# Hacia la Construcción Automática de *Workarounds* para Reparar Programas

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

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina. Email: {uva,naguirre}@dc.exa.unrc.edu.ar

[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

**Abstract.** En este trabajo se propone una técnica de reparación automática de programas basada en el concepto de *workaround*. Esta técnica aprovecha la redundancia que usualmente se encuentra en sistemas basados en componentes, para intentar construir automáticamente una implementación alternativa a una rutina con fallas, que aproveche la interfaz de la componente en la cual ésta se encuentra. La técnica requiere que la componente esté especificada formalmente con *contratos* (pre y post condiciones, invariantes de clase), y garantiza la corrección de la solución construida en ámbitos acotados, dado que se basa en el uso de mecanismos de verificación acotada.
A diferencia de otros enfoques basados en workarounds, la técnica aquí presentada se aplica directamente a código, y construye workarounds permanentes, en el sentido de que los mismos reparan las rutinas a las cuales se aplican para todos los estados (acotados) de ejecución de la misma, y no para entradas o estados particulares.
Además de presentar la técnica, se mostrará un caso de estudio simple, en el cual se evalúa de manera preliminar las capacidades de la misma.

## 1 Introducción

El crecimiento de los sistemas de software a nivel mundial, y la dependencia de actividades humanas en estos sistemas, ha generado un especial interés en las áreas de investigación vinculadas a garantizar la calidad del software, en particular en lo relativo a confiabilidad. Dentro de estas áreas, los métodos formales abarcan un amplio espectro de enfoques que, fundados en sólidas bases matemático-lógicas, apuntan a *garantizar* la corrección del software. En este contexto, tradicionalmente se ha estudiado más en profundidad los mecanismos que ayudan a verificar sistemas, o en su defecto a detectar fallas en el software. Esto último ha ganado popularidad en las últimas décadas, pues se ha puesto mucho interés y esfuerzo en el estudio y desarrollo de técnicas que permitan asistir, de manera automática, en el proceso de descubrimiento de defectos. Algunas de éstas han derivado en poderosas herramientas, como SLAM [1] y Java PathFinder [2], que son capaces de detectar errores en programas de manera automática. Incluso algunas forman parte del set de herramientas elementales para

el desarrollo de ciertas aplicaciones, como es el caso de SDV (Static Driver Verifier) [1]. Estas actividades son sumamente relevantes para la industria del software: se estima que dentro de las fases del desarrollo de un proyecto de software, la cantidad de tiempo y esfuerzo que requieren las actividades orientadas a detectar y corregir defectos pueden llegar a insumir hasta un 50% del esfuerzo total del proyecto. Si bien se han producido enormes avances en el análisis automático de software, el esfuerzo puesto en aplicar estas técnicas para la *reparación* de programas es significativamente menor al puesto en la detección de defectos y la verificación. En este trabajo proponemos una técnica de reparación automática de programas basada en el concepto de *workaround*. Esta técnica aprovecha la redundancia que usualmente se encuentra en sistemas basados en componentes, para intentar construir automáticamente una implementación alternativa a una rutina con defectos, que aproveche la interfaz de la componente en la cual se encuentra. La técnica requiere que la componente esté especificada formalmente con *contratos* (pre y post condiciones, invariantes de clase, etc), y garantiza la corrección de la solución construida en ámbitos acotados, dado que se basa en el uso de mecanismos de verificación acotada. Más precisamente, esta técnica utiliza DynAlloy[3], una extensión al lenguaje formal Alloy[4], que permite utilizar análisis basado en satisfactibilidad booleana para la verificación acotada de programas. Es decir que, indirectamente, nuestra técnica de construcción de workarounds está basada en SAT solving. El concepto de workaround, que tomamos en este trabajo, no es nuevo. Varios enfoques vinculados a sistemas self-healing lo aprovechan. En estos enfoques, los workarounds se construyen a partir de modelos simplificados de software (construidos de manera manual); además, los workarounds son reparaciones a rutinas que fallan, pero para un estado específico, aquel en donde la falla ocurrió; es decir, corrigen un problema en el sistema en un momento específico, pero la solución podría no ser válida en otros casos en que la misma rutina falle. A diferencia de estos enfoques, nuestra técnica se aplica directamente a código fuente, y construye workarounds *permanentes*, en el sentido de que los mismos reparan las rutinas a las cuales se aplica para todos los estados (acotados) de ejecución de la misma, y no para entradas o estados particulares.
Este trabajo está organizado de la siguiente manera: en la sección 2 se introducen los lenguajes Alloy, Dynalloy y la herramienta de análisis Taco[5], utilizada en este trabajo. En la sección 3 se presenta el concepto de *Workaround* junto con algunas de sus aplicaciones. Nuestra técnica es presentada en la sección 4. En la sección 5 se muestra un caso de estudio simple, en el cual evaluamos de manera preliminar las capacidades de la técnica. Este caso de estudio corresponde a la reparación de un programa Java, ubicado en una clase equipada con contratos especificados en JML (Java Modeling Language) [6]. Finalmente, en la sección 6, se exponen conclusiones y trabajos futuros.

