Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Mejorando la usabilidad de herramientas de verificación

Tesis presentada para optar al título de Licenciado de la Universidad de Buenos Aires
en el área de Ciencias de la Computación

Marcos José Chicote

Directores de Tesis: Dr. Juan Pablo Galeotti, Dr. Diego Garbervetsky

Buenos Aires, Mayo de 2012

Mejorando la usabilidad de herramientas de verificación

Resumen

TACO es una herramienta de código abierto que permite el análisis de programas Java realizando una verificación estática del código fuente contra una especificación en JML (Java Modeling Language) o JFSL (Java Forge Specification Language). TACO utiliza la técnica de verificación acotada en la que todas las ejecuciones de un procedimiento son examinadas hasta un límite provisto por el usuario que acota el espacio de análisis. TACO traduce código Java a JDynAlloy (un lenguaje de representación intermedia) que luego es traducido a DynAlloy para su verificación.

En el presente trabajo se presentan diversas mejoras de usabilidad sobre la herramienta TACO que permiten un mayor y mejor análisis del contraejemplo detectado luego de una verificación. Asimismo se presenta el diseño y desarrollo de un plugin para Eclipse, llamado TacoPlug, que utiliza y expande TACO para proveer una mejor experiencia de usuario y una mayor posibilidad de debugging. Si TACO detecta una violación a la especificación, TacoPlug la presenta en términos de código fuente anotado y brinda al usuario la posibilidad de localizar la falla mediante diversas vistas similares a las de cualquier software de debugging.

Finalmente, se demuestra la usabilidad de la herramienta por medio de un ejemplo motivacional tomado de una porción de software industrial.

Improving usability of verification tools

Abstract

TACO is an open source verification tool that statically checks the compliance of a Java program against a specification written in JML (Java Modeling Language) or JFSL (Java Forge Specification Language). TACO uses a bounded verification technique, in which all executions of a procedure are examined up to a user-provided bound that narrows the analysis world. TACO translates Java annotated code to JDynAlloy (an intermediate language representation) which is then translated to DynAlloy for verification.

In this work different improvements over usability issues on TACO are presented that enable a bigger and better analysis of a counterexample found in a verification. Furthermore, this thesis presents the design and implementation of an Eclipse's plugin, called TacoPlug, that uses and expands TACO to provide a more user friendly experience and a profunder debugging capability. If TACO finds a specification violation, TacoPlug presents it in terms of the annotated source code and enables the user the posibility of localizing the defect through different views resembling any software debugger.

Finally, the tool's usability is shown by means of a motivational example taken from an industrial portion of software.

Agradecimientos

A la UBA, a la FCEN y al DC: por recibirme como alumno, formarme y permitir que me desempeñe como docente.

Al jurado, Flavia y Víctor, por tomarse el tiempo de leer la tesis y señalarme las cosas que se podían mejorar.

A mis directores, Juan Pablo y Diego, por darme la oportunidad de hacer la tesis en este tema y guiarme durante el proceso.

A mis amigos de la facu: Jota, Luigi, Eze, Sabi, Fede, PabloB, PabloR, el Lata, la Topadora, Amity, Juanity, PabloH, Píter, Droopy, Mati y Román. No me imagino haber hecho la carrera sin ellos y seguramente seguiría en Algo1 sin su constante apoyo y acompañamiento.

A todos los que compartieron años de Confirmación conmigo.

Al Clus: Salvi, Jorge, Angie, Luchi, Vicky, Dany, Dani y Sofi.

A mis amigos y amigas: Piter, Gabi, Santi, Javi, Oti, Juan, Tomy, la Pola, Gon, Cape, Juani, Chule, Fede, Faca, Ale, Clari, Fla, Domi, Agus, Luli, Guada.

A mis primos, tíos y abuelos. A Manu, Cande e Ine.

Muy especialmente a mis viejos, por darme vida y acompañarme en todo momento y en todo aspecto que pudieran y tuvieran permiso. Gracias por ocuparse de mi educación formal y humana. Les debo más que estas líneas. Gracias Má. Gracias Pá.

Índice

Índice de figuras

1. Introducción

La importancia del software tanto en sistemas críticos como en elementos de la vida cotidiana ha crecido en los últimos años a un ritmo acelerado y se prevee que lo siga haciendo. Este crecimiento implica un incremento en la complejidad y el tamaño del software desarrollado. Actualmente los desarrolladores no lidian con algunos cientos de líneas de código, sino que tratan con programas de millones de líneas. Paralelamente, y aunque se trate de una consecuencia indeseable, la cantidad de fallas en los programas aumenta. Las fallas en los elementos de software varian entre aquellas que pueden ser consideradas molestas y aquellas que traen trágicas consecuencias.

Las condiciones aquí planteadas demandan una mayor robustez en las técnicas utilizadas en la construcción de software en conjunto con herramientas más avanzadas que permitan a los programadores obtener mayores estándares de calidad en los elementos que producen.

Con el objetivo de reducir la cantidad de errores presentes en un software se han desarrollado diversas técnicas y mecanismos durante los últimos años que pueden dividirse en dos grandes áreas: análisis dinámico y análisis estático.

Dentro de las técnicas de análisis dinámico se incluyen todas aquellas que impliquen la ejecución del software que se desea analizar, siendo la más conocida la generación de casos de test y su ejecución ya sea automática o asistida. Para que el análisis dinámico sea efectivo, el software bajo análisis debe ser ejecutado con suficientes datos de entrada como para producir un comportamiento interesante. Técnicas como cobertura de código permiten asegurar que una porción adecuada del comporamiento del software ha sido probada. Algunos ejemplos de herramientas que implementan o facilitan técnicas de análisis estático son: Daikon [1] y Valgrind [2].

Por otro lado, existen metodologías muy utilizadas actualmente que basan su funcionamiento en técnicas de análisis dinámico. El mayor ejemplo de esta práctica actualmente es *Test Driven Development* (TDD) que incluye el diseño de tests de unidad previo al desarrollo de un componente y su posterior ejecución a fin de corroborar que el software se comporte de la manera esperada.

Asimismo, dentro de las técnicas asociadas al análisis estático se incluye todo mecanismo que tenga potencial para reducir la cantidad de errores en el software pero que no incluya la ejecución del mismo. Por ejemplo, los compiladores actuales se benefician de diversas técnicas de análisis de programas como ser chequeo de tipos o análisis data-flow para prevenir posibles errores y notificar a los programadores. Estas técnicas se encuentran bastante desarrolladas y no solo su grado de automatización es alto sino que las IDEs permiten a los programadores entender facilmente la falla.

El análisis estático también es utilizado para la verificación de programas. Dentro de la técnicas más conocidas se incluyen las deductivas basadas en el uso de demostradores automáticos. Esto incluye los verificadores de contratos y los chequeadores de tipos. Por otro lado, las técnicas de *model checking* [3] analizan exhaustivamente todo el espacio de estados del programa. En general, los espacios de estados asociados a un programa suelen ser muy grandes o infinitos transformando esta tarea en muy compleja o imposible. Por este motivo surgen alternativas para reducir el espacio de búsqueda: la abstracción [4], si el espacio de estados es infinito, que puede dar falsos positivos; y la *verificación acotada* que transforma el problema en decidible pero que puede dar falsos negativos.

Actualmente, técnicas de verificación formal son utilizadas por la mayoría o todos los fabricantes de hardware líderes [5]. Esto puede ser atribuido a la importante necesidad de disminuir la cantidad de fallas que pueden tener graves consecuencias comerciales.

Sin embargo, herramientas de *model checking* o *verificación acotada* de no han alcanzado un uso masivo en otros ambientes productivos industriales. Los motivos de la falta de masificación son diversos. Excluyendo cuestiones presupuestarias puede enumerarse los siguientes problemas:

- Reporte de falsos positivos o inconclusividad de los análisis.
- Falta de comodidad en la ejecución.
- Baja facilidad de análisis e interpretación de las fallas encontradas y los resultados reportados.

Para el desarrollo del presente trabajo asumimos que para que la técnica de verficación formal de programas pueda alcanzar un número significativo de usuarios y un uso corriente, interfaces de usuario apropiadas deben ser provistas, por lo que pondremos foco en la segunda causa por la cual este tipo de herramientas permanecen en ambientes académicos o especializados.

El propósito de este trabajo consiste en diseñar e implementar distintas técnicas y mecanismos que aumenten la usabilidad de herramientas de verifación formal y analizar su impacto en el debugging y localización de una falla. Para esto se pondrá el foco sobre herramientas que utilicen la técnica de verificación acotada utilizando la herramienta TACO presentada en [6, 7] como ejemplo de este paradigma y se desarrollará un plugin que permita el uso de esta herramienta desde Eclipse como prueba de concepto.

1.1. Motivación

A continuación se muestra un ejemplo de uso de TACO y su salida. Aunque en el capítulo 2 se presentará esta herramienta con más detalle, de momento alcanza con saber que TACO es una herramienta de verificación formal de programas. No se presenta aquí mayor información sobre TACO con el fin de simular el uso de la herramienta por un usuario inexperto al cual debería alcanzarle con saber que se verificará su programa contra la especificación brindada. Con solamente esta información alcanzará para demostrar que la usabilidad de TACO y su flexibilidad para analizar el resultado encontrado es reducida. Conociendo algunas de las características planteadas en el capítulo 2, esto resultará todavía más evidente.

Considerese la clase `AssertExample` con un método llamado `assertion_method` incluida en la figura 1. Este método está anotado, utilizando el lenguaje JML que se verá más adelante, por una precondición que establece que durante la ejecución del método no se arrojarán excepciones y una poscondición que declara que el resultado del método es `true`. Adicionalmente se establece que el único parámetro del método puede ser nulo.

```
1   public class AssertExample {
2
3     //@ signals (Exception ex) false;
4     //@ ensures \result == true ;
5     public boolean assertion_method(/*@ nullable @*/ Object o) {
6       Object f = o;
7       //@ assert f!=null;
8
9       return true;
10    }
11
12  }
```

Figura 1: Clase `AssertExample`

El código de esta clase es bastante simple. Tras un rápido análisis mental del método contenido es evidente la falla contenida en el mismo: la instrucción `assert` en la línea 7 falla al no poder asegurar que la variable `f` sea no nula (recuérdese que la anotación en el parámetro del método establece que la variable `o`, que es luego asignada a `f`, puede ser nula).

Esta misma falla es detectada mediante un análisis de TACO. En la figura 2 se puede apreciar la salida de la herramienta cuando se la ejecuta mediante la línea de comando. Se omite en esta salida logueo propio de la herramienta que detalla etapas de procesamiento internas y que no modifican el resultado final o el análisis que puede hacerse del mismo. En la línea 23 puede verse que el *Outcome* es **SAT** lo que establece que se encontró un contrajemplo para la verificación realizada.

```
1  ****** Stage Description: Execution of analysis. ******
2
3  00000000  Starting Alloy Analyzer via command line interface
4      * Input spec file          : output/output.als
5
6  00000001  Parsing and typechecking
7      * Command type             : run
8      * Command label            : ar_edu_taco_AssertExample_assertion_method_0
9
10 00000568  Translating Alloy to Kodkod
11     * Solver                   : minisat(jni)
12     * Bit width                : 4
13     * Max sequence             : 7
14     * Skolem depth             : 0
15     * Symmbreaking             : 20
16
17 00000947  Translating Kodkod to CNF
18     * Primary vars             : 36
19     * Total vars               : 228
20     * Clauses                  : 341
21
22 00001344  Solving
23     * Outcome                  : SAT
24     * Solving time             : 419
25
26 00001366  Analysis finished
27
28 ****** END: Alloy Stage ******
```

Figura 2: Salida para el método `assertion_method` de la clase `AssertExample`

Como se puede observar, la única información significativa que brinda la salida de TACO es si se encontró un error o no (en cuyo caso dirá **UNSAT**). El resto de la información aquí mostrada son características del análisis realizado. De esta forma, en caso de encontrar un error, como ser este el caso, no se brinda mayor información sobre cuál es el error, de dónde proviene, en qué línea se encuentra, etc.. Es decir, no se brinda información útil para comprender el origen de la falla.

Por otro lado, TACO brinda la posibilidad de imprimir el contraejemplo encontrado en formato XML. En las figuras 3 y 4 se muestra dicho XML para la verificación aquí considerada separado en los módulos existentes y en los átomos instanciados, respectivamente. Estos XML incluyen información que representa las variables instanciadas durante la ejecución simbólica del método analizado: `this`, parámetros y variables locales.

```
1  <sig label="seq/Int" ID="0" parentID="1" builtin="yes" />
2
3  <sig label="Int" ID="1" parentID="2" builtin="yes" />
4
5  <sig label="String" ID="3" parentID="2" builtin="yes" />
6
7  <sig label="this/null" ID="4" parentID="2" one="yes">
8     <atom label="null$0" />
9  </sig>
10
11 <sig label="this/true" ID="5" parentID="6" one="yes">
12    <atom label="true$0" />
13 </sig>
14
15 <sig label="this/false" ID="7" parentID="6" one="yes">
16    <atom label="false$0" />
17 </sig>
18
19 <sig label="this/boolean" ID="6" parentID="2" abstract="yes" />
20
21 <sig label="this/char" ID="8" parentID="2" abstract="yes" />
22
23 <sig label="this/AssertionFailureLit" ID="9" parentID="10" one="yes">
24    <atom label="AssertionFailureLit$0" />
25 </sig>
26
27 <sig label="this/java_lang_Exception" ID="11" parentID="10" abstract="yes" />
28
29 <sig label="this/java_lang_Throwable" ID="10" parentID="2" abstract="yes" />
30
31 <sig label="this/ar_edu_taco_AssertExample" ID="12" parentID="13">
32    <atom label="ar_edu_taco_AssertExample$0" />
33 </sig>
34
35 <sig label="this/java_lang_Object" ID="13" parentID="2" abstract="yes" />
36
37 <sig label="univ" ID="2" builtin="yes" />
```

Figura 3: XML con la información de diferentes módulos existentes en el contrajemplo

Como puede verse en estas figuras, el contrajemplo no resulta intuitivo ni fácil de entender para un usuario no experto. Inclusive, si es analizado por un usuario avanzado de TACO o Alloy, puede requerir un tiempo prolongado antes de lograr comprender la falla detectada y descripta. Esta tarea permitiría reconstruir las variables involucradas pero no alcanzaría para detectar la falla puesto que faltaría hacer una ejecución, simbólica o no, del método analizado con la información recuperada para visualizar cuál fue el camino de ejecución tomado. Esto resulta en un proceso muy frágil al error humano y requeriría nuevamente una inversión considerable de tiempo.

```
<skolem label="$exit_stmt_reached_1" ID="14">
    <tuple> <atom label="false$0" /> </tuple>
    <types> <type ID="6" /> </types>
</skolem>

<skolem label="$exit_stmt_reached_2" ID="15">
    <tuple> <atom label="false$0" /> </tuple>
    <types> <type ID="6" /> </types>
</skolem>

<skolem label="$var_2_f_0" ID="16">
    <tuple> <atom label="ar_edu_taco_AssertExample$0" /> </tuple>
    <types> <type ID="13" /> </types>
    <types> <type ID="4" /> </types>
</skolem>

<skolem label="$var_2_f_1" ID="17">
    <tuple> <atom label="null$0" /> </tuple>
    <types> <type ID="13" /> </types>
    <types> <type ID="4" /> </types>
</skolem>

<skolem label="$o_0" ID="18">
    <tuple> <atom label="null$0" /> </tuple>
    <types> <type ID="13" /> </types>
    <types> <type ID="4" /> </types>
</skolem>

<skolem label="$return_0" ID="19">
    <tuple> <atom label="false$0" /> </tuple>
    <types> <type ID="6" /> </types>
</skolem>

<skolem label="$return_1" ID="20">
    <tuple> <atom label="false$0" /> </tuple>
    <types> <type ID="6" /> </types>
</skolem>

<skolem label="$throw_0" ID="21">
    <tuple> <atom label="null$0" /> </tuple>
    <types> <type ID="10" /> </types>
    <types> <type ID="4" /> </types>
</skolem>

<skolem label="$throw_1" ID="22">
    <tuple> <atom label="null$0" /> </tuple>
    <types> <type ID="10" /> </types>
    <types> <type ID="4" /> </types>
</skolem>

<skolem label="$throw_2" ID="23">
    <tuple> <atom label="AssertionFailureLit$0" /> </tuple>
    <types> <type ID="10" /> </types>
    <types> <type ID="4" /> </types>
</skolem>
```

Figura 4: XML con la información de diferentes átomos instanciados en el contrajemplo

Este corto y simple ejemplo sirve para entender la motivación del presente trabajo. En primer lugar, la ejecución de herramientas desde línea de comando no suele ser lo más intuitivo o cómodo. Si bien no se niega la posibilidad de que existan herramientas para los cuales este esquema resulte útil, TACO no pareciera ser el caso. Cuando un usuario desea ejecutar un análisis, debe definir una serie de parámetros para lo cual, dependiendo de su experiencia con la herramienta, es probable que deba consultar el manual. La posibilidad de contar con una herramienta visual que permita la ejecución de TACO debería fomentar la verificación de los métodos desarrollados. Siguiendo esta línea, los programadores suelen utilizar IDEs al momento de desarrollar software y la integración con estas herramientas facilitaría la ejecución de análisis.

En segundo lugar, observando el resultado obtenido es evidente que la información brindada no dice más que *hubo error* a un usuario inexperto. Para usuarios expertos, TACO brinda información sobre el contrajemplo encontrado pero esta es muy dificil de leer y su análisis puede resultar en una tediosa tarea. Esto es inaceptable si se espera la masificación de este tipo de herramientas y su adopción en un entorno industrial. Un usuario interesado en la verificación de un programa desarrollado espera, pero sobre todo *necesita*, más información que le permita corregir el programa para poder lograr la verificación del software y que su interpretación se de en el mismo nivel de abstracción que él maneja, es decir, código fuente Java. Por lo tanto, la salida de estándar de TACO resulta insuficiente y su ampliación y clarificación derivan en otra de las motivaciones del presente trabajo.

En la sección 4 se presenta un caso de estudio donde se amplian los problemas de usar TACO mediante la línea de comando y se presentan las ventajas de lo desarrollado en el presente trabajo utilizando un ejemplo más complejo.

1.2. Trabajos Relacionados

Además de TACO, existen otras herramientas de verificación. Antes de inciar el presente trabajo, se realizó un análisis sobre las herramientas disponibles en el mercado que permiten realizar verificación de programas y su usabilidad.

Para el análisis de programas escritos en C/C++:

- **F-Soft**, presentada en [8]. No pudo ser analizada debido a que no se encuentra públicamente disponible.
- **CBMC**, presentada en [9]. Esta herramienta de verificación acotada fue desarrollada en Carnegie Mellon. Aunque detalla la aserción violada e imprime la traza con el error, no permite al usuario inspeccionarla dinámicamente ni evaluar expresiones a lo largo de la misma.

Por el lado de Java, entre las opciones disponibles se hallan:

- **Miniatur**, presentada en [10]. No pudo ser analizada debido a que no se encuentra públicamente disponible.
- **Forge**, presentada en [11]. Forge es la herramienta más parecida a TACO entre las nombradas. 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. La versión actual de Forge incluye un plugin para Eclipse conocido como JForge [12]y, como CBMC, indica qué parte de la especificación ha sido violada. También presenta un grafo con el estado de la memoria pero restringido a los estados incial y final de la traza. El plugin permite al usuario la visualización de la traza que conlleva al contraejemplo pero representada en un lenguaje intermedio.

Finalmente, en lo que se refiere a software desarrollado en C#, en [13] se genera un ejecutable que reproduce el error encontrado por la verificación de Spec#. El objetivo del programa generado

no es solo la comprensión de la falla, sino la detección de errores espúreos. Como aquí la ejecución no es simbólica sino que el programa realmente se ejecuta, moverse hacia atrás en la traza no está permitido.

Continuando con el análisis de herramientas que trabajan sobre código C#, en [14] se presenta una herramienta conocida como *Boogie Verification Debugger* (BVD de ahora en más). Esta herramienta es lo que más se acerca a lo aquí presentado, brindando la posibilidad de navegar la traza hacia adelante y hacia atrás y la evaluación de variables en distintos puntos. Se incluye un plugin para Visual Studio que permite la verificación utilizando los front-ends VCC y Dafny. Aunque BVD permite listar información que puede ser útil al programador, como ser los alias para un objeto determinado, BVD no hace hincapié sobre la visualización de ésta información, presentándola de una forma poco intuitiva.

1.3. Estructura del informe

El presente informe se encuentra estructurado de la siguiente forma.

En el capítulo 2 se incluye una introducción a TACO, su arquitectura y su modo de uso.

Continuando con el capítulo 3 se introducen las contribuciones realizadas sobre TACO que dan soporte a la construcción de un plugin para Eclipse que permite el uso de TACO desde esta IDE. Adicionalmente se enumeran las principales características de TacoPlug en conjunto con los beneficios brindados.

En el capítulo 4 se desarrolla un caso de estudio en el que se muestra y compara el análisis de un programa utilizando TACO y TacoPlug. Se incluye una demostración de los problemas de usabilidad que presentaba TACO anterior al desarrollo de este trabajo.

