Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales

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

Los sistemas auto-reparables, también llamados autónomos, son aquellos que pueden adaptarse dinámicamente a las condiciones cambiantes del entorno (contexto, usuarios, hardware) y a las fallas que puedan producirse, para asegurar su propia estabilidad y utilidad, sin intervención humana.

Existen hoy diversos enfoques en materia de Auto Reparación. Sin embargo, en ninguno de ellos se considera al usuario como un actor crucial en la determinación de requerimientos de auto reparación de un sistema. Con el objetivo de superar esa limitación realizamos una extensión del *framework* "Rainbow", creado por investigadores de la Universidad de Carnegie Mellon. En este *framework* el sistema conoce su arquitectura a través de un modelo creado en un ADL (Lenguaje de Descripción de Arquitectura) y usa ese conocimiento al decidir e implementar la adaptación o reparación. La extensión realizada permite a los *stakeholders* de la aplicación definir cuáles son los requerimientos de atributos de calidad que tiene el sistema y sus prioridades relativas dependientes del contexto de ejecución, así también como sus estrategias y tácticas de reparación asociadas.

En síntesis, en este trabajo establecemos el marco teórico para estudiar el tema, extendemos el *framework* Rainbow para que contemple esta nueva funcionalidad, creamos una herramienta visual para facilitarle al usuario la tarea de configuración y mostramos cómo la flexibilidad introducida enriquece a Rainbow y representa un avance en la idea de lograr sistemas que se adapten y reparen sin intervención humana.

Abstract

Self healing systems, also referred to as Autonomous Systems, are those which can adapt dynamically to changing environments (context, users, hardware) and faults, without human intervention, to ensure stability and utility.

There are currently many different approaches to self-healing. However, none of them considers the users as a critical stakeholder in determining the self healing requirements of a system. With the goal of overcoming this limitation, we implemented an extension to the framework named "Rainbow", created by researchers from Carnegie Mellon University. In this framework, the system knows its architecture through a model created in an ADL (Architecture Description Language) and this knowledge is used when the adaptation or repair has to be decided and implemented. Our extension allows stakeholders to define which are the quality attribute requirements that affect the system and their execution context-dependent relative priorities, as well as their associated repair strategies and tactics.

In summary, in this work we establish the theoretical basis for studying the subject, we extend the Rainbow Framework in order to make it contemplate these new features, we create a visual tool to make it easier for the user to configure these features and we show how the flexibility added makes Rainbow a more powerful framework, and therefore represents and advance in the path to achieving systems that can be adapted and repaired without human intervention.

Índice

Índice de figuras

1. Introducción

1.1. Motivación para este trabajo

La complejidad creciente de los sistemas de software 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 suelen suscitarse 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 Autónoma", "Software consciente" o "Sistemas Auto Reparables" ("Self Healing", en inglés). En sintonía con esta profusión de vocablos, existen distintos términos para referirse a las acciones realizadas por estos sistemas: "Auto Adaptación", "Auto Configuración", "Auto Reparación", etc. En el transcurso del presente trabajo, usaremos los términos "auto reparación" y "auto adaptación" como sinónimos, pese a que ambos poseen semánticas ligeramente diferentes pero que no son relevantes para el alcance de este trabajo. Existe una cantidad en aumento de especialistas en el mundo [GAN/03] que creen que la necesidad de implementar este tipo de mecanismos está dando lugar al nacimiento de una nueva era en los sistemas de software.

La idea subyacente detrás de los nombres antes mencionados 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 Modelos de Arquitecturas". [GAR/02] En este tipo de soluciones, existe un módulo que conoce e interpreta el modelo de la arquitectura del sistema a adaptar y, sobre la base de este conocimiento y el problema detectado, toma una decisión sobre cómo reparar al sistema en cuestión.

Si bien ya existen soluciones de este tipo, en ninguna se considera la participación de los *stakeholders*[1] en el proceso de detección y corrección automática de errores. Esto motivó el presente trabajo, donde se intenta ofrecer una forma de incluir a los *stakeholders* en el proceso, siempre considerando que su participación debe contribuir principalmente en la definición de los potenciales problemas y sus posibles soluciones. Para esto, se consideró el uso de los denominados "Escenarios de Atributos de Calidad" (QAS, de sus siglas en inglés, *Quality Attribute Scenarios*)[BAS/03]; los cuales permiten definir cómo debería responder el sistema, entre otras cosas, ante determinados estímulos y en ciertos entornos de ejecución. Se profundizará sobre el concepto de QAS en la sección 2.5.

Una vez definidos los escenarios por los *stakeholders*, el paso siguiente consiste en utilizar

[1] Típicamente: representantes del cliente, el equipo de arquitectura, representantes de los usuarios finales, operadores y administradores del sistema, *managers*, *testers*, el *sponsor* del proyecto, etc.

toda esa información en tiempo de ejecución para que el sistema sea capaz de auto repararse. Allí nos servimos de un *framework* existente de auto reparación basada en modelos de arquitecturas llamado **Rainbow**.[2] Rainbow propone una manera estática y poco amigable de configurar la información necesaria para tomar decisiones de auto reparación en tiempo de ejecución. Así, el objetivo planteado es superar algunas de dichas limitaciones mediante el agregado del concepto central de este trabajo: los "Escenarios de Atributos de Calidad" como forma fundamental de expresar el conocimiento de los *stakeholders* referido a requerimientos que deben ser contemplados por la arquitectura. El objetivo final que se persigue es involucrar a dichos actores en el proceso de Auto Reparación y proveerles una mayor visibilidad de las posibles consecuencias que dicho proceso conlleva en el desempeño del sistema. El objeto del presente trabajo, al cual denominaremos "Arco Iris", es el extender el comportamiento original de Rainbow modificando ciertos componentes del *framework* para que admitan nuevos conceptos que se explicarán más adelante.

1.2. Organización del presente trabajo

El presente trabajo consta de cinco partes principales, a saber:

1. **Conceptos Preliminares:** en esta sección se introducen todos los conceptos, *frameworks*, herramientas y metodologías necesarias para comprender el trabajo en su conjunto.

2. **Extensión a Rainbow:** aquí se explican en detalle los objetivos de la extensión realizada a Rainbow (llamada "Arco Iris"), su implementación y los problemas que dicha extensión intenta resolver.

3. **Interfaz Gráfica para "Arco Iris":** En esta parte del trabajo se justifica la inclusión de una interfaz gráfica de usuario para facilitar la participación de los *stakeholders* no técnicos; así como también se detallan los alcances y limitaciones de la herramienta, bautizada "Arco Iris UI".

4. **Casos Prácticos:** En esta sección del trabajo se describen y analizan varios casos de prueba del sistema Arco Iris. Dichas pruebas utilizan un sistema ficticio llamado "Znn", desarrollado en una tesis de doctorado de la Universidad de Carnegie Mellon [SHA/08], el cual provee las condiciones de simulación necesarias para poder analizar la extensión realizada a Rainbow de una manera sencilla y efectiva.

5. **Trabajo Futuro y Conclusiones:** En la parte final del presente trabajo se establecen conclusiones con respecto al nivel de cumplimiento de los objetivos que motivaron el trabajo y a la factibilidad de implementación de esta extensión (y, obviamente, de

[2] Página web oficial de Rainbow: http://www.cs.cmu.edu/~able/research/rainbow/

Rainbow) en un sistema real. También se enumeran puntos de continuación, estos son, puntos donde se tomaron decisiones tendientes a acotar el problema pero para los cuales se reconocen posibilidades de continuación, mejora, refinamiento, etcétera.

2. Conceptos Preliminares

2.1. Atributos de Calidad y Restricciones

Desde hace unos años en la bibliografía de la Ingeniería de Software y más específicamente de la Ingeniería de Requerimientos se busca imponer el término **Atributos de Calidad** a las categorías de requerimientos que un sistema debe cumplir. Tradicionalmente los requerimientos eran divididos en **Requerimientos Funcionales** para referirse a la "funcionalidad del negocio" y **Requerimientos No Funcionales** para referirse a otros requerimientos como seguridad, eficiencia, usabilidad, escalabilidad, etc. Sin embargo, esta división resulta poco apropiada, dado que algunos de los llamados **Requerimientos No Funcionales** usualmente se terminan implementando con funcionalidad, como puede ser un módulo de autenticación para lograr seguridad. El término **Atributos de Calidad** aparece entonces como más general, y permite una clasificación más clara entre la funcionalidad y otros atributos. A continuación, algunos ejemplos de requerimientos relacionados con Atributos de Calidad:

1. el sistema debe ser escalable con respecto a la cantidad de usuarios que lo utilizan concurrentemente.
2. el sistema debe implementar políticas de tolerancia a fallos.
3. el sistema debe ser diseñado de manera tal que se minimize el procesamiento y el tiempo de respuesta.

Por otro lado, debemos tener en cuenta que al momento de diseñar una aplicación, el ingeniero de *software* muchas veces está limitado en sus decisiones por distintos factores, a los cuales, en la terminología de la Ingeniería de Requerimientos, se los llama **Restricciones**. Algunos ejemplos de restricciones sobre un sistema de *software* son los siguientes:

4. el código de la aplicación debe ser desarrollado en Java.
5. la base de datos debe ser SQL Server.
6. sólo se utilizarán productos de código abierto (*open-source*).

Es importante conocer los Atributos de Calidad requeridos para un sistema a fin de poder diseñar la arquitectura del mismo. Notar que, los ejemplos de requerimientos de atributos de calidad (1, 2 y 3) se encuentran especificados de manera vaga e imprecisa. Esto es lo que suele ocurrir en la mayoría de los casos en la industria de desarrollo de *software*. Notar también que los requerimientos 1 y 3 pueden llegar a afectarse mutuamente: esto también es muy común y para lograr un buen *tradeoff* entre atributos de calidad los arquitectos suelen tener que tomar un conjunto de decisiones arquitectónicas llamadas **estrategias**, sobre las cuales profundizaremos en la sección 2.3.2.

2.1.1. Atributos de Calidad y *Concerns*

El estándar 1061-1998 de la IEEE [3] que establece una metodología para la definición de métricas con respecto a la calidad del *software*, dice:

> *La calidad del* software *es el nivel que posee de una combinación deseada de atributos (e.g. confiabilidad, interoperabilidad, eficiencia, etc.)*

Algunos ejemplos de atributos de calidad definidos en los estándares IEEE 1061 / ISO 9126 [4] son:

- Eficiencia
- Funcionalidad
- Mantenibilidad
- Portabilidad
- Confiabilidad
- Usabilidad

Imaginemos que el *sponsor* de un sistema a ser desarrollado establece que el sistema debe ser "eficiente". ¿Qué significa esto exactamente? La pregunta es difícil de contestar si no se dispone de más información. Evidentemente, los atributos de calidad son categorizaciones de alto nivel que, si no se dispone de más información, no parecen servir de mucho para tomar decisiones arquitectónicas en pos de alcanzar un nivel aceptable de dichos atributos de calidad. Para superar esa carencia aparecen las denominadas **Incumbencias** o, del inglés y tales como las llamaremos a lo largo de este trabajo, **Concerns**.

Los **Concerns** son parámetros mediante los cuales los atributos de un sistema son juzgados, especificados y medidos. Usualmente, los requerimientos de atributos de calidad son expresados en términos de *concerns*. [BAR/95].

A continuación, enumeraremos algunos ejemplos de *concerns*, junto con el atributo de calidad al que pertenecen:

Atributo de Calidad	***Concerns***
Eficiencia	Comportamiento Temporal, Utilización de Recursos
Funcionalidad	Interoperabilidad, Seguridad
Mantenibilidad	Cambiabilidad, Facilidad de Prueba
Portabilidad	Adaptabilidad, Coexistencia
Usabilidad	Compresibilidad, Atractivo

[3] Para más información, visitar http://standards.ieee.org/findstds/standard/1061-1998.html
[4] Para más información, visitar http://www.sqa.net/iso9126.html

Los *concerns* pueden usualmente relacionarse con propiedades de la arquitectura de un sistema, por ejemplo, en una arquitectura cliente-servidor, el atributo de calidad eficiencia posee varios *concerns* asociados, por ejemplo, el **tiempo de respuesta**, el cuál está relacionado directamente con algunas propiedades de la arquitectura como el ancho de banda de los servidores, la carga del sistema, la cantidad de servidores, etc.

Tanto los atributos de calidad como los *concerns* son conceptos fundamentales en el presente trabajo, no sólo desde un punto de vista teórico sino que también serán utilizados en la práctica.

2.2. Sistemas Autónomos, Auto Adaptación, Auto Reparación...

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 y, hoy en día, operan en ambientes dinámicos, con requerimientos de usuario altamente cambiantes y con la necesidad de operar prácticamente sin interrupción, resultando esto en un aumento en la administración operativa del software, lo cual representa un costo importante para que el sistema pueda mantenerse operativo. Además, la complejidad creciente de los sistemas hace que sea cada vez más difícil para una persona comprender lo que se encuentra ocurriendo en una aplicación cuando deja de brindar sus servicios como era esperado. Para resolver o al menos paliar estos problemas, se puede plantear que los sistemas se adapten de manera dinámica para poder utilizar los recursos existentes, a fin de poder atender los cambiantes requerimientos de atributos de calidad, así también como los errores en el sistema. De forma genérica, a los sistemas de software que cumplen con estas características, se los denomina **Sistemas Autónomos**.

Hilando más fino en la caracterización de sistemas autónomos, encontramos términos en inglés como *Self Configuring* o *Self Adapting* para referirse a sistemas autónomos que tienen la capacidad de auto configurarse (o auto adaptarse) a condiciones cambiantes en el entorno de ejecución.

Por otro lado, cuando la adaptación dinámica del sistema responde a errores o situaciones excepcionales del mismo, el término más utilizado actualmente es *Self Healing* o, en castellano, **Auto Reparación**.

Si bien ya existen mecanismos para mitigar los mencionados problemas, ellos normalmente están intrínsecamente ligados al lenguaje de programación utilizado para construir la aplicación, tales como tratamiento de excepciones, protocolos de tolerancia a fallos, etc. Además, estos mecanismos generalmente dependen del código de la aplicación que se intenta adaptar y consecuentemente, no son fácilmente reutilizables entre distintos sistemas. En resumen: hoy en día, la adaptación de sistemas de software es costosa de construir, difícil de modificar, poco reutilizable y generalmente sólo provee soluciones a fallos de manera puntual.

En cuanto al estado del arte en materia de sistemas autónomos existen diversos enfoques tanto en el ámbito académico como en la industria del software.

Dentro del ámbito de la industria, el concepto de sistemas autónomos se encuentra ampliamente difundido. Sin dudas, el enfoque de IBM, denominado "Autonomic Computing" [HOR/01] es el más completo, apalancado por un gran grupo de investigación y abarcando el problema desde distintos aspectos. También se destaca la iniciativa de Microsoft denominada "Dynamic Systems Initiative", y no tanto la iniciativa de Sun ("Predictive Self-Healing" y "Conscientious Software"), por estar más ligada a adaptar el enfoque al dominio de los sistemas operativos. Para más información sobre estos enfoques, remitirse a [CAS/05]

Por otro lado, existen distintos investigadores ligados a prestigiosas instituciones académicas abocados al estudio de distintos aspectos de la autonomía de los sistemas. Uno de los primeros investigadores en acuñar el término "Self Healing" fue el Dr. David Garlan, de la Universidad de Carnegie Mellon, quien formó un grupo de investigación que dedicó años a estudiar el tema dentro del marco del proyecto ABLE.[5]

El presente trabajo toma como base el trabajo generado por el proyecto ABLE de Carnegie Mellon, el cual implementa el concepto de "Auto Reparación de Sistemas Basada en Modelos de Arquitectura", el cual se describirá en detalle a continuación.

2.2.1. Auto Reparación de Sistemas Basada en Modelos de Arquitectura

En contraste con los mecanismos tradicionales para detección y recuperación de errores que se implementan como parte del código específico de la aplicación, con mecanismos localizados y poco reutilizables entre distintos sistemas; el enfoque propuesto por el Dr. David Garlan, usa **el modelo de la arquitectura** del sistema que se desea adaptar como instrumento para razonar sobre sus propiedades (e.g. tiempo de respuesta de un servidor) y sus correlatos con la dinámica del sistema en tiempo de ejecución.

Diversos investigadores han propuesto usar modelos arquitecturales [ORI/99] que representan al sistema como una mera composición de componentes, sus interconexiones (conectores) y sus propiedades de interés. Este modelo es conocido comunmente como **C&C** (componentes y conectores)[PAN/10]. Tal propuesta ofrece diversos beneficios, el más significativo: un modelo arquitectural abstracto provee una perspectiva global del sistema y expone sus propiedades y restricciones de integridad.

La idea propuesta consiste básicamente en un bucle cerrado (*closed-loop* en inglés), donde existen dos capas (externas al sistema que está siendo ejecutado) que actúan, una encargada del monitoreo del sistema y la otra proveyendo un mecanismo de control y adaptación. Esto

[5] El proyecto ABLE (*"Architecture Based Languages and Environments"*) de la Universidad de Carnegie Mellon lleva a cabo investigaciones que conducen a una base de ingeniería para la arquitectura de software. Para más información, visitar http://www.cs.cmu.edu/~able

ofrece una solución mas efectiva que cualquier mecanismo interno porque permite agrupar todo lo concerniente a la detección y solución del problema en módulos separados, pudiendo ser analizados, modificados, extendidos y reutilizados a través de distintos sistemas.

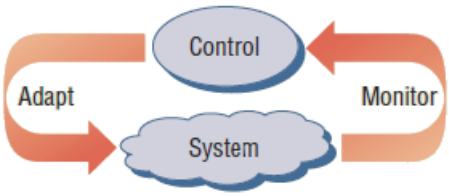


Figura 1: Bucle cerrado

Otro sub proyecto del proyecto ABLE, denominado **Rainbow** (sobre el cual profundizaremos más adelante) utiliza esta técnica de *closed-loop* para monitorear y reparar sistemas.

2.3. Lenguajes de Descripción de Arquitecturas

Un problema fundamental en el diseño de arquitecturas de sistemas ha sido encontrar la notación apropiada para definir dichas arquitecturas.

Un buen lenguaje para describir arquitecturas debería permitir generar una documentación clara sobre los componentes de la arquitectura, que luego servirá como base a los desarrolladores, permitiendo a su vez razonar sobre las propiedades del sistema y automatizar su análisis, hasta pudiendo quizás llegar a utilizarse para la generación automática de parte del código que implementará la arquitectura. También debería ser efectivo para poder validar de manera temprana decisiones arquitectónicas, reduciendo así el tiempo de implementación y evitando utilizar ineficientemente recursos en el desarrollo del sistema.

Una forma de describir dichas arquitecturas es mediante el modelado con UML, si bien este método ha sido ampliamente aceptado y utilizado en la industria, tiene varios inconvenientes: el más importante e invalidante es que no provee un soporte directo para describir propiedades no funcionales. Ésto hace dificultoso razonar sobre propiedades críticas del sistema, como por ejemplo la eficiencia o la confiabilidad. Ésta es la razón principal que ha motivado el avance de los ADLs (*Architecture Description Languages*). Para más información sobre la discusión ADL's vs. UML, remitirse a [PAN/10].

La descripción de arquitecturas de sistemas basada en ADLs ha avanzado considerablemente en las últimas dos décadas, al punto de que ya permiten definir una base formal para su descripción y análisis.

2.3.1. Acme

Acme[GAR/00] es uno de los ADLs más reconocidos y utilizados, ha sido desarrollado en la universidad de Carnegie Mellon, más precisamente por el proyecto 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 sub proyectos utilizan Acme en menor o mayor medida.

Además de los beneficios de todo ADL, el lenguaje Acme y su kit de herramientas *AcmeLib (Acme Tool Developer's Library)* 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 este tipo [6] integrar fácilmente con otras herramientas complementarias. De esta manera, los arquitectos que usan herramientas basadas en Acme tienen un espectro más amplio de herramientas de análisis y diseño que quienes diseñan sus arquitecturas usando otros ADLs.
- Extensibilidad: Acme provee una base sólida, genérica y extensible, y una infraestructura que evita que los desarrolladores vuelvan a construir herramientas de base. Más aún, debido a su idea originaria de lenguaje de intercambio genérico, Acme permite que las herramientas que se han desarrollado utilizándolo sean compatibles con una gran variedad ADLs existentes y con herramientas con un mínimo esfuerzo, y hasta en algunos casos sin esfuerzo alguno.

Actualmente, el lenguaje Acme y *Acme Tool Developer's Library (AcmeLib)*, proveen una infraestructura genérica y extensible para describir, representar, analizar y generar descripciones de arquitecturas de software.

En la figura 2, se observa un breve ejemplo de una arquitectura modelada en el lenguaje Acme, la cual posee un sistema que contiene:

- un servidor HTTP, con algunas propiedades como por ejemplo fidelidad del contenido que provee.
- un cliente HTTP, también con algunas propiedades particulares como el tiempo de respuesta experimentado por el usuario.

Más adelante, en la sección 5, veremos otro ejemplo (más extenso) del lenguaje al mostrar la descripción completa en Acme de la arquitectura del sistema que utilizaremos para mostrar los resultados de la extensión implementada en el presente trabajo.

