# Reparación eficiente de estructuras de datos en tiempo de ejecución basada en SAT

Marcelo Uva[1], Pablo Ponzio[1,2], Germán Regis[1], and Nazareno Aguirre[1,2]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina. Email: {uva,pponzio,gregis,naguirre}@dc.exa.unrc.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

**Resumen** Las fallas de un programa pueden producir estados internos inconsistentes, cuya propagación puede provocar la terminación abrupta del programa, la pérdida de datos del usuario, la incapacidad de realizar alguna tarea, etc. La reparación de estructuras de datos consiste en reemplazar estos estados inconsistentes por estructuras generadas a partir de especificaciones formales, utilizando algún procedimiento de decisión, que permitan al programa continuar su ejecución sin mayores problemas. En este trabajo se presenta un enfoque para resolver este problema basado en SAT solving, cuya característica distintiva es el aprovechamiento de dos técnicas del estado del arte para mejorar su eficiencia y escalabilidad: el uso de rotura de simetrías y de cotas ajustadas. La evaluación experimental preliminar realizada muestra que nuestro enfoque es más eficiente y es capaz de lidiar con estructuras de mayor complejidad que técnicas relacionadas.

## 1. Introducción

La constante evolución y expansión de los sistemas de software en innumerables dispositivos, ha generado la necesidad de contar con aplicaciones cada vez más confiables. Actualmente es posible encontrar software en teléfonos, cámaras digitales, televisores, dispositivos de localización satelital, aviones, electrodomésticos, etc. Todo esto ha generado un especial interés en las áreas de investigación vinculadas a los métodos formales e ingeniería de software. Un alto porcentaje de los recursos asignados a los proyectos de desarrollo de software son destinados a tareas de detección y corrección de fallas. Los problemas se agravan si el sistema ya se encuentra instalado y funcionando. La corrección de fallas en estos casos involucrará una detención del sistema, su corrección y posterior puesta en marcha. Es posible que el sistema no pueda ser detenido durante un período determinado de tiempo por lo que, en estos casos, sea preferible "soportar" un leve deterioro del sistema a someterlo al proceso de corrección descripto.

Una alternativa al procedimiento anterior es aplicar técnicas automáticas, basadas en especificaciones, que permitan restablecer el sistema a un estado consistente luego de la aparición de una falla. En general, estas técnicas utilizan

un procedimiento de decisión para generar un estado que satisface las especificaciones (invariantes, pre y pos condiciones), que sirve de reemplazo al estado del sistema inconsistente producido por un defecto. En particular, diversos trabajos han atacado la reparación de programas que manipulan estructuras de datos dinámicas, como listas, árboles binarios, etc. [1,2,3,4]

En este trabajo, se propone una técnica de reparación de estructuras de datos en tiempo de ejecución basada en satisfactibilidad booleana (SAT). La técnica es aplicada una vez que se ha detectado una estructura que viola un invariante o la poscondición luego de la ejecución de un método (y corresponde a una falla), y emplea procedimiento de decisión basado en SAT (Alloy [5]) para generar una estructura concreta válida respecto al invariante y la poscondición del método erróneo.

Existen en la literatura diversos trabajos vinculados a la reparación de estructuras de datos basada en SAT. En menor o mayor medida, estos enfoques presentan problemas de escalabilidad (como la mayoría de los análisis basados en SAT), y se vuelven ineficientes cuando las estructuras crecen en tamaño y complejidad. Los investigadores han abordado este problema de diferentes maneras. Uno de los trabajos que ha logrado lidiar mejor con los problemas de escalabilidad es el presentado en [1], implementado en la herramienta Cobbler, que propone aprovechar la información de la ejecución del programa que origina la falla. Este enfoque consiste en almacenar información sobre los campos de la estructura a reparar accedidos por el programa, e intentar modificar sólo dichos campos en una primera instancia de reparación, acotando así el espacio de búsqueda de estructuras. Si este primer intento es exitoso, el enfoque obtiene ganancias sustanciales en tiempo de ejecución y escalabilidad respecto de enfoques previos [1]. Sin embargo, en situaciones en las que la reparación implica modificar partes de la estructura que no han sido accedidas por el programa, Cobbler simplemente invoca a un procedimiento tradicional de generación de estructuras basado en SAT solving, y su escalabilidad se ve seriamente afectada (ver sección 2).