La arquitectura de las técnicas diseñadas en el presente trabajo y su implementación como parte de TACO en conjunto con el diseño de TacoPlug son presentados en el capítulo 5. En este capítulo, además, se detallará cómo es la interacción entre TACO y TacoPlug.

El capítulo 6 incluye un resumen del trabajo desarrollado y los resultados obtenidos. Se presentan aquí las conclusiones alcanzadas con respecto a los objetivos planteados.

Finalmente, el capítulo 7 propone una serie de puntos sobre los que puede trabajarse a futuro para seguir incrementando la usabilidad de herramientas de verificación y acercándolas a las manos de los desarrolladores.

2. TACO

2.1. Definiciones previas

Antes de introducir TACO propiamente dicho conviene repasar algunas definiciones previas que utilizaremos frecuentemente.

- **Precondición**: es un predicado que debe ser verdadero en el estado previo a la ejecución del método sobre el cual está 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.
- **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 generadas con *unrolling* puede ser menor a la cantidad real de iteraciones que realizaría el ciclo.
- **Sat-Solving**: un problema de decisión es una pregunta expresada en algún sistema formal cuya respuesta es sí 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 de 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.
- **Análisis Whole Program**: se conoce con este nombre a un análisis que toma el programa como un todo en vez de hacerlo módulo por módulo.
- **Weakest Precondition**: es una técnica para representar un programa como una fórmula lógica para lo cual se le asigna una semántica particular a cada instrucción. Esta técnica requiere la presencia de una poscondición y establece que *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*. Para más información acerca de esta técnica consultar [15].
- **JML** : Java Modeling Language es un lenguaje de especificación que puede ser utilizado para describir el comportamiento de módulos Java. Combina elementos de diseño basado en contratos de Eiffel [16] y la especificación basada en modelos de la familia de lenguajes Larch [17], con algunos elementos de cálculo de refinamiento [18]. Más información sobre este lenguaje puede encontrarse en [19]
- **JFSL** : JForge Specification Language es un lenguaje de especificación formal basado en JML y en la idea de que expresiones en lenguaje Alloy son muy eficientes a la hora de describir el comportamiento de un programa. JFSL fue desarrollado como parte de la herramienta Forge mencionada anteriormente.

2.2. Introducción

TACO es una herramienta desarrollada por el *Relational Formula Method Research Group* [20] 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 o JFSL. TACO utiliza la técnica de verificación acotada en la cual todas las ejecuciones de un procedimiento son examinadas tomando en cuenta dos cotas definidas por el usuario: el máximo número de instancias de una clase presentes en el heap y la cantidad de loop unrolls.

TACO utiliza las técnicas de whole program analysis y weakest precondition para transformar los programas en una fórmula lógica llamada *Condición de Verificación* (VC) y utiliza SAT-Solving para validar o refutar dicha fórmula.

2.3. Arquitectura

Para lograr su objetivo de verificar la correctitud de un programa Java contra su especificación, TACO presenta un pipeline en el que realiza diferentes transformaciones sobre el programa original, simplificándolo en cada etapa, hasta lograr construir la Condición de Verificación.

Este pipeline está explicado, a alto nivel, por la figura 5. En esta figura, los elementos circulares representan artefactos mientras que los rectangulares representan componentes.

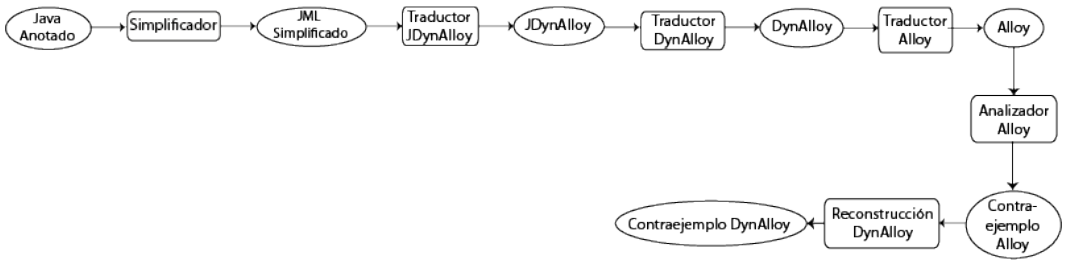

Figura 5: Etapas de transformación del código JML hasta código Alloy

La arquitectura de TACO está fuertemente determinada por el pipeline de transformación de lenguajes en el que basa su funcionamiento. A grandes rasgos, cada una de las etapas del pipeline se encuentra representada por una *stage* a nivel implementativo.

Sin embargo, es importante resaltar algunas características de diseño que escapan a este esquema. La diferencia más significativa se encuentran en la correspondencia entre *stages* y etapas del pipeline. Existen *stages* que no representan etapas del pipline, generalmente asociados a procesos posteriores a la detección de un contraejemplo. Un ejemplo de esta situación es la *stage* conocida como `JUnitStage`, componente que permite la construcción de tests de unidad basados en los contraejemplos detectados por TACO.

Otra característica importante de la arquitectura de TACO es el almacenado de información de transformación del programa original entre las distintas etapas que se ejecutan. Esta información permite tener trazabilidad entre las distintas representaciones del programa brindando la posibilidad de visualizar contraejemplos Alloy (ver más adelante) a nivel Java.

En las siguientes subsecciones se hará un breve repaso de cada una de las etapas involucradas, su objetivo e importancia. Para mayor detalle sobre la arquitectura de TACO se recomienda la lectura de [7].

Antes de detallar las traducciones dadas en cada una de las etapas resulta conveniente definir cada uno de los lenguajes intermedios y revisar sus principales características:

- **Alloy.** Alloy [21] es un lenguaje relacional de especificación formal cuya sintaxis permite expresar tipos abstractos de datos utilizando signaturas. También es posible agregar restricciones sobre dichas signaturas. Ciertas funcionalidades de Alloy se asemejan a características de los lenguajes orientados a objetos. A continuación se muestra un ejemplo del lenguaje Alloy. Supóngase que se desea especificar una libreta de direcciones para lo cual hay que

modelar tipos de datos para los nombres y las direcciones. Para esto es posible empezar indicando la existencia de dos grupos disjuntos para nombres y direcciones. Esto es especificado en Alloy como signaturas como se muestra a continuación:

sig *Address* {} sig *Name* {}

Con estos nombres y direcciones se está en condiciones de definir la libreta. Una posible defínción de la misma es considerar que una libreta consiste de contactos y un mapeo total entre los contactos y sus direcciones. Esto se especifica en Alloy como se muestra a continuación:

sig *Book* {
    contacts:set *Name*,
    addressOf: contacts → one *Address*
}

Más información sobre Alloy puede encontrarse en [22]

- **DynAlloy.** DynAlloy [15] 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. DynAlloy toma el sistema de tipos, expresiones y fórmulas Alloy e incorpora la habilidad de especificar y chequear automáticamente aserciones de correctitud parcial, para lo cual incorpora la keyword `assertCorrectness`. Siguiendo con el ejemplo anterior, si se desea definir una acción para modificar una libreta de direcciones agregando un contacto, el código DynAlloy que modela esta acción es el siguiente:

  {*true*}
  AddContact[b: *Book*, n: *Name*, a: *Adreess*]
  {b'.contacts = b.contacts + n && b'.addressOf = b.addressOf ++ (n→a)}

  Esto implica que cada vez que se cumpla la precondición, en este caso *true*, osea, siempre, y se ejecute la acción *AddContact*, si este termina, se llegara a un estado que satisface la poscondición, es decir, existe un nuevo contacto en la libreta y su dirección fue agregada al mapeo de direcciones. Las variables primadas representan el estado final de la ejecución.

  Información adicional sobre DynAlloy puede encontrarse en [23].

- **JDynAlloy.** JDynAlloy [7] Es un lenguaje de especificación relacional, ya que sus tipos de datos son relacionales, creado con el objetivo de disponer de una representación intermedia entre los lenguajes de alto nivel y DynAlloy 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.

Java anotado a JML Simplificado

La etapa de simplificación toma como entrada código Java con anotaciones JML/JFSL y genera como salida código JML Simplificado. Este lenguaje no es distinto a Java con JML/JFSL y esta transformación simplemente tiene como objetivo reducir la cantidad de estructuras que deberán ser manejadas en la etapa de traducción a JDynAlloy.

Dentro de las principales modificaciones que se realizan podemos mencionar:

- La inicialización de campos es trasladada a 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 los campos.

- Se reduce la cantidad de estructuras de ciclos, transformándolos a la estructura de un `while`.
- Se simplifica el llamado a métodos en anotaciones JML. Básicamente, se introduce un existencial que cuantifica una única variable por cada llamado, luego se asigna el resultado del método a su correspondiente variable cuantificada. Este procedimiento es necesario debido a que en Java los llamados a métodos son expresiones, mientras que en JDynAlloy son predicados.
- Los condicionales son reestructurados para simular la evaluación lazy que se realiza en Java. Para ello se realiza una construcción con sentencias `if` anidadas que computan el resultado del condicional en una variable booleana.
- Con el objetivo de evitar la colisión de nombres entre variables, el simplificador realiza un renombre de las variables miembro de las clases y también de los parámetros de los métodos.

JML Simplificado a JDynAlloy

En esta sección abordaremos el proceso de transformación de código JML Simplificado al lenguaje de especificación JDynAlloy. Gracias a que ciertas estructuras de JDynAlloy emulan estructuras propias de lenguajes orientados a objetos, la transformación de un lenguaje de dicho paradigma a JDynAlloy es bastante directa.

Por cada clase Java, se define un módulo JDynAlloy que contendrá la definición de sus variables miembro representada por los `fields` y cada método será representado por un `program` en JDynAlloy.

La mayoría de las sentencias JML tienen una correspondencia directa con sentencias JDynAlloy por lo que a continuación se presentarán aquellos casos en los que esta regla no se cumple:

- **Manejo de excepciones**. La construcción `try{} catch{}` se traduce a una serie de condicionales que incluyen comparaciones contra una nueva variable, `throw`, que simula la presencia de una excepción.
- **Cortes de control**. Java brinda la posibilidad de realizar cortes de control en el código mediante el uso de keywords como `return` o `throw`. Es decir, dentro de un método, un programador puede introducir múltiples puntos de retorno, lo que produce que ciertos caminos de ejecución no sean alcanzados.

  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.
- **Traducción de expresiones sin correspondencia**. Como se mencionó anteriormente, la traducción de JML Simplificado a JDynAlloy es bastante directa, por lo que la mayoría de estas construcciones tienen un mapeo uno a uno. Por ejemplo, keywords como `assert`, `assume`, `invariant`, `requires`, `ensures`, etc. tiene su contraparte directa en el lenguaje JDynAlloy. Sin embargo, esto no siempre es el caso. Para algunas circunstancias hay que encontrar mecanismos para reproducir el comportamiento. Un ejemplo de esto 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, poscondición del método o al invariante de la clase, dependiendo del caso.

Por otro lado, 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. De la misma forma, luego de la traducción a JDynAlloy se crearán dos casos de especificación, uno para el comportamiento normal y otro para el excepcional.

JDynAlloy a DynAlloy

La transformación de un módulo JDynAlloy a DynAlloy, sin ser un proceso trivial, presenta varias correspondencias directas simplificando la traducción.

En primer lugar, caben algunas definiciones. 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 está *sobrecargado*.

El proceso de resolución de los bindings toma en cuenta dicha particularidad de los programas sobrecargados. JDynAlloy genera bindings utilizando reglas similares a las de Java [24]. La principal diferencia es que Java tiene en cuenta la visibilidad de los métodos, 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.

Por otro lado, en JDynAlloy un módulo puede sobreescribir un programa declarado por su *módulo padre* solo si en el programa original está marcado como *virtual*. Para estos casos, en la traducción a DynAlloy se reescribirá la definición del método virtual. El objetivo es simular el *dispatching* en tiempo de ejecución de los lenguajes orientados a objetos agregando código del dispatcher que invocará al método correcto correspondiente al tipo concreto de la instancia *thiz*.

Otro punto importante de la traducción consiste en la resolución del `callSpec`, única fórmula de JDynAlloy que no posee su contrapartida en DynAlloy. 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`. Como finalmente las expresiones deben transformarse en expresiones DynAlloy, es necesario reemplazar las fórmulas `callSpec` por código DynAlloy equivalente. Para esto se utiliza la técnica de *inlining*, reemplazando la invocación del método por su Condición de Verificación.

La traducción de otras sentencias JDynAlloy se realiza, generalmente, de manera directa, aunque existen aquellas que requieren auxiliares de DynAlloy en su traducción.

Finalmente, se genera en DynAlloy un `assertCorrectness` para realizar la verificación del programa como se muestra en el código 6.

```
1   assertCorrectness check_A_f_0[...] {
2     pre={
3       precondicion_A_f_0[...]
4     }
5     program={
6       call A_f_0[...]
7     }
8     post={
9       postcondition_A_f_0[...]
10    }
11  }
```

Figura 6: 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.

DynAlloy a Alloy

Como se mencionó anteriormente, DynAlloy es una extensión de Alloy y el proceso de traducción de DynAlloy a Alloy está basado en el mecanismo de *Weakest Precondition*. Este proceso es muy complejo e implica la traducción de una noción de estado a un predicado. No se incluye aquí por escapar al alcance de este trabajo. Puede encontrare más información en [15, 23].

Para la verificación final existe un programa llamado *Alloy Analyzer* que permite realizar la verificación automática de una 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.

De esta forma, se agrega al programa Alloy la instrucción para realizar la verificación y se lo envía al *Alloy Analyzer*.

2.4. Uso y resultado

TACO requiere Java 1.6 y, hasta el momento de desarrollo del presente trabajo, la única forma de ejecutar un análisis utilizando esta herramienta era mediante línea de comando siguiendo, a grandes rasgos, el siguiente esquema básico:

```
java -jar taco.jar -cf {configfile} -c {classname} -m {methodname}
```

donde el primer parámetro es un archivo de configuración y el segundo y tercer parámetro representan el nombre de la clase y del método a analizar respectivamente.

TACO está disponible para las siguientes arquitecturas: amd64-linux, x86-freebsd, x86-linux, x86-mac, x86-windows.

Una vez completado el análisis de un programa, TACO informará del resultado utilizando la salida estándar indicando si se pudo satisfacer la Condición de Verificación, informando si se encontró un error o no. Hasta el desarrollo de esta tesis, no se incluía en el informe del error, dónde fue detectado, el estado de las variables en ese momento y tampoco qué aserción falló.

3. TacoPlug

3.1. Introducción

En la presente sección se presenta TacoPlug, un plugin que integra TACO a Eclipse [25]. Eclipse es una IDE ampliamente difundida y utilizada tanto en ambientes académicos como industriales que permite el desarrollo de aplicaciones en distintos lenguajes como ser Java, C++ o Python; se encuentra disponible para todas las grandes plataformas y, como TACO, está desarrollada casi completamente en Java. Una característica distintiva de Eclipse es su soporte para agregar nuevas funcionalides mediante un sofisticado sistema de plugins.

Debido a esto, la construcción de un plugin para esta IDE que permita la integración con TACO pareció ser la opción natural.

TacoPlug es el resultado de un esfuerzo por hacer TACO más amigable y práctico al usuario. Una vez instalado permite realizar una verificación acotada sobre cualquier método de su elección. Se adoptó como premisa que el usuario no necesitase conocer nada sobre la arquitectura de TACO o la presencia de lenguajes intermedios. De esta forma se planteó como objetivo que toda la información que se mostrase al usuario debía ser provista en lenguaje Java o al menos en este nivel de abstracción.

TacoPlug además provee a Eclipse de una nueva perspectiva que incluye un conjunto de nuevas vistas que proveen mecanismos para debuggear el programa bajo análisis.

El desarrollo de este plugin debió ser dividido en dos etapas. Primero debieron diseñarse las técnicas y los mecanismos que se creían necesarios para dar soporte a los requerimientos funcionales definidos para el plugin, para luego implementarlos cómo contribuciones a TACO. En una segunda etapa se diseñaría y construiría el plugin mismo.

Mientras que en la sección 3.2 se presentan las contribuciones realizadas sobre la herramienta TACO, la sección 3.3 describe las distintas funcionalidades del plugin. Finalmente, en el capítulo 5 se presenta la arquitectura y diseño del plugin.

3.2. Contribuciones a TACO

Para mejorar la usabilidad de la herramienta de verificación TACO, debieron diseñarse una serie técnicas que brindaran la posibilidad de cumplir con los requerimientos que se consideraron indispensables para lograr una herramienta más amigable en su uso.

Se trabajó, principalmente, sobre cuatro ejes que serán comentados en las siguientes subsecciones:

1. Trazabilidad del error encontrado a nivel Java.
2. Evaluación de expresiones JML y JFSL.
3. Validación de predicados.
4. Visualización global de la memoria.

3.2.1. Trazabilidad Java

La trazabilidad tiene como principal objetivo la visualización de la traza a nivel Java que genera el error detectado por TACO. Para esto se extendió un mecanismo desarrollado en [26] que permite la obtención de la traza a nivel DynAlloy y a partir de esta, reconstruir su correspondencia en código Java.

De esta forma el usuario cuenta con la posibilidad de hacer un seguimiento de las instrucciones Java que fueron ejecutadas hasta obtener el error y poder así entender de dónde proviene la falla.

3.2.2. Evaluación de expresiones JML y JFSL

Para poder analizar una traza, suele no ser suficiente visualizar los puntos de ejecución que generaron la falla, sino que puede requerirse revisar el contenido de distintas variables que constituyen el contexto de ejecución de esa traza. Con el objetivo de dotar al plugin de esta capacidad, se trabajó para proveer a TACO de un mecanismo que permitiera la evaluación de expresiones JML y JFSL.

De esta forma es posible evaluar expresiones arbitrarias escritas en lenguaje JML/JFSL y chequear su validez para un determinado punto de la ejecución simbólica fallida encontrada por TACO.

3.2.3. Validación de predicados

Como se ha dicho, TACO transforma el programa Java en una fórmula lógica conocida como *Condición de Verificación* compuesta por un predicado que incluye la poscondición del método analizado y los *object invariants*.

Cuando TACO reporta un error, no informa si el problema encontrado proviene del incumplimiento de una poscondición o un object invariant. Esta información resulta de vital importancia al usuario para poder empezar a investigar dónde se encontró la violación a la especificación.

Se incorporó a TACO un mecanismo para identificar qué fórmula es la que falló. Para esto se hace uso de la evaluación de expresiones JML/JFSL, según el lenguaje en el que se encuentre escrita la especificación, en el último punto de la traza obtenida del contraejemplo.

3.2.4. Visualización global de la memoria

Cuando se construyen programas con estructuras recursivas o con objetos complejos que contienen como campos otros objetos complejos, la visualización del contenido de una variable no suele ser suficiente para comprender el contexto de ejecución.

Es por esto que se desarrolló un mecanismo que permite conocer el estado del heap en un punto determinado de la ejecución simbólica desarrollada por TACO. Este mecanismo analiza el programa y el punto de ejecución seleccionado para detectar las variables relevantes (variables miembro, variables locales, parámetros) y construir un grafo que contenga el valor de las mismas en ese punto.

3.3. Características de TacoPlug

La segunda etapa del desarrollo del plugin se focalizó principalmente en aspectos relacionados con la usabilidad y en brindar la usuario herramientas que permitieran el debugging y la localización de un error reportado por TACO. Esto incluyó el diseño y desarrollo de los componentes presentados en las siguientes secciones para los cuales se utilizó la infraestructura PDE [27] además de algunos componentes particulares que se mencionarán en las secciones correspondientes.

3.3.1. Configurando TACO

El primer paso para poder integrar TACO a Eclipse fue brindar la posibilidad de configurar los diversos parámetros que componen TACO. Para esto se incorporó a la pantalla de configuración de Eclipse una subsección llamada *Taco Configuration*. Desde esta pantalla, que se muestra en la figura 7, es posible especificar distintos valores que requiere TACO para su funcionamiento, principal y obligatoriamente el tamaño del scope y la cantidad de loop unrolls.

Como se mencionó anteriormente, TACO realiza una verificación acotada, lo que significa que son examinadas todas las ejecuciones de un procedimiento dentro de los parámetros definidos por el usuario: la cantidad de instancias de una clase presentes en el heap (definido por el tamaño del scope) y la cantidad de loop unrolls. Por este motivo, estos valores siempre deben definirse.

Además de estos parámetros, existen otros valores de configuración para TACO que pueden definirse en esta pantalla.

