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

1.1. Motivación para este trabajo

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.

1.2. Acme

Acme[4] es un simple y genérico ADL (Architecture Description Language), es decir, un lenguaje para describir arquitecturas de sistemas de software, utilizando la vista de C&C. (componentes y conectores) **Acme** está desarrollado por el grupo **Able**[3] de la Universidad de Carnegie Mellon (UCM) en Estados Unidos y fue pensado para proveer un formato común de intercambio entre distintos programas de diseño de arquitecturas; así también como para servir de base a futuros programas de este tipo.

1.3. Rainbow

Uno de los grupos más activos en investigación en materia de auto-reparación es el proyecto **Rainbow**[2], enmarcado dentro del proyecto **Able** de la Universidad de Carnegie Mellon (UCM) en Estados Unidos que, como mencionamos anteriormente, también desarrolla **Acme**.

Rainbow es un *framework* (desarrollado en Java) cuya finalidad es proveer un mecanismo de adaptación para sistemas de software basado en arquitecturas. La adaptación 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 el lenguaje **Acme** 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 y no intrusivo 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.

1.3.1. Ejemplo de auto-reparación con Rainbow

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`"

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

- 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.

1.4. ATAM y QAW

Antes de ahondar en el objetivo de nuestro trabajo, es necesario explicar brevemente dos métodos existentes para analizar y razonar sobre arquitecturas de software y su relación con los atributos de calidad requeridos para un sistema; los cuales utilizaremos posteriormente.

ATAM[5][6] es el acrónimo para "Architecture Tradeoff Analysis Method", es decir, es un método de análisis de arquitecturas que hace foco en el balance que hay hacer al momento de tomar decisiones arquitectónicas para satisfacer ciertos atributos de calidad sin descuidar otros (esto es más conocido en la jerga como hacer "trade-offs") puesto que normalmente es virtualmente imposible satisfacer todos los atributos de calidad al mismo tiempo. (e.g. mas seguridad usualmente conlleva a tener menor *performance*)

Algunos de los objetivos más importantes ATAM son: analizar en una etapa temprana del desarrollo cómo las decisiones arquitecturales satisfacen los atributos de calidad requeridos para un sistema, así también como posibles riesgos, puntos sensibles y trade-offs entre ellas.

Íntimamente ligado al método ATAM aparece otro método llamado **QAW** (Quality Attribute Workshop)[7]. Un QAW es un método facilitado que relaciona los *stakeholders* de un sistema de manera temprana en el ciclo de vida para descubrir los atributos de calidad clave en un sistema de software. En el mismo, los *stakeholders* enuncian, discuten, refinan y priorizan **escenarios** reales de uso de la aplicación, especificando ciertas características de los mismos y lo que es más importante: el atributo de calidad involucrado en dicho escenario. Si bien más adelante ahondaremos en más detalle sobre el concepto de escenario, puesto que es clave en nuestra propuesta de trabajo, un ejemplo de escenario podría ser el siguiente:

Escenario:		**Cuando el sensor de la puerta de un garage detecta a un objeto cerca de la puerta, detiene la misma en menos de un milisegundo.**
Atributos de Calidad relevantes:		Seguridad, performance
Componentes del Escenario	**Estímulo:**	Un objeto está en un radio cercano a la puerta de un garage.
	Fuente del Estímulo:	Un objeto externo al sistema, tal como una bicicleta.
	Entorno:	La puerta del garage está cerrandose.
	Artefacto (si se conoce):	Sensor de movimiento del sistema, componente software de control de movimiento.
	Respuesta Esperada:	La puerta del garage se detiene.
	Medida de la respuesta:	1 (un) milisegundo.

2. Alcance de la tesis

2.1. Idea Básica

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.

Lo antedicho se pretende lograr con cambios importantes en varios de sus módulos. A continuación, el detalle de dichos cambios.

2.2. Cambios en el *Modelo de Arquitectura*

Actualmente Rainbow posee conocimiento sobre la arquitectura del sistema a adaptar mediante su modelo de arquitectura (expresado en **Acme**). 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"), tal cual fueron descriptos en la sección anterior, aunque con algunos agregados de información orientados a la auto reparación.

   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 **Estímulo**, **Artefacto**, el **Entorno** y la **Cuantificación de la Respuesta**.

El **Estímulo** normalmente se asocia a un evento desencadenado en el Artefacto del escenario, que a su vez se encuentra asociado a un componente *runtime* del sistema que se está adaptando.

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.

El **Entorno** condiciona la aplicación del escenario en cuestión a que el sistema se encuentre en un determinado estado. Por ejemplo, se especifica un escenario dónde se dice como el sistema debe responder ante un *request* de determinado tipo, **en un entorno de alta carga**.

