolean importVotesSuccess = false;
4    ...
5    //@ constraint \old(importVotesSuccess) ==> importVotesSuccess;
6    ...
7
8    /*@ normal_behavior
9      ...
10     @ ensures !importVotesSuccess;
11     ...
12   */
13   public static void clear() {
14     ...
15     importVotesSuccess = false;
16   }
17 }
```

Código 6.1: Fragmento de AuditLog

En el código 6.2 se puede apreciar que dicho método no posee definidas explícitamente precondición y poscondición. De hecho la violación de dicho

contrato sólo puede provenir, a nuestro entender, de que el método lance una excepción. Las excepciones *NullPointerException*, *ClassCastException* y *OutOfBoundException* no pueden ser lanzadas en dicho código.

Analizando detenidamente el código de *JMLForge* encontramos un problema en la especificación del constructor de la clase *Date*. En las líneas 12 a la 16 del código 6.3 encontramos que la especificación de la poscondición no predica sobre *throw*, luego una invocación a este constructor podría setear la variable *throw* en algún valor distinto de *null*. Este bug en la especificación de *Date* puede ser corregido modificando las líneas 12 a la 16 del código 6.3 por el código 6.4, que agrega la especificación `throw == null` (línea 4).

Al analizar el método con la versión corregida de *JMLForge* el análisis no encontró contraejemplo, confirmando así nuestra suposición original.

```
1  public class AuditLog {
2    ...
3    // Get and set methods
4    /** Get the current time.
5    */
6    //@ assignable \nothing;
7    public /*@ non_null @*/ static String getCurrentTimestamp() {
8      Date rightNow = new Date(System.currentTimeMillis());
9      return rightNow.toString();
10   }
11 }
```

Código 6.2: Fragmento de AuditLog

Método CandidateListMetadata.Constructor

JMLForge considera que este constructor viola su especificación, mientras que para TACO no existe dicho problema. Esta diferencia es producida por la diferente semántica que ambas herramientas le asignan a la llamada a métodos en la especificación.

En el código 6.5 se puede ver el constructor de la clase *CandidateListMetadata* (líneas 5 a 23) que asigna a la variable *my_kiesKringCount* el valor del parámetro *a_kieskring_count* (línea 21). En la especificación vemos que el constructor asegura igualar el resultado del método *kiesKringCount()* con el parámetro *a_kieskring_count* (línea 8).

```
1  public class DateConverter extends SpecConverter {
2    ...
3    @Override
4    public JForgeSpec convertSpec(JForgeProc jproc) {
5      SootMethod method = jproc.method();
6      SootClass declClass = method.getDeclaringClass();
7      JForgeProg jprog = jproc.jprog();
8      Set<JForgeGlobal> globals = new HashSet<JForgeGlobal>();
9      if (declClass.equals(systemClass)
10       && method.getName().equals("currentTimeMillis")
11       && method.getParameterCount() == 0) {
12         globals.add(jprog.abstractGlobal(clock));
13         return JForgeSpec.simple(jproc,
14                  globals,
15                  jproc.returnVar().eq(jprog.abstractGlobal(clock).variable()),
16                  false);
17     }
18     else if (declClass.equals(dateClass) && JForgeScene.isConstructor(method)) {
19       return JForgeSpec.identity(jproc);
20     }
21     else if ( declClass.equals(dateClass)
22               && method.getName().equals("toString")
23               && method.getParameterCount() == 0) {
24       return JForgeSpec.identity(jproc);
25     }
26
27   }
```

Código 6.3: Fragmento de DateConverter

```
1      return JForgeSpec.simple(jproc,
2                               globals,
3                               jproc.returnVar().eq(jprog.abstractGlobal(clock)
4                                    .variable().and(JForgeSpec.throwNull(jproc))),
5                               false);
```

Código 6.4: Bugfix de DateConverter

Vemos en la definición del método *kiesKringCount()* (líneas 27 a 34) que la especificación asegura únicamente que el resultado sea mayo a cero (línea 30).

En el fragmento de código 6.6 podemos ver cómo queda expresada la llamada al método *kiesKringCount()* en la especificación del constructor de *CandidateListMetadata*. En la línea 16 se muestra la cuantificación existencial sobre el parámetro que representa el valor de retorno.

El enfoque tomado por JMLForge para resolver la llamada a métodos consiste en remplazar cada llamada por una constante. Dicha constante será res-

```
1  class CandidateListMetadata {
2  ...
3  /**
4   ...
5   * <pre><jml>
6   * normal_behavior
7   ...
8   *   ensures kiesKringCount() == a_kieskring_count;
9   ...
10  * </jml></pre>
11  */
12 CandidateListMetadata(final /*@ non_null @*/String a_request_reference,
13                       final /*@ non_null @*/String a_response_reference,
14                       final /*@ non_null @*/String a_creation_time,
15                       final byte a_kieskring_count,
16                       final int a_district_count,
17                       final byte a_kieslijst_count,
18                       final int a_positie_count,
19                       final int a_code_count) {
20   ...
21   my_kiesKringCount = a_kieskring_count;
22   ...
23  }
24
25 /**
26  ...
27  * <pre><jml>
28  * normal_behavior
29  *   ensures \result >= 0;
30  * </jml></pre>
31  */
32  final /*@ pure @*/ byte kiesKringCount() {
33     return my_kiesKringCount_CandidateListMetadataInstance;
34  }
35 }
```

Código 6.5: Constructor CandidateListMetadata

tringida por la especificación del método, en donde los parámetros declarados son reemplazados por las expresiones utilizadas en el llamado.

En el código 6.7 se puede ver la representación de JMLForge de la especificación del método *kiesKringCount()*. Para utilizar dicha especificación se crea el símbolo cuantificado universalmente *kiesKringCount_return0* y se agrega una restricción como la que puede verse en el código 6.8. La diferencia entre el enfoque de TACO y el de JMLForge es que JMLForge considera satisfecha la condición si toda instancia de kiesKringCount_return0 cumple la restricción, mientras que en TACO se considera satisfecha la condición si al menos una instancia la satisface.

Este es el motivo por el cual en circunstancias como la del constructor

analizado, cuando se realiza un llamado a una especificación no deterministica[3] TACO y JMLForge pueden no coincidir en sus conclusiones.

