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

**Director: Santiago Ceria**



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

**Resumen**

Deber incluir un resumen en Castellano e Inglés no más de 200 palabras en donde se exponga con claridad los objetivos y los alcances del trabajo.

# 1. Introducción

Breve referencia del trabajo mencionando la parte novedosa del trabajo.

## 1.1. Self Healing

Self Healing, incluyendo una descripción previa sobre la complejidad creciente de los sistemas y su necesidad de non-stop que justifica este enfoque. Reusar parte de la tesina de Casuscelli de la UCA que está bastante bien escrita (y no vale la pena gastar más tiempo en esto).

A medida que va pasando el tiempo, los sistemas de software se vuelven cada vez más complejos y más exigentes en cuanto a disponibilidad se trata. Hoy en día ya operan en ambientes muy dinámicos, con requerimientos que el usuario va cambiando rápidamente, y con la necesidad de seguir operando prácticamente sin interrupción, resultando esto en un mayor crecimiento en la administración operativa del software, lo cual representa un costo importante para que el sistema pueda funcionar. Para reducir este costo, se espera que los sistemas se adapten de manera dinámica para poder utilizar los recursos existentes, para poder atender los cambiantes requerimientos del usuario y los errores en el sistema. Esto es lo que se conoce como **self healing** (auto reparación).

Si bien ya existen mecanismos para mitigar dichos problemas, los mismos son parte del lenguaje de programación utilizados, tales como tratamiento de excepciones, protocolos de tolerancia a fallos, etc. Pero estos mecanismos son comúnmente dependientes de cada aplicación y generalmente muy atados al código. Así, la adaptación del sistema hoy en día es costosa de construir, dificil de modificar, y generalmente solo provee soluciones a fallos de manera puntual.

Uno de los primeros investigadores en acuñar el término Self Healing fue el Dr. David Garlan, de la Carnegie Mellon University, quien formó un grupo de investigación que dedicó años a estudiar el tema dentro del marco del proyecto Able. El presente trabajo utilizará como base gran parte del trabajo generado por dicho proyecto, los cuales se describirán en detalle más adelante.

## 1.2. Descripción de Sistemas basados en Componentes

Un problema fundamental de las arquitecturas de los sistemas basados en componentes ha sido encontrar la notación apropiada para definir dichos sistemas.

Un buen lenguaje para la descripción de arquitecturas permite generar una documentación clara sobre los componentes del sistema, que luego sirvirá como base a los desarrolladores, permitiendo a su vez razonar sobre las propiedades del sistema y automatizar su análisis y hasta puede llegar a utilizarse para la generación automática del sistema.

Una forma de describir dichas arquitecturas es mediante el modelado de objetos, si bien este método ha sido ampliamente aceptado y utilizado en la industria, tiene varios inconvenientes, el más importante y bloqueante es que no proveen un soporte directo para describir propiedades no funcionales, esto hace dificultoso razonar sobre propiedades críticas del sistema, como son la

performance y confiabilidad. Ésta es la razón principal que ha motivado el avance de los ADLs (Architecture Description Language).[GMW99]

La descripción de arquitecturas de sistemas basada en ADLs ha avanzado muchísimo en las últimas 2 décadas, al punto de que ya permiten definir una base formal para la descripción y el análisis de los mismos.

## 1.3. Auto reparación de sistemas basándose en modelos de arquitectura

Este enfoque ha sido propuesto por el Dr. Garlan, el cual intenta basar la auto reparación en el modelo de la arquitectura del sistema, en lugar de los mecanismos tradicionales para detección y recuperación de errores que se implementan a nivel de código.

La idea propuesta consiste básicamente consiste un *closed-loop* (bucle cerrado), donde existe una capa que actúa como mecanismo de control y otra encargada de monitorear el sistema (ver figura), separada del sistema que está siendo ejecutado, lo cual provee una solución mas efectiva que cualquier mecanismo interno porque agrupan lo concerniente a la detección y solución del problema en módulos separados, pudiendo ser analizados, modificados, extendidos y reutilizados a través de distintos sistemas.

agregar /images/selfhealing-closed-loop.png

El proyecto Rainbow, también del proyecto Able, utiliza esta técnica de *closed-loop* para monitorear y reparar sistemas, más adelante profundizaremos sobre Rainbow.

