Análisis Modular y Recuperación de Contraejemplos en TACO

Raúl Alborodo[1], Nicolás Ricci[1] Juan P. Galeotti[2,3], and Nazareno Aguirre[1,3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
{rnnalborodo,nricci,naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina
jgaleotti@dc.uba.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Resumen. TACO es una herramienta para realizar verificación formal de programas, que permite detectar bugs en los mismos. Esta traduce un programa escrito en lenguaje Java y su especificación en JML a la notación DynAlloy, para luego analizar la especificación obtenida mediante SAT solving, via una traducción adicional al lenguaje Alloy. En este trabajo presentamos dos mejoras significativas a esta herramienta y técnica de análisis. En primer lugar, proveemos un mecanismo de análisis modular de código en presencia de invocación de rutinas. En segundo lugar, automatizamos la construcción de unit tests en Java, que reproducen los bugs detectados por el análisis.
Las mejoras presentadas contribuyen al análisis en dos dimensiones: el análisis modular contribuye a la escabilidad de la técnica de análisis subyacente a TACO, mientras que la recuperación de contraejemplos facilita el uso de la herramienta, ocultando adecuadamente los detalles del método formal subyacente en su aplicación.

1 Introducción

Uno de los problemas más importantes asociados con la construcción de software es la corrección del mismo, es decir, en qué medida el software construido satisface los requisitos (funcionales o no) establecidos durante las etapas tempranas del desarrollo. En busca de proveer garantías del correcto funcionamiento del software, han surgido una variedad de técnicas y metodologías de desarrollo con sólidas bases matemáticas y lógicas conocidas como *métodos formales* [10]. Debido a su naturaleza, la aplicación de métodos formales requiere gran experiencia y conocimientos, sobre todo en lo concerniente a matemáticas y lógica, por lo cual su aplicación resulta costosa en la práctica. Esto ha provocado que su principal aplicación se limite a sistemas críticos [11], es decir, sistemas de software/hardware cuyo mal funcionamiento o falla puede causar daños de magnitud como la pérdida de vidas humanas, aunque los beneficios que sus técnicas proveen son claramente relevantes a todo tipo de software.

Últimamente se ha reconocido que contar con herramientas de análisis automático o semi-automático es un elemento de gran importancia para contribuir a la adopción de métodos formales. Uno de los problemas que se observan en el uso de herramientas de este tipo es que los desarrolladores deben expresar sus programas o modelos en la notación del método formal elegido, además de conocer detalles sutiles de la semántica de este lenguaje. Cuando el método formal funciona como una especie de *backend* de una herramienta de análisis de más alto nivel, y la aplicación del mismo es automática, algo que resulta practicable (muchas veces con dificultad) y de fundamental importancia es ocultar los detalles del método formal subyacente. Por otra parte, las técnicas de análisis automático de software o modelos de software sufren generalmente problemas de escalabilidad, es decir, su aplicación o eficacia decrece significativamente a medida que los tamaños y complejidades de los modelos a analizar crece. Tal es el caso de *model checking* y el problema de la explosión combinatoria de estados [1], o la influencia crítica del tamaño de las especificaciones en el análisis basado en satisfactibilidad booleana.

En este trabajo se proponen métodos para aliviar los problemas antes descriptos en el contexto de TACO [6], una herramienta de análisis de código Java basada en *SAT Solving* como mecanismo de análisis. Esta herramienta realiza verificaciones formales de programas anotados JML [9], permitiendo la detección de bugs en los mismos. TACO toma un programa escrito en lenguaje Java y su especificación en JML, y los traduce al lenguaje DynAlloy [3], para luego analizar la especificación obtenida mediante SAT solving, a través de una traducción adicional al lenguaje Alloy y del uso de *Alloy Analyzer* [7, 8].