```
program CandidateListMetadata::Constructor[
  var thiz:CandidateListMetadata,
  var throw:java_lang_Throwable+null,
  ...
  var a_kieskring_count:Int,
  ...
  ]
Specification
{
  SpecCase #0 {
   ...
   ensures {
      (throw'=null)
      and
      (
        some cs_return_kiesKringCount_377:Int | {
          callSpec kiesKringCount[
                       thiz,throw',
                       cs_return_kiesKringCount_377]
          and
          equ[cs_return_kiesKringCount_377,
              a_kieskring_count']
        }
      )
   }
   ...
  }
}
Implementation
...
```

Código 6.6: program CandidateListMetadata::Constructor

```
(((one get_return)
  && (one get_throw))
  && ((get_throw in Null)
  && (get_return >= 0)))
```

Código 6.7: especificación de kiesKringCount en JMLForge

[3]Nos referimos como especificaciones no deterministicas a aquellas que restringen el valor de la variable de retorno a un rango posible de valores

```
1  (((one kiesKringCount_return0)
2    && (one kiesKringCount_throw1))
3    && ((kiesKringCount_throw1 in Null)
4    && (kiesKringCount_return0 >= 0))))
```

Código 6.8: restricción kiesKringCount_return0

Método KiesLijst.make

El contrato de este método indica que no debería lanzar excepciones, sin embargo *JMLForge* encuentra un contraejemplo en el cual se lanza la excepción runtime *ArrayIndexOutOfBoundsException*. TACO no tiene soporte para analizar si los accesos a arreglos están dentro de valores válidos, por lo que no encuentra contraejemplo en este método.

Método CandidateList.addCandidate

El análisis con TACO de este método arrojó que existía un contraejemplo, mientras que el análisis realizado con *JMLForge* no lo hizo. En el código 6.9 se ve parte del contrato y la implementación de *CandidateList.AddCandidate*, mientras que en el código 6.10 se ve parte del contrato del constructor de *Candidate*.

El método *CandidateList.AddCandidate* depende fuertemente de la llamada al constructor de *Candidate* (línea 30) para cumplir su poscondición, esto se desprende de que la poscondición de *CandidateList.AddCandidate* (líneas 6 a la 14) se encuentra contenida casi en su totalidad por la poscondición del constructor *Candidate* (líneas 5 a la 12). Tanto TACO y *JMLForge* detectan que el constructor de *Candidate* no respeta su contrato. TACO utiliza la técnica de whole program para resolver las llamadas a métodos, utilizando así la implementación defectuosa del constructor de *Candidate*. Ese es el motivo que explica por que se encuentra el mencionado contraejemplo. *JMLForge* por otro lado utiliza el análisis modular, por lo que no detecta contraejemplo en este método.

Método CandidateList.addCandidateCode

En el código 6.11 se muestra un fragmento de la clase *CandidateList*. El contraejemplo proviene de una situación particular en donde dos field de una

```
1    /*
2    * <pre><jml>
3    * normal_behavior
4    ...
5    *   modifies \nothing;
6    *   ensures (* all fields are properly initialized *);
7    *   ensures \result.lastname().equals(a_lastname);
8    *   ensures \result.firstname().equals(a_firstname);
9    *   ensures \result.initials().equals(some_initials);
10   *   ensures \result.gender() == a_gender.charAt(0);
11   *   ensures \result.position_number() == Byte.parseByte(a_position_number);
12   *   ensures \result.cityOfResidence().equals(a_city_of_residence);
13   *   ensures \result.kiesKring().equals(a_kieskring);
14   *   ensures \result.kiesLijst().equals(a_kieslijst);
15   * also
16   * normal_behavior
17   ...
18   * </jml></pre>
19   */
20  final /*@ non_null @*/ Candidate
21    addCandidate(final /*@ non_null @*/ KiesKring a_kieskring,
22                 final /*@ non_null @*/ KiesLijst a_kieslijst,
23                 final /*@ non_null @*/ String a_position_number,
24                 final /*@ non_null @*/ String a_lastname,
25                 final /*@ non_null @*/ String some_initials,
26                 final /*@ non_null @*/ String a_firstname,
27                 final /*@ non_null @*/ String a_gender,
28                 final /*@ non_null @*/ String a_city_of_residence) {
29    ...
30    final Candidate c = new Candidate(a_lastname, a_firstname, some_initials,
31                                      a_gender.charAt(0),
32                                      Byte.parseByte(a_position_number),
33                                      a_city_of_residence,
34                                      a_kieskring, a_kieslijst);
35    return c;
36  }
```

Código 6.9: Fragmento CandidateList.AddCandidate

instancia de *CandidateList* tienen asignados la misma instancia de *Map*. Los fields mencionados son *my_kieskringen* (línea 5) y *my_candidate_codes* (línea 8).

Supongamos que contamos con una instancia de CandidateList llamada C_0 y que también contamos con una instancia de TreeMap llamada M_0. Consideremos que M_0 tiene una cantidad de elementos igual al valor contenido en la constante `MAX_KIESKRINGEN_PER_CANDIDATE_LIST` (línea 3).

Supongamos también que tanto la variable C_0`.my_kieskringen` como C_0`.my_candidate_codes` toman el valor M_0.

Con una invocación a C_0`.addCandidateCode(...)` (línea 15) que agre-