En este trabajo, presentamos un enfoque más general para lidiar con los problemas de escalabilidad de la reparación de estructuras de datos, que involucra la utilización de dos técnicas de optimización recientes de análisis de programas basados en SAT. Estas técnicas son: la canonización del heap para eliminar estructuras "isomorfas" y así acelerar el análisis (comúnmente llamada rotura de simetrías), y el uso de cotas ajustadas para eliminar variables proposicionales innecesarias de la fórmula proposicional a analizar mediante SAT solving. Estas técnicas fueron introducidas originalmente en [6], y aplicadas a diversos tipos de análisis de programas basados en SAT [7,8]. La contribución principal de este trabajo es la adaptación de estas técnicas al contexto novedoso de reparación de estructuras de datos en tiempo de ejecución, y una evaluación experimental preliminar que muestra los beneficios del uso de estas técnicas en este contexto.

Cabe destacar que nuestra técnica permite mejorar la performance y la escalabilidad de la reparación en general, a diferencia de Cobbler, cuyas ventajas sólo se dan bajo ciertas circunstancias (como se discute en 2 y como evidencia

nuestra evaluación experimental). También es importante mencionar que Cobbler y nuestro enfoque son complementarios, y podrían utilizarse conjuntamente para producir mejores resultados.

El resto del trabajo se organiza como sigue. En la sección 2, se describe el lenguaje de especificación Alloy, que se utiliza aquí como lenguaje de especificación y como procedimiento de decisión, y el enfoque de reparación de Cobbler. En la sección 3, se presenta nuestra técnica, y se describen la rotura de simetrías y el uso cotas ajustadas como optimizaciones para mejorar la eficiencia y la escalabilidad de la reparación. La sección 4 presenta un análisis experimental preliminar de nuestra técnica, comparada con la presentada en [1]. En 5 se discuten trabajos relacionados, y en 6 se presentan las conclusiones y posibles trabajos futuros.

## 2. Preliminares

### 2.1. Alloy

Alloy[5] es un lenguaje de especificación declarativo basado en la lógica relacional, una extensión de la lógica de primer orden con operadores como composición y clausura transitiva de relaciones. Alloy es un lenguaje simple, fácil de aprender, y posee una herramienta de software asociada para analizar especificaciones. Esta herramienta, llamada Alloy Analyzer, implementa un análisis automático de las especificaciones Alloy, basado en SAT solving, y es de libre disponibilidad.

Alloy posee un gran poder expresivo, por lo que resulta útil para especificar estructuras de datos con invariantes complejos, como así también los programas que las manipulan (pre y postcondiciones por ejemplo). Por este motivo, y por el análisis eficiente que implementa el Alloy Analyzer, Alloy se ha usado como lenguaje de especificación y como backend de análisis en numersos trabajos, en particular, en [1] y en este trabajo.