[6] Ejemplos de herramientas de descripción de arquitecturas y modelado UML podrían ser: Enterprise Architect (http://www.sparxsystems.com.au/) o Poseidón (http://www.gentleware.com/), entre tantas otras.

```
System system : ClientServerType = {
    Component server : ServerT = new ServerT extended with {
        Port http0 : HttpPortT;
        Property cost;
        Property fidelity;
        Property load;
    }
    Component client : ClientT = new ClientT extended with {
        Port p0 : HttpReqPortT = new HttpReqPortT extended with {
            Property isArchEnabled = true;
        }
        Property deploymentLocation = "127.0.0.1";
        Property isArchEnabled = true;
        Property experRespTime;
    }
}
```

Figura 2: Arquitectura básica modelada con Acme

2.3.2. Tácticas y Estrategias

Hemos mencionado anteriormente que el objetivo de la auto reparación es el alcanzar determinados atributos de calidad definidos para un determinado sistema, ajustando su comportamiento, de ser necesario, de acuerdo a sus condiciones de ejecución. En el libro **Software Architecture in Practice** [BAS/03], Bass, Clements y Kazman caracterizan y formalizan dos herramientas que vienen siendo ampliamente utilizadas desde hace tiempo por los arquitectos de software en la industria, estas son: las **tácticas** y las **estrategias**.

Las **tácticas** se definen como decisiones de diseño tendientes a controlar las respuestas del sistema a determinados estímulos, a fin de satisfacer uno o más atributos de calidad requeridos. La figura 3 muestra gráficamente el concepto de táctica arquitectural.

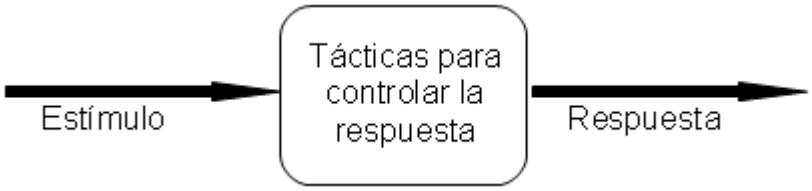


Figura 3: Visión gráfica del concepto de táctica

Cada táctica es una opción de diseño para el arquitecto, un ejemplo concreto podría ser el introducir redundancia en determinados componentes de la arquitectura (e.g. base de datos, servidores web replicados, etc.) para incrementar la disponibilidad del sistema.

Por otro lado, una **estrategia** puede ser entendida como un procedimiento delineado por los arquitectos de *software* para intentar llevar al sistema a un nivel dónde los atributos de calidad se cumplan en el nivel deseado; haciendo uso de una o más tácticas. Cada táctica es ejecutada únicamente cuando el estado del sistema satisface las condiciones impuestas

por la estrategia para dicha ejecución. Por ejemplo, en una arquitectura cliente-servidor y al verse el tiempo de respuesta comprometido, una estrategia podría intentar agregar servidores mientras existan disponibles o, hasta que el tiempo de respuesta haya descendido por debajo de un determinado umbral. Esta lógica sería descrita en la estrategia, mientras que será la táctica `levantar-servidor` la responsable de ejecutar la acción propiamente dicha. Notar que si bien esta estrategia está diseñada para mejorar el tiempo de respuesta, también afecta negativamente al *concern* "cantidad de servidores" (correspondiente al atributo de calidad "costo") puesto que usualmente el utilizar mayor cantidad de servidores suele tener un costo económico no despreciable.

Cuando una estrategia se diseña para mejorar un atributo de calidad en particular, se puede decir que los *stakeholders* que definen dichos atributos de calidad obtienen cierto "beneficio" de las estrategias. Cada estrategia provee un nivel específico de dicho beneficio, pero en contrapartida presenta un costo en tiempo y, sobre todo, en dinero. Es por este motivo que los *stakeholders* deben participar en el proceso de decisión de cúales estrategias se emplearán para satisfacer los atributos de calidad definidos para el sistema. Ellos deberán evaluar el retorno de la inversión (la relación costo-beneficio) de aplicar cada estrategia para elegir la más conveniente.

La estrategia es la herramienta propuesta por Rainbow para quitar al sistema de un estado no deseado.

2.4. Rainbow

2.4.1. Introducción a Rainbow

La herramienta Rainbow, también dentro del marco del proyecto ABLE, tiene como finalidad permitir reducir el costo e incrementar la confiabilidad al realizar cambios en sistemas complejos de software, para esto Rainbow automatiza la adaptación de sistemas de software a traves de sus modelos de arquitectura, tal cual fue descrito en la sección 2.2.1.

Si bien en principio el enfoque de auto adaptación basado en arquitecturas es atractivo, también supone un número significativo de desafíos en el campo de la investigación así también como en el de la ingeniería:

- En primer lugar, uno de los aspectos claves que Rainbow intenta cubrir es la habilidad de manejar una amplia variedad de sistemas con arquitecturas, propiedades de interés y mecanismos que soporten modificaciones dinámicas completamente diferentes.
- Por otro lado, Rainbow intenta ser una solución que permita reducir el costo de agregar control externo al sistema a reparar, puesto que crear los mecanismos de monitoreo y detección de problemas desde cero para un sistema nuevo sería prohibitivamente costoso.

El enfocar la auto reparación de un sistema en su arquitectura permite disponer de una infraestructura reutilizable junto con mecanismos para adaptar dicha infraestructura a las necesidades específicas de cada sistema.

Cabe mencionar que 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.

2.4.2. Arquitectura de Rainbow

La Figura 4 muestra la arquitectura de Rainbow. En resumidas palabras, el *framework* utiliza un modelo arquitectural abstracto para monitorear las propiedades en *runtime* del sistema que está siendo ejecutado, evalúa el modelo para detectar violaciones a alguna de sus restricciones y lleva a cabo adaptaciones en el sistema tendientes a eliminar tales violaciones.

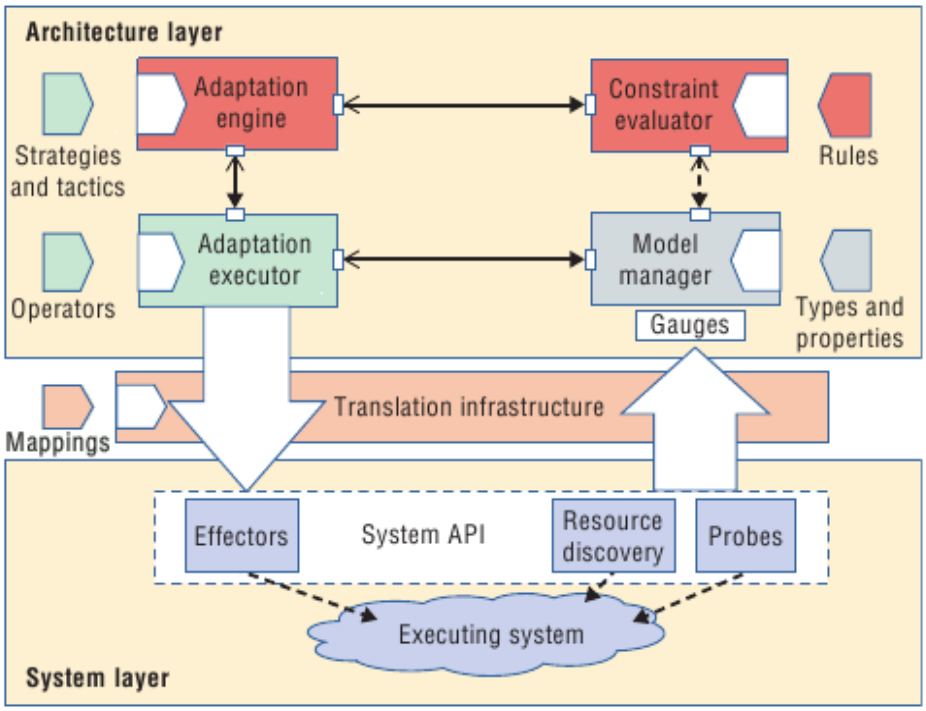


Figura 4: Arquitectura de Rainbow

La infraestructura de adaptación de Rainbow se divide en capas que proveen funcionalidades comunes a distintos sistemas auto adaptables logrando por lo tanto el objetivo de disponer de componentes reutilizables, a saber:

1. **Capa de Sistema**:
   En esta capa se define e implementa una interfaz de acceso al sistema que está siendo ejecutado. Se define un mecanismo para medir variables de interés, materializado en

Probes: componentes que observan y miden diversos estados del sistema, para luego publicarlos.

Adicionalmente, existe un mecanismo para descubrir recursos que puede ser utilizado especificando el tipo de recurso, entre otros criterios. Finalmente, los denominados *Effectors* llevan a cabo las modificaciones propiamente dichas sobre el sistema.

2. **Capa de Arquitectura**:
   En esta capa, los denominados *Gauges* agregan información provenientes de los *Probes* y mantienen constantemente actualizadas las propiedades correspondientes en el modelo arquitectural del sistema (descrito en Acme), el cual es manejado y accedido mediante un componente denominado *Model Manager*. El *Constraint Evaluator* chequea el modelo periódicamente y dispara la adaptación en el caso que ocurra una violación en alguna restricción impuesta sobre el modelo. En ese caso, el motor de adaptación (*Adaptation Engine*) determina el curso de acción y lleva a cabo la adaptación necesaria.

3. **Capa de Traducción**:
   Esta capa es la encargada de cubrir la brecha de abstracción existente entre el sistema en ejecución y el modelo de su arquitectura (en ambos sentidos). En esta infraestructura, un repositorio de traducción mantiene diversos mapeos compartidos por distintos componentes dentro de esta capa, por ejemplo, una operación a nivel modelo de la arquitectura en su correspondiente operación de *runtime*:

```
Componente de Log::desactivar       <==>       Logger.disableLog()
```

Rainbow es un *framework* desarrollado en el lenguaje de programación Java™ y todos los derechos sobre el código fuente pertenecen al grupo ABLE de la Universidad de Carnegie Mellon. Los autores de este trabajo solicitaron permiso a este grupo para poder acceder al código fuente de Rainbow para poder realizar la extensión objeto de este trabajo. En la wiki oficial de Rainbow pueden encontrarse instrucciones para instalar versiones ya compiladas del *framework*. Para más información, visitar http://rainbow.self-adapt.org/RainbowInstall.

2.4.3. Conocimiento específico del sistema

En la sección anterior hemos descrito la infraestructura básica provista por Rainbow. Es de notar que no es suficiente para satisfacer las necesidades puntuales de auto adaptación de un sistema en particular. Para lograr esto, es necesario extender dicha infraestructura, agregando conocimiento específico del sistema que se desea adaptar. Este conocimiento (típicamente no reutilizable entre distintos sistemas) incluye el modelo operacional del sistema, que define parámetros como tipos de componentes y propiedades, restricciones de comportamiento, estrategias de adaptación, interfaz para acceder a la información de *runtime* del sistema, así también como para hacer efectivas las estrategias de reparación, etc.

2.4.4. Stitch

A fin de disponer de una forma suficientemente expresiva de definir tácticas y estrategias, Rainbow incluye un lenguaje de *scripting* de propósito específico llamado **Stitch**, el cual permite plasmar el conocimiento rutinario de las personas sobre adaptación de sistemas de software.

Algunas de las características innovadoras de Stitch:

- **Control del sistema**: La selección de la próxima acción a ejecutar en el contexto de una estrategia depende de los efectos observados luego de la acción previa.
- **Sensibilidad al contexto**: La selección de la mejor estrategia se realiza considerando el estado actual del sistema, mediante la inspección de algunas de sus propiedades.
- **Asincronismo**: Stitch permite especificar un tiempo de demora luego de la ejecución de una táctica para que los efectos de la táctica se puedan ver reflejados en el sistema.

2.4.5. Ejemplo de una Táctica en Stitch

En la figura 5 se puede apreciar un ejemplo de una táctica definida en Stitch para ser utilizada por Rainbow. Primeramente, se importa el modelo de la arquitectura del sistema en cuestión junto con la implementación de un operador que permite impactar al sistema en ejecución (estos operadores suelen ser provistos por el usuario de la aplicación).

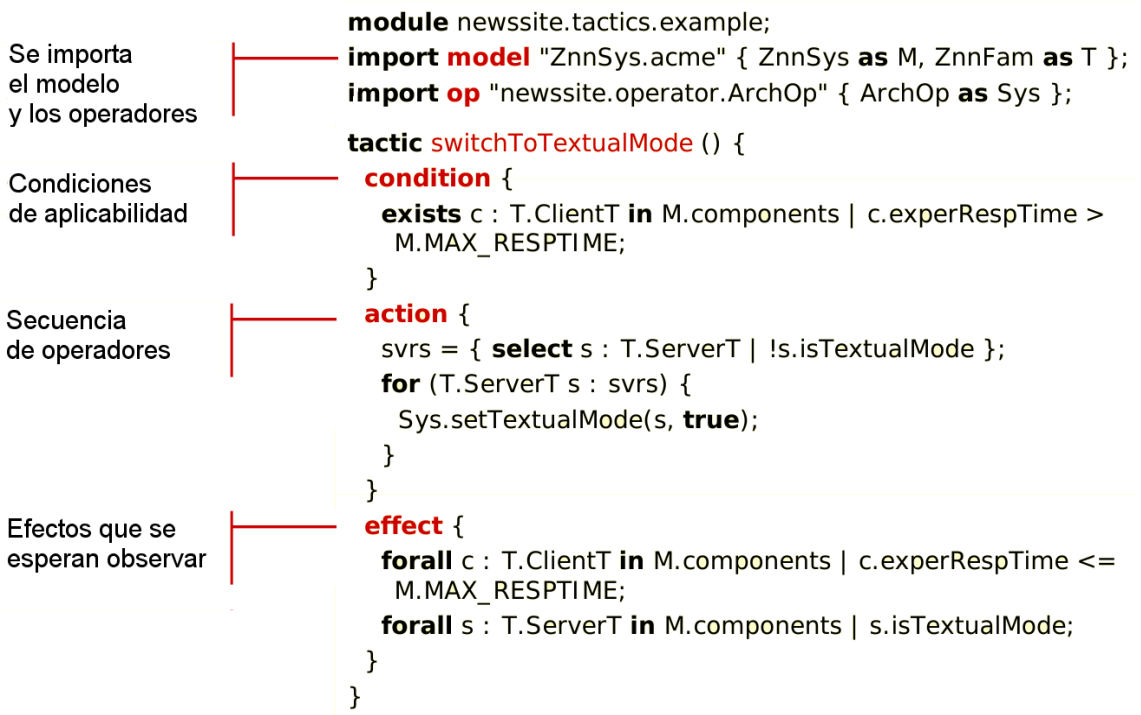


Figura 5: Ejemplo de una táctica en Stitch

La táctica consiste en disminuir la fidelidad (a modo sólo texto) del contenido provisto por todos los servidores cuando se detecta que al menos un cliente experimenta un tiempo de respuesta superior a un determinado umbral. Para lograr esto, Rainbow inspecciona las propiedades del modelo de la arquitectura del sistema, definido en Acme, el cúal se supone constantemente actualizado por Rainbow con respecto al estado actual del sistema en ejecución.

Por último, se especifica que el efecto esperado de ejecutar la táctica consiste en que todos los clientes experimenten un tiempo de respuesta inferior al umbral y que, por otro lado, todos los servidores se encuentren prestando servicio en modo sólo texto.

2.4.6. Ejemplo de una Estrategia en Stitch

En la figura 6 podemos ver un ejemplo de una estrategia definida en Stitch para ser utilizada por Rainbow.

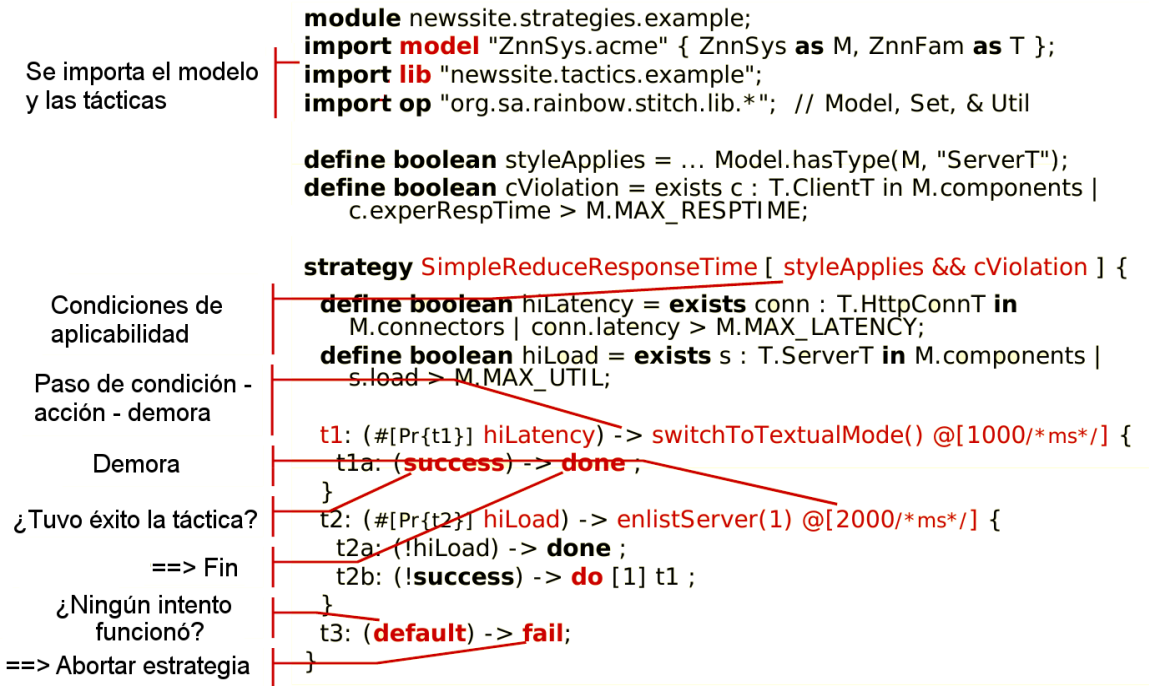


Figura 6: Ejemplo de una estrategia en Stitch

La estrategia representa un algoritmo simple para disminuir el tiempo de respuesta experimentado por el usuario de un sistema cliente-servidor. Se definen algunos predicados de primer orden que predican sobre propiedades del modelo de la arquitectura del sistema, el cual se presume constantemente actualizado por Rainbow con respecto al sistema en ejecución.

El primer paso de la estrategia consiste en verificar que haya en la arquitectura un conector que presente alta latencia, en ese caso, se invoca la táctica definida en la Figura 5, la cual cambia la fidelidad de todos los servidores a modo sólo texto, y se espera 1 segundo antes de determinar el éxito o no de la ejecución de la táctica. Si la táctica tuvo éxito, la estrategia

finaliza satisfactoriamente. Sino, la estrategia chequea la existencia de al menos un servidor con alta carga, y en caso afirmativo, ejecuta otra táctica que consiste en agregar un servidor más en pos de mejorar el *concern* tiempo de respuesta. Si luego de esperar 2 segundos para que los efectos de la táctica puedan verse reflejados en el sistema, todos los servidores operan con carga normal, la estrategia finaliza exitosamente, caso contrario, se considera fallida.

2.4.7. *Exponential Moving Average*

El proceso de adaptar un sistema de manera dinámica puede llegar a ser muy costoso, por ejemplo, en un sistema cliente-servidor, el asignar más servidores para atender las peticiones de los usuarios implica normalmente un coste monetario no despreciable. Otro ejemplo podría ser el suspender temporalmente la reproducción de videos en un sitio de noticias. Si bien esta decisión puede servir para disminuir el tiempo de respuesta experimentado por los usuarios en un contexto de alta carga, también provoca una clara disminución en la calidad del servicio ofrecido. Estos ejemplos concretos sirven para inferir que el disparar un mecanismo de auto reparación en un sistema debe ser una decisión tomada con cierta cautela y en base a datos confiables y sostenidos en el tiempo.

Hemos visto anteriormente, de manera somera, de que forma Rainbow (mediante estrategias y tácticas definidas en el lenguaje Stitch) utiliza datos sobre el sistema en ejecución para decidir el mejor camino a tomar para adaptar el sistema. Consideremos un escenario donde una estrategia debe decidir si el último paso ejecutado fue exitoso o no. Para eso, deberá consultar una o más propiedades del sistema en ejecución, las cuales podrían llegar a no ser representativas del entorno *real* en que el sistema se encuentra, debido a la presencia de (*outliers*), i.e. valores aislados considerablemente distintos al resto de los datos recientes que se usan para tomar decisiones.

Rainbow implementa un mecanismo que permite evitar accionamientos prematuros de la auto reparación debido a la presencia de *outliers*. El mentado mecanismo utiliza *Exponential Moving Average*[7], que permite ponderar los valores históricos de la(s) propiedad(es) del sistema que se consultan para tomar decisiones. La función de suavizado que implementa esta heurística se define inductivamente de la siguiente manera:

$$S_0 = Y_0$$

$$S_t = \alpha \times Y_t + (1 - \alpha) \times S_{t-1}$$

donde α se denomina **factor de suavizado** y $0 < \alpha < 1$.

El valor suavizado S_t no es ni más ni menos que un promedio ponderado de la última observación Y_t y el valor suavizado previo S_{t-1}.

[7]Para más información, visitar http://www.iexplain.org/exponential-moving-average-defined/

Notar que con valores *altos* de α, el último valor observado tendrá más preponderancia que el valor histórico (suavizado) anterior. En este caso, es probable que el sistema se vea afectado por unos pocos valores que se encuentran por fuera del rango considerado como normal, aunque de no ser costoso lanzar una adaptación o revertirla, podría lograrse un rendimiento aceptable del sistema de manera prácticamente inmediata.