```
1   /**
2    * <pre><jml>
3    * normal_behavior
4    ...
5    *   ensures lastname().equals(a_lastname);
6    *   ensures firstname().equals(a_firstname);
7    *   ensures initials().equals(some_initials);
8    *   ensures gender() == a_gender;
9    *   ensures position_number() == a_position_number;
10   *   ensures cityOfResidence().equals(a_city_of_residence);
11   *   ensures kiesKring().equals(a_kieskring);
12   *   ensures kiesLijst().equals(a_kieslijst);
13   *   ...
14   * </jml></pre>
15   */
16  Candidate(final /*@ non_null @*/ String a_lastname,
17            final /*@ non_null @*/ String a_firstname,
18            final /*@ non_null @*/ String some_initials,
19            final char a_gender,
20            final byte a_position_number,
21            final /*@ non_null @*/ String a_city_of_residence,
22            final /*@ non_null @*/ KiesKring a_kieskring,
23            final /*@ non_null @*/ KiesLijst a_kieslijst) {
24    ...
25  }
```

Código 6.10: Fragmento del constructor de Candidate

gue un nuevo elemento a M_0 tendríamos que `M0.size()` supera a la constante `MAX_KIESKRINGEN_PER_CANDIDATE_LIST`. En esas condiciones se rompería el invariante de la línea 6 (del código 6.11).

Cabe aclarar que construir la instancia C_0 no resulta posible utilizando el API que provee dicha clase, pero no esta restringido por la especificación de dicha clase.

La siguiente restricción corrige dicho problema de subespecificación:

```
//@ invariant my_kieskringen != my_candidate_codes;
```

6.3. Relevance Analysis

En esta sección mostraremos el resultado de la utilización del mecanismo de Relevance Analysis (ver sección 5.5) para detectar las clases que son relevantes dentro de una verificación. Para ello se tomaron 15 métodos con-

```
1  class CandidateList implements java.io.Serializable {
2    ...
3    static byte MAX_KIESKRINGEN_PER_CANDIDATE_LIST = 100;
4    ...
5    private /*@ non_null spec_public @*/ SortedMap my_kieskringen;
6    //@ invariant my_kieskringen.size() <= MAX_KIESKRINGEN_PER_CANDIDATE_LIST;
7    ...
8    private /*@ non_null spec_public @*/ Map my_candidate_codes;
9    ...
10    * <pre><jml>
11    * normal_behavior
12    *    ...
13    * </jml></pre>
14    */
15   final boolean addCandidateCode(final /*@non_null@*/Candidate a_candidate,
16                                  final /*@non_null@*/String a_candidate_code){
17     return my_candidate_codes_CandidateListInstance.put(
18          Integer.valueOf(Integer.parseInt(a_candidate_code)),
19                          a_candidate) == null;
20   }
21 }
```

Código 6.11: Fragmento de CandidateList

tenidos dentro de las clases del caso de estudio.

Como puede verse en el gráfico de la figura 6.3 la utilización de este mecanismo redujo considerablemente los tiempos de verificación en todos los métodos en cuestión, tanto en los casos que se usó la semántica de scope tipo *Alloy*, como en los que se utilizó la semántica de scope tipo *JMLForge*.

Cuando introdujimos el Relevance Analysis (sección 5.5) advertimos sobre la posibilidad de que se "pierdan" contraejemplos.

Si bien en casi la totalidad de los métodos analizados el Relevance Analysis no afectó los resultados, hubo un método (*setDecryptNrOfVotes* de la clase *AuditLog*) que dejó de notificar la presencia de un error.

Ésto se debe a que el Relevance Analysis no considera relevante a la clase *KiesLijst* en el análisis del método *AuditLog.setDecryptNrOfVotes*.

En el código 6.12 se muestran fragmentos de *AuditLog* y *KiesLijst*.

El método *AuditLog.setDecryptNrOfVotes* (línea 10) establece el valor del field *decryptNrOfVotes* (línea 4). El problema surge al modificar el valor de dicho field y setearlo a un valor menor que el que posee el field *my_vote_count* (línea 16) de *KiesLijst* (línea 15). De esta manera, se viola el invariante de

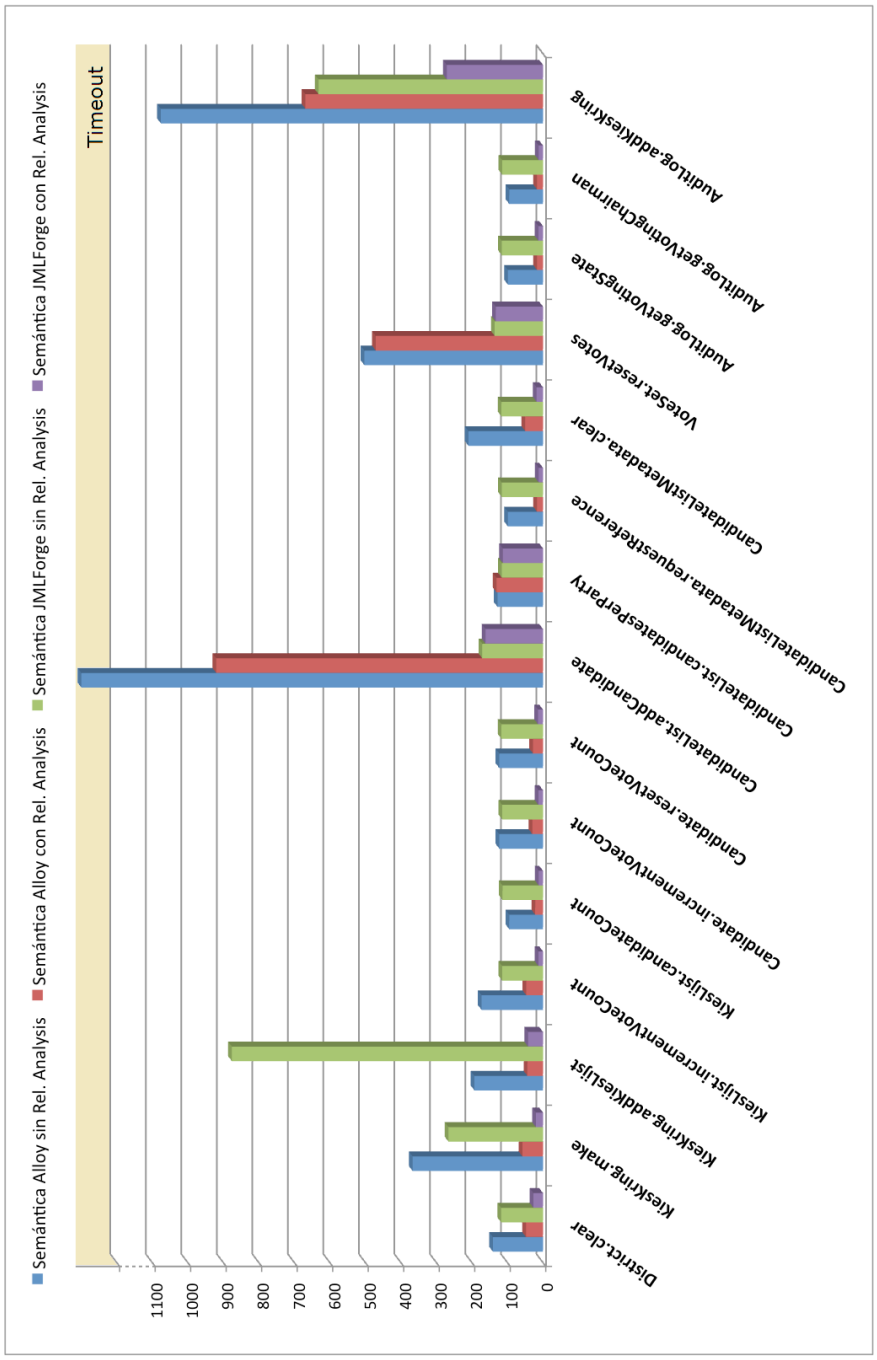

Figura 6.3: Comparación entre las distintos semánticas de scope y la incorporación o no del relevance analysis a la verificación. La zona de timeout indica que la verificación para esos métodos no terminó.

KiesLijst (línea 18).