La primera extensión que presentaremos surge como resultado de observar que, para poder utilizar TACO el desarrollador necesita conocer Alloy, además de Java y JML (los lenguajes utilizados para la entrada de la herramienta); esto se debe a que los bugs encontrados por la herramienta son presentados como modelos Alloy, y es necesario estudiar los mismos para poder depurar el programa. Nuestra extensión apunta a resolver este problema mediante la traducción automática de modelos Alloy a unit tests Java equivalentes, que reproducen el o los bugs detectados por la herramienta. La segunda extensión contribuye a la escalabilidad el análisis usando TACO, incorporando una forma efectiva de modularizar el análisis. Esta modularización aprovecha el hecho de que los programas provistos a la herramienta están equipados con *contratos* (especificaciones formales de los mismos), para tratar las invocaciones a rutinas, sumamente comunes en código orientado a objetos [2], como *supuestos* de la verificación. Más precisamente, en lugar de tratar las invocaciones a rutinas como "inlining" de código, se realiza la verificación *suponiendo que la rutina invocada es correcta*, es decir, se utiliza la especificación de la misma en lugar de su código. Esto da como resultado una forma efectiva de modularización del análisis (se separa el análisis del código invocante de aquel del código invocado), a costa de una pérdida en la precisión del análisis: es posible que la herramienta detecte errores espúrios (no reales) producto de especificaciones correctas pero no lo suficientemente precisas.

2 TACO: Translation of Annotated COde

TACO [6] es una herramienta que permite verificar formalmente programas Java anotados con contratos JML. En esencia, TACO toma un programa Java con su correspondiente especificación descripta en términos de pre- y post-condiciones e invariantes de clase, y busca exhaustivamente alguna ejecución del programa que comience en un estado que satisfaga la precondición, y termine en un estado que viole la postcondición (una idea explotada previamente en [5]). La exhaustividad de la herramienta es *acotada*: el usuario debe proveer cotas que la herramienta utilizará e interpretará como el número máximo de objetos de cada clase que se permite en cualquier ejecución, y el número máximo de iteraciones de los ciclos o invocaciones recursivas. Así, si se provee un programa P con su respectiva especificación, y cotas k_1 y k_2 para objetos e iteraciones respetivamente, entonces TACO buscará exhaustivamente si existe alguna ejecución que involucre hasta k_1 objetos de cada clase y con a lo sumo k_2 ejecuciones de ciclos o invocaciones recursivas, que viole la especificación. Si la herramienta no encuentra ninguna violación, se sabrá entonces que el programa es correcto, en los casos acotados analizados, aunque es posible que existan violaciones "más grandes" que muestren que el programa es incorrecto con más objetos o más iteraciones.

Desde un punto de vista técnico, el programa a ser analizado debe atravesar una serie de etapas para su análisis, en las cuales se toman las cotas provistas, la especificación y el texto del programa para traducirlos a DynAlloy [4], un lenguaje lógico-relacional de especificaciones de sistemas con elementos de lógica relacional (la lógica subyacente a Alloy) [8] y lógica dinámica. Las especificaciones DynAlloy son reducidas mediante cálculo de precondición más débil, más precisamente un tipo de *weakest liberal precondition*, a especificaciones Alloy, que son evaluadas utilizando Alloy Analyzer [4]. El análisis en Alloy se basa en la reducción a fórmulas proposicionales, y la utilización de un SAT-solver para verificar su satisfactibilidad [7, 8].

Como resultado del análisis, Alloy puede concluir que la especificación es insatisfactible, lo cual corresponde a no haber detectado *bugs* (dentro de las cotas provistas) en el programa original, o encontrar un contraejemplo Alloy, que es un modelo en lógica relacional [7]. Obviamente, este modelo representa una ejecución del programa que viola la especificación, pero para poder comprenderla y así depurar el programa es necesario entender cómo se representan el programa y su especificación en Alloy, así como tener experiencia en el uso de Alloy y la comprensión de sus modelos.

A modo de ejemplo, consideremos la porción de código que se muestra en la Figura 1. Esta es un fragmento de código con su respectiva especificación, que implementa la operación de `push` en una pila sobre arreglos. Nótese que la especificación indica propiedades de los atributos de la clase, y para el caso particular de la rutina `pushBuggy` se da una especificación de pre y postcondición: para invocar la rutina `size` debe ser menor que `elems.length-1`, y luego de ejecutar la rutina se debe incrementar `size` en uno, el elemento almacenado en la posición `size-1` es el insertado, y el resto del arreglo no se modifica. Si

```
public class BoundedStack {
    private /*@ spec_public nullable @*/ Object[] elems;
    private /*@ spec_public @*/int size = 0;

    //@ invariant 0 <= size;
    //@ invariant elems != null && (\forall int i; size <= i && i <
        < elems.length; elems[i] == null);

    /*@ requires size < elems.length-1;
      @
      @ assignable elems[size], size;
      @
      @ ensures size == \old(size + 1);
      @ ensures elems[size-1] == e;
      @ ensures (\forall int i; 0 <= i && i < size-1; elems[i] ==
          \old(elems[i]));
      @*/
    public void pushBuggy(Object e) {
        elems[size] = x;
        size++;
    }
}
```

Fig. 1. Implementación de pilas sobre arreglos, con operación `push` errónea, y su especificación JML.

analizamos esta rutina usando TACO, se encontrará un *bug*, que la herramienta reportará con el modelo Alloy que se muestra en la Figura 2.