Por el contrario, con un α tendiendo a cero, la última observación de una propiedad de la arquitectura prácticamente no tendrá relevancia sobre el valor promedio S_t. Esto implicaría que a Arco Iris le tomará más tiempo adaptarse a los cambios en el entorno del sistema. Ésto podría ser útil, por ejemplo, en sistemas con bajo grado de dinamismo o en los cuales la adaptación pueda resultar muy costosa.

No existe un procedimiento formal para determinar el valor de α, en el caso particular de las pruebas realizadas en este trabajo, y al igual que las pruebas realizadas en Znn, se ha elegido $\alpha = 0{,}3$, es decir que, se ponderará con un 30 % al último valor observado mientras que el valor histórico tendrá un peso del 70 %; con esto nos aseguramos que los valores de las propiedades del sistema que son relevantes para la auto adaptación no fluctúen bruscamente debido a unos pocos *outliers*.

2.4.8. Tasa de Fallos de una Estrategia

Rainbow provee un mecanismo que considera la historia de ejecución de las estrategias, el cual es útil para que, dado un umbral predeterminado y no configurable por el usuario, aquellas estrategias que han fallado porcentualmente en un 95 % o más de las veces que fueron ejecutadas, no sean consideradas por el `AdaptationManager`. El *framework* permite especificar al usuario si desea habilitar este mecanismo o no y, de estar habilitado, lleva un registro por estrategia de la historia de sus ejecuciones.

El mecanismo contempla el paso del tiempo como un factor importante al momento de determinar el porcentaje de fallos: el cociente

$$\frac{\text{cantidad de fallos}}{\text{cantidad de ejecuciones}}$$

es sopesado con el tiempo que ha pasado desde la última ejecución de la estrategia, permitiendo así que vuelva potencialmente a ser considerada en futuras ejecuciones.

2.4.9. Znn

Znn es un sistema que simula un sitio web de noticias, el cual nació en el contexto de la tesis de doctorado [SHA/08] de Shang-Wen Cheng, un investigador de la universidad de Carnegie Mellon. En dicha tesis, se evalúa a Rainbow en los siguientes aspectos:

- su efectividad para mantener los atributos de calidad ante condiciones cambiantes.
- la sobrecarga de procesamiento que implica la auto reparación.
- el esfuerzo que implica agregar auto reparación a Znn mediante Rainbow.

Si bien Znn y sus herramientas asociadas (*probes*, *gauges*, tácticas, estrategias, etc.) nacieron con el único objetivo de evaluar la efectividad de Rainbow, cabe mencionar que han sido abiertas a la comunidad para que puedan ser utilizadas para tomar métricas y poder comparar distintos *frameworks* de auto reparación.

Znn provee un entorno de simulación de una arquitectura cliente-servidor ampliamente configurable que permite representar situaciones específicas de ejecución y controlar las variables de simulación permitiendo modificarlas en cualquier punto.

Por ejemplo, es posible configurar a Znn para que inicie con solamente 2 clientes, mostrando allí un desempeño aceptable y que luego se agreguen 10 clientes, comprometiendo así la eficiencia del sistema. De esta manera, Znn permite simular cómo responderían las estrategias implementadas ante dicha situación.

2.4.10. Modos de Ejecución

Rainbow soporta dos modos de ejecución: el **modo normal**, donde el *framework* se conecta con un sistema real, y un **modo simulación**, el cual permite probar las herramientas de auto reparación implementadas por los usuarios sin necesidad de conectar a Rainbow con un sistema real en ejecución.

En modo simulación, el usuario debe configurar mediante un archivo de texto (de extensión `.sim`) consistente en pares clave-valor, los valores iniciales de las propiedades de los componentes del sistema, más los cambios de valores en las propiedades de interés del modelo de la arquitectura siguiendo el siguiente esquema:

```
sim.<numero_simulacion> = <instante_tiempo>:<propiedad>:<valor>
```

Ejemplos de puntos de simulación:

```
sim.1 = 30001:ZNewsSys.c1.perf.service.time:100,30001:ZNewsSys.c2.perf.service.time:100
sim.2 = 40001:ZNewsSys.c1.perf.arrival.rate:4
```

Cabe remarcar, que en este modo los *probes* y los *gauges* no tienen participación, ya que no existe un sistema real en ejecución con el cual interactuar.

En el presente trabajo se utilizará el modo simulación para los casos de prueba, tomando las estrategias y tácticas provistas por Znn. Éstas debieron ser adaptadas para que puedan seguir funcionando con los cambios propuestos en el presente trabajo.

2.5. Escenarios de Atributos de Calidad

Un escenario de atributos de calidad (de ahora en más, simplemente "escenario") es la especificación de un requerimiento para un atributo de calidad en particular. Consiste de seis partes:

- **Fuente del estímulo**: Es una entidad (un ser humano, otro sistema o cualquier otro actor) que generó el estímulo.
- **Estímulo**: la condición que debe ser tenida en cuenta cuando llega al sistema.
- **Entorno**: el estímulo ocurre bajo ciertas condiciones, e.g. entorno de alta carga, entorno normal, sistema no funcionando, etc.
- **Artefacto**: el sistema o partes de él afectadas por el estímulo.
- **Respuesta**: refiere a qué hace el sistema ante la llegada del estímulo.
- **Medición de la respuesta**: cuando ocurre la respuesta, debe ser medible de alguna manera para que el requerimiento pueda ser *testeado*.

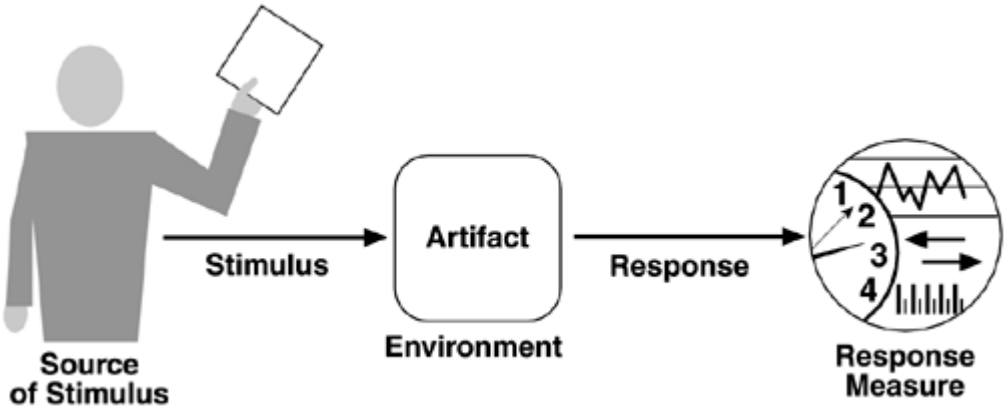


Figura 7: Visión gráfica de un escenario

Los escenarios son pequeñas historias que describen una interacción con el sistema, la cual 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 sobre la recepción del mensaje y continúa su operación sin caídas."*

Este escenario se descompone de la siguiente manera:

- **Fuente del estímulo**: Cualquier fuente externa

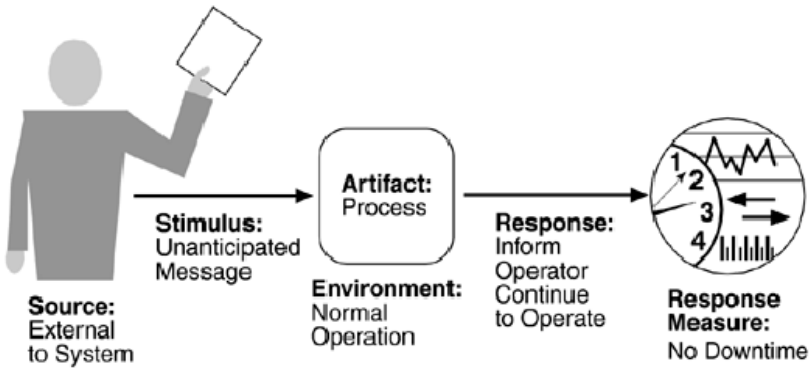


Figura 8: Ejemplo de un escenario de disponibilidad

- **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, el *sponsor*, etc). Estos escenarios pueden luego ser utilizados para analizar y definir la arquitectura del sistema e identificar *concerns* y posibles estrategias para atacar problemas.

2.5.1. QAW

Existe una metodología definida por el Software Engineering Institute (SEI) conocida como **Quality Attribute Workshops (QAW)**, cuya principal herramienta son los escenarios. Los QAW proveen un método para identificar los atributos de calidad críticos de la arquitectura de un sistema, tales como disponibilidad, eficiencia, seguridad, etc, que son derivados de objetivos del negocio. QAW no presupone la existencia de una arquitectura del sistema, sino que fue desarrollado como consecuencia de la necesidad de *stakeholders* y arquitectos de un método que permita identificar los atributos de calidad importantes para el correcto funcionamiento del sistema **antes de definir su arquitectura**.

Los QAW son reuniones en las que participan todos los *stakeholders*, en las que se definen los escenarios que en definitiva representarán los requerimientos de atributos de calidad que el sistema idealmente deberá satisfacer. Una vez definidos todos los escenarios, el siguiente paso del QAW consiste en priorizarlos y refinarlos, especificando claramente todos las partes que los componen y determinando el atributo de calidad asociado a 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 de los participantes, y proporcionando 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, que pueden servir para definir casos de pruebas, o como semillas para el proceso ATAM, sobre el cual discutiremos en la siguiente sección.

Si bien no es condición necesaria para poder usar Arco Iris, el utilizar la metodología QAW, tal cual está descrita en [BAR/03], es recomendado ya que Arco Iris hará uso intensivo de los escenarios de atributos de calidad y, por lo tanto, de su correcta definición dependerá el nivel de optimización y flexibilización que la extensión a Rainbow pueda alcanzar al auto reparar un sistema de *software*.

2.6. ATAM

2.6.1. Terminología necesaria

Antes de profundizar sobre ATAM, es necesario clarificar ciertos términos que serán utilizados en el desarrollo de la presente sección. Estos términos son:

- **Riesgo**: decisión arquitectónica potencialmente problemática.
- **Punto sensible**: propiedad de uno o más componentes (y/o relaciones entre componentes) que es crítica para poder alcanzar un atributo de calidad determinado. Por ejemplo, un módulo de cifrado es un punto sensible para el atributo de calidad "seguridad".
- **Punto de *tradeoff***: se trata de un componente de la arquitectura donde se afecta más de un atributo de calidad (y por ende, también es un punto sensible para dichos atributos de calidad). Por ejemplo, el módulo de cifrado mencionado anteriormente sería un punto de *tradeoff* de haber requerimientos de eficiencia, ya que afecta positivamente la seguridad y negativamente la eficiencia del sistema.
- ***Non risks*** son buenas decisiones de arquitectura que normalmente están implícitas en la arquitectura.

2.6.2. Analizando arquitecturas con ATAM

El método para el análisis de *tradeoffs* o, de sus siglas en inglés, ATAM (Architecture Tradeoff Analysis Method)[KAZ/00], al igual que QAW, ha sido desarrollado por el SEI y es una técnica que permite analizar arquitecturas de *software* con el objetivo de validar

requerimientos de atributos de calidad, su interacción (conocidos como *tradeoffs*), detectar problemas de manera temprana e identificar riesgos y puntos sensibles.

Usualmente, las arquitecturas de *software* son complejas, e involucran muchos *tradeoffs* de diseño. Sin un proceso de análisis formal, no es posible garantizar que las decisiones de arquitectura tomadas –en particular aquellas que afectan al cumplimiento de requerimientos de calidad –son adecuadas para mitigar los riesgos.

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 plantear las preguntas correctas sobre la arquitectura 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 riesgos en la arquitectura y soluciones a dichos riesgos.

Cabe mencionar que los QAW han surgido como consecuencia del uso de ATAM, puesto que era requerida una herramienta o método que permitiera identificar los requerimientos de atributos de calidad más importantes del sistema, **antes de que existiese la arquitectura** sobre la cual ATAM trabajaría. Luego de observar los pasos del método ATAM que detallaremos a continuación, no será difícil para el lector inferir que en realidad ATAM incluye una versión simplificada de QAW.

2.6.3. Pasos del método ATAM

A continuación se describen los pasos del método ATAM:

Presentación

1. **Presentar ATAM.** El método es descrito a los *stakeholders*.
2. **Presentar las metas del negocio.** El *project manager* describe los objetivos del negocio que motivan el desarrollo.
3. **Presentar la arquitectura.** El equipo de arquitectos presenta la arquitectura propuesta, haciendo foco en cómo la misma satisface las metas del negocio.

Investigación y Análisis

4. **Identificar enfoques arquitectónicos.** Los enfoques arquitectónicos son identificados por el equipo de arquitectura, pero no analizados.

5. **Generar el árbol de utilidad.** Se recaban los atributos de calidad que agregan utilidad al sistema y se los especifica en forma de escenarios priorizados.

6. **Analizar enfoques arquitectónicos.** Se delinean enfoques arquitectónicos para los escenarios de mayor prioridad recabados en (5). Durante este paso, se identifican riesgos arquitectónicos, puntos sensibles y puntos de *tradeoff*.

Testing

7. ***Brainstorming* y priorización de escenarios.** Basado en los escenarios del árbol de utilidad, se genera un conjunto mayor de escenarios más específicos. Estos nuevos escenarios son priorizados vía un proceso de votación.

8. **Analizar enfoques arquitectónicos.** Se reitera el paso (6) pero con los escenarios más prioritarios encontrados en el paso (7), con el objetivo de encontrar nuevos riesgos, puntos sensibles y *tradeoffs*.

Reporting

9. **Presentar resultados.** Basado en toda la información recolectada durante el proceso (escenarios, el árbol de utilidad, riesgos, puntos sensibles y puntos de *tradeoff*), el equipo de ATAM escribe un reporte detallando esta información y las estrategias de mitigación propuestas.

Figura 9: Pasos de ATAM

3. Extensión a Rainbow: Arco Iris

3.1. Introducción

Como ya se ha comentado anteriormente, la idea de este trabajo es extender el *framework* Rainbow para poder lograr un mecanismo de auto reparación más flexible y con mayor capacidad expresiva, y con el objetivo de proveer visibilidad a los *stakeholders* de la aplicación sobre dicho proceso, permitiéndoles involucrarse en la definición de escenarios de atributos de calidad del sistema, de sus prioridades relativas y de las estrategias a considerar en la auto reparación del sistema cuando un escenario deja de cumplirse. A fin de lograr lo antedicho, se propone extender el *framework* Rainbow, con la anuencia y apoyo de los integrantes del proyecto ABLE, quienes poseen su propiedad intelectual.

3.2. Rainbow *out of the box*

En el diagrama de colaboración de objetos que se muestra en la Figura 10 se pueden observar los principales componentes de la arquitectura de Rainbow.

Observamos que, en Rainbow, una persona con rol de arquitecto (o similar) es el encargado de configurar el *framework* utilizando, básicamente, dos vías:

1. la creación de un modelo de la arquitectura del sistema al cual Rainbow va a adaptar. Dicho modelo se especifica utilizando el estilo de componentes y conectores (C&C) y el lenguaje de descripción de arquitecturas Acme (ver sección 2.3.1). En dicho lenguaje, los componentes y conectores poseen propiedades (con valores asociados), restricciones y también se ofrece la posibilidad de especificar invariantes a nivel de sistema o sub sistema. Dichos invariantes y restricciones serán evaluados periódicamente por Rainbow para verificar que el sistema funcione dentro de los límites determinados como normales.

2. la generación de un conjunto de archivos de configuración que especifican diversos aspectos relacionados con la definición del sistema a auto reparar, como por ejemplo: ubicación física del archivo Acme que describe la arquitectura del sistema, archivos de tácticas y estrategias de reparación (escritos en *Stitch*), datos para la configuración de la conexión de Rainbow con el sistema en *runtime*, etc.

Podemos ver también en el gráfico anterior que los denominados *Probes* son los componentes designados para interactuar directamente con el sistema a adaptar (el *Target System*), obteniendo así información relevante a los fines de su auto reparación. Dicha información no es interpretada dentro de los *probes*, sino que ellos delegan dicha tarea a los denominados *Gauges*. Estos componentes son los encargados de interpretar la información provista por los *probes* y traducirla a pares `<Propiedad,Valor>` donde `Propiedad` refiere al nombre de

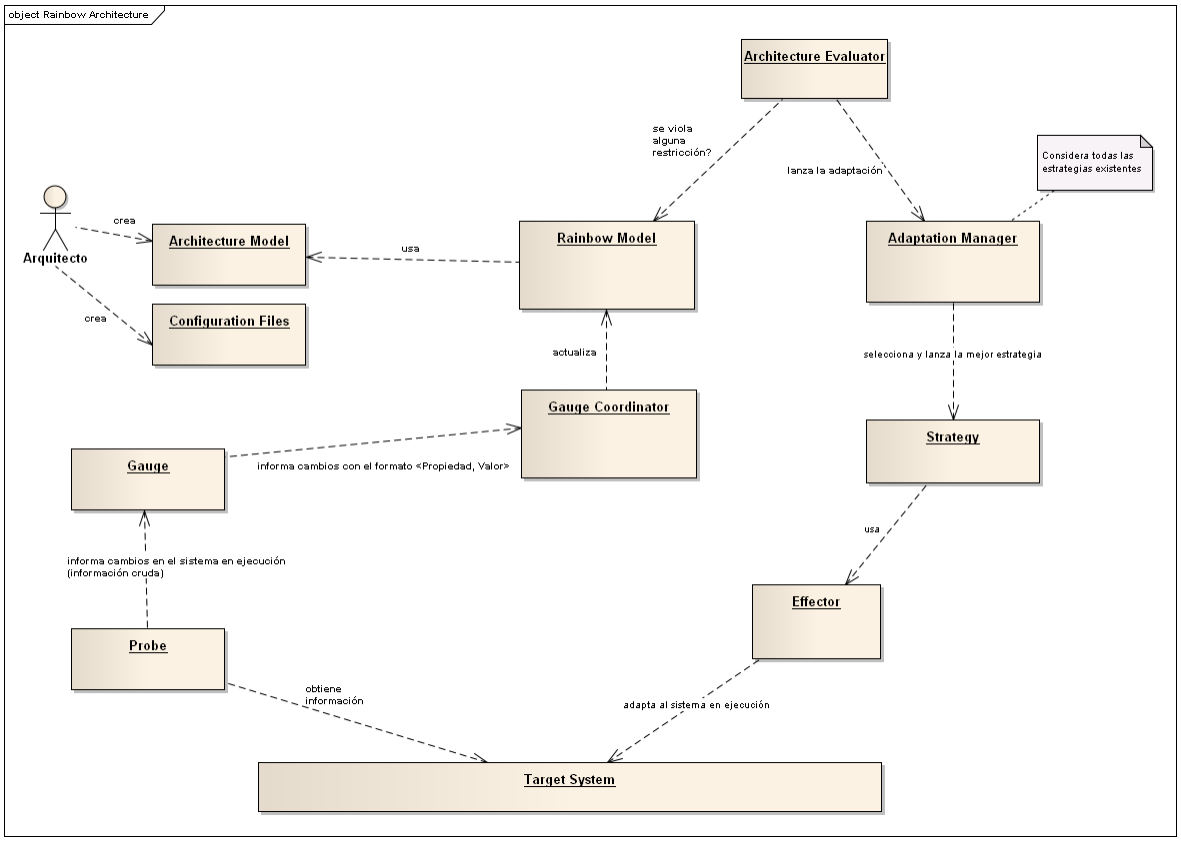


Figura 10: Arquitectura de Rainbow

una propiedad correspondiente a un componente o conector de la arquitectura del sistema a adaptar, mientras que `Valor` es el nuevo valor que poseerá dicha propiedad.

Normalmente, por cada tipo relevante de *concern* que interese ser monitoreado en tiempo de ejecución existe un par `<Gauge,Probe>` asociado. Un ejemplo de *concern* podría ser: "tiempo de respuesta experimentado por el usuario", el cual es un *concern* relacionado al atributo de calidad eficiencia.

El *Gauge Coordinator*, tal como su nombre lo sugiere, coordina la información provista por todos los *gauges* y se encarga de notificar los cambios en el sistema en ejecución al componente *Rainbow Model*.

Rainbow Model, componente clave en la arquitectura del *framework*, tiene como principal responsabilidad hacer efectiva la actualización de los valores de la propiedades de los componentes, conectores, sub sistemas o sistemas del modelo de arquitectura de la aplicación a adaptar. También es quien tiene el conocimiento necesario para detectar violaciones a las restricciones e invariantes presentes en el modelo de arquitectura con el cual el *framework* ha sido configurado.

Por otra parte, existe otro componente llamado *Architecture Evaluator*, el cual consulta periódicamente a *Rainbow Model* para tomar conocimiento de violaciones a restricciones

definidas en el modelo de la arquitectura. En el caso de existir alguna violación, el *Architecture Evaluator* dispara el mecanismo de adaptación invocando al *Adaptation Manager*.

El *Adaptation Manager*, uno de los componentes más importantes de Rainbow, sigue una lógica un tanto compleja (explicada en detalle en la sección 3.9) para determinar la estrategia de reparación que deja al sistema en el mejor estado posible, para luego ejecutarla.

La *Estrategia* contiene la lógica necesaria para reparar el sistema en ejecución mediante el uso de *Tácticas*, que a su vez, utilizan a los denominados *Effectors*, los cuales son componentes que realizan acciones simples y concretas sobre el sistema en ejecución, como por ejemplo "aumentar el nivel de logging de un componente determinado".

3.3. Rainbow + Escenarios = "Arco Iris"

En el presente apartado se presenta someramente las características principales de la arquitectura de Rainbow luego de incorporar las extensiones planteadas en el presente trabajo, a las cuales denominamos "Arco Iris".