```
1  public class AuditLog
2  {
3    ...
4    /*@ spec_public @*/ private static int decryptNrOfVotes = 0;
5    ...
6    /*@ normal_behavior
7    @ requires i >= 0;
8    ...
9    @*/
10   public static void setDecryptNrOfVotes(int i) {
11     decryptNrOfVotes = i;
12   }
13 }
14
15 final class KiesLijst ... {
16   private /*@ spec_public @*/ int my_vote_count;
17   ...
18   //@ invariant my_vote_count <= AuditLog.getDecryptNrOfVotes();
19   ...
20 }
```

Código 6.12: Fragmentos de AuditLog y de KiesLijst

6.4. Diferencias entre las distintas semánticas de scope

Como comentamos anteriormente, a la hora de realizar una verificación con *TACO* se puede optar por dos semánticas distintas en la elección del scope. Una es la semántica tipo *Alloy* y la otra es la semántica tipo *JMLForge* (ver sección 5.4).

Analizando con más detalle los efectos producidos por cada una de estas interpretaciones, surge que la semántica tipo *JMLForge* es, en cierta medida, más restrictiva que la otra, ya que *Alloy* no podrá crear más de n instancias (donde n es el scope indicado) por cada tipo, pero aun así la cantidad total de instancias (contando todos los tipos) es igual a n por la cantidad de tipos no abstractos.

Por otro lado, si restringimos el scope en la semántica tipo *Alloy* a una

cantidad de m instancias, lo que estamos diciendo es que en total no se pueden crear más de m instancias totales. De esta manera, *Alloy* podría crear $m-2$ instancias de un mismo tipo y 1 instancia de otros 2 tipos. Claramente este comportamiento no es posible lograrlo con la semántica de scope tipo *JMLForge*.

Ésto no quiere decir que una sea mejor que la otra, ya que en ocasiones nos encontramos con métodos que no pudieron ser analizados usando la semántica tipo *Alloy*, ya que requerían pocas instancias de cada tipo, pero necesitaban muchos tipos instanciados, y para lograr esto con el scope tipo *Alloy* necesitamos asignar un valor muy grande de scope.

En contrapartida, puede ocurrir que un método necesite n (con $n \geq 6$) instancias de un mismo tipo, y 2 de otro tipo. Para este caso, en la semántica tipo *JMLForge* deberíamos asignar un número mayor o igual a 6, lo que puede producir que la verificación tarde mucho tiempo, por la cantidad de instancias que se crearían. En cambio, con el scope tipo *Alloy* sólo tendríamos que setear un scope de 8, lo que nos permitiría verificar dicho método en un tiempo considerablemente menor.

Capítulo 7

Conclusiones

En la realización de esta tesis hemos definido e implementado el lenguaje de especificación JDynAlloy. Este lenguaje funciona ahora como lenguaje intermedio de representación de la herramienta de verificación TACO. Hemos definido e implementado las traducciones de JML a JDynAlloy y de JDynAlloy a DynAlloy. Hemos implementado el Relevance Analysis y la semántica de scope de JMLForge. Con todo esto hemos logrado mejorar el prototipo preexistente de TACO permitiendo por un lado, que acepte una porción más amplia de JML, y por el otro, mejorando la arquitectura interna de la herramienta.

Se realizó la verificación del caso de estudio KOA usando las herramientas TACO y JMLForge.

Los resultados indican que realizar la verificación usando análisis modular es más veloz que la realizada usando whole program. Esto se aprecia en la pérdida de performance que tiene TACO contra JMLForge en los métodos que poseen sentencias que realizan llamadas a otros métodos. Sin embargo TACO encontró un error gracias a la utilización del análisis whole program (*CandidateList.addCandidate*) que JMLForge pasó por alto.

Consideramos que el análisis modular es más escalable en el tamaño de los programas verificados, habilitando el análisis de sistemas más complejos. Mientras que whole program permite un análisis más preciso, pero más costoso en términos computacionales.

El uso de la semántica estilo JMLForge nos resultó, desde el punto de vis-

ta del desarrollador, más natural que la semántica tipo Alloy, ya que es más intuitivo plantearle a la herramienta que "necesito verificar la especificación con todas las listas enlazadas de hasta tres nodos", que su contrapartida "necesito verificar la especificación con todas las listas posibles tal que no haya más de cuatro instancias de Object".

Encontramos en el Relevance Analysis una técnica fundamental para lograr analizar en un tiempo aceptable el caso de estudio. Las pruebas son concluyentes acerca de las ventajas de usar éste mecanismo a la hora de realizar una verificación en un sistema real, que a diferencia de las pruebas de laboratorio, posee una gran cantidad de clases.

El caso de estudio había sido analizado usando una vasta cantidad de tests de unidad, con la herramienta ESC/Java2 y con la herramienta JML-Forge. Finalmente, el análisis realizado por TACO arrojó violaciones que no habían sido reportadas en ninguno de los análisis realizado anteriormente.

La verificación del software es una disciplina en constante evolución. Basados en los resultados obtenidos consideramos que TACO tiene mucho potencial de contribuir en el desarrollo de esta disciplina.

Capítulo 8

Trabajos futuros

Incorporar soporte completo JAVA y JML: TACO no implementa completamente dichos lenguajes. Esto limita la cantidad de casos de estudio del mundo real que pueden ser analizados por la herramienta. Adicionalmente, JDynAlloy podría sacar provecho de los *generics* de JAVA lo que ayudaría a optimizar el análisis reduciendo las posibles combinaciones de elementos que puede contener una instancia de una colección de JAVA.

Visualización de Contraejemplos JDynAlloy: Pablo Bendersky introduce en [3] una herramienta para la visualización de contraejemplos DynAlloy. La creación de una herramienta similar que permita visualizar los contraejemplos de JDynAlloy simplificaría la tareas de interpretar los contraejemplos generados por JDynAlloy.

Completar el parser JDynAlloy: El parser JDynAlloy no puede analizar actualmente toda la gramática del lenguaje.

Incorporar la técnica de romper la simetría del Heap: En [15] se presenta una técnica para romper la simetría del *heap*. Incorporar dicha técnica a la versión actual de TACO podría reducir el espacio de búsqueda que debe explorar el SAT-Solver, optimizando significativamente el rendimiento de la herramienta.

Incorporar análisis modular: JDynAlloy podría sacar provecho de utilizar la técnica de análisis modular, tal como lo hace JMLForge. Poseer ambas

técnicas en la misma herramienta permitirá no sólo realizar comparativas más detalladas de las mismas, sino también elegir la aplicación de una u otra por cada llamada a un programa.

Soporte para verificar uso correcto de arreglo: JMLForge incorpora una verificación que permite detectar referencias a posiciones inválidas en el uso de arreglos. Incorporar dicha verificación a JDynAlloy habilitaría la detección de este tipo de errores.

Mejorar el Relevance Analysis con información provista por el usuario: En ciertos casos el Relevance Analysis puede no considerar clases que son de especial interés en una verificación concreta. Tal podría ser el caso de una verificación sobre cierta implementación de una jerarquía concreta, donde el usuario considere que toda la jerarquía es relevante, aunque el Relevance Analysis no la incluya. Un mecanismo que permita al usuario informarle al Relevance Analysis cuáles clases deben estar incluidas necesariamente dentro del conjunto de clases relevantes permitiría manejar este tipo de situaciones.

Verificar sistemas multithreaded: La verificación de sistemas multithreaded es compleja debido a las posibles interferencias entre los threads concurrentes. Debido a que los entornos multithreaded son cada vez más comunes, incorporar a TACO la capacidad de verificar estos sistemas incrementaría su ámbito de aplicación.

Apéndice A

Gramática de JDynAlloy

A continuación mostraremos la gramática del lenguage JDynAlloy.

