Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación


Tesis de Licenciatura en
Ciencias de la Computación

# Hacia un modelo más flexible para la implementación de la auto reparación de sistemas de software basada en Arquitectura

Resumen extendido

**Director: Santiago Ceria**



| Alumno | LU | Correo electrónico |
|---|---|---|
| Chiocchio, Jonathan | 849/02 | jchiocchio@gmail.com |
| Tursi, Germán Gabriel | 699/02 | gabrieltursi@gmail.com |

# 1. Introducción

La complejidad creciente de los sistemas desafía de forma permanente el estado del arte de las Ciencias de la Computación y la Ingeniería del Software. La velocidad con la que se producen los cambios, la criticidad de las fallas que aparecen y la necesidad de mantener sistemas funcionando de manera continua a pesar de no pertenecer a lo que tradicionalmente se conoce como "sistemas de misión crítica" ha llevado a los investigadores a buscar novedosas formas de resolver estos desafíos. Una de ellas es la tendencia hacia los sistemas autónomos, que recibe distintos nombres como "Computación Autómona", "Software consciente" o "Sistemas Auto–Reparables" (o "Self Healing" en inglés). Existe una cantidad en aumento de especialistas en el mundo [1] que creen que esta necesidad de implementar este tipo de mecanismos está dando lugar al nacimiento de una nueva era en los sistemas de información.

La idea subyacente detrás de estos nombres es que los sistemas incluyan mecanismos para ajustar su comportamiento a partir de fallas o necesidades cambiantes de sus usuarios y/o el entorno en el que operan. De esta forma, un sistema puede repararse u optimizarse sin intervención humana. Una de las formas de implementar estos mecanismos es la llamada "Adaptación Basada en Arquitecturas". En este tipo de soluciones, el sistema tiene un módulo que conoce su arquitectura, y, en base a este conocimiento y al problema detectado, toma una cierta decisión. Uno de los grupos más activos en investigación de estos temas es el proyecto **Rainbow**[2], enmarcado dentro del proyecto **Able**[3] de la Universidad de Carnegie Mellon (UCM) en Estados Unidos.

La adaptación en Rainbow es lograda a través de varios módulos que colaboran para lograr este objetivo. Estos son:

- **Monitor**: es el módulo que se encarga de obtener la información sobre el funcionamiento del sistema en tiempo de ejecución.
- **Evaluador de Restricciones**: es el que determina si el valor de alguna de las variables que se están monitoreando violó alguna de las restricciones planteadas (por ejemplo, que la *performance* de un proceso dejó de ser aceptable).
- **Modelo de Arquitectura**: es el módulo que contiene una representación en un ADL (Architecture Description Language) de la arquitectura del sistema que se quiere adaptar.
- **Manejador de Reparaciones**: es el módulo que se ocupa de determinar la forma en la que se va a "reparar" o "adaptar" el sistema en función de los problemas detectados.
- **Intérprete**: es el módulo que interpreta los cambios ocurridos en tiempo de ejecución y los "traduce" a cambios en el modelo de arquitectura.
- **Administrador de Runtime**: es el módulo que implementa en tiempo de ejecución el cambio en el comportamiento de la aplicación.

Todos estos mecanismos funcionan de manera externa a la aplicación. Este enfoque tiene varias ventajas, siendo la principal el hecho de contar con un *framework* reusable que pueda ser conectado a distintos tipos de aplicaciones para que implementen mecanismos de adaptación, minimizando el impacto en la aplicación.
El caracter externo de Rainbow representa una ventaja también cuando se desea implementar auto reparación en sistemas cuyo código fuente no está disponible o no es plausible de ser modificado.

A pesar de intentar implementar un mecanismo genérico de auto-reparación, Rainbow tiene varios componentes con conocimiento fijo[1] sobre las reparaciones. Por ejemplo, cuáles son las tácticas para la reparación que se deben implementar cuando una determinada restricción es violada.