Es importante mencionar que TACO realiza lo que se denomina *análisis de programa completo*: las invocaciones a rutinas dentro del programa a verificar se reemplazan por el código de las rutinas correspondientes. Esto, sumado a que el análisis basado en SAT tiene una complejidad exponencial en el tamaño de la entrada, hace que para programas grandes, por ejemplo porque involucran un número grande de invocaciones, el análisis sea impracticable.

3 Recuperación de Contraejemplos

Describimos aquí la primera de las extensiones a TACO. Esta es la encargada de proveer un mecanismo de *retorno*, que muestre al usuario/desarrollador la razón que hace que su programa falle, es decir el resultado del análisis, pero en el contexto de lo que el mismo conoce y maneja, es decir en Java, el lenguaje de programación de la entrada. Para ilustrar la forma en que funciona este mecanismo de recuperación de contraejemplos, consideremos nuevamente el ejemplo de la Figura 1. Si bien no es necesariamente evidente, este programa fallará en el intento de insertar un elemento sobre el arreglo vacío. TACO detectará este problema dando como salida un contraejemplo similar al de la Figura 2. Este contraejemplo Alloy es difícil de comprender si no se conoce cómo se ha codificado el programa Java en Alloy. Nuestra mejora a TACO producir—á, en lugar de la salida anterior, una salida en forma de un unit test en Java. La salida equivalente a la de la Figura 2 que nuestra extensión proveerá se muestra en la

```
0)(precondition_ar_edu_taco_simplecode_BoundedStack_pushBuggy_0[↩
    Object_Array, ar_edu_taco_simplecode_BoundedStack_elems, ↩
    ar_edu_taco_simplecode_BoundedStack_size, thiz, throw, x])?
1)call null/ar_edu_taco_simplecode_BoundedStack_pushBuggy_0[[thiz,↩
     throw, x, saxSourceHashCode, byteHashCode, ↩
    ar_edu_taco_simplecode_BoundedStack_elems, ↩
    JMLObjectSet_contains, Set_contains, Iterator_contains, ↩
    List_contains, auditLogXMLReaderHashCode, stringLength, ↩
    ar_edu_taco_simplecode_BoundedStack_size, dateHashCode, ↩
    stringHashCode, characterHashCode, JMLObjectSequence_contains,↩
     integerHashCode, Map_entries, Object_Array, usedObjects]]
2)(postcondition_ar_edu_taco_simplecode_BoundedStack_pushBuggy_0[↩
    Object_Array, Object_Array', ↩
    ar_edu_taco_simplecode_BoundedStack_elems, ↩
    ar_edu_taco_simplecode_BoundedStack_elems', ↩
    ar_edu_taco_simplecode_BoundedStack_size, ↩
    ar_edu_taco_simplecode_BoundedStack_size', thiz, thiz', throw↩
    ', x', x'])?
```

Fig. 2. Traza de error detectada por TACO, mostrada como modelo Alloy.

Figura 3. Como puede observarse, la legibilidad del contraejemplo es significativamente más sencilla debido a que el mismo está realizado en el mismo lenguaje utilizado para describir el programa a verificar. Esto beneficia a la no necesidad de interpretación o incluso conocimiento de la presencia y uso del lenguaje Alloy como método subyacente de verificación.

3.1 Detalles de técnicos y de implementación

La extensión a TACO que describimos en esta sección se ajusta a la arquitectura original de la herramienta, implementada como un *pipeline* de traducciones y transformaciones. Sumamos a este diseño inicial nuevas componentes, un *singleton* que facilita la recolección de información de importancia de las transformaciones y traducciones, para poder tomar el camino de regreso desde modelos Alloy a Java, y la aplicación del patrón *command* para simplificar la descripción de los estados.