### 2.2. Reparación de estructuras de datos basada en SAT, Cobbler

En esta sección se discute la reparación de estructuras de datos basada en SAT, junto con el enfoque de optimización llamado Cobbler [1], a través de un ejemplo. En la Figura 1 se muestra un fragmento de una implementación tradicional de listas simplemente encadenadas en el lenguaje JAVA. Esta implementación posee un método `addFirst` para agregar un nuevo elemento a la cabeza de la lista. Además, la implementación posee contratos declarativos, especificados en Alloy. El invariante de la clase postula que las listas deben ser acíclicas, y que su tamaño es igual a la cantidad de nodos alcanzables a partir de la cabeza de la lista. En la figura también se muestra (parte de) la poscondición de `addFirst`. Esta indica que luego de la ejecución del método el tamaño de la lista debe ser al tamaño previo a la ejecución más uno (notar que se utiliza la convención usual de que `this` y `this'` representan el estado anterior y posterior a la ejecución del método de `this`, respectivamente).

```
public class Nodo {
    int value;
    Nodo next;
}

public class Lista {
    /*
      Invariante:
        all l: Lista, n: Nodo |
            ((n in l.head.*next => n not in n.^next) and
             l.size = #l.head.*next))
    */
    Nodo head;
    int size;
    /*
      Post:
        . . .
        this'.size = this.size + 1
    */
    public void addFirst(int i) {
      Nodo n = new Nodo();
      n.value = i;
      n.next = this.head;
      this.head = n;
      // size++; ERROR: linea omitida por el programador
    }
}
```

**Figura 1.** Programa de ejemplo: `addFirst` de listas simplemente encadenadas

El método `addFirst` de la figura contiene un defecto: el programador olvidó incrementar el campo `size`, que representa la cantidad de elementos de la lista. Así, ejecutar `addFirst(1)` sobre la primera lista (de arriba hacia abajo) de la figura 2 resulta en la segunda lista de la misma figura: un estado inconsistente que viola el invariante de listas al poseer tres elementos pero tener un tamaño de 2. Una implementación correcta de `addFirst` (que satisfaga la postcondición del método) debería retornar la tercera lista de la figura 2. Una reparación basada en SAT solving para este caso consistiría en generar una especificación Alloy que incluya la primera lista de la figura 2, la postcondición de `addFirst(1)` y el invariante de listas. Alimentando el Alloy Analyzer con esta especificación y los scopes apropiados (límites en la cantidad de nodos y número máximo de enteros, debido a que el análisis implementado por el Alloy Analyzer es acotado), la herramienta produce como output la tercera lista de la figura 2, es decir, una reparación que satisface el invariante de `Lista` y la postcondición de `addFirst()`. Esta lista puede usarse para reemplazar el estado inconsistente producido por la ejecución del defecto, y continuar normalmente con la ejecución de la aplicación

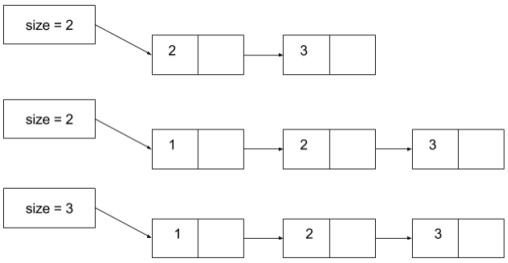


**Figura 2.** De arriba a abajo: entrada de `addFirst(1)`, salida incorrecta de `addFirst(1)`, posible reparación generada con Alloy

que invoca a `addFirst`. Lo descrito anteriormente es el enfoque tradicional de reparación de estructuras de datos basado en SAT, presentado en [2].

Sin embargo, como es usual con los enfoques de análisis basados en SAT, el enfoque anterior sufre de problemas de escalabilidad cuando se aplica a estructuras complejas, y/o cuando se desean reparar estructuras de mayor tamaño (es decir, incrementar los scopes). Una de las propuestas más exitosas para lidiar con este problema es la implementada en la herramienta Cobbler [1]. Cobbler se basa en la observación de que en muchas ocasiones es posible obtener una estructura correcta (reparación) mediante SAT manteniendo fija la parte de la estructura no "visitada" por el código, y permitiendo al SAT solver modificar sólo los campos accedidos por el código. De esta manera, en muchos casos se logra reducir ampliamente el espacio de búsqueda del SAT solver, acelerando su performance y permitiendo reparar estructuras de tamaño mucho mayor.

