Reparación de programas aplicando templates: Generando workarounds permanentes a través de SAT

Marcelo Uva[1], Pablo Ponzio[1,2] y Nazareno Aguirre[1,2]

[1] Universidad Nacional de Río Cuarto (UNRC)
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
{uva,pponzio,naguirre}@dc.exa.unrc.edu.ar

Resumen La aplicación de procesos modernos y sistemáticos en el desarrollo de sistemas informáticos con estrictos estándares de calidad no ha podido evitar la presencia de defectos en el software. A pesar de extensas test-suites, un alto porcentaje de sistemas son implantados con fallas. El desarrollo de técnicas de reparación de programas ha cobrado gran importancia en esta última década. El concepto de *workaround* ha sido utilizado por diversos enfoques para la reparación de programas. Un *workaround* aprovecha la redundancia implícita en un sistema con el propósito de evitar fallas detectadas en runtime. A partir de una técnica de cómputo automático de *state-bound workarounds* (workarounds dependientes de estado) que utiliza SAT-Solving, se propone un procedimiento de búsqueda de *workaround templates* independientes de estado, es decir, *state-free workarounds* que permitan mejorar los tiempos de reparación de programas.

Keywords: Reparación de programas, Workarounds, SAT-Solving

1. Introducción

La aplicación de procesos modernos y sistemáticos en el desarrollo de sistemas informáticos cumpliendo con estrictos estándares de calidad no ha podido evitar la aparición de defectos en el software. A pesar de ejecutar extensas test-suites, un alto porcentaje de los sistemas es implantado con fallas. Muchas de las denominadas fallas de campo, requieren de un período de tiempo largo para ser solucionadas efectivamente por los desarrolladores luego de ser reportadas. Esto se debe, entre otras causas, a la dificultad en reproducir aquellos ambientes en los cuales las fallas ocurren. La complejidad del software, la constante adaptación y extensión de los sistemas, sumada a la presión por disminuir los tiempos de producción y comercialización, define un contexto no muy favorable a la producción de software confiable. Estas circunstancias, combinadas con las demandas de disponibilidad del software hacen que las técnicas, que dotan al software de la capacidad de tolerar y reparar fallas hayan cobrado gran importancia en la actualidad. La noción de *workaround* es utilizada en este sentido y aprovecha

la redundancia intrínseca contenida dentro de un módulo, buscando ejecuciones alternativas a fin de reparar errores introducidos por los desarrolladores o bien provocadas por factores ambientales. En los trabajos [1,2,3] se ha utilizado el concepto de *automatic workaround* como mecanismo de recuperación de fallas sobre dominios específicos como por ejemplo, aplicaciones web. El cómputo de *workarounds* en estos enfoques es realizado sobre modelos abstractos (máquinas de estados) generadas manualmente a partir de un análisis acabado del comportamiento del sistema. En esta misma lína de investigación, en [4,5] se propone un enfoque similar, utilizando en este caso secuencias alternativas equivalentes definidas por los usuarios. Los autores del presente trabajo han propuesto técnicas que utilizan tecnologías basadas en SAT-Solving para el cómputo automático de *workarounds*[6]. A diferencia de los enfoques previos mencionados, estas técnicas son aplicadas sobre abstracciones propias de programas orientados a objetos, generadas a partir de especificaciones formales JML (Java Modeling Language)[7] incluídas en el código fuente de programas Java a modo de anotaciones. Estas técnicas utilizan el estado corriente donde se ha originado la falla como parte activa del cómputo de los *workarounds*, estableciendo así una especialización del concepto de *workaround* en *state-bound workarounds* y *state-free workaround*. En el primer caso, las soluciones alternativas utilizan el estado corriente del sistema para proporcionar una ejecución de reparación y en el segundo caso las soluciones son independientes del estado. Idealmente se desearía contar siempre con *state-free workarounds* para utilizarlos ante la ocurrencia de un error, lamentablemente, este tipo de *workarounds* son más costosos de encontrar y en muchas ocasiones, no existen. Un concepto derivado de estas técnicas es el de *workaround templates*. Estos definen una clase especial de *workarounds* que se mueven entre los dos tipos anteriores. Un *workaround templates* es una solución alternativa generalizable a partir de una serie de *state-bound workarounds* obtenidos. Un *workaround templates* puede ser categorizado en muchos casos como un *state-free workaround* haciendo intervenir minimamente a un SAT-Solver para concretizar los parámetros generalizados.
En este trabajo se propone un mecanismo de cómputo de *workaround templates* independientes de estado, esto permitirá su utilización como *state-free workaround* aplicando SAT-Solving. El resto del presente trabajo está organizado de la siguiente manera: en la sección 2 una revisión de los trabajos vinculados a la propuesta. En la sección 3 se presenta la técnica de reparación de la cual se deriva el procedimiento de cómputo de *workaround templates* independientes de estado. El mecanismo de detección de *workaround templates* y su posterior evaluación como *state-free workaround* es presentado en la sección 4. Luego, en la sección 5 se presentan algunos resultados obtenidos. Finalmente, las conclusiones y posibles trabajos futuros se presentan en la sección 6.