Una vez realizada la recolección y el respaldo de los nombres originales con su correspondiente en Alloy, se procede a realizar el llamado al SAT-Solver. Si se encuentra un contraejemplo, se procede a la construcción de dicha instancia en Java. Esto demanda la utilización de los datos de traducción recolectados y sucesivas *consultas* al módulo de evaluación de Alloy, para poder reconstruir el contraejemplo como un programa. Como puede observarse en la Figura 3, la construcción del unit test Java hace uso de *Reflexión*. Esencialmente, utilizamos la información provista por el modelo Alloy obtenido para construir, en algún sentido a "bajo nivel", el contexto exacto de reproducción de la falla. Construimos todos los objetos presentes en el modelo Alloy, y seteamos sus respectivos estados mediante rutinas provistas por la librería de reflexión. A modo de ejemplo, supongamos que tenemos una instancia de una listas; la forma de creación sería similar a la ilustrada en el código siguiente:

```
import java.lang.reflect.Field;
public class OutputCounterexample {

public static void main(String[] args){
    ar.edu.taco.simplecode.BoundedStack thiz;
    Field f;
    boolean fieldAccess;

    ar.edu.taco.simplecode.BoundedStack ↩
        ar_edu_taco_simplecode_BoundedStack_0 = (ar.edu.taco.↩
        simplecode.BoundedStack) Class.forName("ar.edu.taco.↩
        simplecode.BoundedStack").newInstance();

    f = ar_edu_taco_simplecode_BoundedStack_0.getClass().↩
        getDeclaredField("elems");
    fieldAccess = f.isAccessible();
    f.setAccessible(true);
    f.set(ar_edu_taco_simplecode_BoundedStack_0, {null, null});
    f.setAccessible(fieldAccess);

    f = ar_edu_taco_simplecode_BoundedStack_0.getClass().↩
        getDeclaredField("size");
    fieldAccess = f.isAccessible();
    f.setAccessible(true);
    f.set(ar_edu_taco_simplecode_BoundedStack_0, 0);
    f.setAccessible(fieldAccess);

    thiz = ar_edu_taco_simplecode_BoundedStack_0;

    /* Method invocation */
    void returnValue = thiz.pushBuggy(java_lang_SystemArray_2);
}
```

Fig. 3. Traza de error detectada por TACO, mostrada como unit test Java.

```
myPackage.MyList myPackage_MyList_0 =
        (myPackage.MyList)
                Class.forName("myPackage.MyList").newInstance();
```

Luego, comenzamos a recuperar los valores de los campos de cada átomo mediante consultas al evaluador Alloy, y se construye el código asociado a la modificación del atributo corresopndiente. En el código siguiente puede verse dicha modificación del campo `head`:

```
f = myPackage_MyList_0.getClass().getDeclaredField("head");
fieldAccess = f.isAccessible();
f.setAccessible(true);
f.set(myPackage_MyList_0, null);
f.setAccessible(fieldAccess);
```

Una vez realizada la creación de los objetos y la modificación de sus atributos, concluimos con la recreación del heap al momento de la invocación del método con sus parámetros que viola la aserción. Llegado este punto, identificamos el objeto principal del escenario de falla, e invocamos en éste el método que originó la falla con los parámetros que dieron lugar al bug de acuerdo al modelo Alloy.

4 Análisis Modular

Describimos ahora la segunda de las mejoras propuestas para TACO. El objetivo de ésta es contribuir a la escalabilidad de la técnica de análisis subyacente a la herramienta, proveyendo una forma de realizar el mismo de manera *modular*. La modularización está basada en el aprovechamiento de la misma modularización de código del programa, de acuerdo a abstracción procedimental, es decir la definición de rutinas separadas y su invocación. Esta propuesta ofrece una alternativa al "inlining" de código como forma de tratar las invocaciones a rutinas. Dado que el código de entrada a la herramienta debe constar de especificaciones formales en forma de contratos, nuestra mejora aprovecha dichos contratos y los utiliza en la generación del código a verificar. Así, la complejidad del código a analizar sólo depende de la rutina principal (pues no se realiza inlining), aunque el resultado de la verificación es menos preciso: si el análisis "pasa" la verificación, es correcto siempre y cuando el código invocado también lo sea (lo cual puede verificarse separadamente); si el análisis "no pasa" la verificación, no necesariamente significa que sea incorrecto, sino que puede significar que alguna de las rutinas invocadas no tiene un contrato suficientemente detallado.