3.3.1. Arquitectura de Arco Iris

En la figura 11 podemos observar como luce la arquitectura del *framework* con la incorporación de las extensiones realizadas.

El arquitecto sigue realizando las mismas tareas que realizaba en Rainbow (i.e. "alimentarlo" con el modelo de la arquitectura del sistema a adaptar y con archivos de configuración requeridos por Rainbow) pero ahora se le suma una tarea más: el configurar los escenarios de atributos de calidad; tarea que realiza en conjunto con una o más personas que asumen distintos roles que normalmente se engloban en la palabra *stakeholders* (e.g. analistas funcionales, usuarios del sistema, líderes, clientes, el *sponsor* del proyecto, etc.)

En la versión original de Rainbow (i.e. sin extensiones) el *framework* deja al usuario (e.g. el arquitecto) la responsabilidad de codificar los *probes* y *gauges* que recolectarán información del sistema a adaptar y traducirán esa información a cambios en los valores de las propiedades del sistema. Esto sigue siendo igual en Rainbow con las extensiones provistas por Arco Iris, es decir, el usuario sigue siendo el encargado de crear los componentes que en tiempo de ejecución encuestan al sistema periódicamente para obtener información relevante y mantener actualizado el modelo de la arquitectura subyacente.

En el diagrama se puede observar que, en Arco Iris, tanto los *probes* como los *gauges* ahora poseen conocimiento del concepto de **estímulo** (tal cual es descrito en los escenarios de atributos de calidad de ATAM). A diferencia de los utilizados en Rainbow, los *probes* implementados para trabajar con Arco Iris deberán indicar el estímulo al cual están asociados,

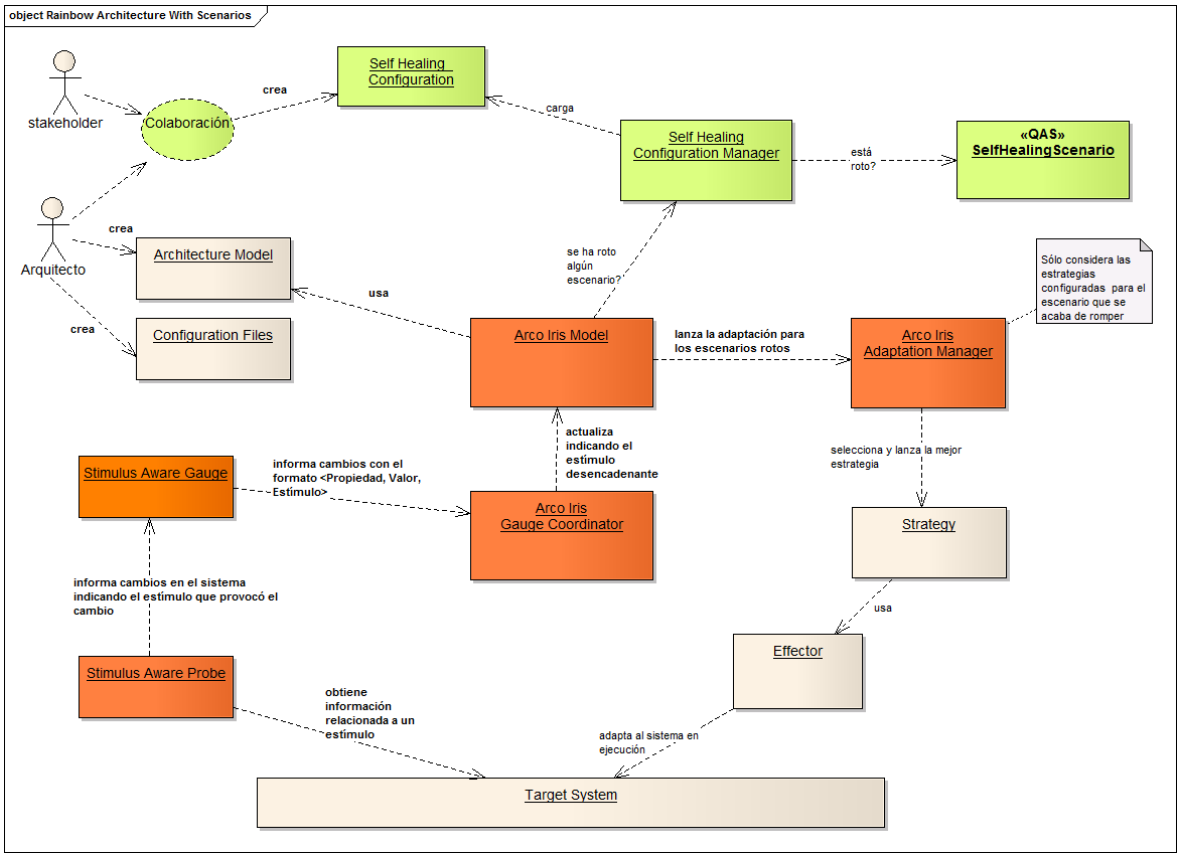


Figura 11: Arquitectura de Arco Iris

mientras que los *gauges* deberán interpretar esta información y notificar al *Arco Iris Model* de la actualización sobre el modelo indicando el estímulo desencadenante. Las novedades sobre los cambios ocurridos en el sistema en *runtime* serán informados con el siguiente formato: `<Propiedad,Valor,Estímulo>` En función de las modificaciones al modelo original de Rainbow descritas en el párrafo anterior, el componente *Gauge Coordinator* ha sido extendido (i.e. *Arco Iris Gauge Coordinator*) conservando el comportamiento original provisto por Rainbow. De esta manera, Arco Iris puede coordinar información proveniente de *gauges* de Rainbow y/o de Arco Iris con su correspondiente conocimiento sobre el estímulo originario de la modificación. De la misma manera, el componente *Arco Iris Model* es una extensión de *Rainbow Model* que permite manipular información de estímulos proveniente del *Arco Iris Gauge Coordinator* sin perder el soporte original provisto por *Rainbow Model*.

Cuando *Arco Iris Model* es notificado de que un estímulo ha sido invocado en el sistema solicita al *Self Healing Configuration Manager* el subconjunto de escenarios habilitados que poseen dicho estímulo y que a su vez han dejado de cumplirse[8]. Luego de obtener este subconjunto de escenarios, invocará al *Arco Iris Adaptation Manager*, extensión del componente original *Adaptation Manager*, indicando el conjunto de escenarios detectados como

[8]De ahora en más, diremos equivalentemente que el escenario en esta situación se encuentra “roto”.

"rotos" a los cuales debe intentar reparar considerando diversas variables como por ejemplo sus prioridades relativas.

Además de detectar los escenarios rotos para un determinado estímulo, el componente *Self Healing Configuration Manager* carga la denominada *Self Healing Configuration*, una abstracción de la mayor parte de la información referente a auto reparación utilizada por Arco Iris. Esto incluye, por supuesto, los escenarios creados por los *stakeholders* y arquitectos.

3.3.2. Modelo de Restricciones en Arco Iris

Tanto Rainbow como Arco Iris, hacen uso del concepto de **restricción**, que si bien se utiliza en distintos contextos debido a que ambos *frameworks* enfocan la auto reparación desde distintos ángulos, ambos responden a la misma finalidad.

Arco Iris utiliza restricciones en dos contextos distintos. Por un lado, las condiciones para conocer el entorno en que se encuentra el sistema no son más que meras restricciones, y por otro, la cuantificación de la respuesta no es más que una restricción que determina la condición necesaria para que el escenario se satisfaga.

Rainbow utiliza el concepto de *restricción* para:

- imponer condiciones sobre el modelo de la arquitectura que el sistema debe satisfacer en tiempo de ejecución, idealmente, en todo momento. Estas condiciones pueden predicar sobre propiedades de componentes, conectores, sub sistemas o directamente sobre todo el sistema.
- especificar precondiciones a la ejecución de tácticas y estrategias, así como también, para condicionar el algoritmo de cada estrategia.

En el primer caso, las restricciones se encuentran expresadas en el lenguaje **Acme**. Si bien, se provee un modelo en Java de los distintos tipos de restricciones que un usuario del lenguaje normalmente querría expresar, dicha implementación se encuentra fuertemente acoplada a la lógica de *parseo* utilizada por el lenguaje, lo cual imposibilitó su reutilización en Arco Iris.

En el segundo caso, las restricciones se encuentran expresadas en el lenguaje **Stitch** y no poseen una contraparte en el lenguaje Java.

Consecuentemente, Arco Iris implementa su propio modelo simplificado de restricciones, el cual se encuentra inspirado en el modelo de Acme, aunque desacoplado de cualquier otra lógica externa. Así, el concepto de restricción presentado en Arco Iris permite agregar nuevos tipos de restricciones simplemente implementando la interfaz `Constraint` [9]. La interfaz definida es la siguiente:

[9] Dicha implementación deberá ser agregada al enumerado ConstraintType en Arco Iris UI para que la UI reconozca el nuevo tipo

```
public interface Constraint {

        boolean holds(Number value);

        String getFullyQualifiedPropertyName();
}
```

Mediante el método `holds`, se define como responsabilidad de la misma `Constraint` el determinar si se cumple o no, recibiendo como parámetro el valor actual de la propiedad del modelo de la arquitectura sobre el cual predica la cuantificación de la respuesta. Es de notar que en el futuro esta interfaz podría ser modificada para soportar propiedades cuyos valores no sean numéricos.

El método `getFullyQualifiedPropertyName` retornará el nombre completo calificado de la propiedad sobre la que predica, incluyendo el sistema y el componente al que pertenece, por ejemplo, `ZNewsSys.ClientT.experRespTime` alude a la propiedad `experRespTime` de todos los componentes del sistema `ZNewsSys` que sean de tipo `ClientT`. De la misma manera, se pueden referenciar propiedades de **instancias** particulares como por ejemplo: `ZNewsSys.Server2.cost`.

Para el presente trabajo se utilizó una única implementación de esta interfaz, consistente en una relación binaria de orden ($<$, $\leq$, $=$, $>$ o $\geq$) entre una propiedad de un componente, conector, etc. de la arquitectura (e.g. `server1.responseTime`) y una constante numérica. La implementación mencionada recibe el nombre de *NumericBinaryRelationalConstraint* y su código puede verse en el apéndice A.

Dado que, como hemos mencionado anteriormente, las restricciones predican sobre propiedades del sistema en ejecución, cuyos valores son sensibles al "ruido" producido por la presencia de posibles *outliers*, Arco Iris utiliza, al igual que Rainbow, el concepto de *Exponential Moving Average* (presentado en la sección 2.4.7) para evitar este efecto negativo que afecta tanto a la detección del entorno actual como a la verificación de los escenarios.

Cuantificación de las Restricciones

Rainbow utiliza las reglas e invariantes provistas en el lenguaje Acme para definir restricciones sobre el modelo de la arquitectura del sistema a reparar. Estas reglas e invariantes, se limitan a predicar únicamente sobre una instancia específica o bien sobre todas las instancias de un determinado tipo de componente.

Ahora bien, en Rainbow, cuando la restricción aplica a un **tipo** de componente de la arquitectura, pueden darse dos casos: que se predique sobre el **valor promedio** de todas las instancias de dicho tipo de componente o bien sobre la **sumatoria**. Para implementar estos casos, Rainbow opta por interpretar las restricciones impuestas sobre el modelo en Acme como si predicaran implícitamente sobre el *promedio* de todos los valores; y por otro lado,

ofrece como mecanismo para expresar restricciones sobre la *sumatoria* de todos los valores, la definición de precondiciones dentro de las estrategias de reparación definidas en en lenguaje *Stitch*. En la figura 12 se observa un ejemplo de cómo Znn impone restricciones sobre la sumatoria del costo de los servidores del sistema.

```
import op "org.sa.rainbow.stitch.lib.Model";
...
define float totalCost = Model.sumOverProperty("cost", servers);
define boolean hiCost = totalCost >= M.THRESHOLD_COST;
...
/* This Strategy is triggered by the total server costs rising above acceptable
 * threshold; this Strategy reduces the number of active servers
 */
strategy ReduceOverallCost
[ hiCost ] {
  t0: (hiCost) -> dischargeServers(1) @[2000 /*ms*/] {
    t1: (!hiCost) -> done;
    t2: (lowRespTime && hiCost) -> do[2] t0;
    t3: (default) -> TNULL;
  }
}
...
```

Figura 12: Estrategia que apaga un servidor para mejorar el costo de servidores.

Se observa la utilización de una clase Java (`Model.sumOverProperty("cost", servers)`) para obtener la sumatoria de los costos de los servidores. Esta forma de establecer restricciones (el "qué") se encuentra claramente acoplada con la forma de reparar el sistema ("el cómo") y además, no existe una forma única de definir restricciones sobre el modelo, lo cual puede conducir a confusiones y potenciales inconsistencias; así como también redunda en poca flexibilidad para el usuario que define restricciones sobre el modelo, ya que esta manera de hacerlo requiere conocimiento técnico no trivial.

Para subsanar esta falencia de diseño, Arco Iris agrega a las restricciones numéricas el concepto de **Cuantificador**. El cual es extensible y actualmente soporta dos opciones: **sumatoria** o **promedio**, los que especifican respectivamente si la condición predica sobre la sumatoria o el promedio de los valores de una determinada propiedad, para todas las instancias en *runtime* de un determinado componente.

3.4. Flexibilización de la Auto Reparación con QAS

Actualmente Rainbow posee conocimiento sobre el sistema a adaptar mediante el modelo de su arquitectura expresado en el lenguaje de descripción de arquitecturas Acme. En dicho modelo también se encuentran definidos invariantes y restricciones sobre el comportamiento esperado del sistema, información que luego es utilizada por Rainbow para detectar cuándo

el sistema se encuentra en un estado no deseado. Existen algunos problemas con respecto a cómo Rainbow lleva a cabo esta tarea:

- Rainbow verifica que se satisfagan **todos** los invariantes y restricciones del modelo. En caso de detectar que alguna de estas condiciones no se satisfacen, no existe un correlato directo entre la restricción o invariante que deja de cumplirse con la o las estrategias de reparación que solucionan dicho problema.
- Como consecuencia del punto anterior, Rainbow debe considerar todas las estrategias definidas para el sistema al momento de determinar la mejor estrategia de reparación a ejecutar.
- A fin de establecer un correlato indirecto entre aquello que causó el funcionamiento no esperado del sistema y las estrategias de reparación candidatas, se **replican** las restricciones o invariantes ya definidas en el modelo como precondiciones en las estrategias.

Se observa que este esquema es poco flexible ya que ante cualquier modificación en los requerimientos de auto reparación definidos para el sistema, el usuario deberá modificar tanto el modelo de la arquitectura como las estrategias de reparación asociadas. Esta redundancia, además de significar un trabajo de configuración innecesario para el usuario, puede llevar a inconsistencias que provocarían un comportamiento indeseado en la auto reparación del sistema.

Por otro lado, el modificar el comportamiento de la auto reparación en Rainbow requiere un conocimiento técnico no trivial, por lo que el cambio debería ser realizado por el arquitecto del sistema, esto hace a Rainbow menos flexible ante cambios en los requerimientos de los usuarios no técnicos, ya que éstos no podrían llevar a cabo la modificación de requerimientos sin la asistencia activa de un usuario técnico.

Con el objetivo de subsanar las falencias anteriormente comentadas, Arco Iris extiende el conocimiento que Rainbow posee sobre el sistema y lo organiza de una manera más adecuada. Esencialmente, se incluye información sobre los atributos de calidad del sistema que son relevantes para los *stakeholders*. Se permite, por ejemplo, poder describir la importancia relativa de la eficiencia, la disponibilidad, etc.; definiendo así una serie de *tradeoffs* entre distintos atributos de calidad requeridos para el sistema. El enfoque propuesto para lograr esto consiste en especificar **Escenarios de Atributos de Calidad**, tal cual fueron descritos en la sección 2.5, extendiendo el concepto con información orientada a flexibilizar la auto reparación. Esta extensión es explicada en detalle en la sección 3.5.

A continuación se detallan las soluciones introducidas por Arco Iris para solucionar los problemas que posee Rainbow mencionados anteriormente:

- Gracias al uso de QAS, el invariante o restricción que en Rainbow se encontraba descrito en el modelo de la arquitectura, pasa a formar parte del *response measure*. Luego, para

solucionar la falta de correlato entre la detección del problema y su reparación, Arco Iris extiende el concepto de escenario, agregándole el conocimiento de cuáles son las estrategias de reparación candidatas para reparar el sistema, en caso de que dicho escenario no se cumpla.

- Conforme a lo explicado en el punto anterior, se evita la necesidad de considerar todas las estrategias como candidatas, tal cual era el comportamiento original de Rainbow.
- Al contar con el correlato explícito entre el problema y sus posibles soluciones, ya no es necesario especificar precondiciones a las estrategias de reparación, flexibilizando y simplificando de esta manera el uso del *framework* así como también eliminando la existencia de posibles inconsistencias en su configuración.

En la Figura 13 se puede observar el modelo diseñado para representar los QAS en Arco Iris. Este modelo cumple una tarea fundamental ya que toda esta información será manipulada constantemente por Arco Iris para llevar a cabo la auto reparación en base a las expectativas plasmadas por los *stakeholders* al crear los escenarios.

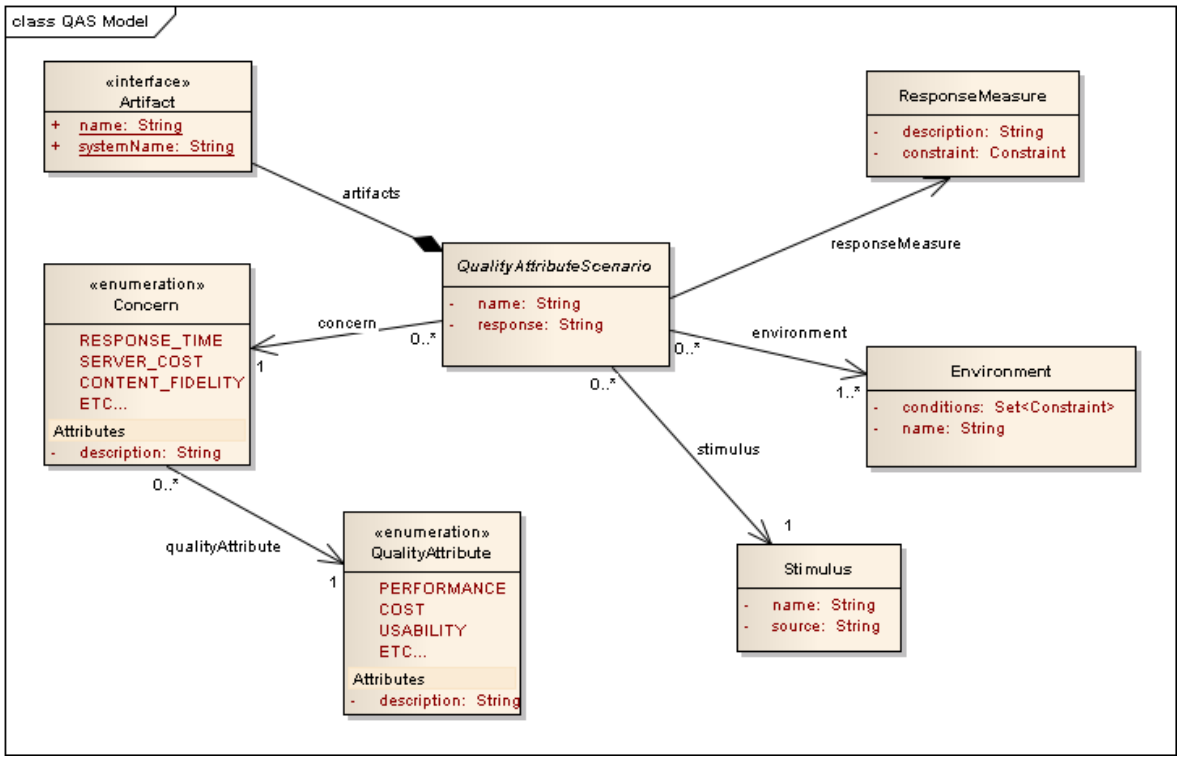


Figura 13: Modelo de QAS

De todos los atributos que posee un QAS, el **Estímulo**, el **Artefacto**, el **Entorno** y la **Cuantificación de la Respuesta** son particularmente relevantes a los fines de establecer información útil para el mecanismo de auto reparación. A lo largo de las próximas secciones

se detalla el uso que hace Arco Iris de esta información para modificar, optimizar y flexibilizar la auto reparación llevada a cabo por Rainbow.

3.4.1. El Estímulo en la Auto Reparación

El **estímulo** de un escenario normalmente se asocia a un evento desencadenado en el sistema por la acción del alguno de sus usuarios. Dicho evento es el punto de entrada del escenario, el disparador (interno o externo) que inicia la interacción con el sistema, y más particularmente, con el artefacto del escenario en cuestión. Por ejemplo, supongamos que en un sistema de administración de cuentas bancarias un cliente intenta hacer una transferencia, en este caso la fuente del estímulo sería el cliente y el estímulo en sí mismo sería *realizar transferencia*. En términos de auto reparación, generalmente el estímulo se encuentra asociado a una operación provista por el artefacto del sistema, el cual puede ser un componente, un conector o un sub sistema.

Saber cuál es el estímulo de cada escenario permite optimizar la auto reparación, ya que habiendo ocurrido determinado estímulo, Arco Iris podrá detectar cuáles son los potenciales escenarios que pueden verse afectados y trabajará verificando dicho subconjunto, acotando así la cantidad de restricciones a verificar y, consecuentemente, optimizando el tiempo de la auto reparación.