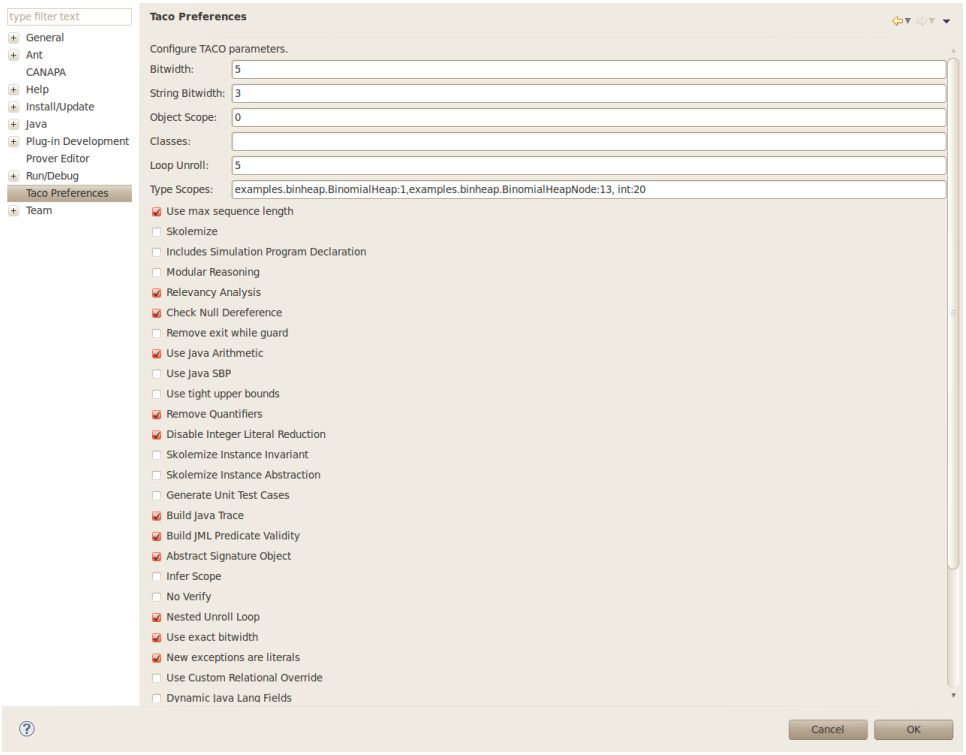

Figura 7: Pantalla de configuración de TACO

Por otro lado, TACO requiere algunos parámetros extra para funcionar que no se configuran desde esta pantalla sino que se calculan automáticamente. Además de lograr mayor usabilidad, al desarrollar un plugin para Eclipse para integrar TACO, se pueden utilizar algunos mecanismos provistos por la IDE para calcular algunos de los parámetros necesitados utilizando conceptos disponibles al trabajar con unidades como *proyectos Java* . Estos parámetros son propios de cada ejecución de un análisis por lo que se explicarán en la siguiente subsección.

3.3.2. Ejecutando un análisis

La ejecución de TACO desde línea de comando, como se mencionó anteriormente, complica su uso. Es por eso que una de las primeras funcionalidades que fueron desarrolladas para el plugin fue la capacidad de ejecutarlo desde Eclipse, de una manera similar a como se ejecuta un test de jUnit [28].

Para esto se desarrolló un *launcher*, llamado *TACO Launcher*, que permite definir el método a analizar en conjunto con el tipo de análisis a realizar. Este launcher es accesible desde la ventana que permite ejecutar aplicaciones Java, tests de jUnit, etc, conocida en Eclipse como '*Run as...*' y puede verse una captura del mismo en la figura 8.

Para esta vista, se decidió que debía indicarse el proyecto que se va a analizar, qué clase y qué método en particular. Para estos tres elementos se desarrollaron buscadores que analizan workspace, proyectos y clases, respectivamente, para acotar la búsqueda del usuario a lo ya seleccionado si se va de lo general a lo particular. Sin embargo, si no se seleccionan elementos generales, por ejemplo, no se selecciona proyecto, el buscador de clases mostrará todas las clases disponibles en el workspace. De la misma forma, si no se selecciona una clase, se mostraran todos los métodos analizables en el workspace. Además, si en estas condiciones se busca y selecciona un método, los campos de clase y proyecto se completaran automáticamente con la información acorde.

Adicionalmente, desde esta pantalla es posible definir si el análisis que se desea realizar es de tipo *Run* o *Check*. Un análisis de tipo *Run* solamente prueba que exista un camino de ejecución

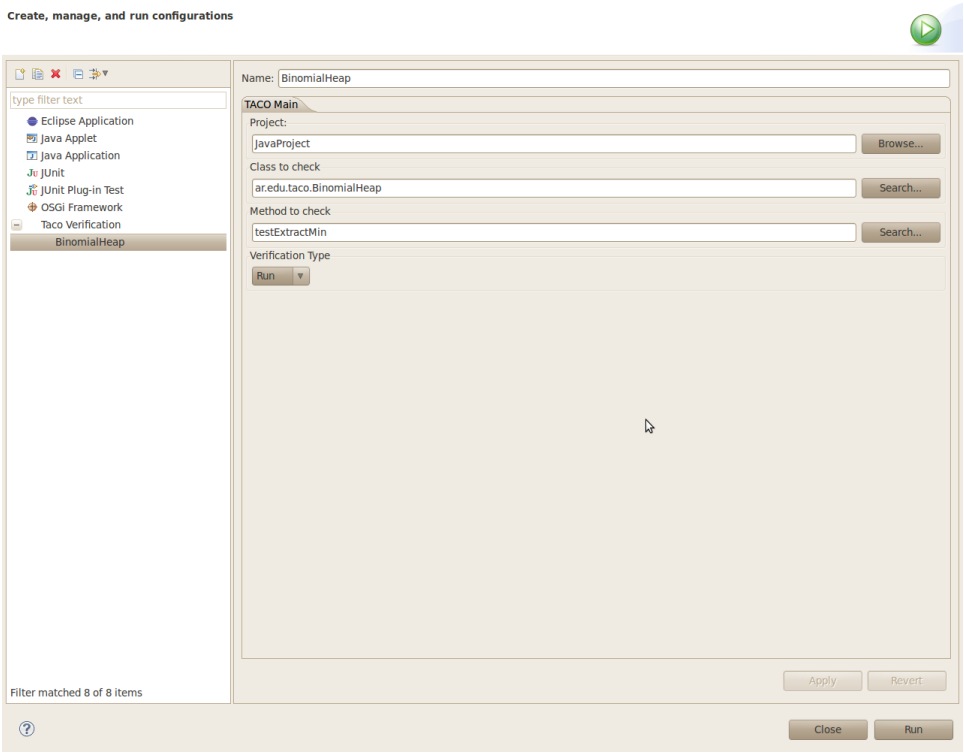

Figura 8: Pantalla de ejecución de análisis TACO

buscando una instancia que ejecute el código, lo cual permite chequear que el invariante sea correcto. Un análisis de tipo *Check* valida el programa contra su especificación.

Como se comentó anteriormente, algunos parámetros propios de cada ejecución de un análisis se calculan automáticamente:

- **Directorio con el código a analizar**. El primer caso en el que se dio esta situación es el caso de `jmlParser.sourcePathStr`, parámetro utilizado para definir dónde se buscan los archivos fuente a ser analizados. Este parámetro se computa automáticamente conociendo qué clase se desea analizar, parámetro registrado en el *launcher*.
- **Conjunto de clases relevantes**. El segundo parámetro que es calculado automáticamente es `relevantClasses` que representa al conjunto de clases relevantes. En este caso, además de lograrse automatización, se logra precisión porque se incluyen exactamente las clases de las cuales un método depende, no las clases de las cuales depende la clase en su conjunto.

3.3.3. TACO Perspective

Cuando se completa un análisis TACO, la *Consola* muestra el resultado del mismo. En caso de encontrarse una falla, se le presenta al usuario la posibilidad de utilizar una serie de vistas desarrolladas con el objetivo de debuggear y localizar la falla en cuestión. Estas vistas se encuentran agrupadas bajo una nueva perspectiva de Eclipse conocida como *TACO Perspective*.

Esta perspectiva se asemeja a la de *Debugging* incluida en Eclipse e incluye vistas que permiten al usuario inspeccionar estáticamente el método bajo análisis, la traza del error y el valor de las distintas variables involucradas en los distintos puntos de ejecución. La figura 9 muestra la nueva perspectiva.

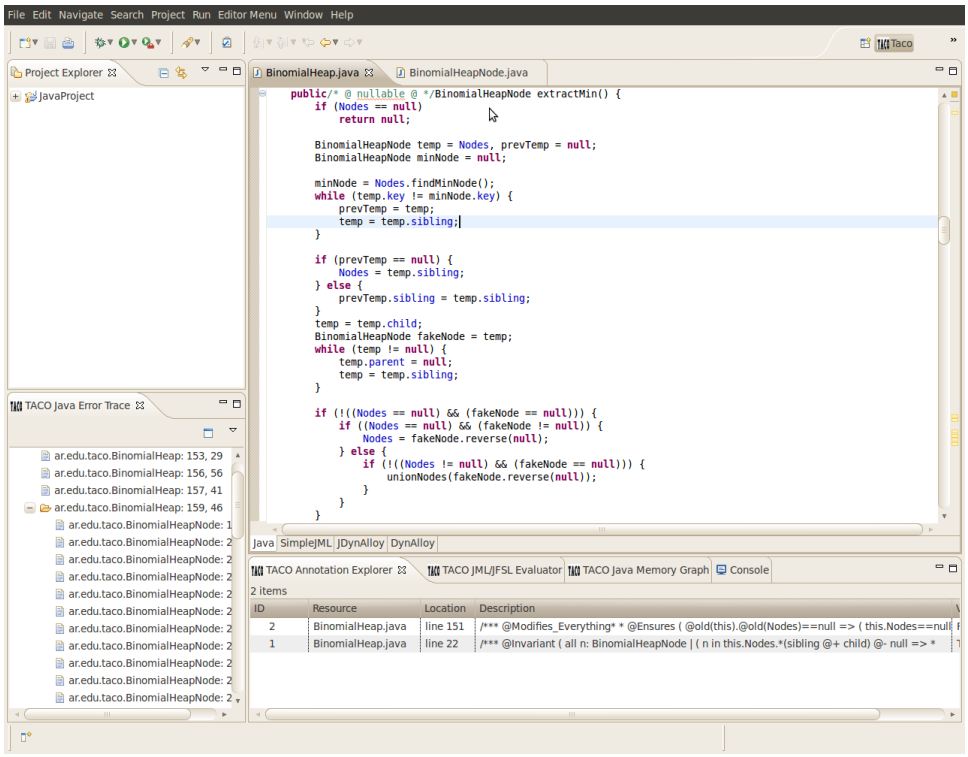

Figura 9: TACO Perspective

3.3.4. JML/JFSL Annotation Explorer

Una vez completado un análisis, la primera vista que resulta útil es la conocida bajo el nombre de *JML/JFSL Annotation Explorer*. Esta vista lista todas las anotaciones del programa analizado y su valor final permitiendo al usuario aislar qué fórmula JML o JFSL de la especificación no fue satisfecha, como se muestra en la figura 10.

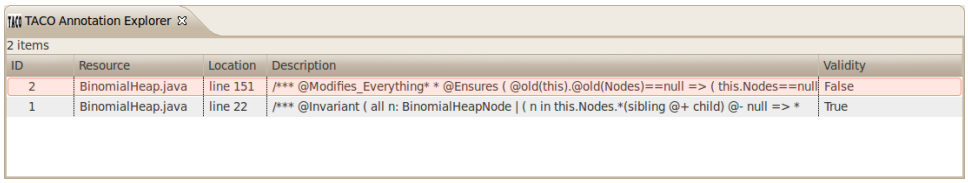

Figura 10: JML/JFSL Annotation Explorer

Para el desarrollo de este componente fue necesaria la evaluación de expresiones JML/JFSL que se comentó en la sección 3.2.

3.3.5. Java Error Trace

Detectada qué fórmula de la especificación no fue satisfecha en la verificación realizada por TACO, resulta útil poder analizar la traza que genera el error detectado. Para lograr este análisis se desarrolló un componente que permite la visualización y navegación de la traza reportada con formato de árbol, como se muestra en la figura 11.

Esta vista permite debuggear un análisis TACO de una manera similiar a como se debuggea un programa usando la vista de *Stack Trace* incluida en la perspectiva de *debugging* provista por

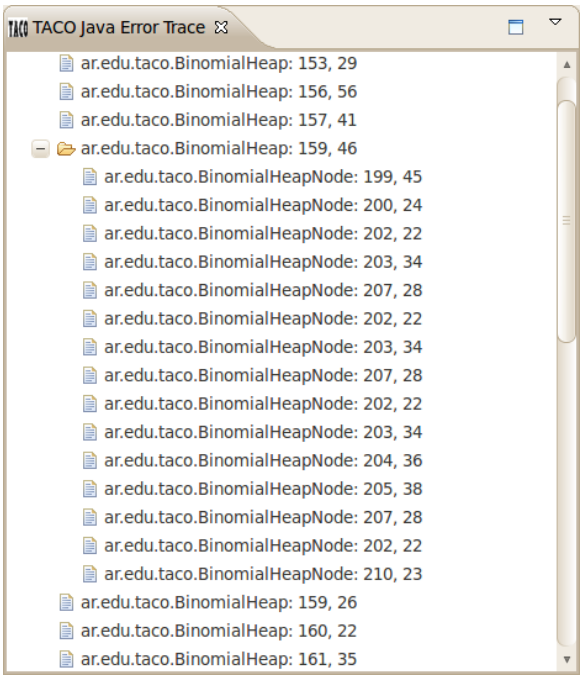

Figura 11: Componente de visualización de la traza Java

Eclipse. La diferencia con esta es que, al tratarse en este caso de una ejecución estática, se puede avanzar y retroceder en la traza según se lo requiera.

Java Error Trace está organizada en forma de árbol donde cada nodo representa un punto en la ejecución y los nodos padre representan llamadas a métodos. Cada nodo contiene una etiqueta compuesta por el nombre de la clase de ese punto de ejecución y los números de línea y columna, en ese orden. Hacer doble click en cualquiera de los nodos causará la localización del foco del cursor en la correspondiente línea Java del archivo fuente.

Es posible que algunos números de líneas se encuentren repetidos en la traza. Esto se debe a que en Java es posible combinar llamados a métodos, como se muestra en la figura 12.

```
1    object.methodA(methodB());
```

Figura 12: Ejemplo de combinación de llamados a métodos en Java

Para estos casos se generarán 2 nodos padres en la traza donde uno contiene el llamado `methodB` y el otro a `methodA`. Estos nodos tendrán el mismo número de línea pero diferirán en el número de columna.

3.3.6. JML/JFSL Evaluator

TACO es una herramienta pensada para ser utilizada con programas que tengan definidas especificaciones en JML o JFSL. Por lo tanto, a medida que se navega la traza, resulta útil saber el valor de las distintas variables que intervienen en un punto de ejecución. Además también podría querer saberse el valor de computar determinada fórmula JML o JFSL.

Con este objetivo, se introduce la vista *JML/JFSL Evaluator*. Esta vista permite definir expresiones JML/JFSL y evaluarlas en distintos puntos de la traza. Es similar a la vista *Expressions* contenida en la perspectiva *Debug* y en la figura 13 se muestra un ejemplo.

Figura 13: JML/JFSL Evaluator

3.3.7. Java Memory Graph

Cuando se analizan programas que incluyen estructuras de datos complejas y vinculadas, la visualización del valor de cierta variable o expresión suele ser insuficiente. En esos casos, el programador debe inspeccionar varias expresiones hasta poder entender cómo los objetos están almacenados en la memoria.

Para facilitar la tarea del programador, se desarrolló la vista *Java Memory Graph* que despliega un grafo donde los nodos son objetos y los arcos son nombres de variables en un determinado punto de ejecución. Los valores primitivos, como ser enteros o booleanos, son visualizados como atributos internos de los objetos.

A medida que el usuario se mueve por la traza, un nuevo grafo es desplegado, como se muestra en la figura 14.

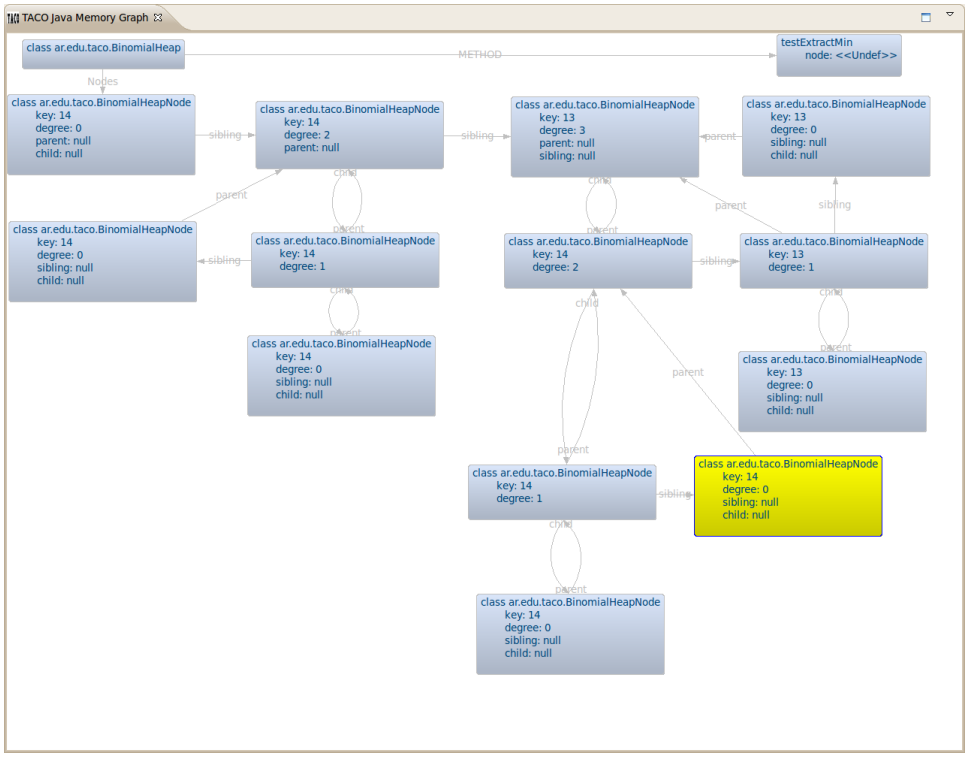

Figura 14: Java Memory Graph

3.3.8. TACO Editor

Por último, se desarrolló un nuevo tipo de editor que permite la visualización de las interpretaciones intermedias del programa Java: JML Simplificado, JDynAlloy y DynAlloy. Este nuevo

editor es conocido como *TACO Editor* e incluye distintas pestañas que muestran, luego de un análisis, las distintas representaciones intermedias como se muestra en la figura 15 donde puede verse el archivo Java original, mientras que en las figuras 16, 17 y 18 pueden verse (mas no editarse) las representaciones en JML Simplificado, JDynAlloy y DynAlloy respectivamente.

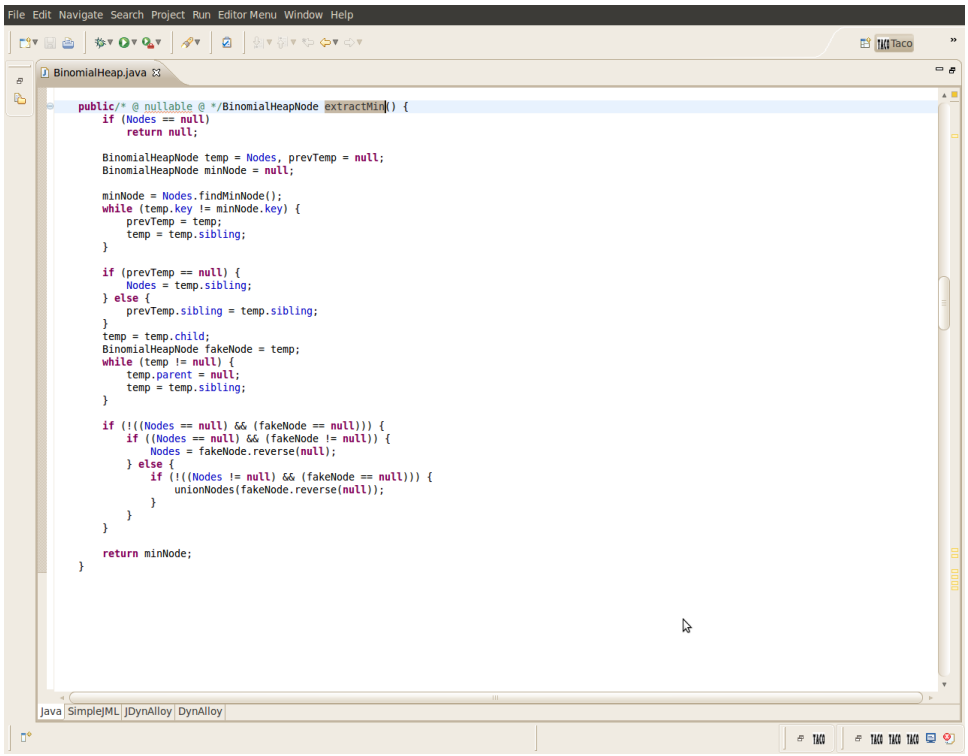

Figura 15: TACO Editor visualizando código Java

Este editor será útil principalmente a usuarios avanzados de TACO que esten interesados en un análisis más profundo de la verificación realizada.