2. Preliminares

El concepto de *workaround* ha sido utilizado en técnicas de reparación de programas a través de diversos enfoques. Se presentan a continuación algunos

de los trabajos más relevantes. También se describen brevemente los lenguajes de modelado Alloy[8] y DynAlloy[9,10] junto con sus correspondientes herramientas de análisis automático: Alloy Analyzer y DynAlloy Analyzer utilizadas en el presente trabajo.

2.1. Automatic workarounds

El concepto de *workaround* fue definido inicialmente en el contexto de sistemas *self-healing* [1]. Un *workaround* aprovecha la redundancia implícita dentro de un sistema de software con el propósito de salvar las fallas que puedan producirse en él. Un *workaround* para una rutina m en un estado s_i define un procedimiento P compuesto por una secuencia de otras acciones pertenecientes al mismo módulo que permiten alcanzar el estado esperado s_f. Con el objetivo de automatizar el cómputo de *workarounds* en [1,3] se presentó un procedimiento que requiere de un modelo (máquina abstracta de estados) con el comportamiento de todo el sistema para poder encontrar estas alternativas de ejecución. El mismo grupo de investigación propuso también en [4] una arquitectura de reparación en donde inicialmente se debe proveer un conjunto de *workarounds* mediante reglas de re-escritura de términos. A partir de las mismas, el mecanismo de reparación tendrá la posibilidad de encontrar nuevas reparaciones combinando las ya conocidas.

2.2. Alloy

Alloy[8] es un lenguaje de especificación de primer orden basado en el uso de relaciones y orientado al modelado de propiedades estructurales de sistemas. Alloy utiliza operadores relacionales, definidos en lógica relacional (formalismo subyacente en el lenguaje) tales como el operador de composición, clausura transitiva y reflexo-transitiva con la finalidad de reducir la complejidad de la expresión de algunas propiedades estructurales comunes en lógica de primer orden. Alloy posee una sintaxis simple y clara, fácil de comprender. Alloy Analyzer es una herramienta de verificación que implementa un análisis automático de las especificaciones Alloy, basado en análisis de satisfactibilidad booleana (SAT). El problema de satisfacibilidad de una fórmula en lógica proposicional consiste en encontrar una asignación de valores de verdad para las variables, de tal forma que la misma se se evalúe como verdadera.
En Alloy los tipos de datos están definidos mediante *Signaturas*. Alloy permite definir restricciones, predicados y aserciones. Mediante la palabra reservada *fact*, es posible establecer propiedades que se asumen válidas en todos los modelos. Por otro lado también es posible definir predicados y aserciones. En en el primer caso para verificar la existencia de instancias de modelos que cumplan con una especificación, y en el segundo caso para verificar que una propiedad se cumple para todo modelo generado dentro de los scopes definidos para cada signatura. En la figura 1(a) se presenta, a modo de ejemplo, una especificación Alloy para listas simplemente encadenadas.