Cabe destacar que también se ofrece la posibilidad de no especificar el estímulo al configurar un escenario, es decir, se permite la opción de que el escenario aplique siempre, independientemente del estímulo que haya impactado al sistema. Esto puede ser útil para casos genéricos, por ejemplo, si se requiere que el tiempo de respuesta experimentado por el usuario nunca sobrepase determinado umbral sin importar la funcionalidad del sistema que el usuario esté utilizando. En particular, se podría no configurar ningún estímulo en ningún escenario, y en ese caso, Arco Iris verificará que se satisfagan todos los escenarios en cada iteración de la auto reparación cada vez que un estímulo perteneciente a cualquier escenario impacte sobre el sistema. Como puede se observar, es recomendable configurar el estímulo específico para cada escenario a fin de conseguir un funcionamiento más preciso del *framework*. Entonces, a diferencia de Rainbow, quien debe verificar todas las restricciones en todo momento, Arco Iris permite refinar el conjunto de escenarios que pueden verse afectados por efecto del estímulo que ocasiona el problema, reduciendo así la cantidad de restricciones e invariantes a verificar.

Para notificar sobre el estímulo desencadente fue necesario extender los *probes*, que, como ya mencionamos anteriormente, son los componentes encargados de extraer del sistema información referente a un determinado *concern*, así también como de volcar dicha información en una cola compartida con otros *probes*. De este *bus*, los *gauges* consumirán la información para luego interpretarla y transformarla en cambios en los valores de propiedades del modelo de la arquitectura.

La extensión de los *probes* implicó la necesidad de extender los *gauges* de manera tal que puedan interpretar la nueva información sobre estímulos agregada en cada uno de los mensajes creados por los *probes*.

Debido a que en el diseño original de Rainbow, los mensajes que los *probes* envían a los *gauges* son meras cadenas de texto, las extensiones realizadas a ambos componentes no presentaron mayores inconvenientes.

Para graficar la diferencia en la implementación que genera el agregar el estímulo en la auto reparación, se ejemplificará una extensión a un *probe* utilizado por Znn. El *probe* escogido para extender y así poder ser utilizado en Arco Iris consiste simplemente en invocar un servicio, esperar su respuesta y medir el tiempo transcurrido entre la invocación y la respuesta; ésta será la información que el *probe* reportará en el *bus* que luego consultará e interpretará el *gauge* correspondiente.

A continuación se puede observar la configuración original utilizada en Znn para instanciar un *probe* del tipo ClientProxyProbe, esta configuración se puede encontrar en el archivo `probes.yml`, donde se declaran todos los *probes* que serán instanciados al iniciar Rainbow:

```
probes:
ClientProxyProbe0:
        alias: clientproxy
        location: "localhost"
        type: java
        javaInfo:
                class: org.sa.rainbow.translator.znews.probes.ClientProxyProbe
                period: 2000
                args.length: 1
                args.0: "http://delegate.oracle/"
```

Un ejemplo de la información que reporta un `ClientProxyProbe` con la configuración presentada anteriormente podría ser:

```
[fri may 13 22:15:04 2011]<clientproxy> localhost: 1532 ms
```

Tanto la configuración como la implementación de `ClientProxy` son muy similares a las de su contraparte de Arco Iris, `ClientProxyWithStimulus`. En el apéndice B se pueden observar ambas implementaciones, mientras que la configuración de una instancia de `ClientProxyWithStimulus` se puede observar en la figura 14.

Por último, se muestra la información reportada por una instancia de `ClientProxyWithStimulus` con la configuración que se acaba de presentar:

```
[fri may 13 22:15:04 2011]<clientproxyWithStimulus> localhost<stimulus:delegateStimulus>: 1532 ms
```

```
ClientProxyProbeWithStimulus0:
        alias: clientproxyWithStimulus
        location: "localhost"
        type: java
        javaInfo:
                class: ar.uba.dc.arcoiris.znn.probes.ClientProxyProbeWithStimulus
                period: 2000
                args.length: 2
                args.0: "http://delegate.oracle/"
                args.1: "delegateStimulus"
```

Figura 14: Configuración de una instancia de ClientProxyWithStimulus.

Algunas formas posibles de implementar *probes* son:

- *Parsing* e interpretación de información existente en archivos de logs del sistema o en el servidor en el cual se ejecuta.
- Invocaciones idempotentes a servicios provistos por el sistema, con el fin de obtener métricas sobre las respuestas.
- El sistema a adaptar puede ofrecer servicios específicos que provean información relevante para su auto reparación.

3.4.2. El Artefacto en la Auto Reparación

El **artefacto** se refiere al componente, conector o sub sistema afectado por el escenario, cualesquiera de ellos se encuentran descritos en el modelo de la arquitectura.

La asociación de un artefacto particular a un escenario permite acotar las propiedades sobre las cuales podrá predicar la **cuantificación de la respuesta** (ver sección 3.4.4).

La arquitectura presentada en Znn servirá para ejemplificar el concepto de artefacto. Cabe recordar que se trata de una arquitectura de estilo cliente-servidor, donde el componente de tipo *ClientT* se define de la siguiente manera:

```
Component Type ClientT extends ArchElementT with {

  Property deploymentLocation : string <<  default : string = "localhost"; >> ;

  Property experRespTime : float <<  default : float = 100.0; >> ;

  Property requestRate : float <<  default : float = 0.0; >> ;
}
```

Figura 15: Definición del tipo *ClientT*.

Suponiendo que se define el siguiente escenario:

> *"Znn news debe servir el contenido de las noticias a los clientes en un tiempo de respuesta menor a 3 segundos en un entorno de operación normal."*

Dado que en la arquitectura de Znn se modela a los clientes como componentes de la arquitectura, con una propiedad específica `experRespTime` que representa el tiempo de respuesta experimentado por el usuario, claramente se observa que en este ejemplo *ClientT* es el artefacto sobre el cual ha de predicar la verificación de validez del escenario anteriormente detallado. Más adelante, en la sección 3.4.4, se podrá observar cómo se configura dicha restricción en un escenario de atributos de calidad.

3.4.3. El Entorno en la Auto Reparación

El Concepto de Entorno en Rainbow

Si bien Rainbow utiliza el concepto de entorno y, para contribuir a la confusión del lector, lo llama *scenario*, cabe aclarar que tal entorno es configurable aunque estático, es decir, no se adapta dinámicamente a los cambios que experimenta el sistema en tiempo de ejecución. Rainbow implementa el entorno como una distribución específica de pesos relativos entre los distintos *concerns* del sistema. A continuación se muestra un ejemplo de un entorno (*scenario*) de Rainbow utilizado por Znn:

```
weights:
        scenario 1:
                uR: 0.35
                uF: 0.4
                uC: 0.25
        scenario 2:
                uR: 0.5
                uF: 0.3
                uC: 0.2
        scenario 2b:
                uR: 0.5
                uF: 0.2
                uC: 0.3
```

Observar que cada entorno asigna un valor a cada *concern* del sistema, en donde, en este caso, `uR` representa al Tiempo de Respuesta, `uF` a la Fidelidad de la información y `uC` al Costo. Notar que la sumatoria de los pesos de los distintos *concerns* para cada entorno debe ser igual a 1.

Esta información debe ser configurada en el archivo de configuración *utilities.yml* utilizado por Rainbow. Aquí, como se puede observar, se definen más de un entorno, uno de los cuales luego deberá ser seleccionado por el arquitecto en el archivo de configuración

`rainbow.properties` previo a la inicialización, intentando predecir cuáles serán las condiciones en las que el sistema deberá responder. Este entorno es utilizado por Rainbow al momento de seleccionar una estrategia para reparar el sistema. En la sección 3.9 se explica en detalle su utilización.

El Entorno en Arco Iris y su Importancia para la Auto Reparación

El concepto de *scenario* de Rainbow es similar al concepto de Entorno planteada por QAS, el cual, a su vez ha sido extendido por Arco Iris. La diferencia fundamental radica en que, mientras en Rainbow el usuario debe predecir en qué entorno operará el sistema, Arco Iris lo detectará automáticamente. Más adelante en esta sección, se explicará en detalle cómo se lleva a cabo esta tarea.

Concentrándonos ahora únicamente en Arco Iris, y de acuerdo a la definición de QAS, se define al **Entorno de Ejecución**, o simplemente, **Entorno** como el estado en el que el sistema se encuentra cuando recibe el estímulo que desencadena el escenario. A modo de ejemplo: al recibir una solicitud de creación de una cuenta bancaria el sistema puede encontrarse en "operatoria normal", en "alta carga" o quizás tal vez se encuentre "fuera de operación".

El entorno condiciona la validez del escenario en cuestión a que el sistema se encuentre en un determinado estado, ya que, por ejemplo, una respuesta esperada aceptable o fácil de cumplir bajo un entorno puede ser inaceptable o muy costosa en otro.

En el escenario planteado en la sección anterior, si el sistema se encontrase en un entorno de "alta carga" el escenario se satisfaría trivialmente, ya que para considerar dicho escenario el sistema debería encontrarse en un entorno de "operación normal".

De esta manera, el entorno del escenario es un elemento que permite a Arco Iris optimizar la búsqueda de escenarios que no se satisfacen en un determinado instante ante la recepción de un estímulo, ya que se ignorarán aquellos escenarios cuyo entorno posea condiciones que no se cumplen en dicho instante. Más adelante se profundizará sobre dichas condiciones.

Escenarios modelados con varios entornos

Si bien el concepto de escenario de atributo calidad, tal cual fue definido en la sección 2.5, incluye únicamente un sólo entorno, en Arco Iris se ha decidido modelar al escenario con una **colección de entornos**. A fin de justificar esta decisión de diseño, supongamos que un *stakeholder* desea que un determinado escenario sea válido en varios entornos. En ese caso, siguiendo la definición estricta de QAS dónde un escenario sólo posee un entorno, el *stakeholder* debería crear varios escenarios idénticos, todos ellos difiriendo únicamente en su entorno. Esto es claramente inconveniente, más aún considerando que la cantidad de entornos configurables por el usuario no está acotada, lo cual podría llevar a una explosión innecesaria de escenarios cuasi idénticos.

Ahora bien, habiendo dicho lo anterior, también puede ocurrir la situación dónde, de acuerdo al entorno de ejecución, se desea tener en cuenta subconjuntos distintos de estrategias de reparación. En este caso, no existe otra opción más que configurar distintos escenarios.

Cabe destacar que, a fines de simplificar el problema y acotar así el alcance de este trabajo, en el modelo de Arco Iris se establece una restricción significativa: se presupone que el usuario de Arco Iris no cargará en el sistema entornos cuyas condiciones de aplicabilidad tengan intersección no nula. Esta simplificación, si bien no representa mayor problema a los fines de este trabajo, sí restringe futuras extensiones del *framework*. La naturaleza de tal restricción y una posible solución son abordadas en detalle en la sección 6.6.

Estructura y Características del Entorno

La estructura del entorno consiste en:

- Un nombre,
- Un conjunto de condiciones, y
- Un mapa `<Concern, Peso>`

El **nombre** es simplemente una cadena de texto que sirve para rápidamente identificarlo.

Las **condiciones** son predicados que predican sobre los valores de las propiedades del modelo de la arquitectura del sistema en tiempo de ejecución. Para que un sistema en ejecución se encuentre en un determinado entorno, deben satisfacerse **todas** sus condiciones. Para más información sobre la implementación en Arco Iris de las condiciones del entorno, ver la sección 3.3.2.

El objetivo del **mapa de pesos** presente en la estructura del entorno es el de definir la importancia relativa de cada *concern* cuando el sistema se encuentra en el entorno en cuestión. Dicho mapa debe contener todos los *concerns* definidos para el sistema y la suma de sus pesos debe ser igual a uno. En la sección 3.9.3 se detalla de qué manera Arco Iris hace uso de estos pesos para escoger la mejor estrategia de reparación.

El Entorno "ANY"

Una característica particular de Arco Iris es la de permitir expresar, mediante la selección de un pseudo-entorno preexistente denominado "ANY", que un determinado escenario aplica bajo cualquier entorno. Esto es equivalente a que el entorno del escenario no posea condición alguna, es decir, que el escenario aplica siempre, trivialmente, sin importar las condiciones actuales del sistema en ejecución. Esto puede resultar útil en algunos casos ya que simplifica la configuración del escenario, pero es importante tener en cuenta que, por otro lado, el rendimiento de la auto reparación se verá reducido puesto que el configurar uno o más entornos específicos por escenario otorga mayor precisión al permitir establecer la importancia (peso relativo) de cada *concern* según el estado en el que el sistema se encuentre.

El entorno "ANY" será asignado automáticamente al escenario si el usuario no especifica ningún entorno o, si por el contrario, decide explícitamente que el escenario aplique **en cualquier entorno de ejecución**. En cualquiera de los dos casos, Arco Iris asignará a todos los *concerns* el mismo peso, con la intención de evitar otorgarle más importancia a algún *concern* en particular. Esta decisión de equidistribuir los pesos relativos por *concern* es, desde ya, arbitraria y se reconoce una posibilidad de mejora en vistas de un trabajo futuro. Para más información, ver 6.5.2.

Para ejemplificar la importancia de configurar un entorno correctamente y no hacer abuso del pseudo-entorno "ANY", considerar, por ejemplo, un sistema el cual se encuentra bajo excesiva carga, y los *stakeholders* consideran que bajo tales circunstancias lo más prioritario es optimizar la eficiencia del sistema. En ese caso, de no especificar el entorno no será posible otorgarle mayor peso a la eficiencia por sobre otros *concerns*, quedando únicamente la opción de aumentar la prioridad[10] de todos los escenarios relacionados con dicho *concern*, tergiversando así dicha información, ya que en realidad lo óptimo sería asignarle más peso al *concern performance* en el entorno de "Alta Carga". Puesto de otra manera, el establecer que un escenario aplica en cualquier entorno, tiene como consecuencia que Arco Iris interprete que el entorno carece de importancia, dejando de lado los pesos de los *concerns* y distribuyendo equitativamente su importancia relativa, lo cual es equivalente a que el concepto de *concern* no exista.

3.4.4. La Cuantificación de la Respuesta en la Auto Reparación

La **cuantificación de la respuesta** (o, en inglés, *Response Measure*) es quizás la propiedad más importante de un escenario, de ella surgen las restricciones que deben ser evaluadas para que, en caso de no cumplirse, se lance la auto reparación. En pocas palabras, la cuantificación de la respuesta se define como la métrica según la cual se decide si la respuesta del sistema ante un determinado estímulo es aceptable o no.

Para que un escenario se considere bien formado debe quedar claro cuál es la métrica o manifestación observable de su respuesta que se debe satisfacer. Latencia y *throughput* son ejemplos de manifestaciones sobre las cuales puede predicar la cuantificación de la respuesta.

Para graficar la importancia de contar con una cuantificación de la respuesta precisa, supongamos que contamos con la siguiente definición:

> *"El sistema debe ser modificable para poder incorporar un nuevo generador de eventos discretos"*

Esta premisa no es suficiente para medir el éxito de la incorporación de la nueva funcionalidad solicitada, ya que con suficiente tiempo y recursos, casi cualquier modificación es posible.

[10] se profundizará sobre este tema en la sección 3.6

Este escenario requiere una métrica, como por ejemplo: *"Utilizando 160 horas/hombre"*. Esto fuerza al arquitecto a asegurar que el sistema sea modificable basándose en un criterio bien definido y con una métrica aplicable.

Con respecto a la implementación, el componente más importante de la cuantificación de la respuesta es la restricción, cuyo modelo en Arco Iris fue anteriormente explicado en la sección 3.3.2.

En Arco Iris las restricciones se sitúan en el contexto de un escenario de atributo de calidad, proveyendo así mayor visibilidad a los *stakeholders*, las cuales antes se encontraban implementadas en el modelo de la arquitectura utilizando el lenguaje Acme, y consecuentemente, eran sólo conocidas y modificadas por los arquitectos y/o técnicos responsables de configurar el *framework*.

El siguiente ejemplo ha sido extraído de la arquitectura de Znn. Aquí se puede observar de qué manera se implementan las restricciones al utilizar Rainbow:

```
Component Type ClientT {
        Property experRespTime : float <<  default : float = 100.0; >> ;
        rule primaryConstraint = invariant self.experRespTime <= MAX_RESPTIME;
}
```

Es importante recordar que en Rainbow, una vez que se detectó que la auto reparación debe ser lanzada debido a una restricción que dejó de cumplirse, se volverán a evaluar todas las restricciones definidas en las precondiciones de **todas las estrategias** para verificar si aplican o no dependiendo del estado actual del sistema. A continuación se muestra un ejemplo extraído de Znn de una estrategia y su precondición:

```
import model "ZNewsSys.acme" { ZNewsSys as M};
...
define boolean cViolation =
    exists c : ClientT in M.components | c.experRespTime > MAX_RESPTIME;
...
strategy BruteReduceResponseTime
[ cViolation ] {
        ...
        execute some tactic...
        ...
}
```

Como se puede observar, tanto la restricción en el modelo que desencadenó la auto reparación como la precondición de la estrategia son equivalentes, por lo que se duplica la lógica aumentando el costo de procesamiento, haciendo que la reparación sea más costosa y menos escalable. Además al duplicar la configuración, el costo de mantener el *framework* es mayor.

En Arco Iris no será necesario contar con las precondiciones de las estrategias, ya que el conocimiento de cuáles estrategias son capaces de reparar una determinada condición se

encuentra plasmado en cada escenario. En definitiva, al utilizar Arco Iris el usuario accede a la posibilidad de configurar el comportamiento esperado del sistema mediante la cuantificación de la respuesta de cada escenario, tarea que se ve sumamente facilitada al utilizar Arco Iris UI (ver sección 4). Otra diferencia esencial con respecto al uso que Rainbow y Arco Iris hacen de las restricciones, es el momento en el que se aplican. Hemos visto anteriormente que Rainbow posee un componente llamado *"Architecture Evaluator"*, cuya responsabilidad consiste en verificar de a intervalos la arquitectura del modelo, siempre y cuando éste haya sufrido algunas modificaciones y no se esté ya ejecutando un proceso de auto reparación, esto implica verificar absolutamente todas las restricciones del modelo. Arco Iris, en cambio, al descubrir un cambio en el modelo, verifica que los escenarios se satisfagan, pero no todos los escenarios, sino solamente aquellos cuyo estímulo coincida con el que desencadenó la actualización del modelo, optimizando así de manera drástica la cantidad de restricciones a verificar, lo que hace más escalable la auto reparación.

3.5. Modelo Extendido de QAS

Hasta aquí se ha detallado el uso que Arco Iris hace del modelo básico de QAS, pero como se mencionó anteriormente, es necesario extender este modelo para agregar información de auto reparación propiamente dicha.

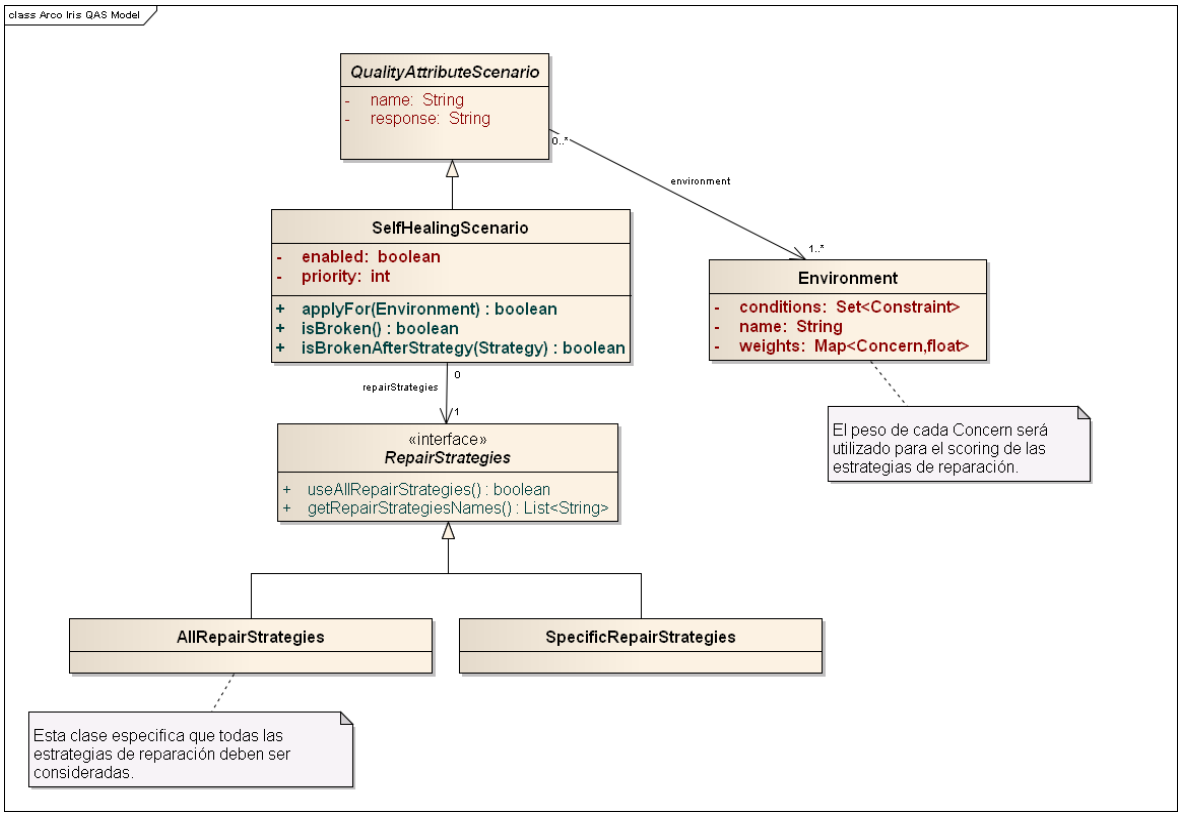


Figura 16: Modelo de Arco Iris

En la figura 16 se puede observar el modelo del **escenario de auto reparación** (`SelfHealingScenario`), el cual, además de ser un QAS tiene toda la información y la lógica necesaria para que Arco Iris pueda llevar a cabo su objetivo de flexibilizar la auto reparación.