## 2 Alloy, DynAlloy y TACO

Alloy [4] es un lenguaje de especificación basado en lógica relacional, una extensión de la lógica de primer orden que soporta operadores relacionales: imagen relacional, clausura, traspuesta, etc. La sintaxis de Alloy es simple y semejante a la de un lenguaje orientado a objetos. El lenguaje fue diseñado con el objetivo de soportar análisis automático (Alloy Analyzer) permitiendo la simulación de modelos y la búsqueda de contraejemplos a propiedades provistas por el usuario. Alloy es considerado un *método formal liviano*. El análisis realizado por Alloy se basa en la reducción a fórmulas proposicionales, y en la utilización de un SAT-solver para verificar satisfactibilidad. Alloy es un lenguaje orientado a modelos, originalmente diseñado para especificar propiedades de estados de sistemas de software. En la Figura 1 se muestra un modelo alloy para listas simplemente encadenadas.

```
one sig Null {}        sig Node {              sig List {
                           next:Node+Null,          head:Node+Null
                           value:Int           }
                       }
```

**Fig.1** Especificación de listas simplemente encadenadas en Alloy

En [7], Jackson propone especificar propiedades dinámicas en Alloy introduciendo manualmente la noción de traza de ejecución como parte de la especificación. Un problema con este enfoque es que la noción de traza de ejecución depende de las signaturas y operaciones de la especificación original, y debe ser definida de manera ad-hoc para cada modelo particular. DynAlloy [3] es un lenguaje que ofrece una alternativa para el problema anterior, mediante la incorporación de una sintaxis (y una semántica) para lidiar con ejecuciones, inspiradas en la lógica dinámica. DynAlloy extiende Alloy con la noción de acción atómica, y agrega operaciones para realizar composiciones de acciones. Esto, sumado a la posibilidad de describir aserciones de correción parcial, provee una forma más sencilla y clara para la especificación de propiedades dinámicas. En la Figura 2 se presenta, a modo de ejemplo, la definición de la acción dynalloy *removeFirst* encargada de eliminar el primer elemento de una lista. En la Figura 3 se muestra la especificación de la aserción *removeAll* que establece un contrato y un programa a verificar. Dynalloy se encargará posteriormente de verificar si el programa definido en la aserción cumple o no con el contrato. El programa definido dentro de *removeAll* puede ser pensado como *while (head(l) != Null) do removeFirst(l)*. Este programa, no requiere ninguna condición previa a ejecutarse y, una vez que el programa termine, la lista debe estar vacia.

```
action removeFirst[l: List] {
     pre { l.head != Null }
     post { l'.head = l.head.next }
}
```

**Fig.2** Especificación de la acción atómica *removeFirst* en Dynalloy

```
assertCorrectness removeAll[l:List]{
    pre = { }
    prg = {([l.head != Null]?;removeFirst(l))*;[l.head = Null]?}
    post = { l'.head = Null }
}
```

**Fig.3** Ejemplo de aserción en Dynalloy

TACO (Translation of Annotated COde) [5] es una herramienta de análisis, de código abierto, que verifica estáticamente y de manera automática la corrección de programas Java. TACO recibe como entrada un programa Java anotado con aserciones JML [6]. JML es un lenguaje de especificación para programas Java, que permite la definición de precondiciones, postcondiciones e invariantes de clase, siguiendo el paradigma de Diseño por Contratos [8]. TACO verifica la conformidad de un programa Java contra su especificación JML, mediante la utilización de técnicas de verificación acotada: todas las ejecuciones de un procedimiento son examinadas exhaustivamente hasta un límite en cantidad de objetos de cada clase (*scope*) y límite de iteraciones, provistos por el usuario. Para realizar las verificaciones, TACO ejecuta una secuencia de transformaciones del programa Java anotado produciendo modelos que posteriormente serán verificados por las herramientas Alloy y Dynalloy.