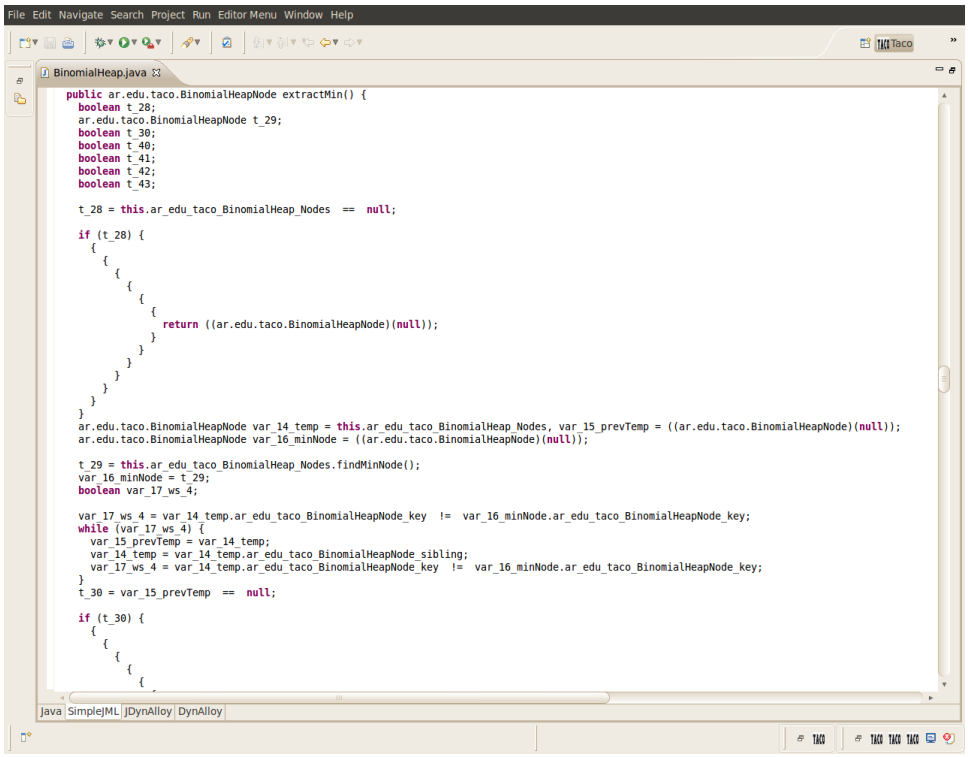

Figura 16: TACO Editor visualizando la versión JML Simplificado

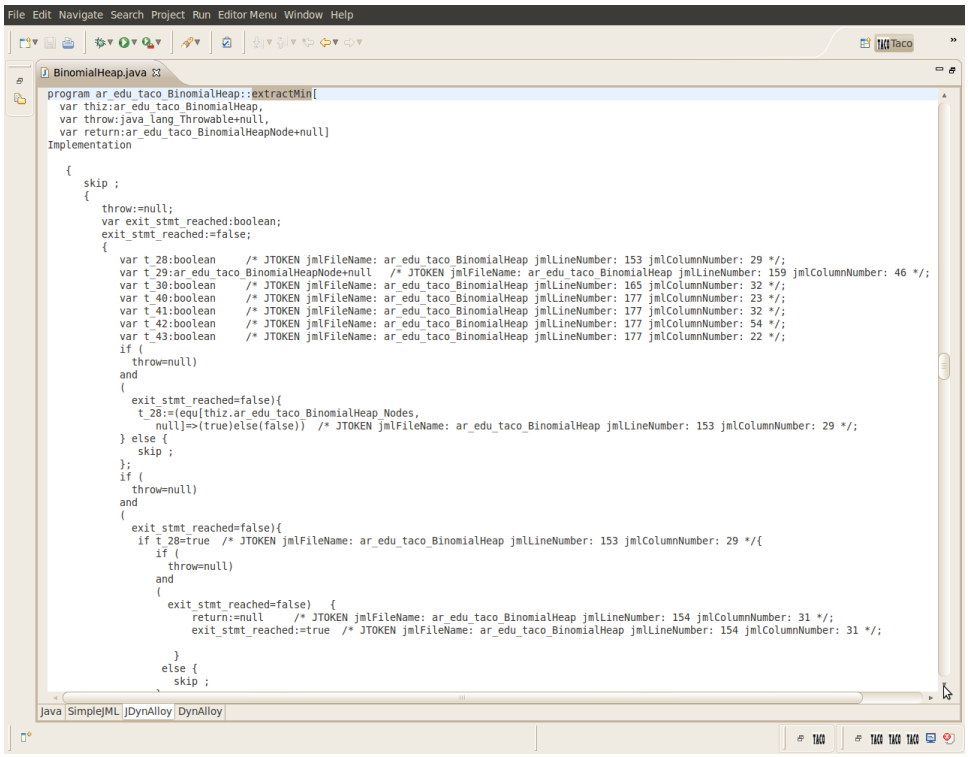

Figura 17: TACO Editor visualizando la versión JDynAlloy

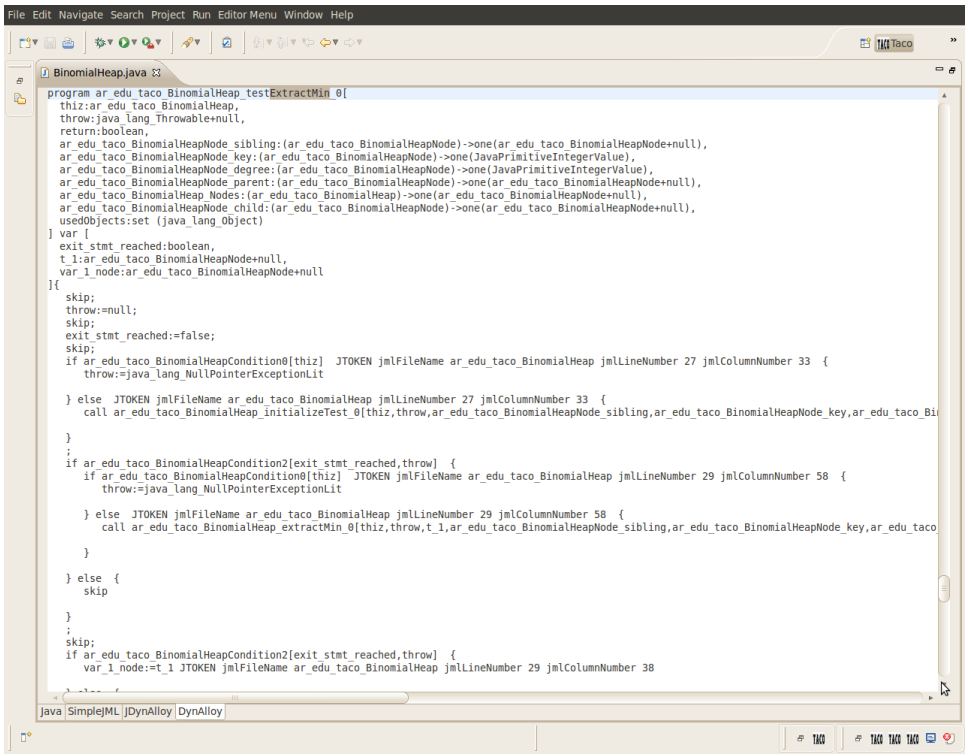

Figura 18: TACO Editor visualizando la versión DynAlloy

4. Caso de estudio: BinomialHeap

Para ilustrar al usuario, consideremos el código fuente Java para extraer el mínimo elemento de un *binomial heap*. Los Binomial heaps [29] son una estructura recursiva diseñada específicamente para implementar colas de prioridad. Por este motivo, una inserción eficiente y la posibilidad de borrar el elemento con menor clave, son características distintivas de esta estructura de datos.

La figura 19 muestra un extracto de la clase `BinomialHeap` que implementa la estructura de binomial heap perteneciente a [30].

```
1  public class BinomialHeap {
2   BinomialHeapNode Nodes; // header
3   ...
4   public BinomialHeapNode extractMin() {
5    if (Nodes == null)
6      return null;
7
8      BinomialHeapNode temp = Nodes, prevTemp = null;
9      BinomialHeapNode minNode = null;
10
11     minNode = Nodes.findMinNode();
12     while (temp.key != minNode.key) {
13       prevTemp = temp;
14       temp = temp.sibling;
15     }
16
17     if (prevTemp == null)
18       Nodes = temp.sibling;
19     else
20       prevTemp.sibling = temp.sibling;
21
22     temp = temp.child;
23     BinomialHeapNode fakeNode = temp;
24     while (temp != null) {
25       temp.parent = null;
26       temp = temp.sibling;
27     }
28
29     if ((Nodes != null) || (fakeNode != null)) {
30       if ((Nodes == null) && (fakeNode != null)) {
31         Nodes = fakeNode.reverse(null);
32       } else if ((Nodes == null) || (fakeNode != null)) {
33           unionNodes(fakeNode.reverse(null));
34       }
35     }
36
37    return minNode;
38   }
39 }
```

Figura 19: Extracto de la clase `BinomialHeap`

Por otro lado, las figuras 20 y 21 muestran el invariante de la clase y la especificación del método `extractMin` respectivamente, escritos usando JFSL.

A diferencia de JML, las construcciones JFSL permiten al usuario escribir fórmulas en lógica de primer orden utilizando operadores relacionales como ser la navegación, unión, disyunción, clausura reflexo transitiva, etc..

```
@Invariant ( all n: BinomialHeapNode |
        ( n in this.Nodes.*(sibling @+ child) @- null =>
          ( ( n.parent!=null => n.key >= n.parent.key ) &&
            ( n.child!=null =>
              n !in n.child.*(sibling @+ child) @- null
            ) &&
            ( n.sibling!=null =>
              n !in n.sibling.*(sibling @+ child) @- null
            ) &&
                  ( ( n.child !=null && n.sibling!=null ) =>
                     ( no m: BinomialHeapNode |
                  ( m in n.child.*(child @+ sibling) @- null &&
                    m in n.sibling.*(child @+ sibling) @- null )
                )
           ) &&
           ( n.degree >= 0 ) &&
           ( n.child=null => n.degree = 0 ) &&
           ( n.child!=null =>n.degree=#(n.child.*sibling @- null) ) &&
           (
            #( ( n.child @+ n.child.child.*(child @+ sibling) ) @- null )
            =
            #( ( n.child @+ n.child.sibling.*(child @+ sibling)) @- null )
           ) &&
           ( n.child!=null => (
            all m: BinomialHeapNode |
                  ( m in n.child.*sibling@-null => m.parent = n ) )
           ) &&
           (
            ( n.sibling!=null && n.parent!=null ) =>
              ( n.degree > n.sibling.degree )
           )
          )
        )
      ) &&
      ( this.size = #(this.Nodes.*(sibling @+ child) @- null) ) &&
      ( all n: BinomialHeapNode | n in this.Nodes.*sibling @- null =>
        ( ( n.sibling!=null => n.degree < n.sibling.degree ) &&
        ( n.parent=null ) )
      ) ;
```

Figura 20: Invariante de clase de `BinomialHeap`

```
@Modifies Everything

@Ensures ( @old(this).@old(Nodes)==null => ( this.Nodes==null &&
                                             return==null ) )
   && ( @old(this).@old(Nodes)!=null => (
        (return in @old(this.nodes)) &&
        ( all y : BinomialHeapNode |
        ( y in @old(this.nodes.key) => y.key >= return.key )
        ) &&
        (this.nodes=@old(this.nodes) @- return ) &&
        (this.nodes.key @+ return.key = @old(this.nodes.key) ) ));
```

Figura 21: Especificación para el método `extractMin` escrita en JFSL

La anotación `@Ensures` para el método `extractMin` establece que toda ejecución del método sobre un binomial heap no vacío termina con otro binomial heap en el que:

- el nodo de retorno fue removido,
- la clave del nodo removido es menor o igual a cualquier otra clave almacenada, y
- ninguna otra clave fue afectada.

4.1. Analizando con TACO

Con todos estos elementos, el usuario ya puede intentar la verificación del programa. Como se dijo anteriormente, TACO requiere un scope de verificación que para este caso se seteara en 5 iteraciones para cada ciclo, una única instancia de `BinomialHeap` y 13 instancias de `BinomialHeapNode`. Luego, como se reportó en [15], TACO responde que la verificación no se mantiene para el scope seleccionado.

Como se muestra en la figura 22, utilizando TACO desde la línea de comandos, esta es la única información que se obtiene. Es posible activar la impresión en formato XML para este contraejemplo pero esto no se incluye aquí por cuestiones de espacio.

En la siguiente sección se presenta cómo se debuggea y localiza el error utilizando TacoPlug.

```
1  ****** Stage Description: Execution of analysis. ******
2
3  00000001  Starting Alloy Analyzer via command line interface
4
5      * Input spec file          : output/output.als
6
7  00000037  Parsing and typechecking
8
9      * Command type             : run
10     * Command label            : examples_binheap_BinomialHeap_extractMin_0
11
12 00027140  Translating Alloy to Kodkod
13
14     * Solver                   : minisat(jni)
15     * Bit width                : 5
16     * Max sequence             : 15
17     * Skolem depth             : 0
18     * Symmbreaking             : 20
19
20 00030060  Translating Kodkod to CNF
21
22     * Primary vars             : 32462
23     * Total vars               : 918439
24     * Clauses                  : 2574455
25
26 00084096  Solving
27
28     * Outcome                  : SAT
29     * Solving time             : 112409
30
31 00142121  Analysis finished
32
33 ***** END: Alloy Stage ******
```

Figura 22: Output de TACO para `extractMin`

4.2. Analizando con TacoPlug

En esta sección se comentará el proceso de debugging y localización de la falla reportada por TACO para el ejemplo de `BinomialHeap`.

El primer paso de este proceso fue la configuración de TACO para su posterior ejecución. Como scope de análisis se utilizó el mismo que fue informado para la verificación utilizando TACO en forma directa en la sección anterior. En este caso, esta configuración se realizó utilizando la vista correspondiente presentada en 3.3.1. Posterior a la configuración se lanzó un análisis de tipo *Check* utilizando el mecanismo presentado en 3.3.2.

Luego de la configuración del plugin para la ejecución y una vez ejecutado el análisis, se procedió a analizar qué parte del código desarrolado no se correspondió con la especificación. Para esto, el punto de partida fue la vista *JML/JFSL Annotation Explorer* la cual permite, como se ha dicho anteriormente, identificar qué fórmula de las constituyentes de la condición de verificación, falló. Utilizando esta vista puede verse que el invariante de la clase se mantiene pero falla la poscondición, como puede verse en la figura 23.

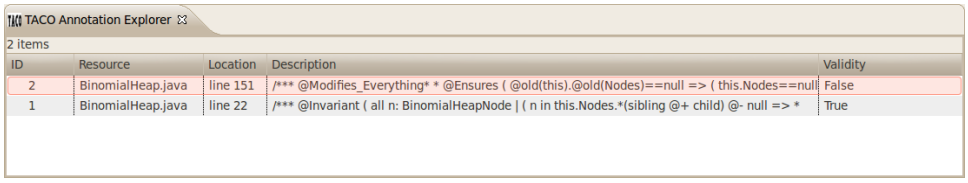

Figura 23: JML/JFSL Annotation Explorer luego del análisis de `extractMin`.

La primera línea del cuadro muestra que la poscondición del método no se mantiene mientras que la segunda dice que el invariante es válido.

Como se marcó con anterioridad, la poscondición del método implica las siguientes condiciones:

- el nodo de retorno fue removido,
- la clave del nodo removido es menor o igual a cualquier otra clave almacenada, y
- ninguna otra clave fue afectada.

Sabiendo que el problema está en la poscondición, debe detercarse cuál de estas condiciones resulta invalida. Para esto resulta conveniente el uso de la vista *Java Memory Graph* y comparar el estado inicial y el estado final del `BinomialHeap`. En la figura 24 se muestra el estado del heap al iniciar la ejecución del método `extractMin` y en la figura 25 se muestra el estado del mismo justo antes de la instrucción de retorno.

En el estado inicial puede observarse que la estructura cumple con el invariante, lo que implica que es un binomial heap válido, y que tiene 13 nodos. Aquí puede verse como se utilizan los parámetros definidos como scope de TACO: existe una única instancia de `BinomialHeap` y 13 instancias de `BinomialHeapNode`.

Por otro lado, en el estado final, puede observarse que se extrajo un nodo del binomial heap (no es alcanzable por ninguna referencia de la estructura) y que además la clave es menor o igual a cualquier otra clave almacenada.

Sin embargo, si se comparan y contrastan los dos estados de la memoria, puede observarse que el binomial heap se encuentra afectado más allá del nodo extraído: en el estado inicial existen 13 nodos alcanzables y en el final solamente 10, es decir 2 menos de los que debería.

Este análisis permite establecer que es la tercer parte de la poscondición la que no se cumple al final la ejecución del método y que el problema está en que se pierden nodos del binomial heap.

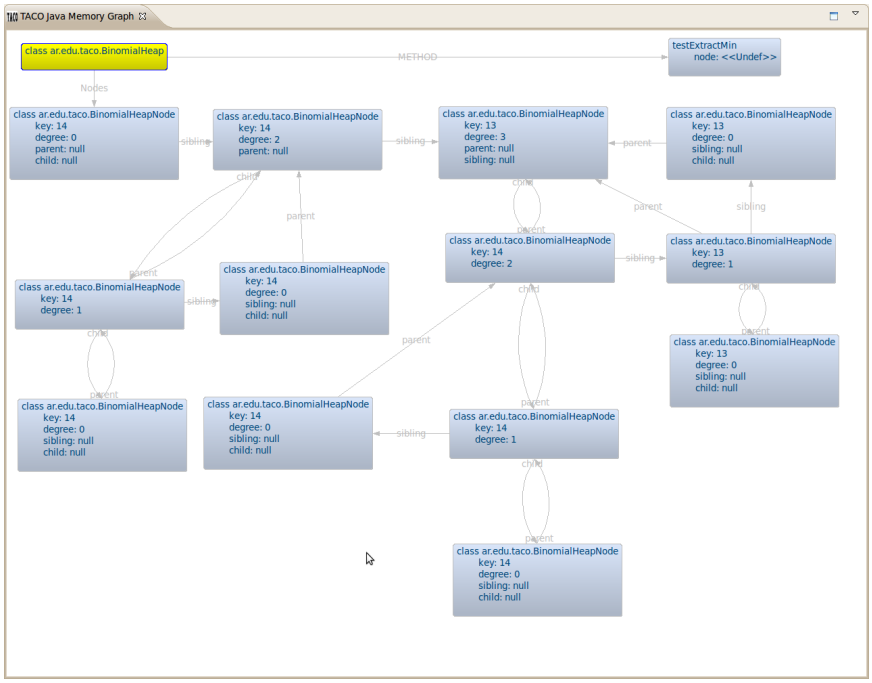

Figura 24: Java Memory Graph inicial para `extractMin`

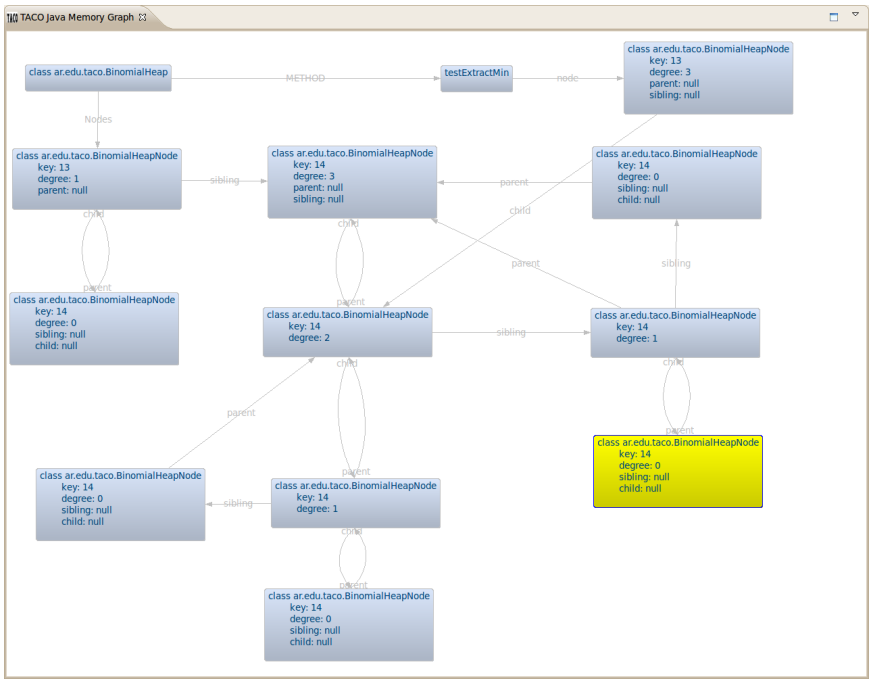

Figura 25: Java Memory Graph final para `extractMin`

Descubierta la discrepancia entre el código y la especificación, el foco se pone en el objetivo de descubrir el origen de la misma. La pregunta que cobra relevancia entonces es cuál es el primer punto de la ejecución dada por la traza que contiene el error en el que en el heap solo existen menos de 13 instancias de `BinomialHeapNode`.