Cobbler instrumenta el código (`addFirst` en nuestro ejemplo) para almacenar los campos accedidos y modificados de las estructuras durante la ejecución. El algoritmo de reparación de Cobbler consta de tres etapas. Primero, Cobbler intenta obtener una reparación permitiendo al SAT solver cambiar sólo los campos modificados por la ejecución particular del código. Si esto no es posible, el siguiente paso generar una estructura correcta permitiendo la modificación de los campos modificados y los accedidos durante la ejecución. Si ninguno de los dos pasos anteriores resulta exitoso, Cobbler utiliza el UNSAT core para determinar cuales campos deben ser modificados para obtener una reparación. En nuestra experiencia con la herramienta, Cobbler es muy eficiente si logra reparar la estructura en las primeras dos etapas. Sin embargo, en la tercera etapa por lo general Cobbler permite modificar la estructura completa (o casi completa), con lo cual en este punto se comporta de manera muy similar a los enfoques tradicionales de reparación basada en SAT. Este es el caso en el ejemplo de la figura 1, y en muchos otros ejemplos prácticos relevantes.

A diferencia de Cobbler, nuestro enfoque apunta a mejorar la eficiencia de la reparación en el caso general.


**Figura 3.** Dos listas isomorfas

## 3. Un enfoque eficiente para la reparación de estructuras de datos basada en SAT

En esta sección se describe la principal contribución de este trabajo: la aplicación de rotura de simetrías y cotas ajustadas a la reparación de estructuras de datos en tiempo de ejecución basada en SAT, con el objetivo de aumentar la eficiencia y la escalabilidad de la reparación a estructuras de mayor tamaño y complejidad. Estos enfoques se explican a continuación.

Es importante destacar que la detección de fallas durante la ejecución del programa y la recuperación de las fallas son problemas diferentes, muy complejos por si mismos, y no se abordarán aquí. Para este propósito podría utilizarse el framework Armor [14], que monitorea la ejecución del programa y en caso de aparición de una falla es capaz de reemplazar estados incosistentes por estados correctos, como los provistos por nuestra técnica, y continuar con la ejecución normal del programa a partir de los mismos.

### 3.1. Rotura de Simetrías

Un problema del modelo Alloy de listas presentado en la sección 2 es que admite estructuras isomorfas, es decir, dos o más estructuras concretas que representan una misma estructura abstracta, y son indistinguibles desde el punto de vista de la reparación.

Por ejemplo, si consideramos 3 nodos e identificadores N0, N1 y N2 para los mismos, las listas de la figura 3 son instancias válidas de nuestra especificación de la sección anterior. Sin embargo, estas estructuras son isomorfas, ya que ambas poseen los mismos valores en el mismo orden, y por lo tanto representan la misma lista (los identificadores de nodos usualmente se consideran abstracciones de direcciones de memoria, y en lenguajes orientados a objetos como JAVA la dirección de memoria donde están alojados los nodos no es importante).

En [6] se propone canonizar el heap para eliminar instancias isomorfas, instrumentando de manera automática las especificaciones con predicados de rotura de simetrías. Estos predicados asignan identificadores a los nodos de las estructuras en orden creciente (N0, N1, N2, etc.) siguiendo un recorrido primero en anchura. Intuitivamente, una vez llevada a cabo la instrumentación de la especificación Alloy con predicados de rotura de simetrías, la lista que se muestra a la izquierda en la figura 3 se convierte en la representación canónica de las listas de la figura, siendo ahora la lista de la derecha una instancia no permitida

de la especificación. La evaluación experimental de [6] indican que la reducción de la cantidad de instancias válidas producto de la rotura de simetrías permite mejorar sustancialmente la escalabilidad y la eficiencia de la verificación acotada de programas basada en SAT.