## 3 Workaround

El concepto de *Workaround* surge dentro del área de los sistemas *Self-Healing*. Este tipo de sistemas posee la capacidad de auto-recuperarse. Ante la detección de una falla el propio sistema, provee los mecanismos para su corrección. Un workaround aprovecha la redundancia implícita en el propio sistema. Dado un estado inicial *Ei*, un programa $P$ y un estado final *Ef*, un workaround define una secuencia alternativa de acciones *P'*, cuya semántica es equivalente a la del programa $P$. Las especificacines algebráicas proveen herramientas eficientes para modelar componentes de software, y a partir de los cuales derivar workarounds. Otras técnicas para detectar workarounds utilizan especificaciones basadas en estados finitos. Se aplican diferentes algoritmos para la búsqueda de caminos desde un estado inicial a un estado final, generando así trazas de ejecución equivalentes. En los trabajos [9, 10] se presenta una arquitectura de monitoreo y recuperación de fallas utilizando Workarounds en tiempo de ejecución, lo que los autores denominan *automatic workarounds*. Como resultado de estos trabajos, el grupo ha desarrollado una herramienta denominada RAW(Runtime Automatic Workarounds) [11] sobre la cual se han realizado una serie de experimentos logrando detectar workarounds en sistemas tales como Google Maps[12] y Flickr[13].
En nuestro trabajo, como se explica en la sección 4 se utiliza el concepto de workaround, de una manera diferente. Se utiliza como una técnica de debugging, es decir, se aplica en tiempo de desarrollo.

# 4 Workaround Permanentes (WaP)

El proceso de debugging se ha convertido en una de las actividades más importantes dentro del proceso de desarrollo de un sofware. Se define debugging[14] como la búsqueda, aislamiento y correción de defectos del software. Este proceso comprende tres actividades: reproducción del error, determinación de posibles causas y aislamiento de las mismas, y finalmente, corrección de la falla. Como fue mencionando en la introducción de este artículo, se estima que dentro de en un proyecto de software, la cantidad de esfuerzo que requieren las actividades orientadas a detectar y corregir fallas pueden llegar a insumir hasta un 50% del esfuerzo total del proyecto y más aún para proyectos clasificados como críticos. En este trabajo se realizan aportes tendientes a lograr automatizar la etapa de debugging haciendo uso del concepto de workaround.
La propuesta concreta consiste en definir un mecanismo que encuentre una solución alternativa ante la presencia de una falla. Para ello es necesario contar con los contratos del programa y de las acciones involucradas.

Dado un escenario en donde durante la etapa de codificación de un programa $P$={$Q$}( $acción_i$)+{$R$}, el programador ha introducido (involuntariamente) una falla, provocando que el contrato {Q,R} no se verifique. Se ha detectado que el error ha sido ocasionado por la ejecución de la acción $acción_k$ contenída en $P$. Para dar solución a este error se propone, utilizando herramientas autmáticas basadas en análisis de satisfactibilidad, un programa *P'* que no contenga a $acción_k$. El nuevo programa *P'* deberá verificar el contrato de $P$, y será clasificado como un *workaround permanente*. Se define *Workaround Candidato (WaC)* a un workaround que verifica el contrato de $P$, pudiendo ser dependiente de un estado particular. Éste es el concepto de workaround utilizado en los sistemas *self-healing*. Por otro lado, un *Workaround Permanente (WaP)* se define como un workaround que verifica el contrato de $P$ independientemente del contexto. Un *WaP* permitirá dar una solución alternativa y permanente a una falla. El siguiente problema a resolver es cómo encontrar el *WaP P'*. En la subsección 4.1 se presenta un esquema del mecanismo propuesto para la detección de *WaPs*.