```
jDynAlloyModules      ::= jDynAlloyModule+

jDynAlloyModule       ::= 'module' id 'abstract'? (signature)?
                          (
                             jObjectInvariant
                             | jClassInvariant
                             | jObjectConstraint
                             | jRepresents
                             | jProgramDeclaration
                          )?

signature             ::= 'sig' id ( ('extends' |'in') id '' '' )? '' form? ''
jField                ::= 'field' id:type ';'

jClassInvariant       ::= 'class_invariant' form ';'
jObjectInvariant      ::= 'object_invariant' form ';'
jObjectConstraint     ::= 'object_invariant' form ';'
jRepresents           ::= 'represents' expr 'such_that' form ';'
jProgramDeclaration   ::= 'virtual'? 'program' id::id '[' jVariableDeclaration ( ',' jVariableDeclaration)*']'
                          'specification {'
                              jSpecCase*
                          '}'
                          'implementation {'
                              jBlock
                          '}'
```

```
jVariableDeclaration          ::= 'var' id:type
jSpecCase                     ::= (jRequires | jModifies | jEnsures)*
jRequires                     ::= 'requires {' form '}'
jModifies                     ::= 'modifies {' expr '}'
jEnsures                      ::= 'ensures {' form '}'

jBlock                        ::= '{' jStatement+ '}'

jStatement                    ::=
                              jAssert
                              | jAssume
                              | jVariableDeclarationStatement
                              | jSkip
                              | jIfThenElse
                              | jCreateObject
                              | jAssignment
                              | jProgramCall
                              | jWhile
                              | jBlock
                              | jHavoc

jAssert                       ::= 'assert' form ';'
jAssume                       ::= 'assume' form ';'
jVariableDeclarationStatement ::= jVariableDeclaration ';'
jSkip                         ::= ';'
jIfThenElse                   ::= 'if' form jBlock 'else' jBlock ';'
jCreateObject                 ::= 'createObject' '¡' id '¿' '[' id ']' ';'
jAssignment                   ::= expr ':=' expr ';'
jProgramCall                  ::= 'call' id '[' expr (',' expr)* ']' ';'
jWhile                        ::= 'while' form ('loop_invariant' form)? jBlock ';'
```

```
jHavoc    ::= 'havoc' expr ';'                                                    havoc

form      ::=
          expr 'implies' expr                                                     implicación
          | expr 'or' expr                                                        disyunción
          | expr 'and' expr                                                       conjunción
          | expr '=' expr                                                         igualdad
          | not expr                                                              negación
          | id '[' expr (',' expr)* ']'                                           predicate
          | ('some' | 'all' | 'lone' | 'no' | 'one' ) (id ':' type)+ '{' form '}' cuantificadores
          | 'callSpec' id '[' expr (',' expr)* ']'                                callSpec

expr      ::= id '[' expr (',' expr)* ']'                                         function
          | id                                                                    variable
          | number                                                                literal Int
          | '(' expr ')'
          | true — false                                                          literal booleano
          | expr '+' expr                                                         union
          | expr '+' expr                                                         intersección
          | expr '.' expr                                                         join
          | expr '-¿' expr                                                        producto
          | expr '++' expr                                                        override