Con el objetivo de localizar este punto de la traza, se utiliza la vista *Java Error Trace* que permite desplazarse a lo largo de la ejecución simbólica reportada por TACO. Realizando una búsqueda binaria sobre los elementos de la traza encontramos que previo al llamado al método `unionNodes`, en la línea 182, todavía son referenciables 13 nodos en el heap como se muestra en la figura 26. Esto muestra que la falla debe estar contenida en el cuerpo del método `unionNodes` y debe analizarse su desarrollo.

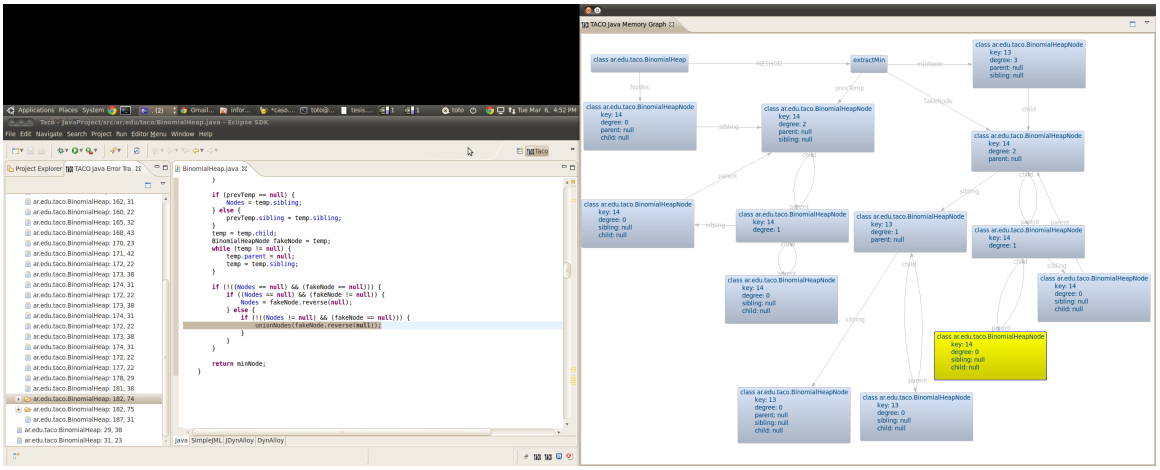

Figura 26: Estado del heap previo a la invocación a `unionNodes`

Una vez en el cuerpo de `unionNodes`, descubrimos que luego de la ejecución del método `merge`, en el heap solo existen 10 instancias de `BinomialHeapNode`, como muestra la figura 27, por lo que la falla debe encontrarse en dicho método.

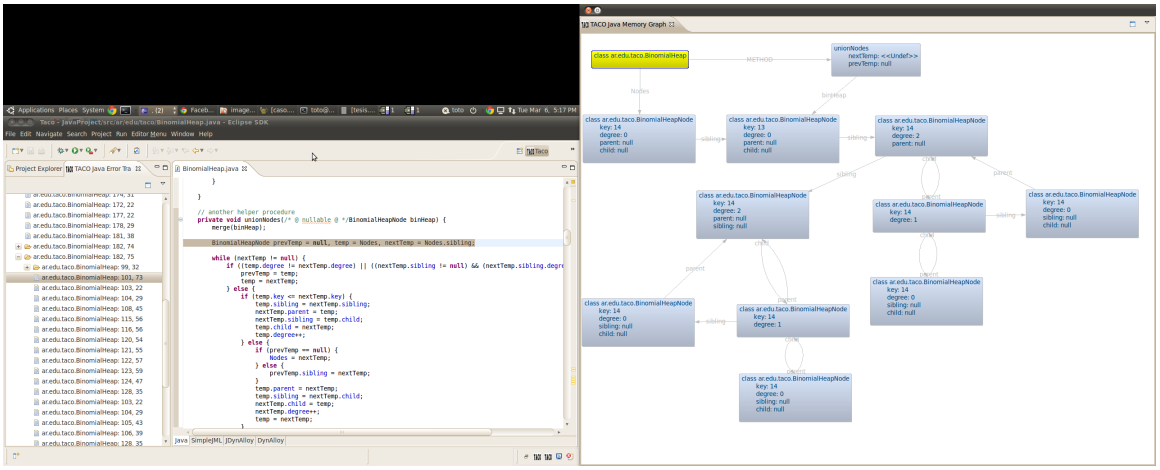

Figura 27: Estado del heap previo a la invocación a `merge`

Analizando el cuerpo del método `merge` nos encontramos con que no existen invocaciones a otros métodos incluidas en este, por lo que acotamos la falla al cuerpo de dicho método.

En este momento, resulta importante analizar el algoritmo que, intuitivamente, une dos binomial heaps. A partir de este punto el análisis involucra un conocimiento profundo del código

ejecutado y una atención especial al algoritmo implementado para unir dos binomial heaps. El código para este algoritmo se incluye en la figura 28

```
1  private void merge(/* @ nullable @ */BinomialHeapNode binHeap) {
2    BinomialHeapNode temp1 = Nodes, temp2 = binHeap;
3    while ((temp1 != null) && (temp2 != null)) {
4      if (temp1.degree == temp2.degree) {
5        BinomialHeapNode tmp = temp2;
6        temp2 = temp2.sibling;
7        tmp.sibling = temp1.sibling;
8        temp1.sibling = tmp;
9        temp1 = tmp.sibling;
10     } else {
11       if (temp1.degree < temp2.degree) {
12         if ((temp1.sibling == null) ||
13             (temp1.sibling.degree > temp2.degree)) {
14
15           BinomialHeapNode tmp = temp2;
16           temp2 = temp2.sibling;
17           tmp.sibling = temp1.sibling;
18           temp1.sibling = tmp;
19           temp1 = tmp.sibling;
20         } else {
21           temp1 = temp1.sibling;
22         }
23       } else {
24         BinomialHeapNode tmp = temp1;
25         temp1 = temp2;
26         temp2 = temp2.sibling;
27         temp1.sibling = tmp;
28         if (tmp == Nodes) {
29           Nodes = temp1;
30         }
31       }
32     }
33   }
34
35   if (temp1 == null) {
36     temp1 = Nodes;
37     while (temp1.sibling != null) {
38       temp1 = temp1.sibling;
39     }
40     temp1.sibling = temp2;
41   }
42 }
```

Figura 28: Algoritmo de merge de dos `BinomialHeap`

A grandes rasgos, el algoritmo de `merge` une los dos binomial heaps analizando las raices de los mismos e iterando sobre los resultados parciales hasta conseguir un único binomial heap. Para esto se utilizan dos variables llamadas `temp1` y `temp2` donde se almacenan las cabezas de los binomial heaps parciales y se compara la variable `degree` (grado) de estas cabezas para determinar cómo se realiza el merge, diferenciando si los grados son iguales, si el grado de `temp1` es menor o si el grado de `temp2` lo es. Para cada uno de estos casos se ejecutará un paso del algoritmo de `merge` distinto.

TacoPlug en este punto resulta útil para recorrer cada uno de los pasos de la traza de ejecución y visualizar el estado de la memoria en estos pasos utilizando los componentes *Java Error Trace* y

Java Memory Graph. Esto permite entender cómo es la ejecución de `merge` sobre un caso particular y encontrar el punto exacto donde empiezan a perderse nodos.

Recorriendo la traza se llega hasta el punto que puede visualizarse en la figura 29 en el que todavía son alcanzables 13 instancias de `BinomialHeapNode`.

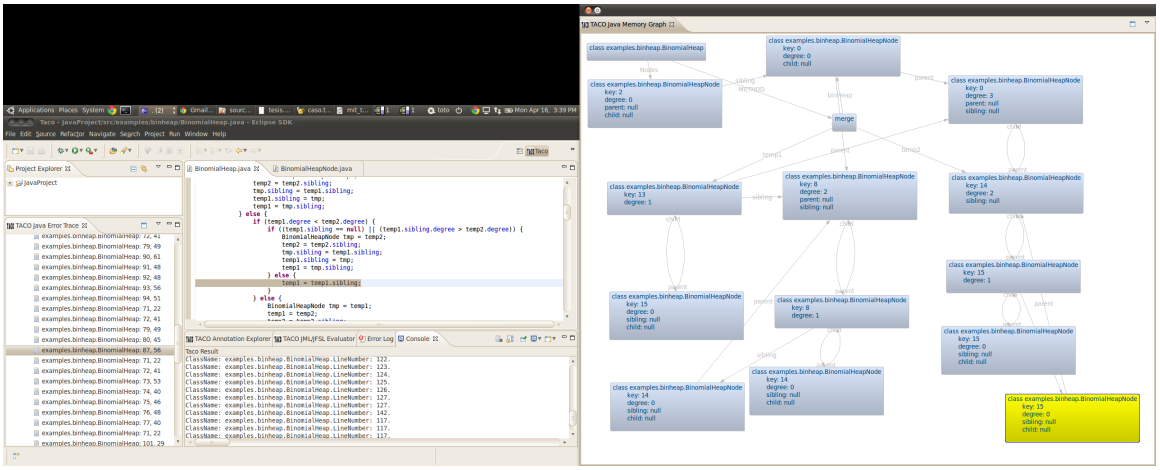

Figura 29: Punto del algoritmo de `merge` previo a la falla

Como puede verse en la imágen capturada, las variables `temp1` y `temp2` apuntan a instancias distintas de `BinomialHeapNode`. Además puede observarse que el grado de la primera de las variables es menor que el de la segunda, lo cual determina qué caso del algoritmo se ejecutará. Para este caso existen dos variantes. El primero requiere que la variable `sibling` de `temp1` sea `null` o que el grado de dicha variable sea mayor que el grado de `temp2`. El segundo caso es simplemente la negación del primero y es el que ocurre en este ejemplo. Para este caso lo único que se ejecuta es la asignación de `temp1` a su hermano directo. De esta forma la variable anteriormente referenciada por `temp1` y las que sean alcanzadas únicamente desde esta (en este caso el nodo hijo), marcadas por el recuardo rojo en la figura 30, son perdidas. Es así como se pierden referencia a 2 nodos del binomial heap y se da una falla en el algoritmo de `merge`, generando una violación en el análisis de TACO.

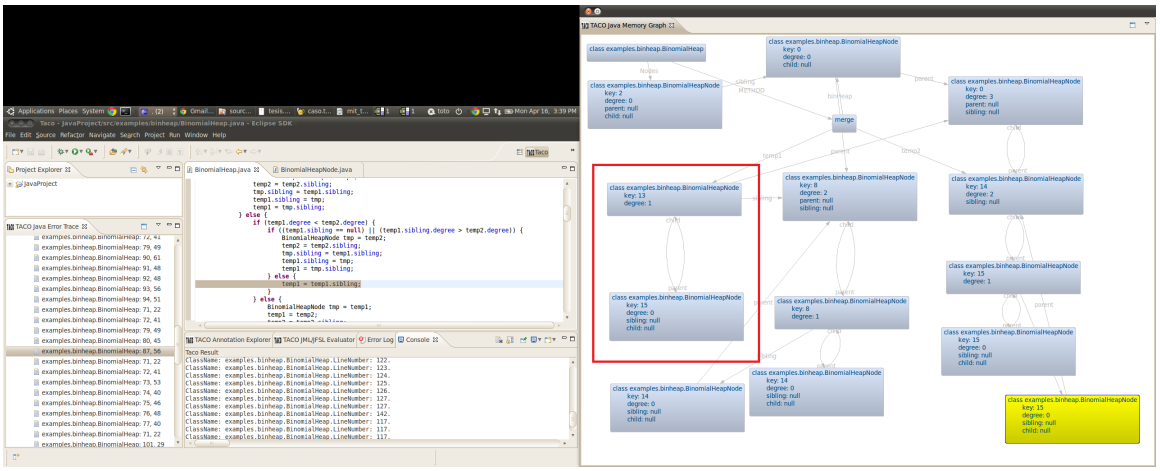

Figura 30: Nodos perdidos durante el algoritmo de `merge`

5. TacoPlug: Diseño y Arquitectura

En la presente sección se presentará con mayor detalle el diseño y la arquitectura de TacoPlug en conjunto con las modificaciones realizadas sobre TACO.

Como se mencionó anteriormente, el proyecto aquí presentado consistió de dos etapas: la primera de modificación de TACO para proveer las condiciones y funcionalidades necesarias para la construcción TacoPlug, y la construcción misma del plugin.

Mientras que en la subsección 5.1 se comentará sobre el diseño de las modificaciones realizadas sobre TACO, en la subsección 5.2 se describirá la arquitectura y diseño de TacoPlug.

5.1. Diseño y Arquitectura de contribuciones a TACO

El desarrollo del presente trabajo obligó a repensar algunos elementos de la arquitectura y diseño de TACO. Al momento de comenzar, TACO funcionaba como una herramienta que procesaba una entrada y generaba un resultado como salida que decía si se había detectado un error o no, pero no tenía mayor interacción con el usuario que la necesaria para inicializar el análisis. Esto, si bien no explícitamente, puede visualizarse en la figura 5. Cuando se empezó a pensar en aumentar la usabilidad de esta herramienta fue evidente que esta concepción de TACO ya no servía. Esto se debe a que TACO dejaría de ser una herramienta de procesamiento lineal y pasaría a ser una con la cual el usuario tiene interacción principalmente para analizar el resultado obtenido. Por otro lado, para poder analizar este resultado y averiguar distintas características del mismo, es necesario, además de guardar el resultado, y debido al diseño de TACO, guardar información sobre cómo se dieron las etapas intermedias y cómo se transformó el código fuente que sirvió como entrada para el análisis. Todo esto obligó a repensar la arquitectura de TACO. Las principales contribuciones a TACO se esquematizan en la figura 31.

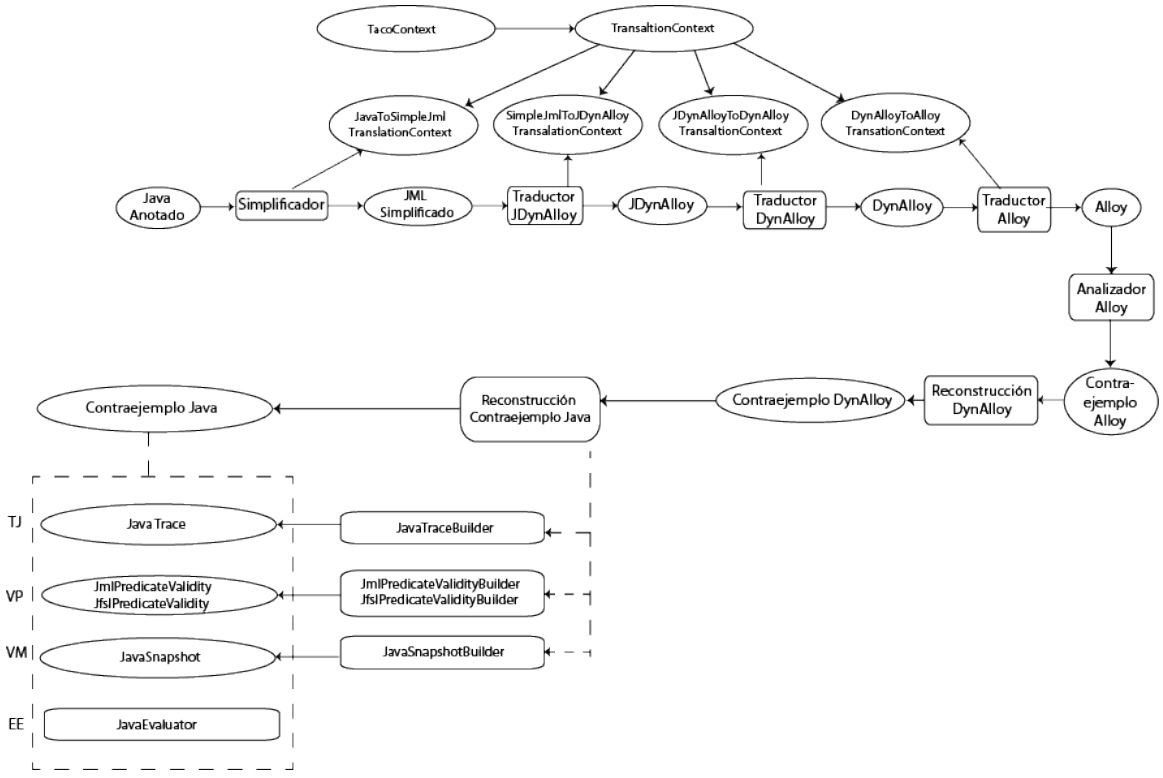

Figura 31: Diagrama de la nueva arquitectura de TACO

En este esquema cada uno de los elementos circulares representa un artefacto, mientras que los rectangulares representan comoponentes. Cada una de estas contribuciones serán explicadas en las siguientes subsecciones, pero valen algunas aclaraciones preliminares que involucran a todos las técnicas y mecanismos aquí presentados.

En primer lugar, como se mencionó anteriormente, se pretende que el usuario no requiera conocer la estructura interna de TACO o la existencia de lenguajes intermedios. Es por esto que toda información que ingrese el usuario debe poder ingresarla en alguno de los lenguajes que maneja: Java, JML o JFSL. Por otro lado, el contraejemplo obtenido se encuentra en lenguaje Alloy y por el mecanismo desarrollado en [26] es posible evaluarlo desde el mundo DynAlloy. Esto implica que el input del usuario debe ser transformado hasta DynAlloy siguiendo las mismas reglas simplificación, renombre de variables y traducciones que se utilizaron para el análisis del programa original. Por este motivo se decidió por la inclusión de contextos de traducción en cada una de las etapas involucradas. Estos contextos de traducción se implementaron como extensiones de la clase `TranslationContext` y existe uno por cada etapa:

- **JavaToSimpleJmlTranslationContext** para la traducción de Java anotado a JML Simplificado.
- **SimpleJmlToJDynAlloyTranslationContext** para la traducción de JML Simplificado a JDynAlloy.
- **JDynAlloyToDynAlloyTranslationContext** para la traducción de JDynAlloy a DynAlloy.
- **DynAlloyToAlloyTranslationContext** para la traducción de DynAlloy a Alloy.

Estos contextos guardan información pertinente a cada transformación. En la figura 32 se incluye un diagrama de clases de los mismos. El desarrollo de estos contextos incluyó, claramente, la modificación de las etapas correspondientes para que la información correspondiente fuera almacenada en los mismos.

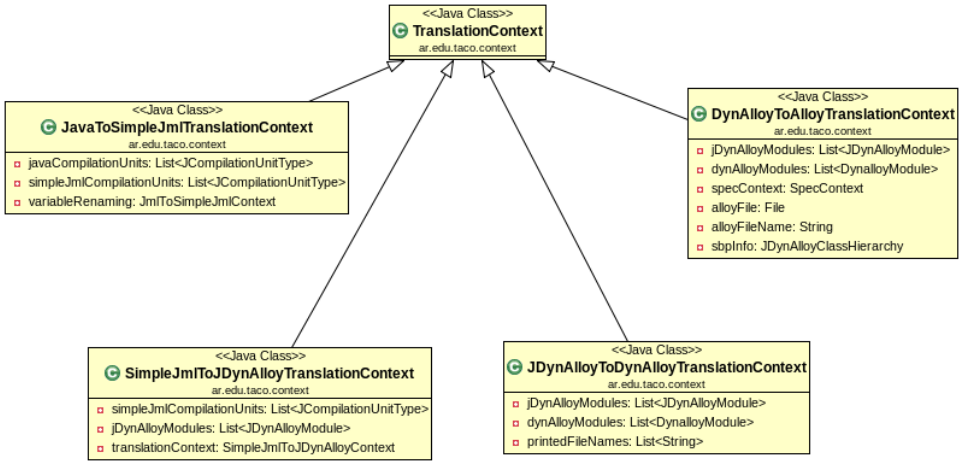

Figura 32: Jerarquía de clases para `TranslationContext`

Como se mencionó, la información de estos contextos debe trascender la ejecución del pipeline de transformación de TACO y su etapa de verificación, por lo que se creo un objeto cuyo objetivo es guardar los distintos contextos de traducción involucrados en el pipeline. Este objeto recibe el nombre de `TacoContext` y, además de guardar los contextos ya mencionados, guarda una copia de las clases originales y el contraejemplo Alloy encontrado durante la verificación.

Estos contextos de traducción pueden visualizarse en la figura 31 en la parte superior de la imagen.

En segundo lugar, como se explicó en el capítulo 2, la arquitectura y diseño de TACO están fuertemente determinadas por el pipeline de TACO. A grandes rasgos, existe una etapa por traducción del pipeline. También existen etapas adicionales, generalmente asociadas a posprocesamiento del resultado encontrado, como la mencionada `JUnitStage`. Siguiendo esta línea, se diseñó una nueva etapa implementada como una *stage*, que recibe el nombre de `JavaCounterexampleStage`. El objetivo de esta etapa es la construcción de un contraejemplo que pueda ser entendido por un usuario inexperto de TACO (es decir, el contraejemplo manejará elementos del mundo Java) a partir del contraejemplo a nivel Alloy.

Esta etapa generará como salida un objeto que encapsulará diversa información útil al usuario a la hora de localizar una falla, como ser: traza Java del error detectado y validez de los predicados (invariantes y poscondiciones) involucrados. Además contará con las capacidades de evaluar expresiones arbitrarias ingresadas por el usuario y de generar un grafo con el estado de la memoria en un punto de la ejecución. Este objeto, al tratarse de un contraejemplo a nivel Java, recibe el nombre de `JavaCounterExample` y también es almacenado en el `TacoContext`. En la figura 33 se incluye un diagrama del clases para este objeto.