Los escenarios de auto reparación pueden ser deshabilitados, considerando Arco Iris solamente aquellos escenarios habilitados al momento de reparar el sistema.

Otra propiedad utilizada por Arco Iris y que no forma parte de la definición original de un QAS, es la prioridad. Cada `SelfHealingScenario` tiene preestablecida su prioridad (ver sección 3.6 para más detalle). Un escenario de Arco Iris también tiene el conocimiento necesario para determinar si se satisface según las condiciones actuales del sistema, y también para predecir si seguirá satisfaciendose luego de una potencial aplicación de una estrategia en particular. Otra extensión a QAS implementada por Arco Iris para optimizar la auto reparación consiste en que cada escenario conozca las estrategias que el arquitecto del sistema y los *stakeholders* consideran que son capaces de reparar el escenario en caso de que este haya dejado de satisfacerse. En la sección 3.7 puede verse en detalle cómo Arco Iris explota esta información. Por último, es importante recordar que se ha extendido el concepto de **entorno** utilizado por QAS, el cual pasa a contar, además de con su nombre, con un conjunto de condiciones que serán utilizadas para saber si un determinado escenario aplica, dado su entorno, para las condiciones actuales del sistema, y además será capaz de determinar la importancia de cada *concern* para el caso en el que el sistema en ejecución se encuentre en este entorno. Esta extensión ya fue explicada en detalle en la sección 3.4.3.

3.6. Prioridades entre Escenarios

Una característica distintiva de Arco Iris, la cual no posee contraparte alguna en Rainbow, es la de ofrecer la posibilidad de **priorizar escenarios** asignándoles prioridades relativas, de modo tal que al momento de escoger una estrategia de auto reparación la estrategia seleccionada no comprometa a alguna otra funcionalidad de la aplicación considerada más importante según la visión de los *stakeholders*.

Para lograr lo antedicho, a cada escenario se le asigna una prioridad, representada por un valor numérico entero positivo que es inversamente proporcional a la importancia que se desea asignarle al escenario. Por ejemplo, un escenario con prioridad 2 es considerado más prioritario que otro con prioridad 6.

Si bien esta nueva funcionalidad abre nuevas posibilidades al usuario, por otro lado, debe manejarse con sumo cuidado ya que es probable que, al configurar escenarios con prioridades *muy grandes*, éstos no tengan prácticamente ningún peso al momento de seleccionar la estrategia de auto reparación a aplicar, con lo cual dichos escenarios nunca serán reparados, careciendo de sentido así su existencia.

La prioridad del escenario es una propiedad fundamental para el correcto funcionamiento

de Arco Iris, veremos su importancia al detallar de qué manera Arco Iris selecciona la estrategia que aplicará para reparar el sistema en la sección 3.9. Por esta razón será de extrema importancia la correcta configuración de las prioridades relativas de los escenarios. Se recomienda realizar la asignación de la prioridad de cada escenario como un paso más del Quality Attribute Workshop (QAW), para más detalle ver la sección 2.5.1.

3.7. Estrategias y su Relación con los Escenarios

Antes de profundizar en el impacto de los escenarios en la selección de la estrategia para reparar el sistema, se repasará brevemente cómo se utilizan las estrategias en Rainbow.

Las estrategias en Rainbow tienen un conjunto de precondiciones que nos indican si dicha estrategia puede ser utilizada para reparar el sistema en un determinado momento. Esto implica que, en cuanto Rainbow detecta un problema en el sistema, deba recorrer todas las estrategias existentes para ver si son aplicables para las condiciones actuales. En otras palabras, Rainbow no tiene forma de determinar **a priori** el subconjunto de estrategias candidatas a reparar el sistema en el preciso momento en que una o más restricciones sobre el modelo de la arquitectura del sistema se dejan de cumplir. Por ende, todas las estrategias disponibles, pasan a ser candidatas.

Por ejemplo, para que una estrategia que soluciona problemas de eficiencia no sea tenida en cuenta al reparar un escenario relacionado a un atributo de calidad distinto, como por ejemplo la mantenibilidad, debe agregarse una precondición como la siguiente en la estrategia:

```
strategy SimpleReduceResponseTime
[ responseTimeConstraintViolation ] {
...
}
```

Donde `responseTimeConstraintViolation` es un predicado que permite determinar si existe algún cliente cuyo tiempo de respuesta haya sobrepasado el umbral máximo permitido:

```
define boolean responseTimeConstraintViolation =
                        exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
```

Con respecto al anterior ejemplo, es importante remarcar que, en caso de que el problema que se está intentando reparar no tenga relación con la eficiencia, Rainbow deberá de todos modos chequear el tiempo de respuesta de todos los clientes, mientras que en Arco Iris existe la posibilidad de evitarlo al configurar correctamente las estrategias que reparan a cada escenario.

Como ya se ha mencionado, en Arco Iris si bien se utilizan el mismo tipo de tácticas y estrategias que en Rainbow (i.e. implementadas con Stitch), se propone desacoplar la detección

del problema de su solución utilizando como medio para ello los escenarios de atributos de calidad. Allí se definen las condiciones para la detección del problema (restricciones dentro de la cuantificación de la respuesta) y se referencian las posibles estrategias de reparación a ser consideradas para su ejecución, en el caso de que el escenario en cuestión se vea comprometido. De esta forma, las estrategias quedan exentas de conocer cuáles son las circunstancias en las que su ejecución tiene sentido. Esto permite al arquitecto tener un mayor control sobre las estrategias a ejecutar en determinadas condiciones, ya que al situarse en un escenario concreto, sabrá cuales serán las soluciones más adecuadas basándose en el entorno del sistema y en la prioridad del escenario en cuestión.

Cabe acotar que Arco Iris también ofrece la posibilidad de configurar el mismo comportamiento brindado por Rainbow: esto se logra simplemente indicando la opción que representa a "todas las estrategias existentes" cuando se configura el escenario (una forma sencilla de especificar ésto, utilizando Arco Iris UI, se explica en la sección 4.5.1). Implementativamente hablando, basta sólo con asignarle al escenario la única instancia de la clase `AllRepairStrategies`.

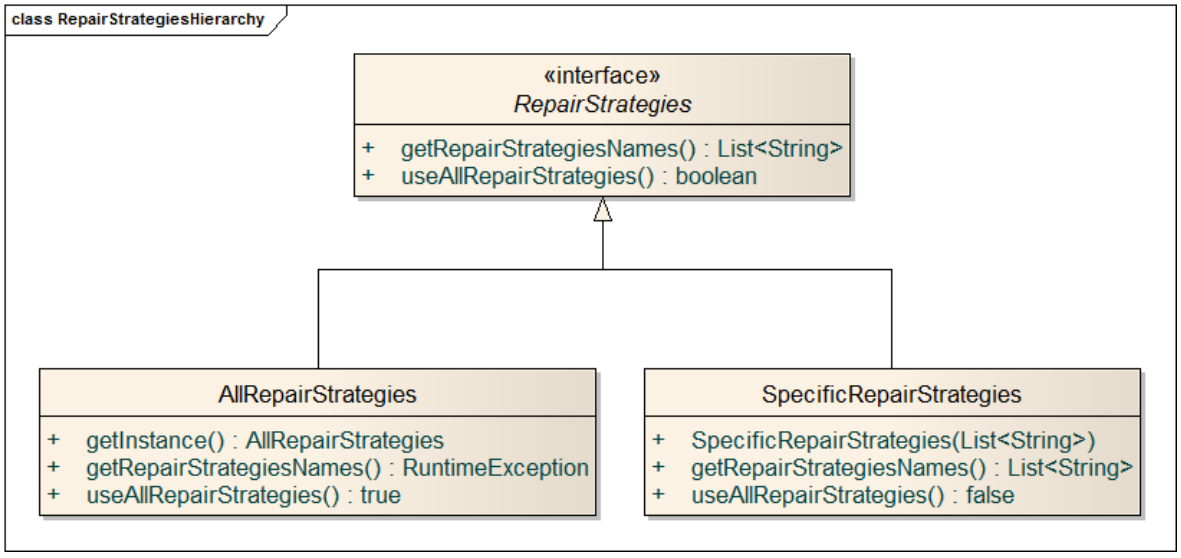


Figura 17: Jerarquía de Estrategias de Reparación en Arco Iris

En Arco Iris, para poder escribir estrategias que involucren varias tácticas, el usuario contará con un mecanismo que le permitirá verificar si los escenarios de un determinado *concern*, y que han sido marcados para reparar, siguen aún sin cumplirse. En base a esta información la estrategia podrá decidir cómo continuar su ejecución.

Como se puede observar en la figura 18, para que este mecanismo funcione, es necesario importar la función `isConcernStillBroken` de la clase `ArcoIrisAdaptationManager`, cuya implementación se observa en la figura 19. Notar que el usuario solamente deberá indicar cual es el *concern* de su interés.

```
import op "org.sa.rainbow.adaptation.ArcoIrisAdaptationManager";
...
define boolean RESP_TIME_STILL_BROKEN =
                ArcoIrisAdaptationManager.isConcernStillBroken("RESPONSE_TIME");
...
/*
 * This Strategy will drop fidelity once, observe, then drop again if necessary.
 */
strategy BruteReduceResponseTime {
  t0: (true) -> lowerFidelity(2, 100) @[5000 /*ms*/] {
    t1: (!RESP_TIME_STILL_BROKEN) -> done;
    t2: (RESP_TIME_STILL_BROKEN) -> lowerFidelity(2, 100) @[8000 /*ms*/] {
      t2a: (!RESP_TIME_STILL_BROKEN) -> done;
      t2b: (default) -> TNULL;  // in this case, we have no more steps to take
    }
  }
}
...
```

Figura 18: Estrategia que baja drásticamente la fidelidad para mejorar el tiempo de respuesta

```
public static boolean isConcernStillBroken(String concernString) {
        Concern concern = Concern.valueOf(concernString);

        boolean result = false;
        for (SelfHealingScenario scenario : currentBrokenScenarios) {
                if (scenario.getConcern().equals(concern) &&
                                scenarioBrokenDetector4CurrentSystemState.isBroken(scenario)) {
                        result = true;
                        break;
                }
        }
        return result;
}
```

Figura 19: Implementación del método *isConcernStillBroken*.

3.8. Activación del Mecanismo de Auto Reparación

3.8.1. Introducción

Con el nuevo enfoque presentado en este trabajo, donde el *Escenario* es el concepto central, es necesario establecer cambios en la lógica aplicada por el *framework* al momento de evaluar restricciones para determinar si es necesario auto reparar el sistema.

En Rainbow, como ya hemos visto, las restricciones del sistema se encuentran embebidas en el modelo de su arquitectura, escrito en lenguaje Acme. Por ejemplo, para especificar que el tiempo de respuesta no debe exceder un determinado umbral es necesario definir una restricción en el tipo de componente `ClientT`, quien es el que posee dicha propiedad

(i.e. `experRespTime`, el tiempo de respuesta experimentado por un cliente). En el siguiente ejemplo tomado de la arquitectura de Znn, se puede apreciar la declaración de tal restricción:

```
Component Type ClientT {
        Property experRespTime : float <<  default : float = 100.0; >> ;
        rule primaryConstraint = invariant self.experRespTime <= MAX_RESPTIME;
}
```

En Arco Iris ya no es necesario definir estas restricciones en la arquitectura, desacoplando así la arquitectura del sistema de la definición de condiciones a evaluar para lanzar la auto reparación. Como se ha explicado en la sección 3.4.4, ahora las restricciones están presentes en la *Cuantificación de la Respuesta* de cada escenario. con el fin de dar mayor visibilidad a los *stakeholders* del sistema. Para esto, se añade la posibilidad de definir las restricciones de manera visual utilizando la herramienta Arco Iris UI (más detalles en la sección 4.2.4).

3.8.2. Desarrollo

Para determinar si el sistema necesita auto repararse, Rainbow utiliza una funcionalidad ofrecida por Acme, que permite evaluar las restricciones descritas en el modelo de la arquitectura. Dicha funcionalidad es provista por una herramienta llamada *Type Checker*. A continuación podemos observar una versión simplificada de cómo Rainbow se sirve de esta utilidad para decidir si debe lanzar la auto reparación o no:

```
public class RainbowModel {
        ...
        RainbowModel() {
                ...
                this.m_acme = /* parse the model from a .acme file */
                this.m_acmeEnv.getTypeChecker().registerModel(this.m_acme);
                // only consider the first system found on the .acme file
                this.m_acmeSys = this.m_acme.getSystems().iterator().next();
                ...
        }

        public void evaluateConstraints () {
                SynchronousTypeChecker typechecker = m_acmeEnv.getTypeChecker();
                typechecker.typecheckAllModelsNow();
                this.m_constraintViolated = !typechecker.typechecks(m_acmeSys);
                if (this.m_constraintViolated) {
                        Set errors = m_acmeEnv.getAllRegisteredErrors();
                        Oracle.instance().writeEvaluatorPanel(m_logger, errors.toString());
                }
        }
        ...
}
```

El componente `ArchEvaluator`, luego de invocar al método `evaluateConstraints` visto más arriba, y habiendo comprobado además que no existe un proceso de adaptación ejecutándose en el sistema, invocará al `AdaptationManager` para que inicie dicho proceso.

Por los motivos explicados en la sección 3.3.2, Arco Iris no hace uso de este mecanismo de *parseo* de restricciones y en cambio, maneja su propio modelo simplificado de restricciones, el cual se encuentra desacoplado del modelado de la arquitectura. Estas restricciones (o *constraints*) serán de vital importancia al momento de activar el mecanismo de auto reparación.

Al iniciar, Arco Iris carga todos los escenarios definidos en su configuración (para más detalle, ver la sección 3.10), y arma un mapa que le permitirá optimizar las verificaciones recurrentes de los escenarios ante la llegada de cada estímulo. El mapa almacena todos los escenarios correspondientes a cada estímulo, así, ante la invocación de un estímulo en el sistema, Arco Iris sólo deberá verificar que se cumplan los escenarios relacionados con aquel estímulo, acotando así de manera sustancial la cantidad de restricciones a verificar.

Paso a paso, el proceso de activación de la auto reparación en Arco Iris consiste en lo siguiente:

1. En el sistema en ejecución ocurre un evento, y uno o más *probes* de Arco Iris lo detectan y publican información sobre dicho evento en una cola compartida (ya existente en Rainbow).

2. Un *gauge* configurado para "escuchar" este tipo de eventos, toma el mensaje de la cola y lo traduce a otro mensaje que consiste en una tripla `<Propiedad,Valor,Estímulo>`, la cual a su vez será depositada en otra cola compartida entre todos los *gauges*.

3. El componente `ArcoIrisGaugeCoordinator` chequea periódicamente esta última cola en búsqueda de nuevas instrucciones para actualizar el modelo arquitectural con los cambios ocurridos en *runtime*. Al encontrar el mensaje dejado por el *gauge*,informará a `ArcoIrisModel` (una extensión de `RainbowModel`) sobre dicho cambio de propiedad y el estímulo que lo generó.

4. *a*) Suponiendo que no existe un proceso de adaptación en progreso, `ArcoIrisModel` procede a actualizar el valor de la propiedad en cuestión en el modelo de la arquitectura del sistema, utilizando la información presente en la anteriormente mencionada tripla.

   b) En el caso en que ya exista un proceso de adaptación ejecutándose, `ArcoIrisModel` no toma ninguna acción ante el pedido de actualizar el valor de la propiedad en cuestión, puesto que el sistema se encuentra en un estado inestable debido a las reparaciones en curso y el modificar los valores a algunas propiedades del modelo podría llegar a ser perjudicial. En otras palabras, cuando se lanza la adaptación, no se reciben mas actualizaciones sobre el modelo hasta que dicha adaptacion finalice.

5. Además de actualizar el valor de o de las propiedades en el modelo arquitectural, `ArcoIrisModel` busca aquellos escenarios que puedan llegar a verse afectados por el estímulo que desencadenó el cambio de propiedad en el modelo (recordemos que dicho estímulo está presente en la tripla que le llega a `ArcoIrisModel`).

6. Una vez identificados los escenarios potencialmente afectados, se verifica que se cumplan las restricciones expresadas en la **cuantificación de la respuesta**, siempre y cuando el entorno del escenario coincida con el entorno en el que se encuentra el sistema actualmente, de lo contrario, el escenario se considera verificado automáticamente.

7. En caso de detectar que uno o más escenarios dejaron de cumplirse, el `ArcoIrisModel` notifica al `ArcoIrisAdaptationManager` (una extensión del `AdaptationManager` de Rainbow) para que inicie la auto reparación del sistema.

8. En primer término, el `ArcoIrisAdaptationManager` debe decidir cuál será la estrategia más conveniente a aplicar, es decir, qué estrategia maximiza la *Utilidad del Sistema* (ver 3.9.1) en el entorno de operación actual. Tanto este proceso como el concepto de *Utilidad del Sistema* serán cubiertos en detalle en las siguientes dos secciones.

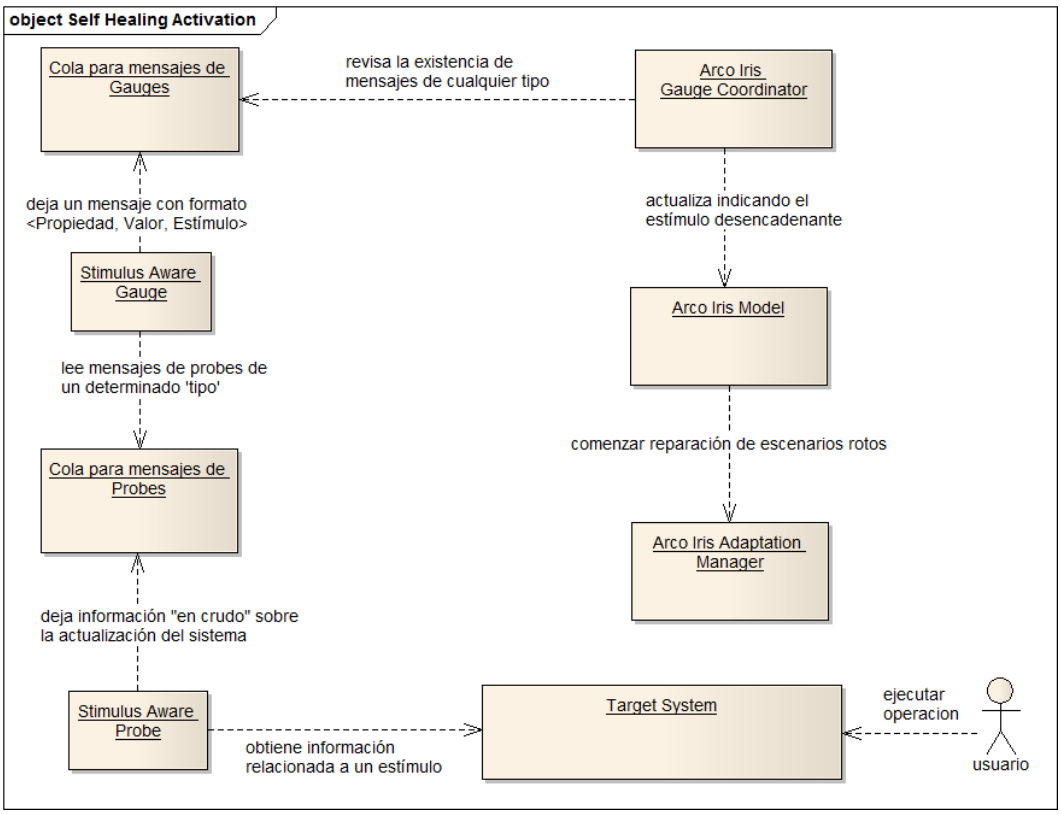


Figura 20: Activación de la estrategia de reparación en Arco Iris

3.9. Selección de la Estrategia

Una vez determinado el subconjunto de estrategias a considerar para su selección, es necesario asignarle un valor a cada una para poder compararlas y seleccionar la estrategia, que luego de ser aplicada, maximice el rendimiento del sistema. Para esto, se definirá el concepto de *Utilidad del Sistema* y se presentará la heurística propuesta para su cálculo en Arco Iris.

3.9.1. Utilidad del Sistema

Rainbow utiliza la **teoría de la utilidad**, concepto forjado en el estudio de la economía, que permite asignar una medida relativa de satisfacción (i.e. *utilidad*) sobre un sistema sobre el cual es necesario medir el impacto ante determinados cambios.

En Rainbow, un sistema brinda un 100 % de utilidad cuando cumple todos sus requerimientos y no viola ninguna de las restricciones planteadas. Desde la perspectiva de Arco Iris, la utilidad se maximiza cuando todos los escenarios definidos se cumplen, consecuentemente, en la medida en que los escenarios dejan de cumplirse, la utilidad del sistema baja.