### 3.2. Cotas Ajustadas

El modelo Alloy de listas de la sección 2 posee un invariante de representación que establece que todas las listas deben ser acíclicas. Ahora bien, bajo el supuesto de que se desea generar un estado de reparación para un método de listas donde se ha producido una falla y donde además las especificaciones cuentan con predicados de rotura de simetrías, el conjunto dominio para el campo *next* contendrá los pares de nodos (Ni, Nj) o (Ni, null), donde el par (Ni, Nj) representa que Ni.next = Nj, para $0 \leq$ i,j $\leq 2$, fijando un scope de 3 para la signatura Nodo. Sin embargo, el invariante de representación afirma que sólo las listas acíclicas son estructuras válidas. Por lo tanto, muchos de los valores posibles en el domino de *next* nunca podrán darse en una reparación válida. Decimos que un par (Ni, Nj) pertenece a la cota ajustada del campo next si Ni.next = Nj en alguna instancia que satisface el invariante de representación. Así, estos valores de campos pueden eliminarse de la codificación del problema de reparación como una fórmula proposicional, simplificando la fórmula y haciéndola más sencilla de analizar mediante SAT solving.

En esta aplicación particular, el cómputo de cotas ajustadas consiste en computar el conjunto de valores que efectivamente puede tomar un campo bajo las restricciones del invariante de la estructura. Para nuestro ejemplo con 3 nodos, los valores permitidos por el invariante para el campo *next* (y la canonización del heap) definen la siguiente cota ajustada:

$$cota\_next = \{(N0, null), (N1, null), (N2, null), (N0, N1), (N1, N2)\}$$

De esta manera, de 12 posibilidades para la codificación del dominio del campo next, la cantidad de valores a representar se reduce a 5 si se dispone de la cota ajustada anterior.

Observar que, mientras más precisa sea la cota ajustada, mayor será el beneficio para el análisis basado en SAT, ya que se podrán eliminar mayor cantidad de variables proposicionales de la fórmula proposicional que codifica el problema de reparación. En [12] se definen una serie de algoritmos para computar cotas ajustadas de manera eficiente. Estos algoritmos fueron los utilizados para computar las cotas ajustadas en este trabajo.

Notar que el cómputo de cotas ajustadas es relativamente costoso computacionalmente, pero en este contexto se puede efectuar previamente a la instalación (deployment) del sistema. Por esta razón, el tiempo de cómputo de cotas ajustadas no se tiene en cuenta en nuestra evaluación experimental de la sección 4.

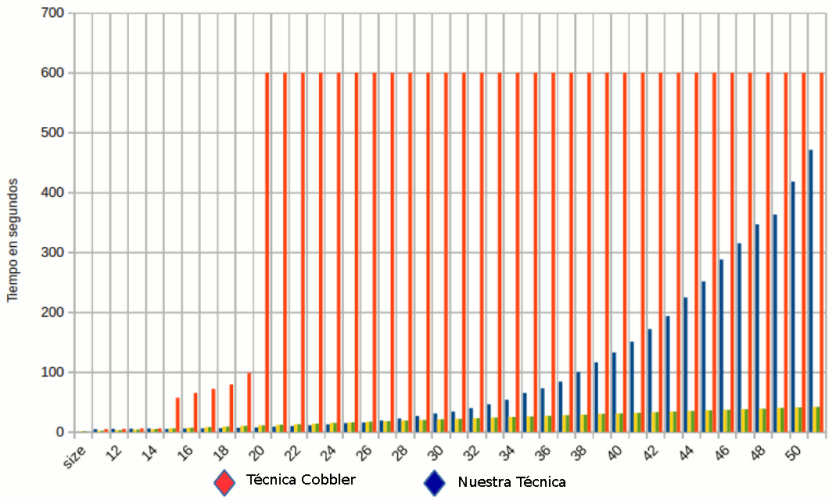


**Figura 4.** Comparación experimental entre Cobbler y nuestro enfoque.

## 4. Evaluación experimental de la técnica

Con el objetivo de evaluar nuestra técnica y comparar su rendimiento con Cobbler[1], se han realizado una serie de pruebas experimentales. Nuestra técnica fue implementada mediante un prototipo desarrollado en Python haciendo uso del Alloy Analyzer. Todos los experimentos fueron ejecutados sobre una PC con un procesador Intel(R) Core(TM) i5-4460 CPU 3.40Ghz con 8GB de RAM corriendo un sistema operativo GNU/Linux 3.2.0.