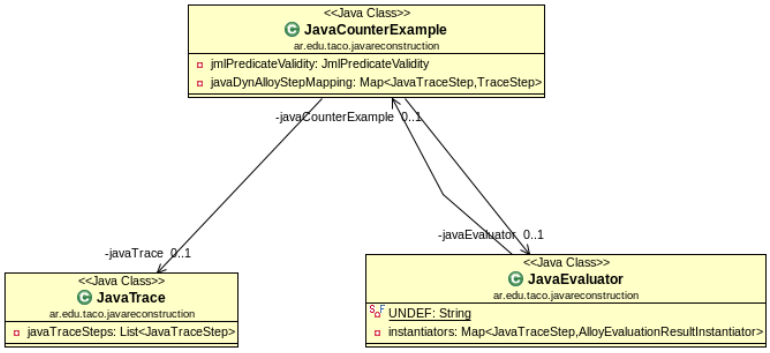

Figura 33: Diagrama de clases para `JavaCounterExample`

La nueva *stage* diseñada puede verse en la parte intermedia de la figura 31, como un cuadro con el texto *Construcción del contraejemplo Java* . De este recuadro se desprende, a modo de resultado, un recuadro punteado que representa al objeto `JavaCounterExample` y que tiene como subitems a los distintos resultados parciales generados por la etapa, traza Java (con la etiqueta TJ), validación de predicados (con el etiqueta VP), en conjunto con las funcionalidades que involucran interacción con el usuario: evaluación de expresiones (con la etiqueta EE) y visualización global de la memoria (con la etiqueta VM). Cada uno de estos puntos es desarrollado en una de las siguientes subsecciones con mayor detalle.

5.1.1. Trazabilidad Java

Como se comentó en 3.2, para la recuperación de la traza que genera una falla se utilizó un mecanismo desarrollado en [26] que permite su reconstrucción a nivel DynAlloy y se incorporó al pipeline de TACO el traslado de una referencia al archivo original con el objetivo de aumentar el nivel de abstracción de dicha traza DynAlloy hasta llegar a algo legible por el usuario, como ser código Java. Además de esto, se modificaron las clases que imprimen las representaciones del archivo original en estos lenguajes intermedios y se modificó el parser de DynAlloy para que parsee un final, opcional, de instrucción que se incorporó a todas las instrucciones DynAlloy siguiendo el siguiente esquema:

`expresion_dyn_alloy JTOKEN` *className lineNumber columnNumber*

donde `expresion_dyn_alloy` representa una expresión DynAlloy, `JTOKEN` es una nueva palabra reservada para indicar que esa instrucción contiene un token de referencia compuesto por *class-*

Name, *lineNumber* y *columnNumber* que representan el nombre de la clase, el número de línea y el número de columna, respectivamente.

Para la reconstrucción de la traza a nivel Java se utilizó la traza a nivel DynAlloy y la información del token reference agregado. De esta forma, utilizando la clase `JavaTraceBuilder`, se construye la traza a nivel Java representada por el objeto `JavaTrace` que luego es guardado como una propiedad de `JavaCounterExample`.

5.1.2. Evaluación de expresiones JML/JFSL

Para este desarrollo se utilizó también un mecanismo desarrollado en [26] que permite la evaluación de expresiones Alloy en distintos puntos de la traza DynAlloy retornando como resultado objetos Alloy.

Por lo tanto, agregar la evaluación de expresiones JML y JFSL requirió de varios etapas que se enumeran a continuación:

1. Parseo de expresiones JML y JFSL.
2. Reconstrucción de expresiones utilizando los renombres de variables correspondientes.
3. Transformación de expresiones JML y JFSL en expresiones DynAlloy.
4. Evaluación de expresiones utilizando el evaluador DynAlloy.
5. Construcción de resultados generando objetos Java.

Algunas de estas etapas se resolvieron utilizando mecanismos ya existentes en TACO o algunas de sus dependencias. Por ejemplo, para el parseo de expresiones se utilizaron clases presentes en las correspondientes dependencias y para la transformación de expresiones JML en expresiones Alloy se utilizó una visitor ya existente en TACO bajo el nombre de `JmlExpressionVisitor`.

Para el punto 2, la reconstrucción de expresiones utilizando los renombres correspondientes de las variables, se utilizaron los contextos de traducción correspondientes y se desarrolló un visitor que clona las expresiones JML/JFSL renombrando las variables según corresponde. Este visitor se encuentra implementado en la clase `JPredicateSimplifierMutator`.

Los dos pasos restantes de la evaluación requirieron un poco más de trabajo y por eso se explican en las siguientes secciones.

La evaluación propiamente dicha de expresiones Alloy depende de un mecanismo diseñado en [26] que evalua este tipo de expresiones. Sin embargo, este mecanismo no tiene en cuenta algunas consideraciones que deben tenerse cuando es el usuario quien decide qué se evalua en un punto determinado: la ausencia de átomos para representar algunas variables en algunos puntos de la traza y el uso del mecanismo de ruptura de simetrías presente en TACO. Estas consideraciones serán explicadas a continuación.

Finalmente se detallará el proceso de construcción de resultados Java a partir de resultados Alloy.

Ausencia de variables

El primer problema que debió atacarse fue el problema de la ausencia de variables. Como se mencionó anteriormente, el problema de SAT-Solving pertenece a la clase de problemas conocida como NP-Completos, lo que significa que el tiempo que demanda su resolución puede ser exponencial con respecto al tamaño de la entrada, es decir la cantidad de variables presentes. Por lo tanto, minimizar el tamaño de la entrada es un objetivo importante en el desarrollo de TACO. Para esto, variables que no son necesarias para determinado análisis, no se traducen a Alloy. Por ejemplo, si un método bajo análisis no realiza ninguna operación sobre una variable miembro de la clase a la cual pertenece, esa variable no existirá en el contexto del análisis.

Por otro lado, en Alloy, al valor de una variable en cada punto de la ejecución se lo conoce como *encarnación*. De esta forma, evaluar una variable en un punto de la ejecución implica evaluar una

encarnación. Sin embargo, por lo mencionado anteriormente, algunas variables pueden no estar encarnadas para determinado nivel de la ejecución aunque siempre se encuentran en el nivel más alto. Por ejemplo, aunque una variable no exista en un método, existirá en el método llamador.

Por lo tanto, para resolver el problema de la ausencia de variables debe encontrarse el punto anterior de ejecución donde la variable sí se encuentre definida, es decir, esté encarnada. Para esto, se recorre el árbol de ejecución hacia atrás, partiendo del punto donde se está evaluando, hasta encontrar el primer lugar donde la variable se encuentre encarnada. El valor de la encarnación, será el valor de la variable en el punto de ejecución seleccionado.

Para la resolución de este problema se desarrolló un visitor que, dado un punto de la traza Java y una expresión Alloy, devuelve una expresion encarnada según la visibilidad de las variables que intervienen. Este visitor está implementado en la clase `JavaIncarnatorMutator`.

Adicionalmente, la evaluación de expresiones Alloy en distintos puntos de la traza Java se encapsuló en el componente conocido como `JavaEvaluator`.

Manejando la ruptura de simetrías

En segundo lugar, cuando se utiliza un mecanismo disponible en TACO conocido como ruptura de simetrías (SBP por sus siglas en inglés que representan *Symmetry Breaking Predicate*), una variable Java puede encontrarse representada por más de un átomo Alloy en el estado inicial de la ejecución simbólica. Por lo tanto, para evaluar expresiones que contengan alguna de estas variables, debe saberse si la variable se encuentra partida o no. Esta información se almacena en la clase `JDynAlloyClassHierarchy` a la que se debe consultar antes de encarnar una expresión Alloy. La instancia utilizada para ruptura de simetrias (en caso de estar habilitada), se almacena como parte del contexto de traducción DynAlloy-Alloy. De esta forma, siguiendo el esquema comentado anteriormente, se incorporó al mecanismo de encarnación un chequeo que analiza si la variable se encuentra partida y en tal caso realiza las modificaciones necesarias para que la encarnación se complete de forma correcta.

Construcción de resultados Java a partir de resultados Alloy

En este punto, dos elementos debieron tenerse en cuenta: la construcción de objetos Java y la posible presencia de aliasing.

Para la resolución del primer problema se diseñó e implementó una clase que, dado un resultado Alloy, lo traduce a Java utilizando *reflection*. `AlloyEvaluationResultInstantiator` es el nombre de dicha clase. El proceso de reconstrucción de una variable Java a partir de un átomo Alloy depende del tipo de átomo. Existen tres tipo de resultados posibles luego de la evaluación de una expresión Alloy, además de `null`:

- `Integer`
- `Boolean`
- `A4TupleSet` con aridad unaria o binaria.

Los dos primeros casos son los tipos básicos de Java y no requieren mayor análisis puesto que son devueltos automáticamente. El tercero de estos valores, `A4TupleSet`, representa tipos complejos, como podrían ser `List`, `Set`, `Map`, `Exception` o tipos particulares de cada análisis definidos por el usuario. Intuitivamente un objeto de `A4TupleSet` puede ser pensado como una tupla, unaria o binaria.

La tarea de construir un objeto Java a partir de uno Alloy de tipo `A4TupleSet` no es trivial. El primer paso para esta construcción es la inferencia del tipo Java del resultado. Esto se logra accediendo a un átomo particular de la tupla: el único presente en el caso de tuplas unarias y el segundo para tuplas binarias. Este átomo contendrá la información necesaria para conocer el tipo Java del resultado. El valor de este átomo siempre es de tipo `String` y es apartir de este

string que se conoce el tipo del resultado. Por ejemplo, el siguiente podría ser el átomo con la información de tipo del `A4TupleSet` que se obtiene luego de evaluar la variable `this` en el ejemplo de la sección 1.1:

{AssertExample$0}

Este string puede interpretarse de la siguiente forma. Siendo el símbolo $ un delimitador, el prefijo representa el nombre calificado del tipo del resultado Java y el sufijo es el *id* de ese objeto en particular (similar a un *object id* en Java). Simplemente extrayendo el prefijo se conoce el tipo del resultado en el mundo Java.

Una vez obtenido el tipo del resultado se procede a instanciar una variable de esta clase, utilizando *reflection*. Utilizando también este mecanismo brindado por Java, se conocen los campos presentes en un objeto de este tipo y se procede a construir una expresion Alloy que, luego de ser evaluada, recorrerá este mismo proceso hasta obtener un resultado en el mundo Java que será asignado al campo en cuestión.

Por otro lado, para resolver el problema de aliasing, cuando se instancia un objeto, se guarda en un diccionario el id del objeto Alloy del cual provino en conjunto con el objeto instanciado. De esta forma, la siguiente vez que se intente instanciar el objeto Alloy, se consultará el diccionario y, de encontrarse el id del objeto Alloy, se devolverá el objeto ya construido, resolviendo el problema del aliasing. Este problema está resuelto en la misma clase anteriormente mencionada.

Una esquematización del proceso de instanciación de variables puede verse en la figura 34. En este esquema, los recuadros representan componentes, mientras que los circulos representan artefactos.

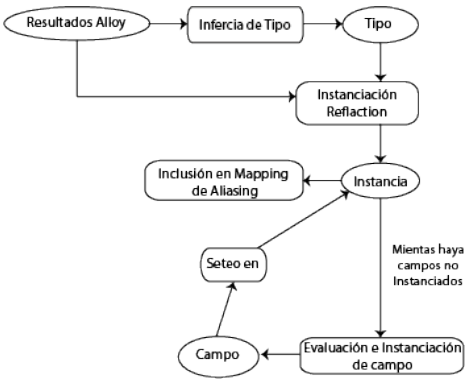

Figura 34: Esquema instanciación resultados Alloy

Finalmente cabe aclarar que para que la instanciación de variables utilizando el mecanismo aquí descripto funcione, es necesario que las clases compiladas se encuentren disponibles al *system class loader*. Esto se resuelve mediante un artilugio de bajo nivel implementado a nivel de plugin al momento de ejecutar un análisis. Se comentará sobre este tema más adelante.

5.1.3. Validación de predicados

Como se explicó en la sección correspondiente, es útil saber cuál de las fórmulas que componen la VC no pudo ser satisfecha. Para esto se diseñó y desarrolló un mecanismo que determina qué parte de la especificación JML/JFSL no pudo verificarse.

El componente que determina si una fórmula de la especificación es correcta o no, funciona distinto dependiendo del lenguaje en el que se encuentre escrita la misma:

- **JML.** Para especificaciones escritas en JML, se guarda un diccionario como parte del contexto de traducción de Java a JML Simplificado que mapea las fórmulas JML a fórmulas Alloy. De esta forma, para saber si una fórmula de la especificación es válida, alcanza con consultar

el diccionario para obtener la fómula Alloy correspondiente y evaluar dicha fórmula utilizando el mecanismo anteriormente desarrollado. Sin embargo, la evaluación de fórmulas Alloy no es directa. Para esto se desarrolló un visitor bajo el nombre `AlloyFormulaEvaluatorVisitor` que descompone la fórmula, evalua los predicados que la componen y recomponen el valor de verdad de la fórmula basándose en los resultados de los predicados evaluados y la estructura de la fórmula.

- **JFSL.** Para especificaciones escritas utilizando JFSL se utiliza un diagrama de evaluación directo ya que las fórmulas JFSL ya se encuentran escritas en lenguaje Alloy. La evaluación de estas fórmulas presenta el mismo problema que la evaluación de fórmulas Alloy traducidas desde JML por lo que se utiliza el mismo visitor para su evaluación.

5.1.4. Visualización global de la memoria

Finalmente, el mecanismo de evaluación de expresiones JML/JFSL permitirá a TACO contar con un componente capaz de generar un grafo con el estado del heap en el punto de la traza que el usuario haya seleccionado, permitiendole obtener una visualización clara del estado de la memoria. El proceso de construcción de este grafo implica dos etapas.

La primer etapa tiene como objetivo el construir un diccionario con un mapeo entre todas las variables involucradas en un determinado punto del programa y sus valores. Esta etapa se encuentra implementada en la clase `JavaSnapshotBuilder` que, dado un `TacoContext` y un punto de la traza Java, retorna una instancia de la clase `JavaSnapshot` que contiene el diccionario buscado. Para esto evalua, utilizando el mecanismo de evaluación de expresiones anteriormente descripto, la variable `this`, las variables locales y los parámetros del método al cual pertenece el punto de ejecución.

La segunda etapa de la construcción del grafo parte del resultado obtenido en la etapa anterior. La clase `JavaSnapshot` contiene un método que devuelve una representación de la información en formato de grafo modelado por la clase `JavaSnapshotGraph`.

5.2. Diseño y Arquitectura de TacoPlug

Como se comentó anteriormente, para el desarrollo de TacoPlug se utilizó la infraestructura provista por Eclipse, Plugin Development Environment (PDE) [27]. Este entorno de desarrollo de plugins provee herramientas para crear, desarrollar, testear, debuggear, construir y deployar plugins para Eclipse. PDE no está incluido en la versión básica de Eclipse sino que debe bajarse como un plugin mismo.

El primer paso en el construcción de TacoPlug fue la creación de un proyecto de Eclipse bajo el template *Plug-in Project* provisto por PDE sobre el que se diseñaron y desarrollaron las distintas vistas en el mismo orden en el que fueron presentadas en la sección 3, con la excepción de la nueva perspectiva que fue desarrollada hacia el final del proyecto.

En PDE, los distintos componentes que uno crea se conocen como *Extensions* y generalmente extienden un *Extension Point*. En las siguientes subsecciones se comentará como fue construido cada extension y qué extension point sirvió de base.

Adicionalmente, deben definirse los plugins de los cuales depende el plugin que se está desarrollando. Generalmente se agregan distintos plugins que contienen los extension points que se van a extender, pero pueden agregarse otros que brinden funcionalidades para la construcción de extensiones pero que no sirvan como base (un ejemplo de esto se verá en 5.2.6)

Finalmente, aunque no forma parte de la arquitectura de TacoPlug, existe una herramienta que resultó muy útil al momento de desarrollar este plugin. Esta herramienta se llama PDE Incubator Spy [31] y permite la introspección de Eclipse en términos de lo que a un desarrollador de plugins le resulta útil. La combinación de teclas **ALT+SHIFT+F1** abre el Spy y presenta al usuario información relevante a donde se haye el foco. En la figura 35 puede verse esta herramienta cuando se la aplica sobre la ventana de problemas de compilación provista por Eclipse.

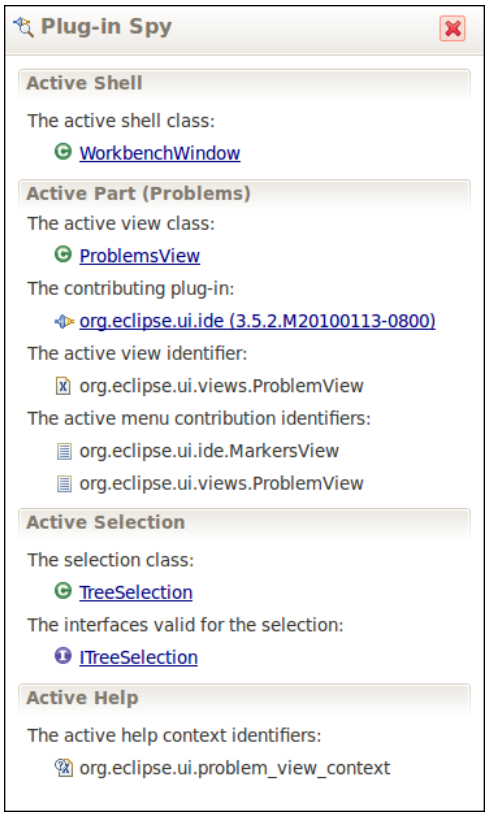

Figura 35: PDE Incubator Spy para la vista de problemas de compilación de Eclipse

5.2.1. TACO Preferences

Esta pantalla, que permite la configuración de TACO, utiliza el punto de extensión `org.eclipse.ui.preferencePages` y está compuesta por 3 componentes que se encuentran en el paquete `tacoplugin.preferences`:

- **PreferenceConstants**: implementado por la clase `PreferenceConstants`, contiene los identificadores de las distintas variables de configuración.
- **PreferenceInitializer**: implementado en la clase `PreferenceInitializer`, contiene la inicialización de las distintas variables de configuración. Esta clase extiende a la clase de nombre `AbstractPreferenceInitializer` provista por PDE.
- **PreferencePage**: implementado en la clase `TacoPreferencePage`, que extiende a la clase `FieldEditorPreferencePage` provista por PDE. Contiene la construcción de la vista propiamente dicha y es donde se define qué componentes son utilizados para la configuración de los distintos parámetros, el tipo de datos permitido para cada uno, etc. Por ejmplo, para parámetros booleanos puede utilizarse un checkbox, para campos donde se requiere que el usuario ingrese información puede definirse si se aceptan valores numéricos o no, etc. En esta clase también se detalla la descripción de cada parámetro para que el usuario entienda qué es lo que configura.

5.2.2. TACO Launcher

Para la construcción de esta pantalla, que permite la ejecución de análisis TACO, se utilizaron diversos extension points según se explica a continuación. Las clases que componen esta extensión están contenidas en el paquete `tacoplugin.launcher`.

En primer lugar, se extendió `org.eclipse.debug.ui.launchConfigurationTabs` para la construcción de una clase que permite definir la pantalla de configuración de parámetros propios del análisis que se va a realizar. El nombre de esta clase es `TacoLaunchConfigurationTab` y desde aquí se definen las distintas partes que compondrán esta vista, los parámetros que deben definirse, cómo se inicializan, etc.

En segundo lugar, toda extensión de `org.eclipse.debug.ui.launchConfigurationTabs` debe ser incluída en un TabGroup para poder ser utilizada. Por este motivo, se extendió `org.eclipse.debug.ui.launchConfigurationTabGroups` y se incluyó la pantalla anterior en este grupo.

Finalmente, debió definirse la propia ejecución. Es decir, qué es lo que se ejecuta una vez que el usuario configuró los parámetros e hizo click en *Run*. Para esto se extendió `org.eclipse.debug.core.launchConfigurationTypes` con la clase `TacoLauncher` que es la que finalmente recupera las preferencias globales, los parámetros recien definidos sobre el análisis y ejecuta TACO. Adicionalmente, luego de la ejecución de TACO, dependiendo del resultado, actualiza las distintas pantallas del plugin.

Por otra parte, al momento de ejecutar un análisis TACO se ejecutan ciertos procesos que calculan automáticamente algunos parámetros de TACO como se mencionó en 3.3.2. Desde el punto de vista implementativo, los procesos que tienen particular interés son los de cálculo de clases relevantes y la modificación del *class loader*.

Para el cálculo de clases relevantes se aprovechó un mecanismo provisto por Eclipse desde la clase `org.eclipse.jdt.internal.corext.callhierarchy.CallHierarchy` y se lo utilizó para desarrollar un análisis que, dado el método que se desea verificar, computa las clases relevantes. De esta forma, se agregan como clases relevantes:

- Clases utilizadas en el método a analizar.
- Clases relevantes de los métodos invocados desde el método a analizar.