## 4.1 Búsqueda de workarounds permanentes utilizando TACO y Dynalloy

El procedimiento de búsqueda de *WaPs* toma como entrada un programa Java con sus respectivos contratos JML, y parte del supuesto de que en un método se ha producido un error. En la Figura 4 ilustramos el procedimiento propuesto.

- ⋆ La primera actividad a realizar para encontrar un *WaP* que permita dar solución a la falla consiste en la de traducción de un programa Java $P$ a un modelo Dyalloy[3] utilizando la herramienta TACO. El contrato de $P$ es {$Q$,$R$}.

- ⋆ Al modelo dynalloy obtenido en el punto anterior, se le incorpora el programa dynalloy $P_{wac}$, con el contrato {$Q$,$\neg R$} para poder encontrar los WaC. $P_{wac}$

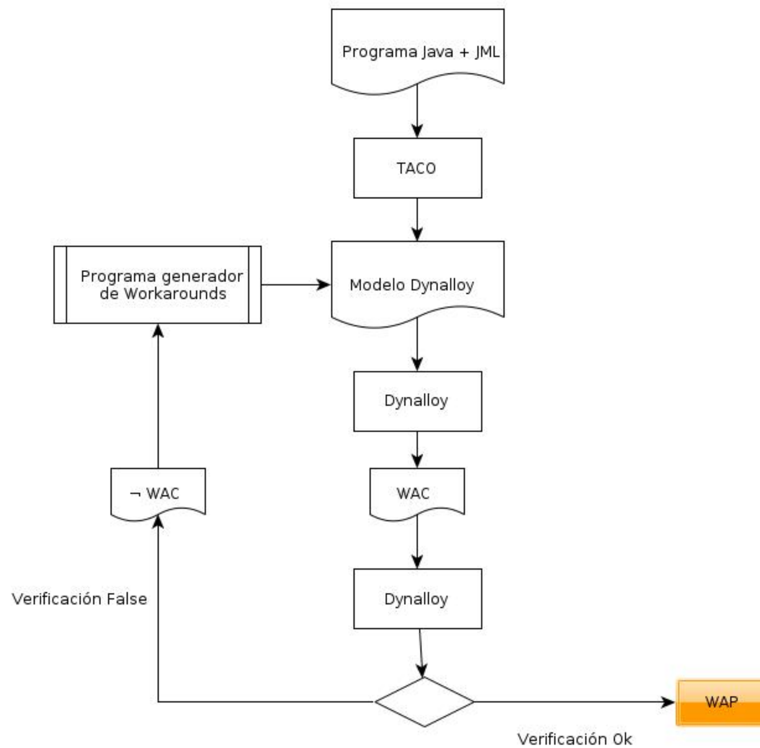


**Fig.4** Procedimiento de búsqueda propuesto

está definido por una composición paralela de todas las acciones disponibles en el modelo.

$$P_{wac} = (acción_1 + ... + acción_n)^* \text{ , donde } acción_k \notin P_{wac}.$$

- ⋆ Dynalloy intentará verificar la corrección de $P_{wac}$. Para ello, tomará el contrato $\{Q,\neg R\}$ e intentará construir un $WaC$ combinando todas de acciones disponibles en el modelo y dentro del alcance establecido. Si Dynalloy no encuentra un contraejemplo indicará que todos los programas posibles de construir verifican el contrato $\{Q,\neg R\}$, por lo que ninguno cumple con el contrato $\{Q,R\}$. Es decir, no existirá (dentro del alcance definido) un programa que pueda ser utilizado alternativamente ante la presencia de un error.

- ⋆ En el caso de que en el punto anterior, Dynalloy, hubiese encontrado un contraejemplo $WaC_i$ para $P_{wac}$, el paso siguiente será verificar si este $WaC$ es independiente del ambiente, es decir, si es un $WaP$ que pueda ser utilizado para solucionar la falla. Para realizar esta operación, se utilizará nuevamente Dynalloy. Esta vez, intentando verificar el $WaC_i$ con el contrato $\{Q,R\}$. Si

Dynalloy no encuentra ningún contrajemplo, indicará que $WaC_i$ es un $WaP$ y que puede ser utilizado para dar sulución a la falla. En caso contrario, existirá al menos un ambiente para el cual $P_{wac}$ no verifica el contrato $\{Q,R\}$, por lo que será descartado como un $WaP$, e incorporado al programa $P_{wac}$ de tal forma de que al repetir todo el proceso nuevamente sea excluído, puediendo encotrar un $WaC_j$ diferente con posibilidades de ser un $WaP$.