2.3. DynAlloy

Alloy es un lenguaje conveniente, simple y expresivo para construir modelos de software *estáticos*. Para el caso del modelado de sistemas *dinámicos* que capturan la ejecución de sistemas mediante, cambios de estado, Alloy no posee una forma sencilla para representar este comportamiento. DynAlloy [9,10] es una extensión de Alloy que incorpora constructores que permiten capturar fácilmente la noción de cambio de estado. La sintaxis y la semántica de DynAlloy está basada sobre lógica dinámica. DynAlloy extiende Alloy con *acciones básicas*, *programas*, y *aserciones de corrección parcial*. Las acciones básicas son definidas a través de pre y pos condiciones. Por ejemplo, una acción que elimina todos los elementos de una lista simplemente encadenada podría definirse como `removeAll` en la figura 1 (b). Esta acción atómica actualiza el `head` y el `size` de la lista utilizando el operador relacional (`++`) de sobre-escritura. Por un lado la acción `removeAll` posee los campos de `List`, `head` y `size` con parámetros explícitos, en lugar de utilizar `this` se utiliza `thiz` (ya que la primera es una palabra reservada en Alloy). Los programas DynAlloy se construyen utilizando acciónes de asignación (`:=`), `skip`, tests (acción skip con guardas, `[expr]?`) y acciones atómicas como casos base, combinadas mediante la aplicación de composición secuencial (`;`), elección no determinística (`+`) e iteración (`*`).

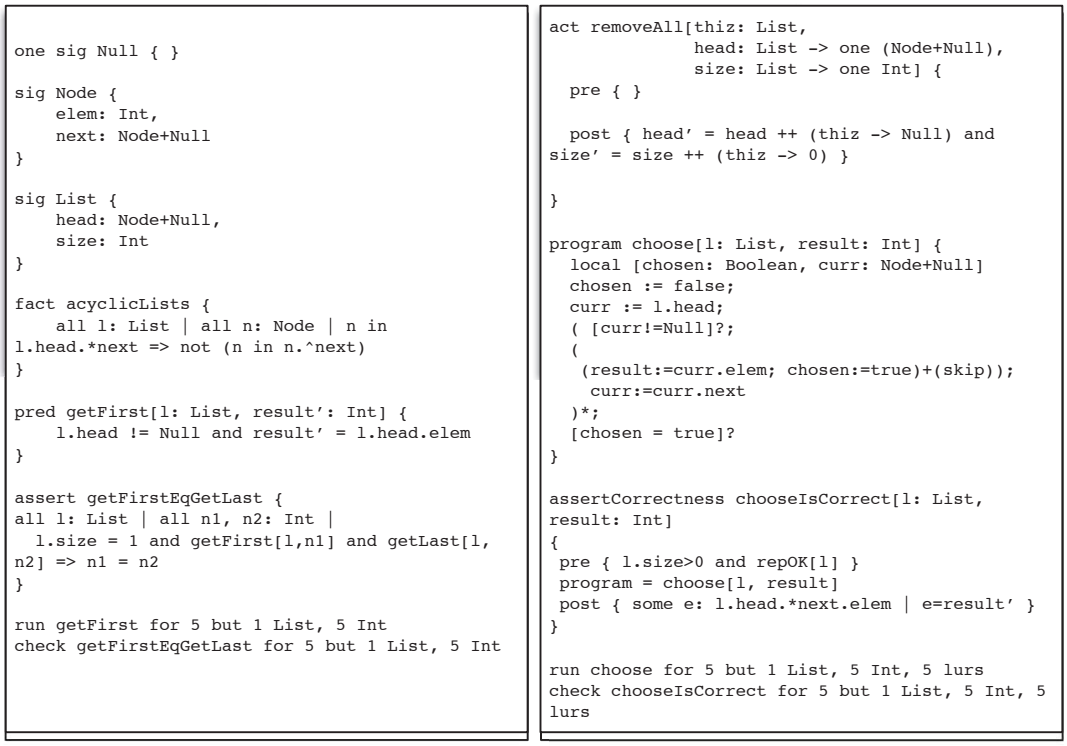
Alloy y DynAlloy son lo suficientemente expresivos como para modelar abstracciones de programas Java[11] y especificaciones JML, y se han utilizado como lenguajes intermedios para diversos análisis, incluída la verificación acotada y la generación de casos de tests de programas Java con anotaciones JML [12].