id        ::= ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')* "'"?
number    ::= ('0'..'9')+;
```

Apéndice B

Verificación de KOA: Resultados completos

B.1. Clase District

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	5 / 3 / 1	44	NO	NO
clear	5 / 3 / 1	43	NO	NO
number	5 / 3 / 1	51	NO	NO
name	5 / 3 / 1	43	NO	NO
kiesKring	5 / 3 / 1	43	NO	NO

Cuadro B.1: Resultados de la verificación de la clase District usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.2. Clase KiesKring

Método	Limites de verificación	Tiempo (sec)	Violación (TACO)	Violación (JMLForge)
Constructor	5 / 3 / 1	151	SI	SI
make	5 / 3 / 1	170	SI	SI
addDistrict	5 / 3 / 1	92	SI	SI
districtCount	5 / 3 / 1	43	NO	NO
getDistrict	5 / 3 / 1	43	NO	NO
getDistricten	5 / 3 / 1	42	NO	NO
hasDistrict	5 / 3 / 1	43	NO	NO
name	5 / 3 / 1	42	NO	NO
number	5 / 3 / 1	43	NO	NO
kieslijstCount	5 / 3 / 1	42	NO	NO
addKiesLijst	5 / 3 / 1	175	SI	SI
getKiesLijst	5 / 3 / 1	44	NO	NO
getKiesLijsten	5 / 3 / 1	42	NO	NO
hasKiesLijst	5 / 3 / 1	45	NO	NO
clear	3 / 3 / 3	624	SI	SI

Cuadro B.2: Resultados de la verificación de la clase KiesKring usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.3. Clase KiesLijst

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	5 / 3 / 1	502	NO	NO
make	2 / 3 / 1	468	NO	SI
number	5 / 3 / 1	46	NO	NO
name	5 / 3 / 1	43	NO	NO
voteCount	5 / 3 / 1	44	NO	NO
resetVoteCount	5 / 3 / 1	166	NO	NO
incrementVoteCount	5 / 3 / 1	44	NO	NO
candidateCount	5 / 3 / 1	44	NO	NO
addCandidate	5 / 3 / 1	303	SI	SI
candidates	5 / 3 / 1	42	NO	NO
clear	5 / 3 / 3	1140	SI	SI

Cuadro B.3: Resultados de la verificación de la clase KiesLijst usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.4. Clase Candidate

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	4 / 3 / 1	347	SI	SI
lastname	5 / 3 / 1	43	NO	NO
firstname	5 / 3 / 1	43	NO	NO
initials	5 / 3 / 1	44	NO	NO
gender	5 / 3 / 1	74	NO	NO
position_number	5 / 3 / 1	43	NO	NO
cityOfResidence	5 / 3 / 1	42	NO	NO
kiesKring	5 / 3 / 1	43	NO	NO
kiesLijst	5 / 3 / 1	43	NO	NO
incrementVoteCount	5 / 3 / 1	43	NO	NO
voteCount	5 / 3 / 1	42	NO	NO
resetVoteCount	5 / 3 / 1	102	NO	NO

Cuadro B.4: Resultados de la verificación de la clase Candidate usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.5. Clase CandidateList

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	4 / 3 / 1	1152	SI	SI
addKiesKring	3 / 3 / 1	1229	SI	SI
addDistrict	4 / 3 / 1	1744	SI	SI
addKiesLijst	3 / 3 / 1	579	SI	SI
addCandidate	3 / 3 / 1	567	SI	NO
addCandidateCode	2 / 3 / 1	632	SI	NO
getCandidate(String)	4 / 3 / 1	993	NO	NO
getCandidate(Int)	4 / 3 / 1	995	NO	NO
validCandidate(String)	4 / 3 / 1	980	NO	NO
validCandidate(Int)	4 / 3 / 1	983	NO	NO
metadata	4 / 3 / 1	1003	NO	NO
candidatesPerParty	4 / 3 / 1	978	NO	NO
totalPartyCount	4 / 3 / 1	1031	NO	NO
blancoCount	4 / 3 / 1	1001	NO	NO
clear	2 / 3 / 1	526	NO	NO
getKiesKringen	4 / 3 / 1	992	NO	NO

Cuadro B.5: Resultados de la verificación de la clase CandidateList usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.6. Clase CandidateListMetadata

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	5 / 3 / 1	259	NO	SI
requestReference	5 / 3 / 1	41	NO	NO
responseReference	5 / 3 / 1	41	NO	NO
creationTime	5 / 3 / 1	41	NO	NO
kiesKringCount	5 / 3 / 1	42	NO	NO
districtCount	5 / 3 / 1	42	NO	NO
kiesLijstCount	5 / 3 / 1	43	NO	NO
positieCount	5 / 3 / 1	43	NO	NO
codeCount	5 / 3 / 1	43	NO	NO
clear	5 / 3 / 1	340	NO	NO

Cuadro B.6: Resultados de la verificación de la clase CandidateListMetadata usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.7. Clase VoteSet

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
Constructor	3 / 3 / 1	415	NO	NO
validateRedundantInfo	—	Timeout	—	SI
validateKiesKringNumber	—	Timeout	—	SI
validCandidate(String)	3 / 3 / 1	412	NO	NO
validCandidate(Int)	3 / 3 / 1	409	NO	NO
initializeVote	3 / 3 / 1	415	NO	NO
addVote(Int)	3 / 3 / 1	931	SI	SI
addVote(String)	—	Timeout	—	SI
finalizeVote	3 / 3 / 1	416	NO	NO
voteCount(String)	3 / 3 / 1	413	NO	NO
voteCount(Int)	3 / 3 / 1	425	NO	NO