### 1.1. Breve ejemplo de auto-reparación

Dado que estos sistemas y sus conceptos son relativamente novedosos, proponemos un pequeño ejemplo de auto-reparación para afirmar conceptos:

- Supongamos una aplicación Web que brinda servicios a millones de usuarios. Es crítico que el tiempo de respuesta ante un pedido de una página se mantenga dentro de rangos razonables.
- El sistema está compuesto por varias páginas, siendo la más crítica su página principal. Esta página está formada por varias "partes", cada una con su respectiva funcionalidad.
- El *Monitor* implementa un mecanismo de monitoreo del tiempo de respuesta del sistema ante un pedido, y lo informa al *Intérprete*, el que a su vez se encarga de traducir dicha información en cambios en las propiedades del sistema. e.g.
  "el tiempo de respuesta fue 4.300 ms. $\Rightarrow$ `client.experiencedResponseTime` $\leftarrow$ `4300`"
- El *Evaluador de Restricciones* determina si las restricciones del sistema siguen cumpliendose o no. Cuando no se respeta el tiempo máximo durante una predeterminada cantidad de veces, decide implementar una auto-reparación.
- El *Manejador de Reparaciones*, en función de su conocimiento de la arquitectura del sistema, decide **desactivar cierta funcionalidad de la página principal**, resignando funcionalidad para ganar en *performance*.
- Ese cambio se implementa a través del *Administrador de Run-Time*.
- Al desactivar esa funcionalidad, la performance del sistema mejora.

[1] Más conocido comúnmente como "hardcodeado" o "cableado".

## 2. Alcance de la tesis

La idea de esta tesis es extender el *framework* Rainbow para poder lograr una implementación más flexible y poderosa del mecanismo de auto-reparación, ofreciendo a su vez la posibilidad de hacer visible dicho mecanismo a los *stakeholders* de la aplicación, permitiéndoles involucrarse en la definición de escenarios uso real del sistema y su relación con los atributos de calidad requeridos para el mismo; y en la definición de prioridades y/o estrategias a considerar en la auto-reparación del sistema.

Esto se pretende lograr con cambios importantes en varios de sus módulos:

- Actualmente Rainbow posee conocimiento sobre la arquitectura del sistema a adaptar mediante su modelo de arquitectura (expresado en el ADL Acme[4]). Uno de los objetivos de este trabajo es extender dicho conocimiento, para eso, en el módulo de *Modelo de Arquitectura*, se implementarán las siguiente mejoras:

  1. Incluir información sobre los atributos de calidad que la arquitectura implementa y que son relevantes *para el o los stakeholders del sistema* en tiempo de ejecución. Por ejemplo, poder describir la importancia (relativa) de la *performance*, la usabilidad, la disponibilidad, etc.
     Nuestro enfoque para lograr esto consiste en especificar **Escenarios de Atributos de Calidad** (de ahora en más, simplemente "Escenarios"), un concepto ya especificado por ATAM[5] [6] (Architecture Tradeoff Analysis Method) en el contexto de los "Workshops de Atributos de Calidad" (Quality Attribute Workshops QAW[7]).
     Un Escenario modela una situación concreta y real de uso del sistema ante la cual el mismo debe comportarse de una manera esperada. Los escenarios están compuestos por la siguiente información:

     - Fuente del Estímulo
     - Estímulo
     - Artefacto
     - Entorno
     - Respuesta
     - Cuantificación de la Respuesta

     De todos estos atributos, son particularmente importantes el **Artefacto** y la **Cuantificación de la Respuesta**.

     El **Artefacto** se refiere al componente, subsistema o parte del sistema afectada por el escenario. Dado que en Acme se especifican los componentes y conectores del sistema, el escenario tendría entonces una vinculación directa en la especificación con los componentes afectados.

     La **Cuantificación de la Respuesta** es también importante ya que de ella surgen las restricciones que deben ser evaluadas por el *Evaluador de Restricciones*. Al hacer

estos cambios, las restricciones que se usaban anteriormente en Rainbow pasarían a ser en realidad una parte de un Escenario de Atributo de Calidad.