Esto implica que los archivos fuente de las clases relevantes estarán disponibles para ser utilizados durante el análisis realizado por TACO.

Por otro lado, y aunque no se trate de un parámetro, es importante mencionar que el conjunto de clases relevantes también deben encontrarse disponibles para el *class loader* de la *JVM* en la que corra TACO. Esto es necesario porque al momento de evaluar expresiones puede ser necesario instanciar objetos pertenecientes a una clase relevante o a la misma clase bajo análisis. Si estas clases no estuviesen disponibles para el *class loader*, se obtendría una excepción al momento de instanciarlas.

Agregar estas clases al *class loader* no es una tarea fácil y requiere un poco de manejo de bajo nivel de Java. Esto se logró mediante el uso de *reflection*, agregando el directorio de *output* del proyecto (es decir, donde se almacenan las clases compiladas) a los directorios visible al *class loader*. De esta forma, cualquier clase compilada en el proyecto analizado se encuentra disponible para ser instanciada.

5.2.3. JML/JFSL Annotation Explorer

En la construcción de esta pantalla se utilizó un mecanismo bien conocido por usuarios de Eclipse: *markers*. Para esto se creó un identificador nuevo de tipos de markers, implementado como variable de la clase enumerada `TacoMarker` del paquete `tacoplugin.markers`.

Adicionalmente se creó una vista, utilizando los extension points `org.eclipse.ui.views` y `org.eclipse.ui.ide.markerSupport`, implementada por la clase `TacoPredicateValidityView` que, extendiendo la vista `MarkerSupportView` provista por PDE, muestra los markers del tipo con el que se la configure.

Este mecanismo, simple pero efectivo, permite al usuario hacer doble click sobre los markers y lograr que el foco del cursor se situe sobre el predicado correspondiente en el código fuente.

5.2.4. Java Error Trace

Esta extensión utiliza como extension point a `org.eclipse.ui.views` en conjunto con algunos componentes provistos por PDE. Las clases que componen esta vista están contenidas en el paquete `tacoplugin.views.javaTrace`, siendo la principal `JavaTraceView`.

Dicha clase utiliza un componente provisto por JFace [32], framework que provee clases y elementos para el desarrollo de interfaces gráficas incluido en PDE, llamado TreeViewer que, dados:

- Un proveedor de contenido
- Un etiquetador
- Un ordenador

dibuja un arbol con dicha información. Estos elementos son provistos por las clases de nombre `JavaTraceViewerContentProvider`, `JavaTraceViewLabelProvider` y `JavaTraceViewerSorter` respectivamente, que extendiendo determinadas clases y definiendo algunos métodos definidos como abstractos, proveen la funcionalidad necesaria para construir el árbol que muestra la traza del contraejemplo encontrado por TACO.

5.2.5. JML/JFSL Evaluator

Esta vista es similar a *Java Error Trace* en el sentido de que también es una extensión de `org.eclipse.ui.views` y que también utiliza componentes de JFace.

Sin embargo, a diferencia de la descripta anteriormente, la extensión *JML/JFSL Evaluator* no utiliza un `TreeViewer`, sino que hace uso de los componentes `TableViewer` y `TextViewer`. El primero de estos componentes tiene como objetivo mostrar el listado de expresiones JML/JFSL que se definieron para ser evaluadas, mientras que el objetivo del segundo es mostrar el resultado de la evaluación de las mismas.

Esta extensión está construida en el paquete `tacoplugin.views.jmlEvaluation` siendo la principal clase `JmlExpressionEvaluationView`.

De la misma forma que el `TreeViewer`, un `TableViewer` requiere de:

- Un proveedor de contenido
- Un etiquetador

que son provistos por las clases de nombre `JmlExpressionEvaluationViewerContentProvider` y `JmlExpressionEvaluationViewerLabelProvider` respectivamente.

5.2.6. Java Memory Graph

Este componente es otro de los construidos sobre la extensión `org.eclipse.ui.views`. Aunque, a diferencia de los dos anteriores, cuenta con la particularidad de que está basado integramente en el plugin Zest [33]. Como se mencionó anteriormente, Zest es un plugin que provee un conjunto de componentes para visualización en Eclipse. Esta librería está desarrollada completamente utilizando SWT/Draw2D.

Para poder utilizar este plugin se debió, además de instalarlo, agregarlo como dependencia al archivo *plugin.xml* y configurar el launcher de Eclipse para que lo cargue cuando se ejecute TacoPlug. En un esquema productivo de TacoPlug, es decir, cuando se lo instale en una versión limpia de Eclipse como cualquier plugin, solamente será necesario instalar Zest como dependencia.

Con Zest disponible para ser utilizado desde TacoPlug, se construyó esta extensión en el paquete `tacoplugin.views.heap` donde la principal clase es `JavaHeapView`. Esta clase utiliza Zest para construir un `GraphViewer` que, dados:

- Un proveedor de contenido

- Un etiquetador

dibuja un grafo con la información y estilo configurado. El proveedor de contenido está dado por `JavaHeapViewerContentProvider` mientras que el etiquetador queda descripto por la clase `JavaHeapViewerLabelProvider`.

5.2.7. TACO Editor

El desarrollo de esta extensión está basado en el extension point *org.eclipse.ui.editors*. El componente se encuentra compuesto por las clases incluidas en el paquete `tacoplugin.editors` siendo la principal clase que lo compone `TacoEditor`.

La clase `TacoEditor` extiende a la clase `MultiPageEditorPart`, provista por PDE, que permite la construcción de editores con múltiples páginas. Se definieron la siguientes variables de instancia de esta clase:

- `CompilationUnitEditor javaEditor`
- `CompilationUnitEditor simpleJmlVisualizer`
- `TextEditor jDynAlloyVisualizer`
- `TextEditor dynAlloyVisualizer`

Como puede imaginarse, cada una de estas variables representa los editores de Java, JML Simplificado, JDynAlloy y DynAlloy respectivamente. Para el editor de Java se utilizó el mismo editor utilizado por Eclipse, lo que automáticamente brinda *syntax highlighting* y todas las funcionalidades provistas por el mismo. Para el visualizador de JML Simplificado también se utilizó este editor pero se le desactivo la posibilidad de modificar el contenido. Los visualizadores de JDynAlloy y DynAlloy son simples editores de texto con la capacidad de edición desactivada (ver 7).

5.2.8. TACO Perspective

Esta extensión fue construida solo con objetivos de usabilidad ya que no agrega funcionalidad a lo ya presentado pero sí lo organiza para comodidad del usuario. Está basada en los extension points `org.eclipse.ui.perspectives` y `org.eclipse.ui.perspectiveExtensions` y su clase constituyente es `TacoPerspective` en el paquete `tacoplugin.perspectives`. En este clase se define que vistas forman parte de la perspectiva y donde se las posiciona.

5.3. Integración TacoPlug-TACO

En primer lugar, para poder interactuar con TACO fue necesario agregar este proyecto (y sus dependencias) al *classpath* del proyecto TacoPlug. A diferencia de los proyectos Java, para este tipo de proyectos no es posible agregar como referencia otros proyectos salvo que sean del mismo tipo, es decir plugins para Eclipse. Es por esto que para agregar TACO al classpath fue necesario exportar el *jar* y guardarlo en la carpeta *lib* del nuevo proyecto. Lo mismo debió hacerse con los proyectos *jdynalloy* y *dynalloy4*.

En la figura 36 se incluye un diagrama de la interacción entre TacoPlug y TACO a partir del cual se explicará la integración entre TACO y TacoPlug.

Como se puede ver en este diagrama, la interacción comienza cuando el usuario inicia un análisis a través de la vista *TacoLauncher* cuya primera tarea es configurar el contexto en el cual se ejecutará el análisis. Para esto recupera las preferencias globales seteadas por el usuario a través de `TacoConfigurationBuilder` y utilizando *TacoPreferences*, calcula el conjunto de clases relevantes por medio del componente `RelevantClassAnalysis` (y las agrega al *class loader*), obtiene el directorio donde se encuentra el código fuente y finalmente ejecuta el análisis propiamente dicho, punto en el cual ya se utiliza TACO exclusivamente.

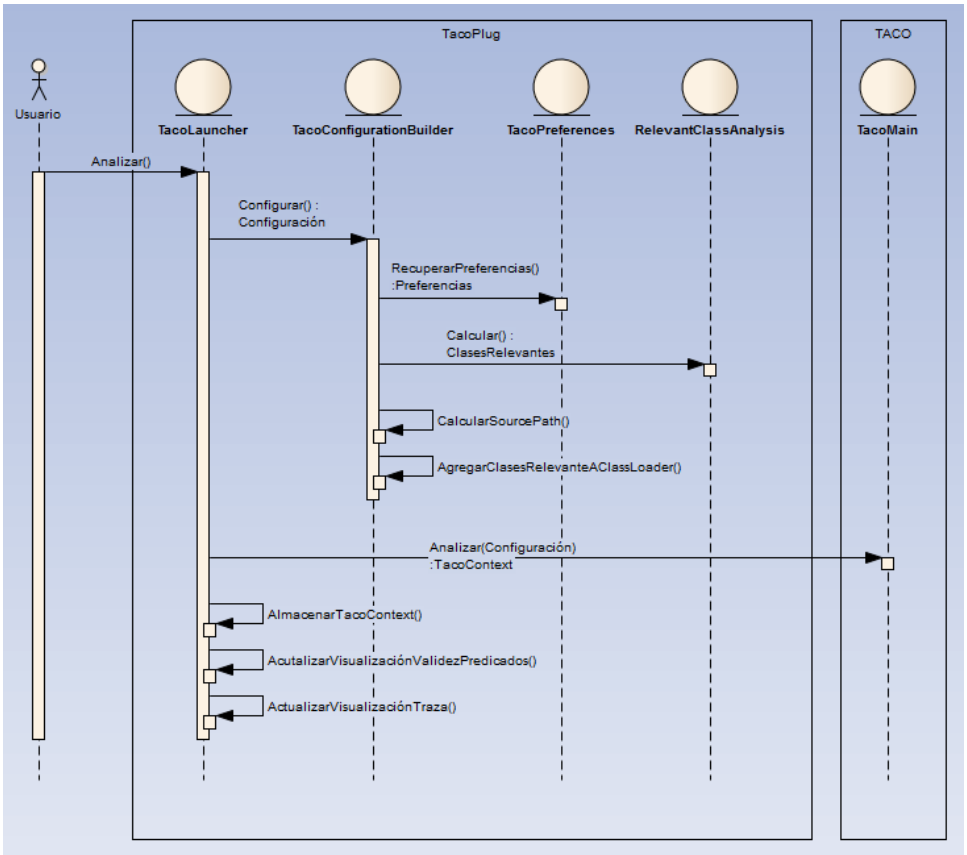

Figura 36: Diagrama de interacción entre TacoPlug y TACO

Una vez concluido el análisis, TACO retorna el contexto de traducción (que contiene el contraejemplo Java), que es almacenado para futuras interacciones del usuario. Asimismo también se actualizan las vistas que permiten la visualización de la validez de los predicados pertenecientes a la VC y actualiza el compoente que permite la visualización de la traza Java.

A partir de este punto, cómo siga la interacción entre TACO y TacoPlug depende del usuario, pero en cualquier caso, el punto de entrada va a ser el contraejemplo Java reportado y almacenado como parte del contexto de traducción obtenido durante el análisis. Este interacción se da, primordialmente, cuando el usuario selecciona un punto de la traza. Es ese momento, por un lado es posible evaluar expresiones JML/JFSL, y por el otro, se dispara automáticamente el mecanismo que computa el estado del heap en el punto seleccionado. A continuación se incluyen diagramas de interacción entre TacoPlug y TACO para estos mecanismos.

En la figura 37 se muestra la interacción entre el usuario, TacoPlug y TACO necesaria para poder evaluar una expresión.

Como se puede apreciar en este diagrama, el primer paso es la selección de un punto de la traza Java. Esta selección se da a través de la vista *Java Error Trace* y contextualiza la evaluación que se realizará en ese punto de la traza. El siguiente paso es, utilizando la visa *JML/JFSL Evaluator*, ingresar una expresión a ser evaluada. Una vez ingresada la expresión, se dispara un mecanismo que, a través del contraejemplo Java presente en la instancia de `TacoContext` obtenida como resultado del análisis realizado, permitirá el parseo, traducción, encarnación, evaluación e instanciación del resultado según el mecanismo descripto en 5.1.2. Una vez completada esta evaluación, se mostrará al usuario el `toString` del resultado.

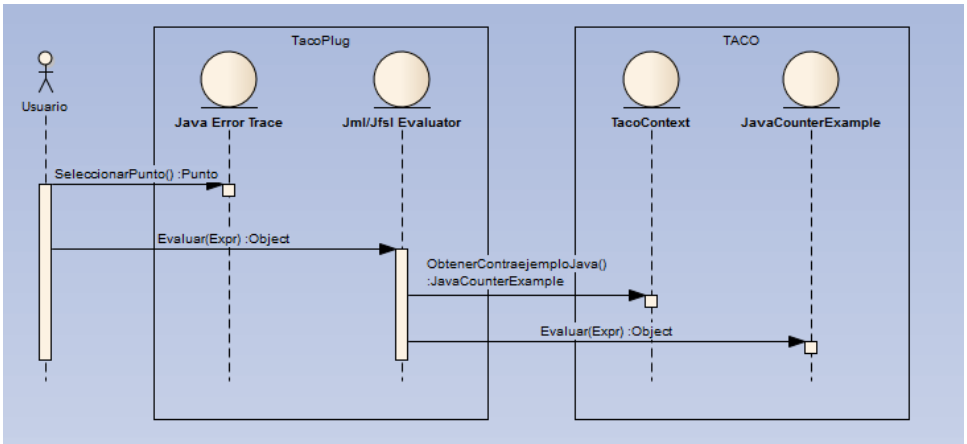

Figura 37: Diagrama de interacción para la evaluación de una expresión

Por otro lado, en la figura 38 se muestra la interacción entre el usuario, TacoPlug y TACO para la construcción del grafo con la visualización global del estado de la memoria.

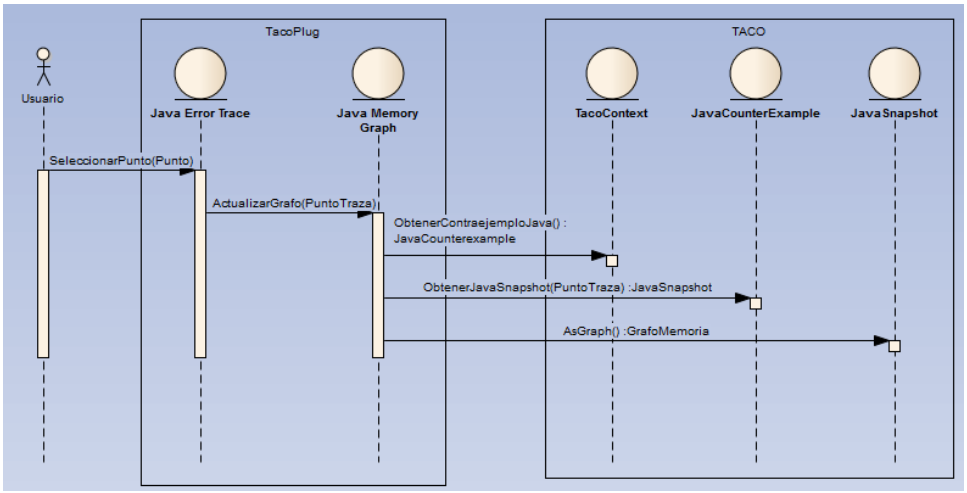

Figura 38: Diagrama de interacción para la construcción del grafo con el estado de la memoria

En este diagrama se muestra como, luego de la selección de un punto de la traza de la misma forma que se hace para la evaluación de expresiones, se construye el grafo respresentativo del estado de la memoria para ese punto según el mecanismo descripto en 5.1.4. A diferencia del mecanismo antes descripto, la construcción del grafo no requiere mayor interacción de parte del usuario, sino que su construcción es disparada automáticamente. La construcción del grafo se dispara a través del contraejemplo Java presente en la instancia de `TacoContext` obtenida como resultado del análisis realizado. Como resultado de esta ejecución, se genera un objeto de la clase `JavaSnapshot` que, utilizando un método llamado `asGraph`, devuelve la información contenida en formato de grafo. Este grafo es luego tomado por *Java Memory Graph* para ser visualizado por el usuario.

6. Conclusiones

Durante la realización del presente trabajo se diseñaron y desarrollaron diversas técnicas que permiten un mejor análisis de un contraejemplo reportado por TACO luego de la verificación de un programa. Asimismo se desarrolló un plugin para Eclipse que permite la integración de TACO a dicha IDE facilitando la utilización de TACO y ampliando su audiencia a un público no experimentado en herramientas de verificación acotada.

Desde el punto de vista de la herramienta TACO se presentó una técnica para reconstruir la traza a nivel Java que resulta en la violación de la especificación reportada. Adicionalmente se exhibió un mecanismo que permite la evaluación de expresiones JML/JFSL en distintos puntos de la traza. Este mecanismo permitió, en primer lugar, el diseño y construcción de un módulo capaz de reconstruir el estado del heap en cualquier punto de la ejecución simbólica. En segundo lugar, posibilitó la evaluación del invariante de clase y la postcondición del método en el estado final de la ejecución, lo cual permite acotar la violación a una sola fórmula en el mejor caso.

Por otro lado, desde el punto de vista del plugin, se introdujeron distintos componentes modelados como vistas de Eclipse que permiten la utilización de TACO de una forma mucho más amigable. En primer lugar se integró la posibilidad de ejecutar análisis desde Eclipse permitiendo la configuración de casi todos los parámetros existentes en TACO. Los parámetros que quedaron por fuera de esta definición se computan automáticamente lo que facilita su uso.

Asimismo, se diseñó una nueva perspectiva para Eclipse que incluye las distintas vistas que permiten el debugging y localización de la falla detectada por TACO. Entre las vistas desarrolladas se incluye la posibilidad de recorrer la traza reportada, visualizar la fórmula violada, evaluar expresiones y visualizar el estado del heap en distintos puntos de la traza.

Finalmente se presentó un caso de estudio complejo donde se utilizan estructuras de datos recursivas (binomial heap) y diversas construcciones del lenguaje Java (`if`, `while`). Se mostró cómo se utilizó TACO para detectar una falla y cómo el plugin TacoPlug facilitó su aprovechamiento para detectar dicha falla.

Considerando las técnicas y mecanismos aquí presentados, en conjunto con el caso de estudio realizado y cómo fueron utilizados los componentes que los implementan para localizar la falla presente en el código, se puede concluir que la utilización de la herramienta TACO se ha vuelto mucho más amigable y una mayor comprensión del contraejemplo reportado por la misma es posible. La extensión TacoPlug probó ser útil a la hora de localizar una falla reportada. Por estos motivos, la existencia de herramientas como TacoPlug resulta indispensable para dejar la verificación de programas al alcance de un amplio espectro de usuarios.

El código fuente de TACO como de TacoPlug está disponible públicamente a través del sitio `http://www.dc.uba.ar/taco`.

7. Trabajo futuro

En lo que concierne a perspectivas futuras, existen diversos enfoques que pueden seguirse, tanto a nivel social como técnico.

Desde un punto de vista social, se plantea la posibilidad de realizar experimentos de verificación de la usabilidad de la herramienta con un rango amplio de programadores, los cuales podrían incluir estudiantes. Esto permitiría asentar la usabilidad de los componentes desarrollados e identificar otros que podrían resultar útiles para analizar errores detectados por TACO y localizar la falla en el código fuente.

Desde un punto de vista técnico, las posibilidades son amplias incluyendo mejoras de usabilidad e integración con funcionalidades de TACO que quedaron fuera del alcance de este trabajo.
Dentro de las mejoras de usabilidad propuestas se encuentran:

1. **JDynAlloy y DynAlloy syntax highlighting**. Como se presentó en la sección 2, se desarrolló un editor que permite la visualización de las transformaciones JDynAlloy y DynAlloy del programa original. Para mejorar la lectura de estas transformaciones resultaría útil la implementación de syntax highlighting en estos editores.

2. **Administrador de análisis**. Actualmente existe la posibilidad de ejecutar de a un análisis a la vez. Esto puede resultar incómodo si se desea analizar un método con distintos parámetros o si al momento de analizar una traza se desea realizar un análisis sobre algún método interno. Estos problemas de usabilidad podrían enfrentarse desarrollando un administrador de análisis TACO donde se guarden los distintos resultados y se contextualice el uso del plugin de acuerdo al análisis seleccionado.

3. **Visualización del heap**. Existen numerosos trabajos sobre visualización del heap [34] cuya integración al plugin podría beneficiar el análisis que debe realizar el usuario cuando utiliza estructuras de datos recursivas.

4. **Mejoras de performance**. Algunos de los algoritmos relacionados con la recuperación de la información tienen una performance un poco por debajo de lo esperado. Esto obstruye la usabilidad del plugin y debe mejorarse.