Más precisamente, la transformación que esta mejora realiza consiste en el reemplazo de la instrucción de llamada a rutina por la siguiente secuencia de instrucciones:

```
JAssert method_precondition[actual_parameter_1, .., actual_parameter_T ]
JHavoc[field_1, .. , field_Q]    // descriptos en la clausula
                                 // modifies
JAssume method_postcondition[actual_parameter_1, .., actual_parameter_N ]
JAssume class_invariant[param_1, .. , param_M]  // clase a la cual
                                                // pertenece el metodo
                                                // invocado
```

Estas instrucciones indican que debe comprobarse que el método invocante cumple con la precondición del método invocado (*assert*), que todas las variables y campos que la rutina invocada puede modificar podrían cambiar arbitrariamente, pero que se asume verdadera la postcondición del método invocado, y el invariante de la clase de la rutina invocada. La combinación de estas sentencias equivalen a la utilización de la especificación de la rutina invocada, en lugar de su código.

Para aclarar el funcionamiento de este proceso, consideremos un método adicional de la clase `BoundedStack` que ya introdujimos anteriormente; este método, denominado `pushCalledBuggyInside`, está definido como se indica a continuación:

```
public class BoundedStack {
    ...
    /*@ requires size < elems.length-1;
      @
      @ assignable elems[size], size;
      @
      @ ensures size == \old(size + 1);
      @ ensures elems[size-1] == x;
```

```
    @ ensures_redundantly (\forall int i; 0 <= i && i < size-1; ↩
        elems[i] == \old(elems[i]));
    @*/
  public void pushCalledBuggyInside(Object x) {
      pushBuggy(x);
  }
  ...
}
```

TACO trataría normalmente la invocación a `pushBuggy` como un inlining de código, y si el código invocado a su vez invoca otra rutinas, se realizará el inlining de estas últimas igualmente. Con nuestra extensión, cada invocación es reemplazada de la forma antes descripta por tres pasos. La precondición se añade en forma de un `assert` (debe cumplirse siempre que se invoque al método). Todos los elementos descriptos tanto en el `modifies` de la clase como en el método se convierten a "havocs"[1]. Finalmente se realiza el agregado de la postcondición y el invariante de clase utilizando `assume` (todos los elementos involucrados deben modificarse de modo que la postcondición se satisfaga).

4.1 Detalles técnicos y de implementación

Para poder realizar la implementación de esta segunda extensión a la herramienta, se estudió y se eligió la posición en el pipeline de transformaciones de la herramienta en la cual esta extensión mejor se adecúe. El lugar elegido fue la transformación desde código intermedio *jdynalloy* a DynAlloy. Se utilizó principalmente el patrón de diseño *Visitor* para tomar el código intermedio, en este caso en forma de árbol abstracto de sintaxis (AST), y recorrer el mismo para localizar invocaciones a rutinas. Afortunadamente, esta codificación ya trata a las invocaciones de manera modular, a través de rutinas de llamada (una implementación a bajo nivel de proceso de invocación a rutina y retorno al invocador). Al identificar las invocaciones, las mismas se procesan para saber cómo deben reemplazarse, como se explicó anteriormente.

5 Limitaciones

Las extensiones propuestas poseen algunas limitaciones, propias del carácter prototípico de las mismas. En primer lugar, el soporte de tipos es relativamente limitado. Por ejemplo, en el caso de arreglos, se soportan sólo arreglos de tipos básicos. Otras limitaciones tienen que ver directamente con TACO y no con las extensiones en si. Un ejemplo de esto es la ausencia de soporte para código que maneje reflexión o instrospección (éstos no son soportados en los programas analizados por TACO).

[1] Havoc genera valores arbitrarios para los parámetros recibidos dependiendo el tipo de éstos. Es decir, equivale a decir que los parámetros cambian arbitrariamente.

6 Experimentación

Hemos experimentado con las extensiones propuestas para un número de casos de estudio de código de estructuras de datos en Java. Lamentablemente, la exigencia de requerir que el código esté acompañado de sus respectivas especificaciones formales en JML hace que sea difícil encontrar casos de prueba, y que podamos evaluar las herramientas con grandes volúmenes de código real.