3. Cómputo de *state-bound workarounds*

En [6] se propuso una técnica de cómputo automática de *state-bound workarounds*. Este procedimiento de búsqueda utiliza el estado corriente del programa como información para encontrar una solución alternativa ante una eventual falla. Esta técnica es aplicable a programas Java anotados con especificaciones JML (Java Modeling Language)[7]. El procedimiento de búsqueda de *state-bound workarounds* se realiza a través de un programa DynAlloy generado la partir de las acciones atómicas DynAlloy mapeadas de las especificaciones JML de los métodos del módulo. Dada una clase C , y $m_1, m_2, \ldots, m_k$ métodos públicos en C, donde cada método m_i posee su contrato JML, pre_{m_i} y $post_{m_i}$, las acciones DynAlloy $a_1, a_2, \ldots, a_k$, correspondientes a cada uno de los métodos $m_1, m_2, \ldots, m_k$ son utilizadas para generar el programa de búsqueda DynAlloy. Cada acción atómica a_i es definida como:

$$\texttt{act}\ a_i\ \{\texttt{pre}\{\ pre^A_{m_i}\ \}\,\texttt{post}\{\ post^A_{m_i}\ \}\}.$$

El programa de reparación modela todas las posibles composiciones secuenciales de las acciones $a_1, a_2, \ldots, a_k$. El operador $*$, es interpretado como un bucle de cero o más iteraciones [10], por lo que al momento de ejecutar DynAlloy Analyzer será necesario fijar este valor de cota superior de iteraciones o *loops*



```
one sig Null { }

sig Node {
    elem: Int,
    next: Node+Null
}

sig List {
    head: Node+Null,
    size: Int
}

fact acyclicLists {
    all l: List | all n: Node | n in
l.head.*next => not (n in n.^next)
}

pred getFirst[l: List, result': Int] {
    l.head != Null and result' = l.head.elem
}

assert getFirstEqGetLast {
all l: List | all n1, n2: Int |
  l.size = 1 and getFirst[l,n1] and getLast[l,
n2] => n1 = n2
}

run getFirst for 5 but 1 List, 5 Int
check getFirstEqGetLast for 5 but 1 List, 5 Int
```

(a)

```
act removeAll[thiz: List,
              head: List -> one (Node+Null),
              size: List -> one Int] {
  pre { }

  post { head' = head ++ (thiz -> Null) and
size' = size ++ (thiz -> 0) }

}

program choose[l: List, result: Int] {
  local [chosen: Boolean, curr: Node+Null]
  chosen := false;
  curr := l.head;
  ( [curr!=Null]?;
  (
   (result:=curr.elem; chosen:=true)+(skip));
    curr:=curr.next
  )*;
  [chosen = true]?
}

assertCorrectness chooseIsCorrect[l: List,
result: Int]
{
 pre { l.size>0 and repOK[l] }
 program = choose[l, result]
 post { some e: l.head.*next.elem | e=result' }
}

run choose for 5 but 1 List, 5 Int, 5 lurs
check chooseIsCorrect for 5 but 1 List, 5 Int, 5
lurs
```

(b)

Figura 1. Especificaciones Alloy and DynAlloy para listas enlazadas.

unrolls. Este valor determinará el alcance o *scope* máximo de composiciones de acciones, es decir, la cantidad de acciones que podrá contener el *workaround*. Bajo el suspuesto de que un método m_i de la clase produce una falla en tiempo de ejecución en un estado concreto s_i (modelado con un predicado Alloy s_i^A) se contruye la siguiente aserción de corrección parcial:

$$\{\ s_i^A\ \}\ (a_1 + a_2 + \ldots + a_{i-1} + a_{i+1} + \ldots + a_k) *\ \{\ \neg post_{m_i}^A\ \}$$

La aserción será ser verificada automáticamente utilizando DynAlloy Analyzer. Un contra-ejemplo encontrado que no satisfaga la aserción anterior estará compuesta por una secuencia de estados $s_{A_0}, \ldots, s_{A_j}$ tal que:

- s_{A_0} es un estado s_i^A;
- existe una secuencia $a_{p(1)}; a_{p(2)}; \ldots; a_{p(j)}$ de operaciones tales que $\langle s_{A_i}, s_{A_{i+1}} \rangle$ están relacionados por la relación de transición $a_{p(i)}$; y
- s_{A_j} es un estado s_f^A que *no* satisface $\neg post_{m_i}^A$, es decir , que satisface $post_{m_i}^A$.