El caso de estudio seleccionado ha sido *listas simplemente encadenadas* y el método a reparar `addFirst()` el cual cuenta con el error de no incrementar el campo *size* (como se discute en la sección 2). Para probar la efectividad de nuestra técnica se generaron aleatoriamente 100.000 listas simplemente encadenadas por medio de Randoop[13] (herramienta de generación automática de casos de tests basada en aleatoriedad), de las cuales fueron seleccionadas aleatoriamente 41 listas de tamaño entre 11 y 51. Estas listas se utilizaron como entradas para `addFirst()` (con el defecto mencionado), para producir los estados erróneos que se utilizarán para medir la eficiencia y la escalabilidad de la reparación, utilizando Cobbler y nuestro prototipo. En ambos casos se estableció un tiempo máximo de reparación de 10 minutos, y se interrumpieron los experimentos que superaron este límite.

La figura 4 resume los resultados de los experimentos. En ella puede observarse que nuestra técnica logra reparar listas de hasta 51 elementos, mientras que Cobbler alcanza el timeout de 10 minutos sin lograr una reparación para todas las listas de 19 elementos o más. Por otro lado, Cobbler repara listas con un máximo de 19 elementos en 98,79 segundos, versus los 6,89 segundos que tarda nuestra técnica, esto es, nuestra técnica es un orden de magnitud más rápida en

este caso. Esta evaluación preliminar muestra resultados prometedores respecto de la eficiencia y la escalabilidad de nuestra técnica.

Cabe destacar que este es un caso difícil para Cobbler, ya que es las estructuras resultantes debido al defecto en `addFirst()` no pueden repararse modificando sólo los campos accedidos por `addFirst()`. Si bien pueden existir casos en los que Cobbler sea más eficiente que nuestro enfoque (cuando el código accede a pocos campos, y es posible obtener una reparación cambiando sólo estos campos), es importante mencionar que Cobbler y las técnicas de rotura de simetrías y uso de cotas ajustadas utilizadas por nuestro enfoque pueden combinarse para obtener los beneficios de ambas a la vez.

## 5. Trabajos Relacionados

Los trabajos relacionados de reparación más relevantes fueron discutidos anteriormente en nuestra sección de preliminares.

Nuestro trabajo utiliza cotas ajustadas para optimizar el análisis. Estas han sido explotadas exitosamente en trabajos previos pero en otros contextos. Por ejemplo, para la generación de casos de test basados en SAT [7], para la generación de casos de test mediante model checking acotado y ejecución simbólica [8], y para la verificación exhaustiva acotada de código [6]. En todos estos casos, las cotas ajustadas han sido de suma importancia para mejorar la eficiencia y la escalabilidad de las técnicas.

La rotura de simetrías para SAT solving fue introducida en [6]. Sin embargo, esta técnica ha sido aplicada a muchos otros contextos anteriormente, como por ejemplo, en model cheking [9] [10], o en la generación de casos de tests basada en invariantes operacionales [11].

## 6. Conclusiones y trabajos futuros

En este trabajo, se ha presentado una técnica basada en SAT de reparación estructuras de datos dinánicas en tiempo de ejecución. Si bien, como se menciona en las primeras secciones, existen algunos trabajos en la literatura en donde se aplica SAT con estos fines, nuestro enfoque realiza contribuciones en cuanto a mejorar escalabilidad en estruturas de datos de mayor tamaño y complejidad. Esto se ha logrado incorporando las técnicas de rotura de simetrias y uso de cotas ajustadas, aplicadas anteriormente a distintos análisis de software como generación automática de casos de test y verificación acotada, al contexto novedoso de reparación de estructuras de datos dinámicas. Esto agrega evidencia de que la rotura de simetrías y en particular las cotas ajustadas, pueden aprovecharse fuera de análisis de código más tradicionales como el testing y la verificación, y alienta la investigación de nuevos contextos de aplicación que puedan beneficiarse de estas técnicas.