## 1.4. Acme

Acme es uno de los ADLs más reconocidos y utilizados, ha sido desarrollado en la Carnegie Mellon University, más precisamente por el proyecto Architecture Based Languages and Environments (Able), liderado por el Dr. David Garlan.

Acme es un pilar fundamental dentro del proyecto Able, ya que todo el proyecto gira en torno a la arquitectura de software de los sistemas, y es Acme quien permite describir formalmente dichas arquitecturas, por lo tanto todos los restantes subproyectos dentro de Able utilizan Acme en menor o en mayor medida.

Además de los benecifios de todo ADL, el lenguaje Acme y su kit de herramientas proveen las siguientes capacidades fundamentales:

- Intercambio Arquitectural: al proveer un formato de intercambio genérico para diseñar arquitecturas de software, Acme permite a los desarrolladores de herramientas de Arquitectura de Software integrar fácilmente con otras herramientas complementarias basadas también en Acme. De esta manera, los arquitectos que usan herramientas basadas en Acme tiene un espectro más amplio de herramientas de análisis y diseño que quienes diseñan sus herramientas usando otros ADLs.

- Extensibilidad: Acme provee una base sólida y extensible y una infraestructura que evita que los desarrolladores de herramientas deban modificar la infraestructura sobre la cual

trabajan.

agregar ejemplo de ACME?

## 1.5. Rainbow

### 1.5.1. Introducción a Rainbow

La herramienta Rainbow, desarrollada también dentro del marco del proyecto Able, tiene como finalidad permitir reducir el costo e incrementar la confiabilidad al realizar cambios en sistemas complejos, para esto Rainbow automatiza la adaptación de sistemas dinámicos via el modelo de la arquitectura del sistema, representaciones explícitas de las tareas del usuario, y estimaciones orientadas a la performance en tiempo de ejecución.

La tecnología en la cual se apoya Rainbow esta basada en innovaciones en tres áreas críticas:

- Detección: la habilidad de determinar dinámicamente, o sea en run-time, propiedades de sistemas complejos y distribuidos.
- Resolution: la habilidad de determinar cuándo las propiedades observadas del sistema violen los supuestos críticos del diseño del sistema.
- Adaptación: la habilidad de automatizar la adaptación del sistema en respuesta a las violaciones de los supuestos del diseño del sistema.

Éstas habilidades proveerán:

- La posibilidad de manipular los cambios en el sistema con respecto a estimaciones soportadas por Rainbow.
- Un framework extensible para manipular estimaciones y estrategias de adaptación creadas por terceros.

Tomando todo esto en conjunto, el objetivo de Rainbow es reducir drásticamente la necesidad de intervención humana en la adaptación de los sistemas ante cambios en el ambiente para alcanzar los objetivos de calidad requeridos, aumentando así la confianza en los cambios al proveer sistemas autoreparables.

hablar sobre closed-loop: In contrast to these internal mechanisms, recent work uses external models and mechanisms in a closed-loop control fashion to achieve various goals by monitoring and adapting system behavior at runtime

### 1.5.2. Tácticas y Estrategias

Definiciones del libro de Bass y Clements. Explicar la idea de una "táctica para atributo de calidad"(ejemplo, redundancia para disponibilidad o late binding para anticipación al cambio)

La estrategia es la herramienta propuesta por Rainbow para quitar al sistema de un estado indeseable. Por ejemplo, al verse la performance del sistema comprometida, una estrategia podría agregar servidores hasta llevar la performance a un nivel adecuado.

Las estrategias son desarrolladas para un *concern* específico. Un *concern* de un sistema abarca un subconjunto de propiedades del sistema, como por ejemplo performance, costo o confiabilidad. En una arquitectura cliente-servidor, el *concern* performance abarcaría las propiedades ancho de banda y carga del sistema, mientras que el *concern* costo se encargaría del costo de los servidores.

Las estrategias se componen de tácticas, las cuales realizan la modificación en el sistema. Por ejemplo, una estrategia podría ser: agregar servidores mientras haya disponible, o hasta que el tiempo de respuesta haya alcanzado un determinado valor. Esta lógica sería descripta en la estrategia, mientras que va a ser la táctica levantar-servidor la responsable de ejecutar la acción propiamente dicha.