resetVotes	3 / 3 / 1	435	NO	NO

Cuadro B.7: Resultados de la verificación de la clase VoteSet usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

B.8. Clase AuditLog

Método	Limites de verificación	Tiempo (sec)	Violación (TACO)	Violación (JMLForge)
setLogTimestamp	5 / 3 / 1	42	NO	NO
getVotingElectionTimestampEnd	5 / 3 / 1	42	NO	NO
getImportCandidatesFileName	5 / 3 / 1	41	NO	NO
getDecryptTimestampEnd	5 / 3 / 1	42	NO	NO
setVotingExportTimestamp	5 / 3 / 1	42	NO	NO
setImportCandidatesSuccess	5 / 3 / 1	44	NO	NO
getVotingState	5 / 3 / 1	41	NO	NO
getVotingChairman	5 / 3 / 1	42	NO	NO
getCountTimestampStart	5 / 3 / 1	42	NO	NO
getImportCandidatesRefNr	5 / 3 / 1	42	NO	NO
getImportPubKeySuccess	5 / 3 / 1	43	NO	NO
setCountTimestampEnd	5 / 3 / 1	42	NO	NO
getImportPubKeyError	5 / 3 / 1	41	NO	NO
setVotingNrOfRegisteredVoters	5 / 3 / 1	42	NO	NO
getSourceForAuditLog	5 / 3 / 1	41	NO	NO
setImportCandidatesError	5 / 3 / 1	43	NO	NO
setVotingBureau	5 / 3 / 1	42	NO	NO
setImportCandidatesRefNr	5 / 3 / 1	43	NO	NO
getLogTimestamp	5 / 3 / 1	42	NO	NO
setDecryptTimestampEnd	5 / 3 / 1	43	NO	NO
getCountTimestampEnd	5 / 3 / 1	42	NO	NO
getDecryptNrOfVotes	5 / 3 / 1	42	NO	NO
getDecryptErrors	5 / 3 / 1	42	NO	NO
setKeypairSuccess	5 / 3 / 1	44	NO	NO
getVotingNrOfRegisteredVoters	5 / 3 / 1	42	NO	NO
getCountSuccess	5 / 3 / 1	42	NO	NO
getImportPrivKeyFileName	5 / 3 / 1	42	NO	NO
getImportCandidatesError	5 / 3 / 1	42	NO	NO
setImportPrivKeyFileName	5 / 3 / 1	43	NO	NO
setImportVotesSuccess	5 / 3 / 1	43	NO	NO
setImportVotesNrOfVotes	5 / 3 / 1	42	NO	NO
addKiesKring	3 / 3 / 1	779	SI	SI
setDecryptNrOfVotes	5 / 3 / 1	42	NO	NO
getCountErrors	5 / 3 / 1	42	NO	NO
getImportPubKeyFileName	5 / 3 / 1	42	NO	NO

Cuadro B.8: Resultados de la verificación de la clase AuditLog usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
setVotingElectionTimestampStart	5 / 3 / 1	42	NO	NO
setImportPrivKeyError	5 / 3 / 1	43	NO	NO
setImportCandidatesNrOfLists	5 / 3 / 1	42	NO	NO
setImportPubKeySuccess	5 / 3 / 1	43	NO	NO
setCountNrOfVotes	5 / 3 / 1	43	NO	NO
setImportVotesFileName	5 / 3 / 1	42	NO	NO
getVotingBureau	5 / 3 / 1	41	NO	NO
getCountNrOfVotes	5 / 3 / 1	42	NO	NO
setImportPubKeyFileTimestamp	5 / 3 / 1	42	NO	NO
getImportPrivKeyError	5 / 3 / 1	42	NO	NO
getVotingExportTimestamp	5 / 3 / 1	42	NO	NO
setCountSuccess	5 / 3 / 1	43	NO	NO
getImportVotesFileName	5 / 3 / 1	42	NO	NO
getVotingElection	5 / 3 / 1	42	NO	NO
setVotingElectionTimestampEnd	5 / 3 / 1	42	NO	NO
getImportCandidatesNrOfBlanco	5 / 3 / 1	42	NO	NO
setImportVotesNrOfKieskringen	5 / 3 / 1	42	NO	NO
getImportCandidatesNrOfCandidates	5 / 3 / 1	43	NO	NO
setImportCandidatesFileTimestamp	5 / 3 / 1	42	NO	NO
clear	3 / 3 / 1	719	SI	NO
getImportCandidatesSuccess	5 / 3 / 1	42	NO	NO
getVotingElectionTimestampStart	5 / 3 / 1	42	NO	NO
getImportPubKeyFileTimestamp	5 / 3 / 1	42	NO	NO
hasKiesKring	4 / 3 / 1	1029	NO	NO
setCountTimestampStart	5 / 3 / 1	42	NO	NO
setImportCandidatesFileName	5 / 3 / 1	43	NO	NO
setImportVotesError	5 / 3 / 1	43	NO	NO
getVotingInterval	5 / 3 / 1	42	NO	NO
setDecryptTimestampStart	5 / 3 / 1	42	NO	NO
setImportCandidatesNrOfBlanco	5 / 3 / 1	41	NO	NO
setImportCandidatesNrOfCandidates	5 / 3 / 1	41	NO	NO
setDecryptSuccess	5 / 3 / 1	42	NO	NO
getImportVotesNrOfVotes	5 / 3 / 1	42	NO	NO
getDecryptSuccess	5 / 3 / 1	42	NO	NO
setCountErrors	5 / 3 / 1	42	NO	NO

Cuadro B.9: Resultados de la verificación de la clase AuditLog usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

Método	**Limites de verificación**	**Tiempo (sec)**	**Violación (TACO)**	**Violación (JMLForge)**
getCurrentTimestamp	5 / 1 / 1	37	NO	SI
getImportVotesSuccess	5 / 3 / 1	43	NO	NO
getDecryptTimestampStart	5 / 3 / 1	42	NO	NO
setDecryptErrors	5 / 3 / 1	43	NO	NO
setImportPrivKeySuccess	5 / 3 / 1	43	NO	NO
setVotingInterval	5 / 3 / 1	42	NO	NO
setVotingState	5 / 3 / 1	42	NO	NO
getKeypairSuccess	5 / 3 / 1	43	NO	NO
getImportPrivKeyFileTimestamp	5 / 3 / 1	42	NO	NO
setImportPubKeyFileName	5 / 3 / 1	43	NO	NO
getImportCandidatesFileTimestamp	5 / 3 / 1	41	NO	NO