Dado que s_i^A y $post_{m_i}^A$ son predicados Alloy que caracterizan los estados s_i y la pos condición del método m_i, respectivamente, cada contra-ejemplo será en efecto un *workaround* formado por una secuencia de acciones, vinculadas a los métodos de la clase C, que producirán una transición de estados del programa desde s_i con el propósito de arribar a un estado que satisfaga $post_{m_i}$. En el caso de que DynAlloy Analyzer no encuentre un contra-ejemplo a la aserción anterior, dentro del *scope* previsto, no será garantía de que no existe un *workaround* en scopes superiores. El procedimiento de búsqueda se realiza de manera incremental, para poder determinar la alternativa de menor tamaño (en cantidad e acciones involucradas). A continuación, y a modo de ejemplo, se muestran algunos *state-bound workarounds* encontrados para listas simplemente encadenadas. En este caso se supone que un programa Java, al ejecutar una línea $Integer\ v = lista.set(0, 4)$, la cual debería almacenar en la variable v el valor contenido en la posición 0 de la lista y luego almacenar en esa posición 4, produce un incumplimiento en su contrato. En el cuadro siguiente se muestran algunos de los *state-bound workarounds* encontrados en función del número de iteraciones(*).

Invocación a reparar: Integer v = lista.set(0,4)	
Estado inicial: {lista = [10,5,20]}	
Estado final: {v=10 , lista= [4,5,20]}	
Nro. de Iteraciones (*)	state-bound workarounds
1	*No existen workarounds de longitud 1.*
2	*Integer v= lista.remove(0) ; lista.addFirst(4);*
	Integer v= lista.remove(10) ; lista.addFirst(4);
	lista.addFirst(4) ; Integer v = remove(1);
	...
3	...
4	...
5	*lista.clear() ; lista.addFirst(20); lista.addFirst(5) ; lista.addFirst(4) ; Integer v= lista.get(0)*

Cuadro 1. State-bound workaronds para un escenario de falla sobre listas

4. Búsqueda de *workaround templates* y *state-free workarounds*

La reparación de fallas utilizando *state-bound workaronds* posee un condimento adicional de complejidad vinculado a los parámetros de invocación. Luego de analizar los *state-bound workaronds* obtenidos en la evaluación experimental de la técnica presentada en la sección 3, se pudieron identificar *workarounds templates*. Éstos establecen soluciones generalizables fijando la secuencia de acciones de invocación y dejando "libres" los parámetros a ser completados por un

SAT-Solver. Se propone a continuación un procedimiento que permite identificar a un *workaround templates* como un *state-bound workaronds*, es decir que pueda ser utilizable para cualquier estado. Esto también se realiza aplicando técnicas de SAT-Solving. El procedimiento está dividido en las siguientes etapas:

- Etapa 1: Se aplica el algoritmo de cómputo de *state-bound workarounds* para el método m obteniendo como resultado un *state-bound workaround wkt* .
- Etapa 2: Se evalúa si *wkt* puede ser generalizable a un *workaround template*. Para ello se fija la secuencia de las acciones contenidas en *wkt* y se "liberan" los parámetros de los métodos involucrados. El SAT-Solver (Alloy Analyzer), buscará los valores apropiados para alcanzar la pos condición del método m.
- Etapa 3: Para verificar si *wkt* es un *workaround template* válido, se generan una serie de escenarios aleatorios en donde se asume una falla en m y se intenta repararlo utilizando el template derivado de *wkt*.
- Etapa 4: Como se mencionó anteriormente, un *workaround templates* es un *state-bound workaround* cuya "confianza" es brindada por la prueba del template bajo escenarios generados aleatoriamente. A pesar de que este mecanismo brinda confianza acerca del las posibilidades de reparación del *workaround templates*, no asegura una reparación efectiva para todos los casos. Es decir, no es un *state-free workaround*. Para promover a este *workaround templates* a la categoría de *state-free workaround* se utiliza Alloy Analyzer, en este caso. El SAT-Solver intentará generar un estado (dentro de los alcanzables) que satisfaga la precondición del método asociado con el template y que al ejecutar el mismo (dando la posibilidad de que el SAT-Solver defina los parámetros actuales) no satisfaga la pos condición del método. Es decir, buscará generar un estado inicial para el cual ejecutando el template no se repare la falla. En caso de encontrarse el contra-ejemplo el *workaround templates* no podrá ser promovido a *state-free workaround* dentro de los scopes definidos. En caso de que no se encuentre un contra-ejemplo, el *workaround templates* se incluirá en una lista de *state-free workaround* los cuales serán invocados en un primer momento a la hora de reparar una falla.