## 5 Caso de Estudio

Para ilustrar nuestra propuesta, mostraremos el caso de estudio *lista simplemente encadenada* de enteros implementada sobre arreglos. Recordemos que la técnica propuesta se fundamenta en la idea subyacente de aprovechar al máximo la redundancia implícita contenida en los sistemas computacionales.
El programa *ListaEnlazada.java* contiene el código fuente con sus respectivos contratos. A continuación listaremos los perfiles de los métodos de la clase.

```
//  addFirst, inserta un entero al comienzo de la lista.
/*@
  @ requires (this.size < this.elementos.length);
  @ ensures this.elementos[0] == val;
  @ ensures  (\forall int i;
             i>0 && i< (this.size) ;
             this.elementos[i]== \old(this.elementos[i-1]));
  @ ensures \old(this.size)+1 == this.size;
  @ ensures this.header == 0;
  @*/
 public void addFirst(int val){...}

//  Inserta un entero al final de la lista.
 public void addLast(int val){....}


//  clearList, elimina todos los elementos de la lista.
/*@
  @ requires this.size > 0;
  @ ensures  this.size== 0;
  @*/
 public void clearList(){...}


// sizeList, retorna la cantidad de elementos de la lista.
 public int sizeList(){...}


// isEmptyList, indica si la lista esta vacia.
 public boolean isEmptyList(){...}
```

```
// removeFirst, elimina el primer elemento de la lista.
 public void removeFirst() {...}

// remplace, remplaza el elemento val en el indice index.
 public void remplace(int index, int val) {...}

// getFirst, retorna el primer elemento de la lista.
 public int getFirst() {...}


// add, inserta el elemento val en el indice index.
 public void add(int index, int val){...}

/*@
  @ requires this.size > 0;
  @ ensures  this.size== 0;
  @*/
 public void principal(){
clearList();
  }

// removePairAtHead, elimina el primer elemento de
// la lista siempre que este sea un elemento que
// contenga a un nro. par.
/*@
  @ requires this.size > 0 ;
  @ requires (this.elementos[0]%2)==0;
  @ ensures  (\forall int i;
             0 <= i && i< this.size;
             this.elementos[i]== \old(this.elementos[i+1]));
  @ ensures  this.size == \old(this.size)-1;
  @*/

 public void removePairAtHead() {... }

}
```

Para simplificar el ejemplo sólo hemos incluído los contratos de algunos de los métodos. El método *removePairAtHead* fue agregado para incrementar la redundancia del caso de estudio y mostrar de una forma más clara la diferencia entre *WaC* y *WaP*. Bajo la suposición de que en el método *principal* se hubiese detectado una falla al utilizar el método *clearList*, la técnica propuesta deberá encontrar un *WaP* que no utilice *clearList*. Siguiendo el diagrama planteado en la sección anterior, el primer paso a realizar es la traducción de *ListaEnlazada.java*

a un modelo dynalloy. Para ello utilizamos la herramienta TACO obteniendo el modelo *ListaEnlazada.dals*. A cotinuación, incorporamos al modelo dynalloy, el programa $P_{wac}$ para la búsqueda de los WaCs.

$$P_{wac} = (addFirst + addLast + removeFirst + add + remplace + removePairAtHead)^*$$

El primer $WaC$ encontrado por Dynalloy durante el proceso de búsqueda es $WaC_1$= *removePairAtHead*. Posteriormente, se verifica si éste es un $WaP$. Dynalloy responde que $WaC_1$ no es un $WaP$ ya que es dependiente del contexto, lo cual es cierto ya que sólo resuelve el problema para listas formadas por un número par. Para el caso de aquellas listas compuestas por un número impar, el $WaC_1$ no da solución a la falla por lo que es descartado como un $WaP$. $WaC_1$ es incorporado al modelo *ListaEnlazada.dals* para evitar encontrarlo en la próxima búsqueda. Al ejecutar nuevamente Dynalloy, este encuentra a $WaC_2$=*removeFirst;removeFirst* el cual sí es verificado posteriormente como un $WaP$, es decir, es independiente del contexto.
Este caso de estudio tiene un tamaño pequeño, a pesar de ello hemos tenido que acotar a 4 la cantidad de loops unrolls de Dynalloy. Fue realizado en una notebook Dell Inspiron 1525, procesador Dual Core Intel con 2 GB de RAM. Uno de los puntos a estudiar es como lograr escalar la técnica para que sea aplicable a ejemplos de mayor complejidad y con tiempos de respuesta razonables para una tarea de debugging.