### 1.5.3. ZNN: Testeando Rainbow

Dentro del proyecto Rainbow se ha desarrollado un caso de estudio para validar el enfoque propuesto, dicho proyecto ha recibido el nombre de ZNN, que cumple la función de simular un sitio web de noticias.

ZNN provee un entorno de simulación de una arquitectura cliente servidor ampliamente configurable que permite representar situaciones y controlar las variables de simulación permitiendo modificarlas en cualquier punto de la simulación. Por ejemplo, el sistema que se está simulando podría comenzar con solamente 2 clientes, mostrando allí una performance aceptable, pero a partir de un momento dado, se agregan 10 clientes comprometiendo así la performance del mismo. De esta manera permite simular como responderían las estrategias implementadas ante dicha situación.

En el presente trabajo se utlizará el modo simulación para realizar los casos de prueba, tomando las estrategias y tácticas de ZNN. Éstas debieron ser adaptadas para que puedan continuar funcionando con los cambios propuestos en la tesis.

## 1.6. Escenarios de Atributos de Calidad y QAW

Los atributos de calidad representan los requerimientos no funcionales de un sistema, y suelen estar pobremente especificados, o directamente no especificados (“un requerimiento que no es testeable no es implementable”).

Los atributos de calidad pueden ser representados mediante escenarios. Los escenarios están compuestos por:

- **Fuente del estímulo**: Interna o externa
- **Estímulo**: condición que debe ser tenida en cuenta al llegar al sistema
- **Entorno**: condiciones en las cuales ocurre el estímulo

- **Artefacto**: el sistema o partes de él afectadas por el estímulo
- **Respuesta**: qué hace el sistema ante la llegada del estímulo
- **Medición de la respuesta**: cuantificación de un atributo de la respuesta

agregar imagen /images/scenario.jpg

Los escenarios son pequeñas historias que describen una interacción con el sistema, que impacta sobre un atributo de calidad en particular. Por ejemplo, un escenario sobre disponibilidad podría ser:

> "Un proceso del sistema recibe un mensaje externo no anticipado durante un modo de operación normal. El proceso informa al operador y continúa su operación sin caídas."

Este escenario se descompone de la siguiente manera:

- **Fuente del estímulo**: Sistema externo
- **Estímulo**: Mensaje no anticipado
- **Entorno**: Operación normal
- **Artefacto**: Proceso interno
- **Respuesta**: Informar al operador y seguir operando
- **Medición de la respuesta**: sin caídas (downtime)

Los escenarios permiten obtener el punto de vista de un grupo diverso de stakeholders (arquitectos, desarrolladores, usuarios, sponsors, etc). Estos escenarios pueden luego ser utilizados para analizar la arquitectura del sistema e identificar concerns y posibles estrategias para atacar problemas.

fuente de lo que sigue: http://www.sei.cmu.edu/architecture/tools/qaw/index.cfm

Existe una metodología definida por el Software Engineering Institute (SEI) conocida como Quality Attribute Workshop (QAW), cuya principal herramienta son los escenarios. Esta metodología provee un método para identificar los atributos de calidad críticos de la arquitectura de un sistema, tales como disponibilidad, performance, seguridad, etc, que son derivados de objetivos del negocio del sistema. QAW no asume la existencia de una arquitectura del software, sino que fue desarrollado en base a reclamos de los clientes que necesitaban un método para identifar los atributos de calidad importantes, y aclarar requerimientos del sistema antes de que exista la arquitectura del mismo.

En los QAW se organizan reuniones con todos los stakeholders del sistema, en las que se definen los escenarios que en definitiva representarán los requerimientos de atributos de calidad que el sistema idealmente deberá satisfacer. Un vez definidos todos los escenarios, el siguiente

paso del QAW consiste en priorizar y refinar los escenarios, por ejemplo agregando detalles adicionales tales como los participantes involucrados, la secuencia de actividades, y preguntas sobre el requerimiento que representa cada escenario. El proceso de refinar los escenarios permite a los stakeholders comunicarse entre ellos, exponiendo supuestos que pueden no ser tan claros para el resto del grupo. Dicho refinamiento también proporciona una visión de cómo interactúan los atributos de calidad entre sí, sirviendo de base para definir tradeoffs entre estos atributos.