El cómputo de *workaround templates* permite encontrar soluciones alternativas que mejoran el rendimiento de la técnica de búsqueda de *state-bound workaround*. Este cómputo es incorporado a la técnica presentada en [6]. Ante un escenario de falla, el algoritmo de búsqueda de reparaciones buscará inicialmente en la lista de los *state-free workaround* y *workaround templates* para intentar reparar el problema. En la sección 5 podemos ver algunos resultados obtenidos al utilizar el procedimiento de búsqueda y la mejora en los tiempos de reparación utilizando *workaround templates* y *state-free workaround*.

5. Evaluación de la propuesta

En en cuadro 2, se muestran algunos de los *workaround templates* identificados para métodos de la clase `java.util.TreeSet`, el valor $True$ en la columna *Condidición de Aplicabilidad* indica que el templates es un *state-free workarounds*. En el cuadro 3 se presenta la aceleración obtenida por la técnica al incorporar *workaround templates* independientes de estado.

Cuadro 2. Ejemplos de *state-bound workarounds* y *workaround templates* encontrados para `java.util.TreeSet`

Método	State-Bound Workaround	Workaround Template	Condición de Aplicabilidad
add(e)	{1,7,10}.add(10)=={1,7,10}.contains(-1)	s.add(X):=s.contains(Y)	X belongs to s
ceiling(e)	{1,7,10}.ceiling(10)=={1,7,10}.floor(10) {1,7,10}.ceiling(8)=={1,7,10}.floor(11) {1,7,10}.ceiling(20)=={1,7,10}.floor(-1)	s.ceiling(X):=s.floor(Y)	True
contains(e)	{2,4}.contains(5)=={2,4}.isEmpty() {2,4}.contains(4)=={2,4}.add(5);{2,4,5}.remove(5)	s.contains(X):=s.isEmpty() s.contains(X)==s.add(Y);s.remove(Z)	s not empty X in s
first()	{2,3,4}.first()=={2,3,4}.floor(2)	s.first():=s.floor(X)	True
floor(e)	{1,7,3,10}.floor(10)=={1,7,3,10}.ceiling(10) {1,7,3,10}.floor(8)=={1,7,3,10}.ceiling(5) {1,7,3,10}.floor(-1)=={1,7,3,10}.ceiling(100)	s.floor(X):=s.ceiling(Y)	True
higher(e)	{2,3,5}.higher(4)=={2,3,5}.floor(5) {2,3,5}.higher(-10)=={2,3,5}.floor(-1)	s.higher(X)==s.floor(Y)	True
last()	{2,3,4}.last()=={2,3,4}.floor(4)	s.last():=s.floor(X)	True
lower(e)	{-1,3,4}.lower(2) == {-1,3,4}.floor(-1) {1,3,24}.lower(-1) == {1,3,24}.floor(-2)	s.lower(X)==s.floor(Y)	True
pollFirst()	{2,4,7}.pollFirst()=={2,4,7}.first();{2,4,7}.remove(2)	s.pollFirst():=s.first();s.remove(X)	True

6. Conclusiones y trabajos futuros

El concepto de *workaround* ha sido utilizado en técnicas de reparación de programas a través de diversos enfoques. En trabajos anteriores hemos propuesto una técnica de cómputo automático de *state-bound workarounds*. Esta técnica

Cuadro 3. Aceleración lograda usando wokarounds templates para `Java.util.TreeSet`