La evaluación experimental preliminar realizada muestra que nuestra técnica tiene el potencial de superar ampliamente a Cobbler. Estos escenarios desfavorables para Cobbler son relativamente frecuentes en la práctica (el ejemplo utiliza-

do es un error común de programación), lo que muestra la relevancia práctica de nuestra técnica. Es sencillo imaginar otros escenarios problemáticos para Cobbler, como por ejemplo, muchas veces la parte de la estructura visitada por el código debe modificarse si la estructura actual debe rebalancearse para satisfacer su invariante de representación (en AVLs, árboles rojos y negros, etc.). Como trabajo futuro se llevará a cabo una comparación extensiva de estas técnicas, lo que permitirá determinar sus virtudes y sus desventajas con mayor precisión.

Respecto a los escenarios favorables a Cobbler, en tales casos ambas técnicas podrían usarse en conjunto, potenciando sus ganancias. En otras palabras, las optimizaciones de rotura de simetrías y aprovechamiento de cotas ajustadas pueden incorporarse en Cobbler, mejorando su performance. Nuestro objetivo aquí fue mostrar la importancia de estas optimizaciones, debido a nuestra observación de que Cobbler no siempre es exitoso en mejorar la eficiencia de la reparación. Evaluar la incorporación de las optimizaciones mencionadas en Cobbler se deja como trabajo futuro.

## Referencias


1. R. Zaeem, D. Gopinath, S. Khurshid. "History-Aware data structure repair using SAT". Proceeding TACAS'12 Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems Pages 2-17
2. B. Elkarablieh; I. Garcia; Y.L. Suen, S. Khurshid, S. " Assertion-based repair of complex data structures". ASE (2007).
3. F. Zaeem; R. Nokhbeh; S. Khurshid "Contract-Based Data Structure Repair Using Alloy". ECOOP 2010.
4. H. Samimi; H. Aung; E. Millstein. "Falling Back on Executable Specifications". ECOOP 2010.
5. Jackson, D. Alloy: a lightweight object modeling notation. ACM Transactions on Software Engineering and Methodology 11, 2, 2002, 256–290.
6. J.P. Galeotti, N. Rosner, C. Lopez Pomboy M. Frias. "Analysis of invariants for efficient bounded verification" ISSTA 2010. ACM.
7. P. Abad; N. Aguirre; V. Bengolea; D. Ciolek; M. Frias et al. "Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. 2013 IEEE International Conference on Software Testing, Verification and Validation.
8. N. Rosner; J. Geldenhuys; N. Aguirre; W. Visser; M.Frias. "BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support." IEEE Trans. Software Eng. 41(7): 639-660 (2015)
9. R. Iosif, "Symmetry reduction criteria for software model checking". 9th International SPIN Workshop on Model Checking of Software. 2002.
10. M. Musuvathi and D. L. Dill, "An incremental heap canonicalization algorithm". 12th international conference on Model Checking Software, SPIN, 2005.
11. C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates" International Symposium on Software Testing and Analysis 2002.
12. P. Ponzio, Tesis doctoral.
*http : //digital.bl.fcen.uba.ar/Download/Tesis/Tesis_*5547*_Ponzio.pdf*
13. C. Pacheco and M. Ernst. "Randoop: Feedback-directed Random Testing for Java." OOPSLA 2007 Companion, Montreal, Canada, Oct. 2007, ACM.
14. A. Carzaniga; A. Gorla y otros. "Automatic Recovery from Runtime Failures". ICSE 2013.