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. Asociar las estrategias de reparación 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, gracias a los escenarios, los problemas y sus posibles soluciones (i.e. estrategias de reparación), pueden ser visibles a los usuarios y stakeholders de la aplicación. Dichas estrategias (serían más de una, ordenadas por prioridad al igual que los escenarios) poseen la información necesaria para que el **Manejador de Reparaciones** pueda simular su aplicación y estimar cómo quedaría el sistema luego de haber aplicado dicha estrategia. (estimación de la nueva "utilidad del sistema")

2.3. Cambios en el *Evaluador de Restricciones*

Debido a que, con este nuevo enfoque, todo pasa por el concepto de escenario, es necesario establecer cambios en la lógica de evaluación de restricciones (o invariantes) del sistema. Dichas restricciones ya no estarían más embebidas en los componentes de la arquitectura (modelados con Acme) sino que las *constraints* a evaluar serían aquellas presentes en la cuantificación de la respuesta de cada escenario.

Ante la invocación de un estímulo en el sistema en ejecución, el **Monitor** del sistema informa de esta situación al **Intérprete**, que a su vez actualiza las propiedades del modelo de la arquitectura (recordemos: en Acme) e invoca finalmente al **Evaluador de Restricciones** para que busque aquellos escenarios que posean al estímulo ejecutado y que estén definidos para

el **Entorno** de ejecución actual y que las restricciones asociadas a sus **Cuantificaciones de Respuesta** no se cumplan. Aquellos escenarios que cumplan dichas condiciones serán aquellos escenarios que el Manejador de Reparaciones deberá intentar reparar.

2.4. Cambios en el *Manejador de Reparaciones*

El módulo *Manejador de Reparaciones* se modificará para que utilice el conocimiento plasmado en los escenarios para optar por la estrategia de reparación a aplicar, teniendo en cuenta:

- el estado del sistema actual,
- el atributo de calidad asociado al escenario,
- el entorno de aplicación del mismo y
- las prioridades relativas de los escenarios

El objetivo final es, además de reparar el inconveniente hallado, evitar perjudicar algún otro escenario de mayor prioridad y mediante el uso de heurísticas, poder aproximar la mejor estrategia de reparación a llevar a cabo de modo que la *utilidad del sistema* se maximize.

2.5. Desarrollo de una GUI

Se propone el desarrollo de una interfaz de usuario gráfica (GUI, de sus siglas en inglés: Graphical User Interface) para que los distintos *stakeholders* (incluyendo usuarios y arquitectos) puedan colaborar creando y editando escenarios que luego serán importados y utilizados por Rainbow.

2.6. Resúmen

En resumidas palabras, el alcance del trabajo consiste en:

- Definir las siguientes extensiones:
  - Posibilidad de modelar escenarios de atributos de calidad siguiendo los principios de ATAM y QAW.
  - Posibilidad de relacionar escenarios con componentes de la arquitectura.
  - Posibilidad de especificar prioridades relativas entre escenarios, a ser utilizadas en la elección de la estrategia de reparación a ejecutar.
  - Posibilidad de modelar estrategias de reparación y asociarlas a escenarios.
  - Posibilidad de definir Entornos de ejecución para el sistema, los cuales tengan un papel clave en el algoritmo de elección de estrategias de reparación.
- 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.

Plantear la coexistencia de Rainbow with Scenarios con el Rainbow de siempre? Si hacemos eso, podríamos poner el diagramita "Rainbow Architecture With Scenarios" hecho en Enterprise Architect...

Todo lo antedicho tiene como consecuencia 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. Como contrapartida, 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*.

Por otro lado, luego de la extensión, muchos aspectos del framework de auto-reparación serán más automáticos y no estarán tan fijos como lo están actualmente en Rainbow. Asímismo, se agregan nuevos conceptos que amplían la participación y visibilidad de los usuarios, lo que conlleva múltiples beneficios asociados.

3. Trabajo a futuro

Actualmente existe una herramienta de creación y edición de arquitecturas modeladas en Acme, llamado AcmeStudio[8] la cual se encuentra integrada en la popular herramienta de desarrollo Eclipse[9] como un *plug-in*.

Se propone como trabajo a futuro extender la herramienta AcmeStudio para que dé soporte a las extensiones propuestas en el presente trabajo, permitiendo así la integración del modelado de la arquitectura con el modelado de los escenarios que la complementan, utilizando la misma herramienta.

Enumerar las posibles "puntas"???

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

[8] Acme Studio Tool, Software Engineering Institute (SEI)
http://www.cs.cmu.edu/~acme/AcmeStudio/

[9] Eclipse Platform
http://www.eclipse.org/