## 6 Conclusiones y trabajos futuros

La detección y corrección de errores en las etapas finales de un proceso de desarrollo de software insume un alto porcentaje de los recursos asignados al proyecto. Por otro lado, la utilización de Métodos Formales para la realización de tareas vinculadas a garantizar calidad en sistemas informáticos estaba más bien relegada sólo a aquellos grupos altamente formados en áreas tales como Álgebra, Lógica y análisis matemáticos complejos. En la década del 90' surgieron proyectos tales como SLAM y Java PathFinder (entre otros) los cuales instalaron la idea de Métodos Formales Livianos. Fruto de estos proyectos surgieron una serie de herramientas que posibilitaron extender técnicas formales a un gran dominio de proyectos de desarrollo. Nuestra propuesta realiza aportes orientados en esta misma lína de trabajo, vinculados a detectar y corregir los defectos en el software utilizando técnicas de análisis automático y permitiendo optimizar los resursos.

Nuestro objetivo final es el desarrollo de una herramienta que asista al programador en la etapa de debugging. Uno de los puntos que habitualmente se critican a las herramientas basadas en métodos formales es que exigen la definición de un modelo que preserve las propiedades del problema original. Nuestra propuesta se aplica directamente a código fuente y de manera automática se genera el modelo inicial, lo que evita el problema de la construcción del modelo por parte del

usuario. Por otro lado, las especificaciones JML exigidas establecen una descripción formal del comportamiento de las clases y los métodos. La aplicación del concepto de *workarround* durante la etapa de debugging es novedosa en el sentido de que combina los conceptos de los sistemas *self-healing* con la aplicanción de herramientas provenientes del campo de los métodos formales.

Actualmente se está trabajando en la aplicación de esta técnica con ejemplos de tamaño superior al planteado en este trabajo. Se analizan una serie de propiedades, donde la *escalabilidad* es una de las prioritarias a observar. Al mismo tiempo se está trabajando en la contrucción de un prototipo que implemente automáticamente todo el proceso descripto en el trabajo.

## References


1. T. Ball, B. Cook, V. Levin and S. K. Rajamani, “SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft”, Integrated Formal Methods 2004. Lecture Notes in Computer Science, 2004.
2. http://javapathfinder.sourceforge.net/. Robust Software Engineering Group at the NASA Ames Research Center.
3. Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, Nazareno Aguirre: DynAlloy: upgrading alloy with actions. ICSE 2005: 442-451
4. Jackson, D. Alloy: a lightweight object modeling notation. ACM Transactions on Software Engineering and Methodology 11, 2, 2002, 256–290.
5. TACO: Translation of Annotated COde.
$http://www.dc.uba.ar/inv/grupos/rfm_folder/TACO$
6. The Java Modeling Language (JML). $http://www.eecs.ucf.edu/\ leavens/JML/$
7. Jackson,D. “Software Abstractions: Logic, Language, and Analysis”, The MIT Press, 2006.
8. Meyer, Bertrand: “Design by Contract, in Advances in Object-Oriented Software Engineering.”, D. Mandrioli and B. Meyer, Prentice Hall, 1991.
9. A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, ”Automatic Workarounds for Web Applications,” in FSE’10: Proceedings of the 2010 Foundations of Software Engineering conference, New York, NY, USA, 2010, pp. 237-246
10. A. Carzaniga, A. Gorla, and M. Pezzè, ”Healing Web applications through automatic workarounds,” International Journal on Software Tools for Technology Transfer (STTT), vol. 10, iss. 6, pp. 493-502, 2008.
11. A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, ”RAW: runtime automatic workarounds,” in ICSE’10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Tool Demo), New York, NY, USA, 2010, pp.
12. Google Maps. https://maps.google.com/
13. Flickr. http://www.flickr.com/
14. Andreas Zeller. ”Why Programs Fail”.Elsevier ISBN 978-0-12-374515-6. U.S.