2. Asignar prioridades a estos escenarios, para que a la hora de escoger una estrategia de auto-reparación, se tengan en consideración otros aspectos del sistema (especificados como atributos de calidad) de modo tal que la estrategia de reparación elegida no comprometa a alguna otra funcionalidad de la aplicación considerada más importante para el usuario.
3. ~~Relacionar estos Escenarios (en realidad los artefactos afectados por el escenario) con los modelos de la arquitectura usando la terminología de ATAM[5] (Architecture Tradeoff Analysis Method), para indicar cuáles son los puntos sensibles y los puntos de tradeoff dentro de la arquitectura para los distintos escenarios. De esta manera, el *Manejador de Reparaciones* puede saber dónde debe cambiar el sistema si un escenario no se está cumpliendo.~~

- El módulo *Evaluador de Restricciones* se extenderá para que evalúe la información de restricciones, presentes en la cuantificación de la respuesta, asociadas a los Escenarios que figuren como activos en el modelo de arquitectura.
- En el módulo *Manejador de Reparaciones*, utilizar el conocimiento plasmado en los escenarios para optar por la estrategia de reparación a aplicar, teniendo en cuenta los puntos sensibles, los puntos de tradeoff relacionados y las prioridades de los escenarios, con el objetivo de, además de reparar el inconveniente hallado, evitar perjudicar algún otro escenario de mayor prioridad.
- Una última idea que puede resultar interesante es la de asociar las reparaciones a los escenarios. Por ejemplo, un escenario podría incluir información del estilo "Si este escenario se ve comprometido, implementar tal reparación". Esto agrega la ventaja de que ahora los problemas y las soluciones (estrategias), gracias a los escenarios, pueden ser visibles a los usuarios y stakeholders de la aplicación. Dichas estrategias (podrían ser más de una, ordenadas por prioridad al igual que los escenarios) también deberían declarar que otros concerns perjudican, para que el *Manejador de Reparaciones* pueda contemplar tradeoffs y decidir la mejor estrategia que no perjudique otros escenarios con dichos concerns.

En resumidas palabras, el alcance del trabajo consiste en:

- Definir las siguientes extensiones:
  - Posibilidad de modelar escenarios.
  - Posibilidad de relacionar escenarios con componentes de la arquitectura.
  - Posibilidad de especificar **Puntos Sensibles** y **Puntos de Tradeoff** en las relaciones entre un escenario y los componentes y conectores que son afectados por él.
  - Posibilidad de modelar reparaciones y asociarlas a Escenarios.

- Proponer los cambios en el *Manejador de Reparaciones* y el *Evaluador de Restricciones* de Rainbow, e implementar algunos casos prácticos que permitan mostrar cómo esta estrategia puede funcionar y llevar a un Framework de Adaptación más flexible y poderoso. Esto tiene como contrapartida que los usuarios tendrán más control sobre lo que el sistema decida hacer para auto-repararse: sólo con modificar la información de los escenarios el sistema modificará su comportamiento. Por otro lado, las extensiones propuestas en este trabajo hacen que el sistema se comporte de manera más autónoma a medida que se agregan escenarios. Esto paradójicamente le quita control al usuario, ya que el procedimiento de auto-reparación se vuelve más complejo, complicando el seguimiento de las decisiones tomadas por el framework.

## 3. Trabajo a futuro

Se propone como trabajo a futuro extender la herramienta AcmeStudio para que dé soporte a las extensiones propuestas en el presente trabajo, permitiendo así modelar escenarios mediante una herramienta gráfica, lo que sería muy útil para stakeholders no técnicos. Dicha herramienta es la más utilizada hoy en día para diseñar gráficamente arquitecturas basadas en el ADL Acme.

## Referencias

[1] Ganek, Alan G. y Corbi, Thomas A. The dawning of the autonomic computing era. IBM Syst. J., 42(1):5-18, 2003. ISSN 0018-8670.
http://www.cs.cmu.edu/~garlan/17811/Readings/ganek.pdf

[2] Proyecto Rainbow de la Universidad de Carnegie Mellon.
http://www.cs.cmu.edu/~able/research/rainbow/

[3] Proyecto Able de la Universidad de Carnegie Mellon.
http://www.cs.cmu.edu/~able

[4] Proyecto Acme de la Universidad de Carnegie Mellon.
http://www.cs.cmu.edu/~acme

[5] ATAM: Method for Architecture Evaluation, Software Engineering Institute (SEI)
http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm

[6] The Architecture Tradeoff Analysis Method (ATAM), Software Engineering Institute (SEI)
http://tinyurl.com/ye5ub9l

[7] Quality Attribute Workshops (QAWs), Third Edition, Software Engineering Institute (SEI)
http://www.sei.cmu.edu/library/abstracts/reports/03tr016.cfm