getImportCandidatesNrOfLists	5 / 3 / 1	42	NO	NO
getImportVotesNrOfKieskringen	5 / 3 / 1	43	NO	NO
setVotingElection	5 / 3 / 1	43	NO	NO
getImportVotesError	5 / 3 / 1	42	NO	NO
setImportVotesFileTimestamp	5 / 3 / 1	42	NO	NO
setImportPubKeyError	5 / 3 / 1	42	NO	NO
getImportPrivKeySuccess	5 / 3 / 1	43	NO	NO
setVotingChairman	5 / 3 / 1	42	NO	NO
Constructor	5 / 3 / 1	42	NO	NO
setImportPrivKeyFileTimestamp	5 / 3 / 1	42	NO	NO
getImportVotesFileTimestamp	5 / 3 / 1	41	NO	NO

Cuadro B.10: Resultados de la verificación de la clase AuditLog usando TACO. Los limites de verificación representan (scope/tamaño de enteros/ unrollings)

Bibliografía

[1] Randy Allen and Ken Kennedy. *Optimizing Compilers For Modern Architectures*. Morgan Kuafmann, 2001.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *In Formal Methods for Components and Objects (FMCO'05)*, 2005.

[3] Pablo Gabriel Bendersky. Hacia un entorno integrado para la verificación de contratos utilizando SAT Solvers. Master's thesis, Universidad de Buenos Aires, 2010.

[4] Víctor Braberman, Diego Garbervetsky, and Alfredo Olivero. Improving the verification of timed systems using influence information. 2280:21–36, apr 2002.

[5] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, , and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[6] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and junit way. *Lecture Notes in Computer Science, Springer-Verlag*, 2002.

[7] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft, 2005.

[8] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. *Lecture Notes in Computer Science*, 5295/2008:130–145, 2008.

[9] Gregory D. Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, Massachusetts Institute of Technology, 2009.

[10] D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program checking. *Research Report 178, Compaq SRC*, 2002.

[11] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, 2002.

[13] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, 2001.

[14] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. Dynalloy: upgrading alloy with actions. *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 442–451, 2005.

[15] Juan P. Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias. Distributed sat-based computation of relational tight bounds. *Symposium on Automatic Program Verification*, 2009.

[16] Juan Pablo Galeotti and Marcelo Frias. Dynalloy as a formal method for the analysis of java programs. *Springer*, 227:249–260, 2006.

[17] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002.

[18] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] Joseph R. Kiniry and David R. Cok. Esc/java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. *CASSIS*, 2004.

[20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, , and J. Kiniry. *JML reference manual*. Department of Computer Science, Iowa State University, 2004. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`.

[21] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Technical report, Available from: http://www.jmlspecs.org, 2004.

[22] Nancy G. Leveson. The therac-25 accidents. *IEEE Computer*, 1993.

[23] Darko Marinov and Sarfraz Khurshid. Valloy - virtual functions meet a relational language. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 234–251, London, UK, 2002. Springer-Verlag.

[24] Sun Microsystem. JDC tech tips: March 14, 2000. `http://java.sun.com/developer/TechTips/2000/tt0314.html`.

[25] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 2001.

[26] Kathy Schwalbe. *Information Technology Project Management, 5 edition*. Course Technology, 2007.

[27] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. In *SOFSEM '09: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, pages 570–581, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Kuat T. Yessenov. A lightweight specification language for bounded program verification. Master's thesis, Massachusetts Institute of Technology, 2009.