En cuanto a la primera de las extensiones, hemos experimentado con un conjunto de casos de código con bugs, y con casos en los cuales hemos mutado ligeramente las especificaciones formales para que no se ajusten al código (una forma de inyectar bugs modificando las especificaciones en lugar de código). En general la evaluación ha tenido por propósito detectar problemas en nuestra herramienta, la cual ha funcionado satisfactoriamente. Sin embargo hemos observado que en muchos casos una parte importante del heap reconstruido para reproducir el bug no juega ningún rol importante en el mismo. Por lo tanto, estamos estudiando formas de remover las porciones redundantes o irrelevantes del heap, y así hacer los unit tests producidos más simples y fáciles de utilizar para la depuración.

La experimentación con la segunda de nuestras extensiones nos ha permitido detectar especificaciones correctas pero débiles en rutinas invocadas dentro de otras verificadas, lo cual ha dado lugar a un número significativo de contraejemplos espúrios. Depurar estos contraejemplos, y descubrir su espuriedad, es costoso en tiempo. Por esta razón, estamos trabajando en automatizar el proceso de evaluación de contraejemplos, un elemento aún no disponible en nuestra extensión.

7 Conclusiones

Hemos presentado dos mejoras a una herramienta de análisis automático de programas Java anotados con JML, denominada TACO. En primer lugar, proveemos un mecanismo de análisis modular de código en presencia de invocación de rutinas. En segundo lugar, automatizamos la construcción de unit tests en Java, que reproducen los bugs detectados por el análisis.

Las mejoras presentadas contribuyen al análisis en dos dimensiones: el análisis modular contribuye a la escabilidad de la técnica de análisis subyacente a TACO, mientras que la recuperación de contraejemplos facilita el uso de la herramienta, ocultando adecuadamente los detalles del método formal subyacente en su aplicación. Estas extensiones son en nuestra opinión sumamente relevantes, ya que contribuyen a la mejora de herramientas automáticas de análisis basadas en métodos formales, muy poderosas, pero cuya adopción se ve en general afectada por razones de escalabilidad o por los requisitos, en cuanto a capacidades requeridas para su uso, que se demanda a los desarrolladores. Estos desarrollos contribuyen a paliar ambos problemas.

Varias oportunidades de mejora de las herramientas y técnicas descriptas han sido identificadas como producto de este desarrollo. Algunas de estas han sido

mencionadas en la sección anterior. Una línea interesante de trabajo, que apunta a que la técnica detrás de TACO sea más ampliamente aplicada, es la construcción de especificaciones en lugar de requerir las mismas por parte de los usuarios. Existen algunos enfoques para conseguir esto, por ejemplo reemplazando las pre y post condiciones por unit tests "positivos" y "negativos" (que describen qué debería y qué no debería hacer el software mediante unit tests) y la construcción de resúmenes para rutinas invocadas, como en [12], que planeamos explorar en nuestro trabajo.

Referencias

1. E. Clarke, O. Grumberg y D. Peled, *Model Checking*, MIT Press, 1999.
2. S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, The Object-Oriented Series, Prentice-Hall, 1994.
3. M. Frias, J. Galeotti, J. López Pombo y N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, en Proceedings de International Conference on Software Engineering ICSE 2005, ACM Press, 2005.
4. M. Frias, J. P. Galeotti, J. López Pombo y N. Aguirre, *Efficient Analysis of DynAlloy Specifications*, Transactions on Software Engineering and Methodology (ACM TOSEM), ACM Press, 2007.
5. J. P. Galeotti y M. Frias, *DynAlloy as a Formal Method for the Analysis of Java Programs*, Software Engineering Techniques: Design for Quality, SET 2006, IFIP 227, Springer, 2006.
6. J. Galeotti, N. Rosner, C. López Pombo y M. Frias, *Analysis of Invariants for Efficient Bounded Verification*, en Proceedings de International Symposium on Software Testing and Analysis ISSTA 2010, ACM Press, 2010.
7. D. Jackson, *Alloy: a lightweight object modelling notation*, ACM Transactions on Software Engineering and Methodology (ACM TOSEM), Vol. 11, Nro. 2, 2002.
8. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press. 2006.
9. G. Leavens et al., *JML Reference Manual*, Dept. of Computer Science, Iowa State University, 2004.
10. J.-F. Monin y M. Hinchey, *Understanding Formal Methods*, Springer, 2003.
11. N. Storey, *Safety-Critical Computer Systems*, Pearson, 1996.
12. M. Taghdiri, *Inferring Specifications to Detect Errors in Code*, en Proceedings de 19th International Conference on Automated Software Engineering ASE 2004, September 2004, Austria.