Rainbow posee un mecanismo que permite predecir cuál estrategia dejaría al sistema en una mejor situación, de acuerdo al valor arrojado por el cálculo de la utilidad del sistema luego de la supuesta ejecución de cada una de las estrategias candidatas. Si bien este mecanismo de predicción es reutilizado por Arco Iris, el cálculo de la utilidad del sistema, es sustancialmente diferente.

En el archivo de configuración llamado `utilities.yml`, Rainbow permite configurar:

- cúales son los *concerns* que manejará el sistema y la propiedad del modelo de la arquitectura asociada a él.
- como se vió anteriormente en la sección 3.4.3, se configura antes de que Rainbow inicie, el entorno estático de ejecución en el que se desea que funcione, y dicho entorno no cambia de acuerdo a la situación real del sistema en *runtime*. Estos entornos estáticos (llamados “scenarios”, que nada tienen que ver con los escenarios de Arco Iris) son ni más ni menos que un diccionario que especifica qué peso (o importancia) tiene en el sistema cada *concern*.
- por cada uno de los *concerns* que manejará el sistema, se define una función de utilidad, la cual, dada un valor de la propiedad del modelo relacionada (e.g. tiempo de respuesta), arroja un valor entre 0 y 1, cuyo uso se explicará en breve.

Se observa a continuación, un ejemplo de una configuración de este tipo, tomada de Znn:

```
...
utilities:
        uR:
                label:       Average Response Time
                mapping:     "[EAvg]ClientT.experRespTime"
                description: "Client experienced response time in milliseconds, R, defined as
                              a float property 'ClientT.experRespTime' in the architecture"
                utility:
                        0:    1.00
                        100:  1.00
                        200:  0.99
                        500:  0.90
                        1000: 0.75
                        1500: 0.50
                        2000: 0.25
                        4000: 0.00
        uF:
                label:        Average Fidelity
                mapping:      "[EAvg]ServerT.fidelity"
        ...
        weights:
                scenario 1:
                        uR: 0.35
                        uF: 0.4
                        uC: 0.25
                scenario 2:
                        uR: 0.5
                        uF: 0.3
                        uC: 0.2
        ...
```

En las primeras líneas se observa la configuración del *concern* tiempo de respuesta, cuyo identificador será `uR` y la propiedad del modelo con la cual se corresponde es `experRespTime` del tipo de componente `ClientT`. A continuación se encuentra definida la función de utilidad para dicho *concern*, la cual, dado un valor de tiempo de respuesta en milisegundos, retorna un valor interpolado entre 0 y 1, que determina el porcentaje de utilidad que el sistema brinda para el *concern* en cuestión; dónde 0 representa ausencia de utilidad alguna y 1 representa que el sistema brinda el máximo de utilidad posible. Por ejemplo, para un tiempo de respuesta de 2000 milisegundos, el sistema brinda un 25 % de la utilidad para dicho *concern*.

Se observa también la configuración estática de entornos de ejecución (denominados *scenarios* por Rainbow), la cual fue explicada en detalle en la sección 3.4.3. Recordar que uno de estos entornos será seleccionado en la configuración de Rainbow como el entorno actual y, por ende, sus pesos serán utilizados en el cálculo de la *Utilidad del Sistema*, cuyo pseudocódigo se presenta a continuación:

```
utilidadDelSistema = 0
valoresPorConcern = valores actuales del sistema para cada concern (e.g. uR=1500, uF=5)
Por cada concern
        uf = Funcion de Utilidad para el concern actual
        valorConcern = valoresPorConcern.get(concern) (e.g. 1500)
        valorUtilidad = uf(valorConcern) (e.g. 0.50)
        valorPonderado = valorUtilidad * peso estático asignado al concern (e.g. 0.35)
        utilidadDelSistema = utilidadDelSistema + valorPonderado
```

El concepto de *Utilidad del Sistema* es clave para comprender los mecanismos utilizados por Rainbow y Arco Iris para puntuar las estrategias candidatas a ser elegidas para reparar una situación anómala del sistema que se está adaptando.

3.9.2. Puntuación de Estrategias según Rainbow

Una vez obtenido el subconjunto de estrategias aplicables, Rainbow procede a asignarle un puntaje a cada una y luego **aplica la estrategia de mayor puntaje**.

Para calcular el puntaje de una estrategia, Rainbow obtiene dos datos:

1. los pesos asignados a cada *concern* de acuerdo al entorno estáticamente configurado antes de que inicie el *framework*. (para más detalle sobre estos pesos, remitirse a la sección 3.4.3)

2. calcula estimaciones del valor de cada *concern* luego de la potencial aplicación de la estrategia en cuestión y aplica a dichos valores las respectivas funciones de utilidad definidas en el archivo de configuración `utilities.yml`. (ver sección 3.9.1)

Para estimar el valor de cada *concern* luego de la ejecución de una determinada estrategia sobre el sistema, y considerando que las estrategias se componen de tácticas, cada táctica deberá explicitar los *concerns* sobre los cuales impacta y en qué medida lo hace. Por ejemplo, en el caso de "bajar un servidor" en Znn se especifica la siguiente meta información asociada a la táctica, representando el impacto estimado sobre el sistema:

```
dischargeServers:
        uR: +144
        uF: 0
        uC: -1.00
```

Esto significa que una vez ejecutada esta táctica, y de manera aproximada, la eficiencia se verá degradada en 144 milisegundos y la fidelidad permanecerá intacta, mientras que el costo se reducirá en 1 unidad.

A fin de comprender la heurística aplicada por Rainbow para obtener las estimaciones explicadas en el punto 2, es útil recordar que las estrategias poseen líneas etiquetadas (e.g. `t0, t1`, etc...), las cuáles pueden entenderse como "ramas" del algoritmo, por ejemplo:

```
strategy SimpleReduceResponseTime
[ cViolation ] {
  t0: (/*hiLoad*/ cViolation) -> enlistServers(1) @[1000 /*ms*/] {
    t1: (!cViolation) -> done;
    t2: (/*hiRespTime*/ cViolation) -> lowerFidelity(2, 100) @[3000 /*ms*/] {
      t2a: (!cViolation) -> done;
      t2b: (default) -> TNULL;  // in this case, we have no more steps to take
    }
  }
}
```

Así, Rainbow estima el valor de cada *concern* para una estrategia dada, considerando a) la **probabilidad** que tiene cada rama de la estrategia de ser ejecutada y b) los atributos de las tácticas presentes en cada rama, esto es, el impacto que cada táctica tiene sobre los *concerns* del sistema. Estos valores luego serán ponderados por el peso estático de cada *concern* (punto 1) para obtener finalmente el puntaje de la estrategia.

Se presenta a continuación, el pseudocódigo del cálculo de puntuación de una estrategia llevado a cabo por Rainbow:

```
puntaje = 0
estimacionPorConcern = Estimar los valores para cada concern luego de aplicar la estrategia
Por cada concern
        uf = Funcion de Utilidad para el concern actual
        estimacion = estimacionPorConcern.get(concern)
        valorUtilidad = uf(estimacion)
        valorPonderado = valorUtilidad * peso asignado al concern en la config. estática de Rainbow
        puntaje = puntaje + valorPonderado
```

3.9.3. Puntuación de Estrategias según Arco Iris

En el caso de Arco Iris el primer paso será calcular la utilidad del sistema antes de iniciar la auto reparación, asegurándose así de que la estrategia a aplicar no perjudique el rendimiento actual de la aplicación, es decir, la estrategia que resulte tener la puntuación máxima deberá mejorar la utilidad actual del sistema, de lo contrario Arco Iris no procederá a ejecutarla. El mecanismo utilizado para calcular la utilidad actual del sistema es idéntico al algoritmo de puntuación de una estrategia explicado a continuación, salvo que los cálculos son realizados en base a los valores actuales (reales) del sistema, dejando de lado las estimaciones heurísticas necesarias para predecir el impacto de cada estrategia.

La heurística aplicada por Arco Iris para el cálculo de la puntuación de una estrategia utilizará el conocimiento plasmado en los escenarios para determinar la mejor estrategia de reparación a aplicar, teniendo en cuenta lo siguiente:

- el entorno de ejecución en el que se encuentra la aplicación y el peso que cada *concern* tiene según dicho entorno,

- el *concern* asociado al escenario, y
- las prioridades relativas entre los escenarios.

Es importante mencionar que para calcular la utilidad del sistema, Arco Iris sólo considerará aquellos escenarios que se encuentren "habilitados" (*enabled*, en el modelo de Arco Iris presentado en la figura 16). En otras palabras, para calcular la utilidad del sistema, Arco Iris asignará un determinado puntaje a cada escenario habilitado que se satisfaga, el resto de los escenarios serán ignorados, o lo que es equivalente, tendrán puntaje cero.

Por cada escenario habilitado, Arco Iris verificará si aplica para el entorno actual del sistema, en caso de no aplicar, el escenario se satisface trivialmente, por lo que sumará a la utilidad del sistema. Para los escenarios que sí apliquen al entorno actual, Arco Iris evaluará si las restricciones establecidas en la cuantificación de la respuesta del escenario se satisfacen y sólo en caso afirmativo sumarán a la utilidad del sistema.

```
scoreMaximo = utilidad actual del sistema

Por cada estrategia
        scoreStrategia = 0
        estimacionPorConcern = simular la aplicación de la estrategia como lo hace Rainbow y
                               obtener el valor resultante para los concerns
        Por cada escenario habilitado
                Si (!escenario.applyFor(entornoActual) o
                   (escenario.applyFor(entornoActual) y
                       escenario.responseMeasure.holds(estimacionPorConcern)))

                        prioridadRelativa = calcular prioridad relativa del escenario
                        pesoConcern = peso que el entorno actual asigna a escenario.concern
                        puntajeEscenario = prioridadRelativa * pesoConcern
                        scoreStrategia = scoreStrategia + puntajeEscenario
                Fin
        Fin
        Si scoreStrategia > scoreMaximo
                estrategiaSeleccionada = estrategia
                scoreMaximo = scoreStrategia
        Sino
                Si scoreStrategia = scoreMaximo
                        utilityDiff = rainbowSystemUtility(estrategia) -
                                      rainbowSystemUtility(estrategiaSeleccionada)
                        Si utilityDiff > 0
                                estrategiaSeleccionada = estrategia
                        Sino Si utilityDiff = 0 y
                             tasaDeFallos(estrategia) < tasaDeFallos(estrategiaSeleccionada)
                                estrategiaSeleccionada = estrategia
                        Fin
                Fin
        Fin
Fin
```

Figura 21: Pseudocódigo del cálculo del puntaje de escenarios y estrategias.

Ahora bien, una vez que se tienen los escenarios que pesarán sobre la utilidad del sistema, veamos cómo Arco Iris asignará un puntaje (*score*) a cada uno. En la figura 21 se puede ver plasmado en pseudocódigo el cálculo del puntaje de un escenario y cómo es utilizado para calcular el puntaje final de la estrategia, seleccionando Arco Iris aquella de mayor puntaje para ser ejecutada.

Como se dijo anteriormente, si la mejor estrategia no agrega más utilidad que la actual, Arco Iris no aplica ninguna estrategia. Luego, por cada estrategia, Arco Iris simulará su aplicación reutilizando para este punto la implementación de Rainbow. Lo importante de la simulación es obtener los valores estimados para cada *concern*, los cuales se almacenan en un diccionario `<Concern, Estimación>`.

Una vez obtenidos los valores estimados, Arco Iris iterará por los escenarios habilitados y por cada escenario que se satisfaga[11] deberá calcular la **prioridad relativa** del escenario. Recordemos que cada escenario posee una prioridad asignada, pero esa prioridad no puede ser tomada directamente para calcular la utilidad del sistema ya que aquellos escenarios cuyas prioridades son numéricamente menores, poseen una mayor importancia para los *stakeholders*. Para calcular la prioridad relativa y permitir extender y/o modificar la lógica de su cálculo, se ha definido la interfaz `ScenarioRelativePriorityAssigner`, la cual define el siguiente método:

```
public abstract int relativePriority(SelfHealingScenario scenario);
```

Por defecto, se provee la implementación `DefaultScenarioRelativePriorityAssigner`, que calcula la prioridad relativa ponderando los valores absolutos de las prioridades de los escenarios con el escenario de menor prioridad, esto es, el escenario con máxima prioridad numérica entre todos los escenarios:

```
public class DefaultScenarioRelativePriorityAssigner
         implements ScenarioRelativePriorityAssigner {

        private SelfHealingConfigurationManager selfHealingConfigurationManager;

        public DefaultScenarioRelativePriorityAssigner(SelfHealingConfigurationManager shcm) {
                this.selfHealingConfigurationManager = shcm;
        }

        @Override
        public int relativePriority(SelfHealingScenario scenario) {
                int maxPriority = selfHealingConfigurationManager.getMaxPriority();
                return (maxPriority - scenario.getPriority() + 1) / maxPriority;
        }
}
```

[11] Recordar que, que un escenario se "satisfaga", significa que o bien el entorno del escenario no coincide con el actual, o bien, si coincide y la cuantificación de la respuesta del escenario se satisface

Algunas aclaraciones:

- en la sección siguiente se entrará en detalle sobre el rol que cumple en Arco Iris el objeto `SelfHealingConfigurationManager`.
- se toma como prioridad máxima la del escenario menos prioritario.
- al calcular la prioridad relativa, se suma uno al dividendo para evitar que la prioridad relativa del escenario menos prioritario sea cero y que sea ignorado en el cálculo de la utilidad del sistema. De esta manera también se logra que la prioridad relativa de un escenario de prioridad uno (i.e: el escenario más importante) sea exactamente igual a uno.

Una vez calculada la prioridad relativa del escenario, el siguiente paso será obtener el peso que el entorno actual asigna al *concern* del escenario en cuestión. Por ejemplo, si el entorno actual es de alta carga, es lógico que los escenarios de eficiencia posean un mayor peso que los escenarios de costo, por esto se pesan los escenarios según su *concern* y el entorno en el que se encuentra el sistema.

Luego, y en caso de que la estrategia actual posea el mismo puntaje que la estrategia seleccionada hasta el momento, el algoritmo pasa a escoger para reparar el sistema a aquella que brinde mayor utilidad del sistema calculada según el mecanismo provisto por Rainbow (ver sección 3.9.2), i.e. aplicando las funciones de utilidad de cada concern a los valores estimados por la simulación. Por ejemplo, si dos estrategias reparan exactamente los mismos escenarios, su puntaje coincidirá, pero al analizar las estimaciones realizadas para cada estrategia se detecta que una deja al tiempo de respuesta en un valor promedio de 1000 milisegundos, mientras que la otra brinda un valor promedio de 2000 milisegundos. Gracias a la comparación de utilidades antes mencionada, la primer estrategia ha de ser la seleccionada por Arco Iris para reparar el sistema.

Por último, cuando ambas estrategias coincidan en el puntaje asignado por Arco Iris y en la utilidad del sistema según Rainbow, el desempate estará dado por aquella estrategia que posea menor tasa de fallos (ver sección 2.4.8).

En la figura 22 se presenta el modelo conceptual utilizado por Arco Iris para seleccionar la mejor estrategia de reparación.

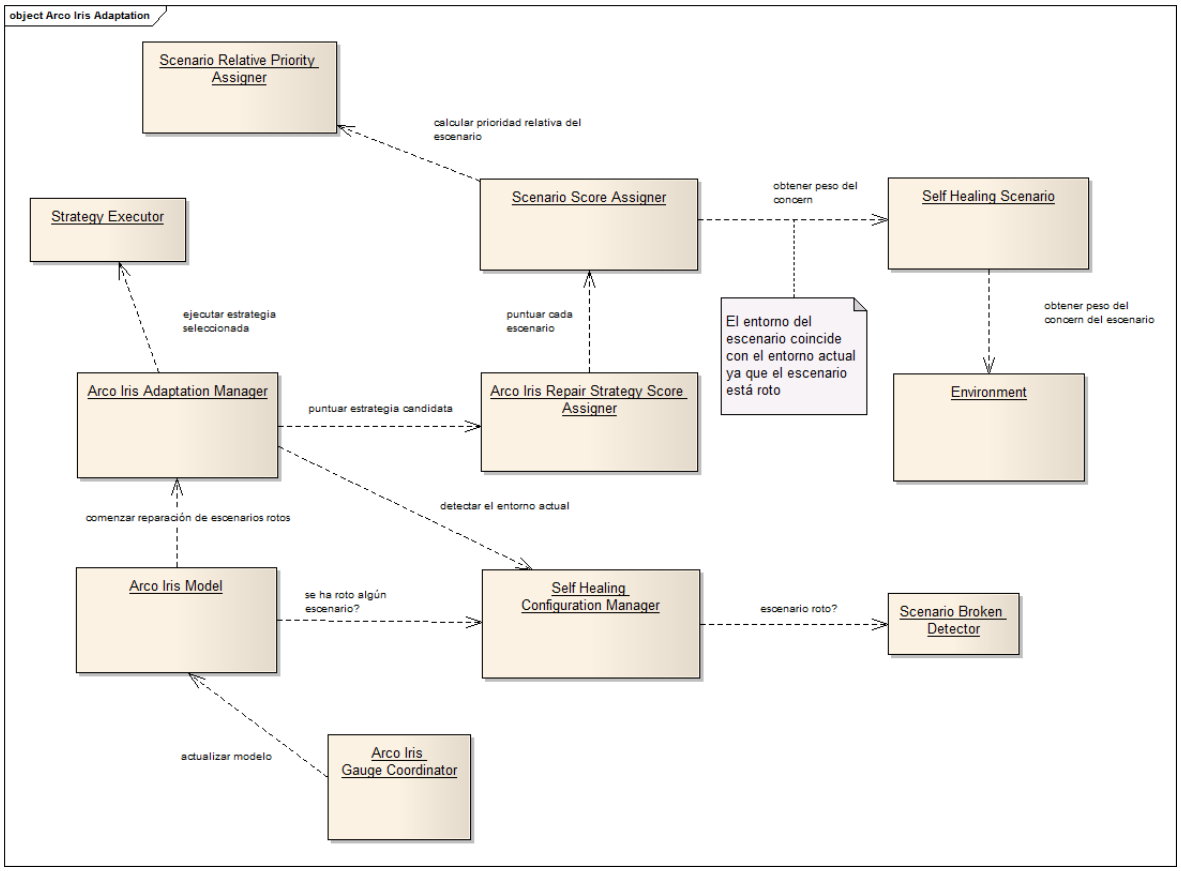


Figura 22: Puntuación y selección de estrategias en Arco Iris

3.10. Configuración de Arco Iris

La configuración de Arco Iris se encuentra modelada en `SelfHealingConfiguration`, el cual que posee la siguiente estructura:

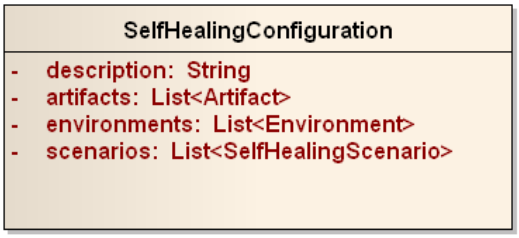


Figura 23: Estructura de la configuración de Self Healing utilizada por Arco Iris

Arco Iris obtiene todo lo que necesita de la configuración utilizando un objeto llamado `SelfHealingConfigurationManager`, el cual funciona como un intermediario entre el archivo de configuración y los componentes centrales de Arco Iris. A continuación, se enumeran algunas de las funcionalidades más importantes de `SelfHealingConfigurationManager`:

- `findBrokenScenarios`: Dado un estímulo, obtiene la colección de escenarios con dicho

estímulo que no se cumplen. Este método es invocado desde `ArcoIrisModel` al momento de actualizar una propiedad en el modelo de la arquitectura del sistema a adaptar.(ver sección 3.8.2)

- `getEnabledScenarios`: Obtener todos los escenarios que se encuentran habilitados. Este método es invocado desde `ArcoIrisAdaptationManager` durante el cálculo de la utilidad del sistema.(ver sección 3.9.3)

- `getMaxPriority`: obtiene la prioridad de mayor valor absoluto configurada en alguno de los escenarios de la configuración subyacente. Este método es utilizado por `DefaultScenarioRelativePriorityAssigner` para el cálculo de la prioridad relativa de un escenario.(ver sección 3.9.3)

A fin de simplificar y amenizar para el usuario la creación y edición de la configuración que Arco Iris utiliza, se provee la herramienta Arco Iris UI, la cuál será explicada en detalle en la sección 4.

3.10.1. Formato del Archivo de Configuración

En el apéndice C, se puede observar un ejemplo de un archivo de configuración de *Self Healing*.

El archivo de configuración de *Self Healing* que Arco Iris acepta debe poseer un formato determinado. Se ha decidido que el mejor formato para dicho archivo es XML, ya que es comprensible no sólo por sistemas de *software* sino también por personas. Esto es particularmente útil en el caso en que el usuario desee efectuar modificaciones mínimas sobre la configuración y no desee o, por algún motivo, no pueda ejecutar la herramienta Arco Iris UI.

3.10.2. Mecanismo de Lectura de la Configuración

La clase `SelfHealingConfigurationPersister` provee un método (`readFromFile`) para leer un archivo XML de configuración de *Self Healing* y otro método (`saveToFile`) para escribir una configuración en Java a un archivo XML. Para ambos fines, la clase utiliza el reconocido parser "XStream" [12], el cual permite fácilmente convertir objetos Java en XML y viceversa.