El proceso QAW termina con la lista de escenarios refinados y priorizados. Los mismos pueden servir para definir casos de pruebas, o como semillas para el proceso ATAM.

## 1.7. ATAM

fuente: svn/doc/atam/ATAM Method for Architecture Evaluation.pdf

ATAM (Architecture Tradeoff Analysis Method) ha sido desarrollado por el SEI, al igual que QAW, y es una técnica que permite analizar arquitecturas de software. Las arquitecturas son complejas, e involucran muchos tradeoffs de diseño. Sin un proceso de análisis formal, no se puede estar seguro de que las decisiones de arquitectura tomadas - en particular aquellas que afectan el cumplimiento de requerimientos de calidad como performance, disponibilidad, seguridad y modificabilidad - son adecuadas para mitigar los riesgos.

El nombre ATAM proviene no solo del hecho de que refleja cuán bien una arquitectura satisface objetivos de calidad específicos, sino también del hecho de que provee una visión de cómo esos objetivos de calidad interactúan entre sí, esto es lo que conocemos como tradeoffs.

La meta de evaluar una arquitectura con ATAM es entender las consecuencias de las decisiones arquitectónicas con respecto a los requerimientos de atributos de calidad del sistema. Otro objetivo fundamental de ATAM es determinar si dichos requerimientos pueden ser alcanzados con la arquitectura concebida, antes de destinar grandes cantidades de recursos a la construcción del software.

ATAM es un método estructurado y repetible, ayudando así a que las preguntas correctas sobre la arquitectura sean planteadas de manera temprana en el proyecto, durante las etapas de análisis de requerimentos y de diseño, en las cuales los problemas detectados pueden ser corregidos sin mayores costos. ATAM guía a los usuarios del método (stakeholders) para que busquen conflictos en la arquitectura y soluciones a dichos conflictos.

Cabe aclarar que los QAW han surgido como consecuencia del uso de ATAM, ya que usuarios de éste último solicitaban una herramienta o método que les permitiera identificar los requerimientos y los atributos de calidad más importantes del sistema, pero antes de que existiese la arquitectura sobre la cual ATAM trabajaría.

(Sensitivity points, tradeoffs, etc.)

## 2. Limitaciones de Rainbow

Tabla del Google Doc expandida

# 3. Propuesta

Capítulo 3 de la propuesta. Ojo que vamos a tener que revisarla.
Explicacion de lo que hicimos con mas detalle (funcional).
Explicar concepto de Utility Function.

## 3.1. Selección de la estrategia a aplicar

## 3.2. UI de Escenarios

# 4. Resultados

Ademas de puntas sueltas... Sobre que hablariamos aca?

## 4.1. Temas abiertos para investigación

puntas sueltas: - Cómo seleccionar la mejor estrategia. Lo del ütility function.es limitado. - Cómo incluir algo de aprendizaje sobre el resultado de las reparaciones anteriores (más sofisticado, lo que hay ahora es muy limitado) - Mi idea original de usar para algo los .ªrtifacts"vinculados al escenario - Expandir la idea del "system environment"(tenemos una implementación muy restrictiva, no se pueden solapar e.g. alta carga y super alta carga (a.k.a. hasta las tetas) ). - No alcanza con la UI (y su XML asociado) para configurar 100 % nuestra solución (hace falta configurar cosas por fuera). Por ejemplo?

# 5. Conclusiones

Alcances de los resultados obtenidos, ventajas y desventajas, futuros trabajos, etc.

## Bibliografía

[GIO/82] Gioan A. “Regularized Minimization Under Weaker Hypotheses”, applied Mathematics Optimization, Springer Verlag, Volumen 8 numero1 - pag 59-68.1982.

[GMW99] Garlan, D., Monroe R. T., Wile D. “Acme: Architectural Description of Component-Based Systems´´.

ver svn/doc

Libros de Arquitecturas (Ver biblio de IS2)

Artículos del SEI de ATAM y QAW

Tesis de Owen Chen

Todos los artículos que nos pasó Garlan sobre Self Healing

Tesis del flaco de la UCA