5. ***Diff* entre versiones del grafo de memoria**. Cuando se intenta localizar una falla en un programa que utiliza estructuras de datos recursivas, la vista *Java Memory Graph* resulta un elemento clave. Sin embargo, en muchos casos resulta frustrante tener que analizar los cambios en el heap entre dos pasos consecutivos de la traza generada por TACO. Para estos casos resultaría muy útil la incorporación de un componente que permita visualizar facilmente las diferencias entre dos grafos de memoria.

6. **Evaluación parcial de fórmulas**. Luego de la detección de una falla, el usuario intenta identificar el origen de la violación a la especificación utilizando la vista *JML/JFSL Annotation Explorer*. Este componente indica qué formula falló pero no permite el análisis de las subformulas que pudieran componerla. Sería deseable facilitar al usuario la capacidad de analizar esta subformulas por separado pudiendo achicar el espacio de análisis para la localización de la falla.

Dentro de las mejoras que incluyen la integración con funcionalidades de TACO no cubiertas por el presente trabajo se incluye:

- **Integración con generación de casos de test**. TACO presenta la posibilidad de generar distintos casos de test automáticos utilizando el framework de test jUnit. La integración de esta funcionalidad a TacoPlug podría facilitar el debugging de un método utilizando un acercamiento dinámico al problema.

- **Slice del programa original**. Se está trabajando para lograr proveer desde TACO un *slice* del programa original, dado por el contraejemplo encontrado, que descarte todas las instrucciones que no están directamente relacionadas con la violación de la especificación encontrada.

Finalmente cabe aclarar que cualquier trabajo sobre TACO que no ataque el problema de la eficiencia temporal de la herramienta, sino que brinde una nueva funcionalidad, puede constituir un potencial trabajo futuro sobre el plugin aquí presentado que incluya la integración de la nueva funcionalidad.

En este sentido, una posibilidad a considerar sería la construcción de un plugin orientado a desarrolladores de la herramienta. Entre las caracterísiticas con las que contaría esta versión del plugin se podrían incluir:

- Edición de los archivos JDynAlloy, DynAlloy, Alloy.
- Visualización de los contextos de traducción de las distintas etapas.
- Visualización del repositorio de cotas que calcula TACO.
- Visualización de TACO Flow, framework de dataflow integrado a TACO.
- Integración con Alloy Analyzer y sus herramientas de análisis del contrajemeplo.

A. Código fuente BinomialHeap

```
1  //
2  // Copyright (C) 2006 United States Government as represented by the
3  // Administrator of the National Aeronautics and Space Administration
4  // (NASA).  All Rights Reserved.
5  //
6  // This software is distributed under the NASA Open Source Agreement
7  // (NOSA), version 1.3.  The NOSA has been approved by the Open Source
8  // Initiative.  See the file NOSA-1.3-JPF at the top of the distribution
9  // directory tree for the complete NOSA document.
10 //
11 // THE SUBJECT SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY WARRANTY OF ANY
12 // KIND, EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING, BUT NOT
13 // LIMITED TO, ANY WARRANTY THAT THE SUBJECT SOFTWARE WILL CONFORM TO
14 // SPECIFICATIONS, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
15 // A PARTICULAR PURPOSE, OR FREEDOM FROM INFRINGEMENT, ANY WARRANTY THAT
16 // THE SUBJECT SOFTWARE WILL BE ERROR FREE, OR ANY WARRANTY THAT
17 // DOCUMENTATION, IF PROVIDED, WILL CONFORM TO THE SUBJECT SOFTWARE.
18 //
19 package ar.edu.taco;
20
21 /**
22  * @SpecField nodes: set BinomialHeapNode from this.Nodes, this.nodes.child, this.nodes.s
23  *                   | this.nodes = this.Nodes.*(child @+ sibling) @- null ;
24  */
25 /**
26  * @Invariant ( all n: BinomialHeapNode | ( n in this.Nodes.*(sibling @+ child) @- null =>
27  *               ( ( n.parent!=null => n.key >= n.parent.key ) &&
28  *               ( n.child!=null => n !in n.child.*(sibling @+ child) @- null ) &&
29  *               ( n.sibling!=null => n !in n.sibling.*(sibling @+ child) @- null ) &&
30  *               ( ( n.child !=null && n.sibling!=null ) =>
31  *                 ( no m: BinomialHeapNode | ( m in n.child.*(child @+ sibling) @- null &
32  *                                              m in n.sibling.*(child @+ sibling) @- nul
33  *               ( n.degree >= 0 ) &&
34  *               ( n.child=null => n.degree = 0 ) &&
35  *               ( n.child!=null =>n.degree=#(n.child.*sibling @- null) ) &&
36  *               ( #( ( n.child @+ n.child.child.*(child @+ sibling) ) @- null ) = #( ( n
37  *               ( n.child!=null => ( all m: BinomialHeapNode | ( m in n.child.*sibling@-
38  *               ( ( n.sibling!=null && n.parent!=null ) => ( n.degree > n.sibling.degree
39  *
40  *            ( this.size = #(this.Nodes.*(sibling @+ child) @- null) ) &&
41  *            ( all n: BinomialHeapNode | n in this.Nodes.*sibling @- null =>
42  *               ( ( n.sibling!=null => n.degree < n.sibling.degree ) &&
43  *               ( n.parent=null ) )) ;
44  */
45 public class BinomialHeap{
46
47   private/*@ nullable @*/BinomialHeapNode Nodes;
48
49   private int size;
50
51   public BinomialHeap() {
52     Nodes = null;
53     size = 0;
54   }
55
56   /**
```

```
57     * @Modifies_Everything
58     *
59     * @Requires some this.nodes ;
60     * @Ensures ( some x: BinomialHeapNode | x in this.nodes && x.key == return
61     *           ) && ( all y : BinomialHeapNode | ( y in this.nodes && y!=return
62     *           ) => return <= y.key ) ;
63     */
64    public int findMinimum() {
65      return Nodes.findMinNode().key;
66    }
67
68    private void merge(/* @ nullable @ */BinomialHeapNode binHeap) {
69      BinomialHeapNode temp1 = Nodes, temp2 = binHeap;
70      while ((temp1 != null) && (temp2 != null)) {
71        if (temp1.degree == temp2.degree) {
72          BinomialHeapNode tmp = temp2;
73          temp2 = temp2.sibling;
74          tmp.sibling = temp1.sibling;
75          temp1.sibling = tmp;
76          temp1 = tmp.sibling;
77        } else {
78          if (temp1.degree < temp2.degree) {
79            if ((temp1.sibling == null) || (temp1.sibling.degree > temp2.degree)) {
80              BinomialHeapNode tmp = temp2;
81              temp2 = temp2.sibling;
82              tmp.sibling = temp1.sibling;
83              temp1.sibling = tmp;
84              temp1 = tmp.sibling;
85            } else {
86              temp1 = temp1.sibling;
87            }
88          } else {
89            BinomialHeapNode tmp = temp1;
90            temp1 = temp2;
91            temp2 = temp2.sibling;
92            temp1.sibling = tmp;
93            if (tmp == Nodes) {
94              Nodes = temp1;
95            }
96          }
97        }
98      }
99
100     if (temp1 == null) {
101       temp1 = Nodes;
102       while (temp1.sibling != null) {
103         temp1 = temp1.sibling;
104       }
105       temp1.sibling = temp2;
106     }
107
108   }
109
110   private void unionNodes(/* @ nullable @ */BinomialHeapNode binHeap) {
111     merge(binHeap);
112
113     BinomialHeapNode prevTemp = null, temp = Nodes, nextTemp = Nodes.sibling;
114
```

```
115       while (nextTemp != null) {
116         if ((temp.degree != nextTemp.degree) ||
117           (
118             (nextTemp.sibling != null) &&
119             (nextTemp.sibling.degree == temp.degree)
120           )
121         ) {
122           prevTemp = temp;
123           temp = nextTemp;
124         } else {
125           if (temp.key <= nextTemp.key) {
126             temp.sibling = nextTemp.sibling;
127             nextTemp.parent = temp;
128             nextTemp.sibling = temp.child;
129             temp.child = nextTemp;
130             temp.degree++;
131           } else {
132             if (prevTemp == null) {
133               Nodes = nextTemp;
134             } else {
135               prevTemp.sibling = nextTemp;
136             }
137             temp.parent = nextTemp;
138             temp.sibling = nextTemp.child;
139             nextTemp.child = temp;
140             nextTemp.degree++;
141             temp = nextTemp;
142           }
143         }
144
145         nextTemp = temp.sibling;
146       }
147     }
148
149     /**
150      * @Modifies_Everything
151      *
152      * @Ensures some n: BinomialHeapNode | ( n !in @old(this.nodes) &&
153      *           this.nodes = @old(this.nodes) @+ n && n.key = value ) ;
154      */
155     public void insert(int value) {
156       if (value > 0) {
157         BinomialHeapNode temp = new BinomialHeapNode(value);
158         if (Nodes == null) {
159           Nodes = temp;
160           size = 1;
161         } else {
162           unionNodes(temp);
163           size++;
164         }
165       }
166     }
167
168     /**
169      * @Modifies_Everything
170      *
171      * @Ensures ( @old(this).@old(Nodes)==null => (
172      *                          this.Nodes==null &&
```

```
173     *                           return==null
174     *                           )
175     *       )
176     *        && ( @old(this).@old(Nodes)!=null => (
177     *                  (return in @old(this.nodes)) &&
178     *                  ( all y : BinomialHeapNode | (
179     *                  y in @old(this.nodes) => y.key >= return.key
180     *                  ) ) &&
181     *                  (this.nodes=@old(this.nodes) @- return ) &&
182     *                  (this.nodes.key @+ return.key = @old(this.nodes.key) )
183     *         )
184     *       );
185     */
186    public/* @ nullable @ */BinomialHeapNode extractMin() {
187      if (Nodes == null)
188        return null;
189
190      BinomialHeapNode temp = Nodes, prevTemp = null;
191      BinomialHeapNode minNode = null;
192
193      minNode = Nodes.findMinNode();
194      while (temp.key != minNode.key) {
195        prevTemp = temp;
196        temp = temp.sibling;
197      }
198
199      if (prevTemp == null) {
200        Nodes = temp.sibling;
201      } else {
202        prevTemp.sibling = temp.sibling;
203      }
204      temp = temp.child;
205      BinomialHeapNode fakeNode = temp;
206      while (temp != null) {
207        temp.parent = null;
208        temp = temp.sibling;
209      }
210
211      if ((Nodes == null) && (fakeNode == null)) {
212        size = 0;
213      } else {
214        if ((Nodes == null) && (fakeNode != null)) {
215          Nodes = fakeNode.reverse(null);
216          size--;
217        } else {
218          if ((Nodes != null) && (fakeNode == null)) {
219            size--;
220          } else {
221            unionNodes(fakeNode.reverse(null));
222            size--;
223          }
224        }
225      }
226
227      return minNode;
228    }
229
230    public void decreaseKeyValue(int old_value, int new_value) {
```

```
231     BinomialHeapNode temp = Nodes.findANodeWithKey(old_value);
232     decreaseKeyNode(temp, new_value);
233   }
234
235   /**
236    *
237    * @Modifies_Everything
238    *
239    * @Requires node in this.nodes && node.key >= new_value ;
240    *
241    * @Ensures (some other: BinomialHeapNode | other in this.nodes &&
242    *           other!=node && @old(other.key)=@old(node.key)) ? this.nodes.key
243    *           = @old(this.nodes.key) @+ new_value : this.nodes.key =
244    * @old(this.nodes.key) @- @old(node.key) @+ new_value ;
245    */
246   public void decreaseKeyNode(final BinomialHeapNode node, final int new_value) {
247     if (node == null)
248       return;
249
250     BinomialHeapNode y = node;
251     y.key = new_value;
252     BinomialHeapNode z = node.parent;
253
254     while ((z != null) && (node.key < z.key)) {
255       int z_key = y.key;
256       y.key = z.key;
257       z.key = z_key;
258
259       y = z;
260       z = z.parent;
261     }
262   }
263
264   public void delete(int value) {
265     if ((Nodes != null) && (Nodes.findANodeWithKey(value) != null)) {
266       decreaseKeyValue(value, findMinimum() - 1);
267       extractMin();
268     }
269   }
270 }
```

```
1  package ar.edu.taco;
2
3  public class BinomialHeapNode{
4    int key;
5
6    int degree;
7
8    /*@ nullable @*/BinomialHeapNode parent;
9
10   /*@ nullable @*/BinomialHeapNode sibling;
11
12   /*@ nullable @*/BinomialHeapNode child;
13
14   public BinomialHeapNode(int k) {
15     key = k;
16     degree = 0;
17     parent = null;
```

```
18      sibling = null;
19      child = null;
20    }
21
22    public int getKey() {
23      return key;
24    }
25
26    private void setKey(int value) {
27      key = value;
28    }
29
30    public int getDegree() {
31      return degree;
32    }
33
34    private void setDegree(int deg) {
35      degree = deg;
36    }
37
38    public BinomialHeapNode getParent() {
39      return parent;
40    }
41
42    private void setParent(BinomialHeapNode par) {
43      parent = par;
44    }
45
46    public BinomialHeapNode getSibling() {
47      return sibling;
48    }
49
50    private void setSibling(BinomialHeapNode nextBr) {
51      sibling = nextBr;
52    }
53
54    public BinomialHeapNode getChild() {
55      return child;
56    }
57
58    private void setChild(BinomialHeapNode firstCh) {
59      child = firstCh;
60    }
61
62    public int getSize() {
63      return (
64          1 +
65          ((child == null) ? 0 : child.getSize()) +
66          ((sibling == null) ? 0 : sibling.getSize())
67      );
68    }
69
70    BinomialHeapNode reverse(BinomialHeapNode sibl) {
71      BinomialHeapNode ret;
72      if (sibling != null)
73        ret = sibling.reverse_0(this);
74      else
75        ret = this;
```

```
76      sibling = sibl;
77      return ret;
78    }
79
80    BinomialHeapNode reverse_0(BinomialHeapNode sibl) {
81      BinomialHeapNode ret;
82      if (sibling != null)
83        ret = sibling.reverse_1(this);
84      else
85        ret = this;
86      sibling = sibl;
87      return ret;
88    }
89
90    BinomialHeapNode reverse_1(BinomialHeapNode sibl) {
91      BinomialHeapNode ret;
92      if (sibling != null)
93        ret = sibling.reverse_2(this);
94      else
95        ret = this;
96      sibling = sibl;
97      return ret;
98    }
99
100   BinomialHeapNode reverse_2(BinomialHeapNode sibl) {
101     BinomialHeapNode ret;
102     if (sibling != null)
103       ret = sibling.reverse_3(this);
104     else
105       ret = this;
106     sibling = sibl;
107     return ret;
108   }
109
110   BinomialHeapNode reverse_3(BinomialHeapNode sibl) {
111     BinomialHeapNode ret;
112     if (sibling != null)
113       ret = sibling.reverse_4(this);
114     else
115       ret = this;
116     sibling = sibl;
117     return ret;
118   }
119
120   BinomialHeapNode reverse_4(BinomialHeapNode sibl) {
121     BinomialHeapNode ret;
122     if (sibling != null)
123       ret = sibling.reverse_5(this);
124     else
125       ret = this;
126     sibling = sibl;
127     return ret;
128   }
129
130   BinomialHeapNode reverse_5(BinomialHeapNode sibl) {
131     BinomialHeapNode ret;
132     if (sibling != null)
133       ret = sibling.reverse_6(this);
```

```
134       else
135         ret = this;
136       sibling = sibl;
137       return ret;
138     }
139
140     BinomialHeapNode reverse_6(BinomialHeapNode sibl) {
141       BinomialHeapNode ret;
142       if (sibling != null)
143         ret = sibling.reverse_7(this);
144       else
145         ret = this;
146       sibling = sibl;
147       return ret;
148     }
149
150     BinomialHeapNode reverse_7(BinomialHeapNode sibl) {
151       BinomialHeapNode ret;
152       if (sibling != null)
153         ret = sibling.reverse_8(this);
154       else
155         ret = this;
156       sibling = sibl;
157       return ret;
158     }
159
160     BinomialHeapNode reverse_8(BinomialHeapNode sibl) {
161       BinomialHeapNode ret;
162       if (sibling != null)
163         ret = sibling.reverse_9(this);
164       else
165         ret = this;
166       sibling = sibl;
167       return ret;
168     }
169
170     BinomialHeapNode reverse_9(BinomialHeapNode sibl) {
171       BinomialHeapNode ret;
172       if (sibling != null)
173         throw new RuntimeException();
174       else
175         ret = this;
176       sibling = sibl;
177       return ret;
178     }
179
180     BinomialHeapNode findMinNode() {
181       BinomialHeapNode x = this, y = this;
182       int min = x.key;
183
184       while (x != null) {
185         if (x.key < min) {
186           y = x;
187           min = x.key;
188         }
189         x = x.sibling;
190       }
191
```

```
192     return y;
193   }
194
195   BinomialHeapNode findANodeWithKey(int value) {
196     BinomialHeapNode temp = this, node = null;
197     while (temp != null) {
198       if (temp.key == value) {
199         node = temp;
200         return node;
201       }
202       if (temp.child == null)
203         temp = temp.sibling;
204       else {
205         node = temp.child.findANodeWithKey(value);
206         if (node == null)
207           temp = temp.sibling;
208         else
209           return node;
210       }
211     }
212
213     return node;
214   }
215 }
```

Referencias

[1] "Daikon." [Online]. Available: http://groups.csail.mit.edu/pag/daikon/

[2] "Valgrind." [Online]. Available: http://valgrind.org/

[3] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.

[4] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994.

[5] J. Harrison, "Formal verification in industry." Intel Corporation, 1999.

[6] J. Galeotti, N. Rosner, C. Lopez Pombo, and M. Frias, "Analysis of Invariants for Efficient Bounded Verification," in *International Symposium on Software Testing and Analysis*, 2010.

[7] D. Dobniewski, G. Gasser Noblia, and J. Galeotti, "Verificación de un sistema de votación usando SAT-Solving," *Master's thesis, Universidad de Buenos Aires*, 2010.

[8] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, "F-Soft: Software Verification Platform." *In CAV'05, pp. 301–306*, 2005.

[9] E. Clarke, D. Kroening, and L. F., "A Tool for Checking ANSI-C Programs," *In proceedings of TACAS 2004. LNCS (2988) pp. 168–176.*, 2004.

[10] D. J., V. M., and T. F., "Finding Bugs Efficiently with a SAT Solver," *ESEC/FSE'07, pp. 195–204, ACM Press*, 2007.

[11] G. Dennis, "A Relational Framework for Bounded Program Verification," *MIT PhD Thesis*, 2009.

[12] "JForge Eclipse Plug-in." [Online]. Available: http://sdg.csail.mit.edu/forge/plugin.html

[13] M. P. and R. J., "Using Debuggers to Understand Failed Verification Attempts," *In Formal Methods (FM) 2011, pp. 73–87*, 2011.

[14] C. Le Goues, M. Leino, and M. Moskal, "The Boogie Verification Debugger," *SEFM*, 2011.

[15] M. Frías, J. Galeotti, L. P. C., and N. Aguirre, "Dynalloy: upgrading alloy with actions," *ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 442-451*, 2005.

[16] "Eiffel Software." [Online]. Available: http://www.eiffel.com/

[17] "Larch." [Online]. Available: http://www.eecs.ucf.edu/~leavens/larch-faq.html

[18] "Refinement Calculus Tutorial." [Online]. Available: http://users.ecs.soton.ac.uk/mjb/refcalc-tut/home.html

[19] "Java Modeling Language." [Online]. Available: http://www.eecs.ucf.edu/~leavens/JML/

[20] "Relational Formula Method Research Group." [Online]. Available: http://www.dc.uba.ar/inv/grupos/rfm_folder

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

[22] "Alloy." [Online]. Available: http://alloy.mit.edu/alloy/

[23] J. Galeotti, "Verificación de Software usando Alloy," *PHD's thesis, Universidad de Buenos Aires*, 2010.

[24] "Sun Microsystems. JDC techtips." 2000. [Online]. Available: http://java.sun.com/developer/TechTips/2000/tt0314.html

[25] "Eclipse." [Online]. Available: http://www.eclipse.org/

[26] P. Bendersky, "Hacia un entorno integrado para la verificación de contratos utilizando SAT Solvers," *Master's thesis, Universidad de Buenos Aires*, 2010.

[27] "Plugin Development Environment." [Online]. Available: http://www.eclipse.org/pde/

[28] "jUnit." [Online]. Available: http://www.junit.org/

[29] T. Cormen, L. C., R. R., and S. C., "Introduction to Algorithms (3. ed.)," *MIT Press*, 2009.

[30] V. W., P. C. S., and P. R., "Test Input Generation for Java Containers using State Matching," *ISSTA 2006, pp. 37–48*, 2006.

[31] "Eclipse Incubator Spy." [Online]. Available: http://www.eclipse.org/pde/incubator/spy/

[32] "JFace." [Online]. Available: http://wiki.eclipse.org/JFace

[33] "Zest." [Online]. Available: http://www.eclipse.org/gef/zest/

[34] T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," *Software Visualization pp. 191–204*, 2001.