`Treesets`: 136 estructuras analizadas.; tam. mín.: 11, tam. máx.: 22, Tam. Promedio:13.16			
Método a reparar	Tiempo Promedio de rep.sin templates	Tiempo de rep. con templates	Aceleración
add	0:00:42	0:00: 03	14
ceiling	0:00:43	0:00: 13	3.31
clear	0:00:44	0:00: 02	22
contains	0:00:44	0:00: 02	22
first	0:00:28	0:00: 06	4.67
floor	0:00:43	0:00: 12	3.6
higher	0:00:27	0:00: 13	2.1
is_empty	0:00:22	0:00: 02	11
last	0:00:30	0:00: 06	5
lower	0:00:22	0:00: 07	3.14
poll_first	0:00:44	0:00: 05	8.8
remove	0:00:15	0:00:07	8.5

es aplicable a programas Java anotados con especificaciones JML y utiliza SAT-Solving como mecanismo de búsqueda. La reparación de fallas utilizando *state-bound workarond* posee un condimento adicional de complejidad vinculado a los parámetros de invocación. Se observó que los *state-bound workaronds* pueden ser generalizables liberando los parámetros de invocación. A esos workarounds genéricos los denominamos *workaround templates*. En este trabajo se propuso un mecanismo de cómputo automático de *workaround templates* independientes de estado (*state-free workaronds*). Los resultados obtenidos en la evaluación experimental han sido muy positivos. La utilización de *state-free workaround* permite mejorar notablemente el rendimiento de la técnica de búsqueda de *workarounds*. En el cuadro3 observamos una aceleración promedio de reparación muy significativa, las mismas van de 3.31 a 22 veces más rápida utilizando *workaround templates* comparado con el enfoque tradicional. Por otro lado, la detección de *workaround templates* independientes de estado (*state-free workaronds*) módulo intervención del SAT-Solver para fijar los parámetros de invocación, brinda la posibilidad de computar previamente workarounds permanentes, independientemente de los tiempos requeridos. Este proceso de SAT se lleva a cabo en una instancia previa, y pueden ser almacenados en un repositorio para su utilización. Como trabajos futuros, se preve continuar con optimizaciones aplicables a la técnicas de análisis de SAT utilizadas. Se espera incrementar los límites de exploración alcanzados hasta el momento.

Referencias

1. A. Carzaniga, A. Gorla and M. Pezzè, *Self-healing by means of automatic workarounds*, in Proceedings of 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems SEAMS 2008, Leipzig, Germany, May 12-13, ACM, 2008.
2. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *Automatic Workarounds for Web Applications*, in Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering FSE 2010, Santa Fe (NM), USA, ACM, 2010.

3. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *RAW: runtime automatic workarounds*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010, New York (NY), USA, ACM, 2010.
4. A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino and M. Pezzè, *Automatic recovery from runtime failures*, in Proceedings of the 35th International Conference on Software Engineering ICSE 2013, San Francisco (CA), USA, IEEE/ACM, 2013.
5. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications*, ACM Trans. Softw. Eng. Methodol. 24(3), ACM, 2015.
6. Marcelo Uva, Pablo Ponzio, Germn Regis, Nazareno Aguirre, Marcelo F. Frias: *Automated Workarounds from Java Program Specifications Based on SAT Solving*. FASE 2017: 356-373.
7. P. Chalin, J. R. Kiniry, G. T. Leavens and E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, in Proceedings of 4th International Symposium on Formal Methods for Components and Objects FMCO 2005, LNCS 4111, Springer, 2005.
8. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
9. J. Galeotti and M. Frias, *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proceedings of Software Engineering Techniques SET 2006: Design for Quality, Warsaw, Poland, IFIP 227, Springer, 2006.
10. M. Frias, J. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proceedings of International Conference on Software Engineering ICSE 2005, St. Louis, Missouri, USA, ACM, 2005.
11. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification and Object-Oriented Design*, Addison-Wesley, 2000.
12. J.P. Galeotti, N. Rosner, C. López Pombo and M. Frias, *Analysis of Invariants for Efficient Bounded Verification*, in Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, ACM, 2010.