3.10.3. Actualización Dinámica de Configuración

Una de las principales limitaciones de Rainbow es la imposibilidad de actualizar la configuración del *framework* sin tener que reiniciar su ejecución. A fin de superar (al menos

[12] Para más información, visitar http://xstream.codehaus.org/

parcialmente) tal limitación, se propone un simple mecanismo de detección de cambios en el archivo de configuración de Arco Iris, el cual, al detectar cualquier cambio, **reemplazará dinámicamente la configuración de Arco Iris cargada en memoria por la nueva configuración**, todo esto sin necesidad de reiniciar el *framework*.

El mecanismo se ejecuta periódicamente cada X milisegundos, siendo X configurable utilizando el archivo de configuración estándar de Rainbow `rainbow.properties`. La propiedad a configurar recibe el nombre de `customize.scenarios.reloadInterval`, cuyo valor inicial por defecto será 5000, es decir, el mecanismo de actualización se ejecutará cada 5 segundos.

Se utiliza el patrón *Observer*[13] como modo de notificar a todos aquellos objetos interesados en llevar a cabo alguna acción como consecuencia de un cambio en la configuración. El componente (*observer* siguiendo la nomenclatura utilizada comúnmente en el patrón) más interesado en conocer cuando un cambio en la configuración tiene lugar es el denominado `SelfHealingConfigurationManager`, el cual, en ese caso, descarta toda la configuración de los escenarios cargada en memoria, tomando luego la nueva configuración del archivo recientemente actualizado. A partir de este momento Arco Iris continuará trabajando con la nueva configuración de escenarios sin detener en ningún momento su ejecución.

El mecanismo descrito en este apartado se encuentra implementado en la clase `FileSelfHealingConfiguracionDao`, cuya implementación se puede observar en el apéndice D.

[13] Para más información acerca del patrón Observer, visitar http://www.oodesign.com/observer-pattern.html

4. Interfaz Gráfica: Arco Iris UI

4.1. Introducción

4.1.1. Motivaciones para una "GUI"

Aún considerando las mejoras provistas por la extensión a Rainbow propuesta en el presente trabajo, uno de los escollos más notorios para poder utilizar de manera amena, ágil y productiva a Arco Iris es, sin dudas, la ausencia de una interfaz visual para que los *stakeholders* y arquitectos de la aplicación puedan crear, editar y eliminar escenarios, entornos, *artifacts* y otros conceptos introducidos en Arco Iris; así también como otros conceptos ya existentes en Rainbow.

A fin de superar esta dificultad, 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 expresar los atributos de calidad (o "ilities") del sistema en formato de escenarios de QAW, que luego serán importados y utilizados por Arco Iris.

En el presente apartado nos dedicaremos a repasar los aspectos más importantes de la aplicación, bautizada "Arco Iris UI".

4.1.2. Herramienta de escritorio

Arco Iris UI es una aplicación de escritorio, hecha en Java y utilizando la librería *JFace-SWT*[14], la cual provee un conjunto de *widgets* para construir interfaces gráficas de usuario sobre SWT.

Se prefirió una herramienta de escritorio por sobre una aplicación web principalmente porque las aplicaciones del primer tipo suelen proveer interfaces de usuario más ricas, intuitivas, ágiles y fáciles de usar que las aplicaciones web; todas estas, características deseables para que Arco Iris UI sea útil para los usuarios finales.

4.1.3. Idioma por defecto: inglés

Dado que Rainbow es una aplicación originalmente creada en Estados Unidos, y previendo que tanto Arco Iris como Arco Iris UI puedan ser aplicaciones tomadas por otros investigadores para ser extendidas, se ha decidido que lo más razonable es que el idioma de estas aplicaciones sea el inglés. En el caso de Arco Iris UI, todos los textos mostrados en pantalla

[14] JFace fue desarrollado por IBM para facilitar la construcción del entorno de desarrollo Eclipse, pero su uso no está limitado a éste. JFace proporciona una serie de construcciones muy frecuentes para desarrollar interfaces gráficas de usuario, tales como cuadros de diálogo, evitando al programador la tediosa tarea de lidiar manualmente con los widgets de SWT.

también están escritos en inglés, aunque todas las etiquetas se encuentran externalizadas a archivos de configuración, simplificando así la traducción a otros idiomas, en caso de que en un futuro fuera necesario.

4.1.4. Multi plataforma

Arco Iris UI es, como la gran mayoría de las aplicaciones Java, ejecutable desde los sistemas operativos más importantes. Debido a la utilización de las librerías SWT-JFace para la creación de la herramienta, las cuales poseen dependencias nativas del sistema operativo (i.e. el encargado de dibujar y refrescar los componentes visuales no es la máquina virtual de Java sino el sistema operativo) es necesario empaquetar la aplicación con diferentes dependencias java (i.e. archivos JAR) en el *classpath* para poder correr la aplicación en distintas plataformas. En el apéndice E se explica en detalle los pasos necesarios para instalar y ejecutar las aplicaciones Arco Iris y Arco Iris UI.

4.2. Conceptos básicos de uso de la herramienta

4.2.1. Formato de entrada y salida: XML

Arco Iris UI lee y graba la configuración que genera en un archivo XML, de acuerdo al formato explicado en detalle en la sección 3.10.1.

El mecanismo utilizado para serializar una configuración de *Self Healing* a un archivo XML es exactamente el mismo que el mecanismo utilizado por Arco Iris para importar dicho archivo; y se encuentra detallado en la sección 3.10.2.

4.2.2. Una configuración de *Self Healing* subyacente

La idea básica de uso de Arco Iris UI es que, en todo momento, exista una configuración de *Self Healing* subyacente. Bien sea que el usuario la haya creado "desde cero" o bien, que el usuario haya abierto una configuración preexistente desde un archivo.

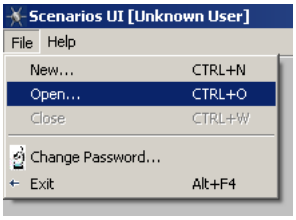


Figura 24: Diálogo de creación o apertura de archivo

Ante cualquier acción del usuario de Arco Iris UI dónde se guarden cambios hechos en la configuración activa, la aplicación **guardará automáticamente dichos cambios en el archivo subyacente**. Esto es particularmente útil en el caso en que el usuario se encuentre

editando el archivo de configuración que Arco Iris está utilizando en *runtime*. En ese caso, cualquier cambio automáticamente guardado por Arco Iris UI será detectado al instante por Arco Iris, el cuál, tal como se explica en la sección 3.10.3, recargará la configuración, dinamizando considerablemente el comportamiento original presente en *Rainbow*.

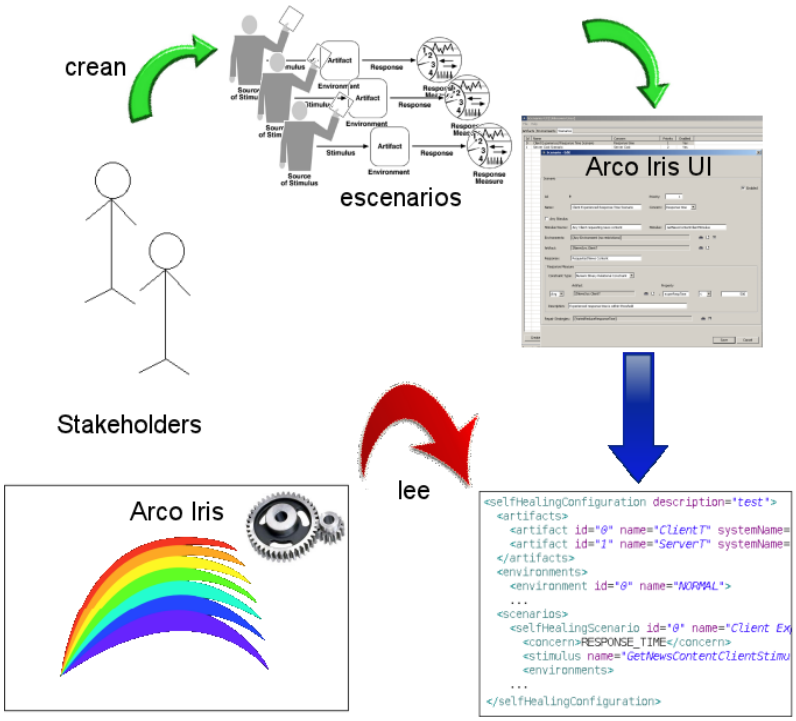


Figura 25: Flujo conceptual entre Arco Iris UI y Arco Iris

4.2.3. Auto refresco de tablas

Como veremos más adelante en detalle, en Arco Iris UI se visualizan los componentes (i.e. escenarios, entornos, *artifacts*, etc.) existentes en el sistema con una tabla que sumariza sus propiedades más relevantes. En general, estas tablas se refrescarán automáticamente de acuerdo a los cambios hechos en los objetos subyacentes que componen la configuración de *Self Healing*.

4.2.4. *Constraints*

Tanto en el contexto de creación/edición de un entorno (al imponer restricciones sobre su ocurrencia) como al crear o editar el *response measure* de un escenario, las llamadas "restricciones" (o *constraints*) son una parte fundamental de la configuración requerida para que Arco Iris provea su funcionalidad adecuadamente.

El componente visual asociado a una restricción fue diseñado para ser fácilmente extendido pero, sólo a fines del presente trabajo y con el objetivo de acotar la funcionalidad a los límites

establecidos en sus objetivos, sólo se ha implementado un sólo tipo de restricción: la restricción numérica relacional binaria (ver apéndice A para más detalle).

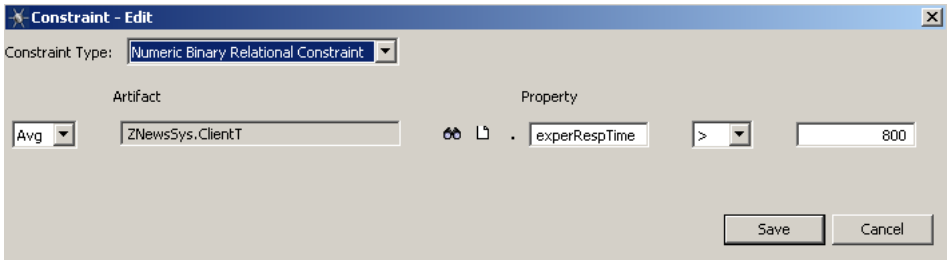


Figura 26: Diálogo de creación o edición de una restricción numérica binaria relacional

4.3. Administración de *Artifacts*

Uno de los componentes relevantes a un escenario de auto reparación es el llamado *Artifact*, el cual representa al componente de arquitectura que está involucrado en el escenario en cuestión.

Arco Iris UI provee la funcionalidad de alta, baja y modificación (ABM) básica para administrar *artifacts* junto con una solapa dentro de la pantalla principal de la aplicación para poder visualizar las propiedades más importantes de dichos objetos.

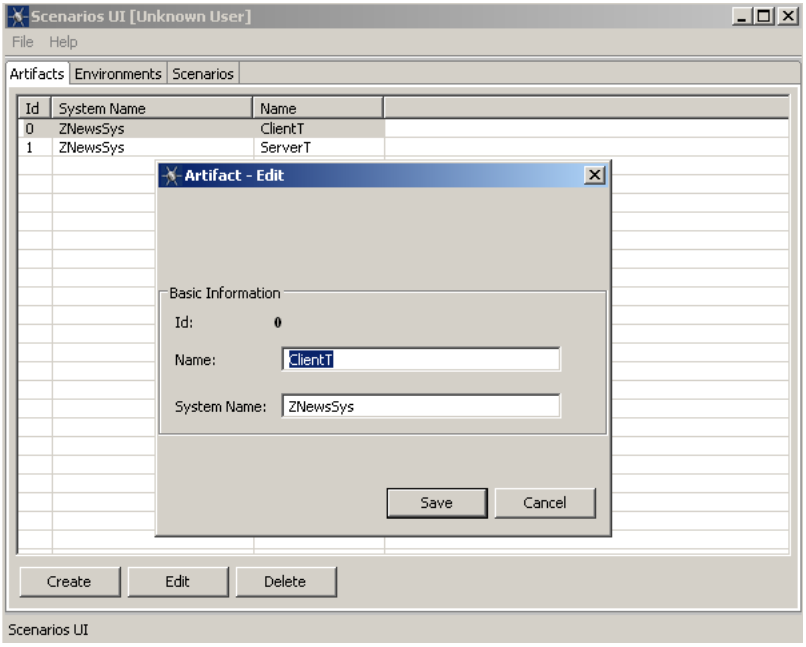


Figura 27: Diálogo de creación o edición de un *artifact*, con su solapa ABM en segundo plano

4.4. Administración de Entornos

El "Entorno" es uno de los componentes más relevantes dentro de un escenario de auto reparación: representa la situación concreta (en tiempo de ejecución) donde el escenario en

cuestión va a aplicar.

Al igual que para los *artifacts*, Arco Iris UI provee la funcionalidad de ABM para administrar escenarios junto con una solapa para visualizar sus propiedades más relevantes.

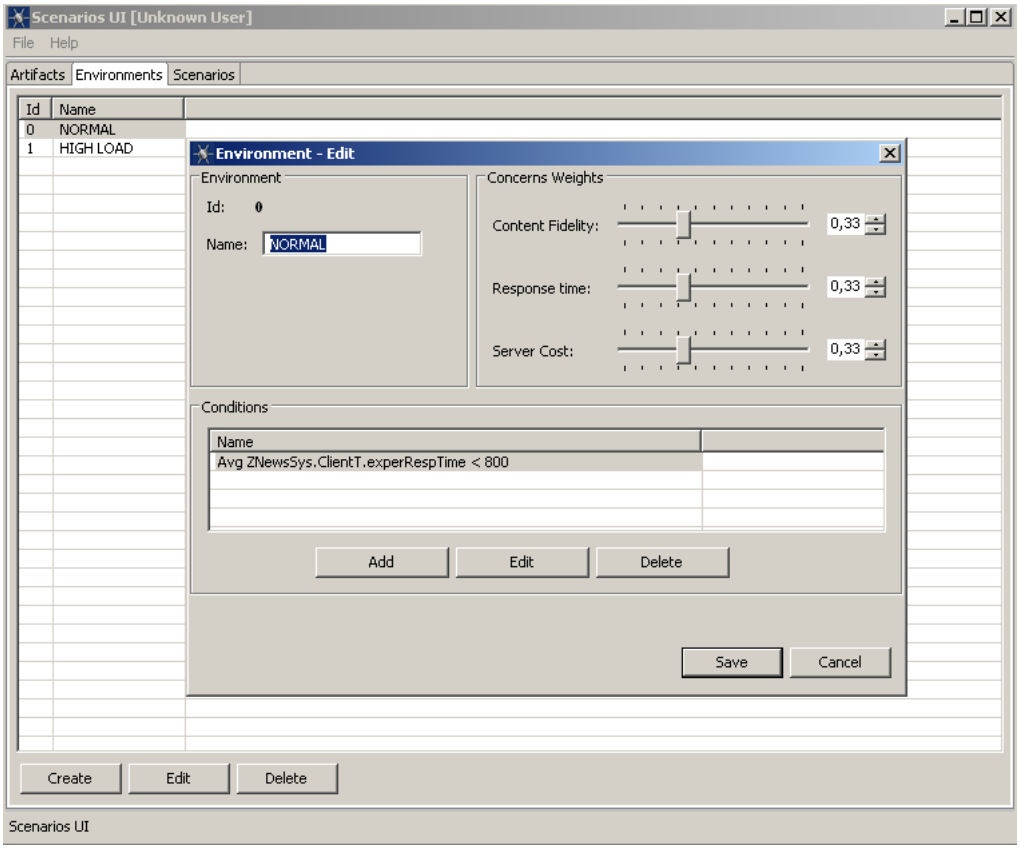


Figura 28: Diálogo de creación o edición de un entorno, con su solapa ABM en segundo plano

El cuadro de diálogo que muestra las propiedades de un entorno está diseñado para recoger dinámicamente todos los *concerns* existentes en el sistema e inicialmente asignarle a cada uno un peso equidistribuido P determinado de la siguiente forma:

$$P = \frac{1}{K} \tag{1}$$

donde K es la cantidad de *concerns* existentes en el sistema.

Se observa también la funcionalidad de ABM para las condiciones que se deben cumplir para que el entorno se cumpla en *runtime*. Aquí se reutiliza un componente genérico para restricciones, también utilizado en el diálogo de creación de un escenario (Ver figura 26)

4.5. Administración de Escenarios

Esta sección trata sobre el concepto central de este trabajo: el escenario de auto reparación, quien contiene los elementos ya vistos anteriormente junto con otros atributos que se repasarán a continuación. Por supuesto que existe, al igual que para los otros conceptos

anteriormente vistos, una solapa dedicada para maneja las altas, bajas y modificaciones de escenarios.

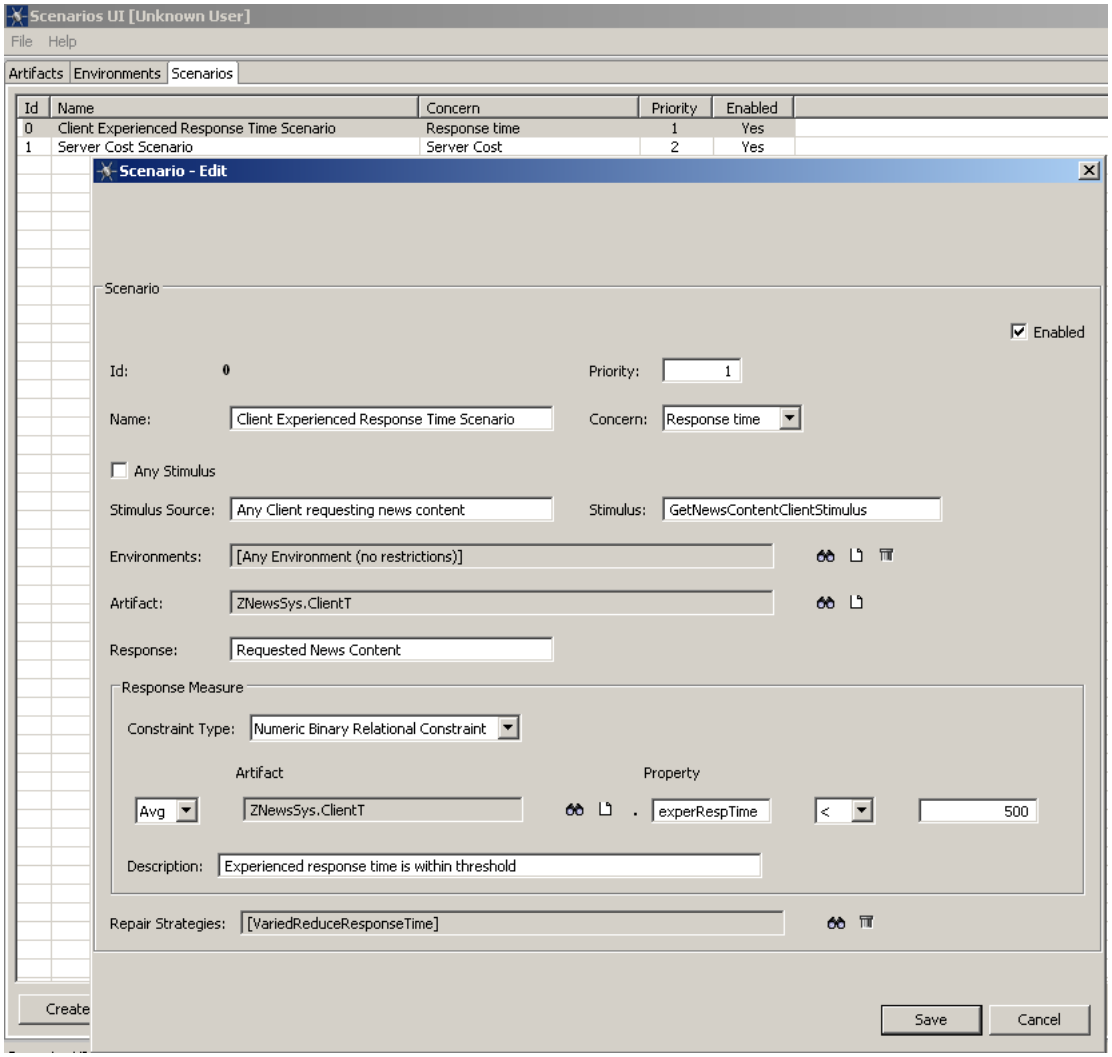


Figura 29: Diálogo de creación o edición de un escenario, con su solapa ABM en segundo plano

En la figura anterior observamos que:

- Existe la posibilidad de marcar un escenario como habilitado o deshabilitado. Recordemos que Arco Iris únicamente considera escenarios habilitados.
- El combo box de *Concerns* se llena con los *concerns* existentes en el sistema al momento de cargarse el cuadro de diálogo.
- Se permite especificar que el estímulo correspondiente al escenario es "cualquiera".
- Se permite elegir *artifacts*, considerando los existentes en el sistema o creando ad-hoc uno nuevo.

- En el componente “Response Measure” se reutiliza el componente mencionado anteriormente en la sección 4.2.4 para crear y editar constraints.

4.5.1. Selección de Estrategias de Reparación

Una funcionalidad interesante de Arco Iris UI es su capacidad para, en el contexto de la creación o edición de un escenario, poder elegir una estrategia de reparación directamente desde el archivo *stitch* en el cual se encuentra definida. Primeramente, se elige el archivo que contiene las estrategias, para luego seleccionar la estrategia deseada, visualizando el código (en el lenguaje de scripting *stitch*) de la misma.

Es de notar que para el caso de las estrategias de reparación, y a diferencia del resto de los otros conceptos explicados hasta este momento, no es posible crear una estrategia ad-hoc, sino que se deben usar únicamente las ya existentes.

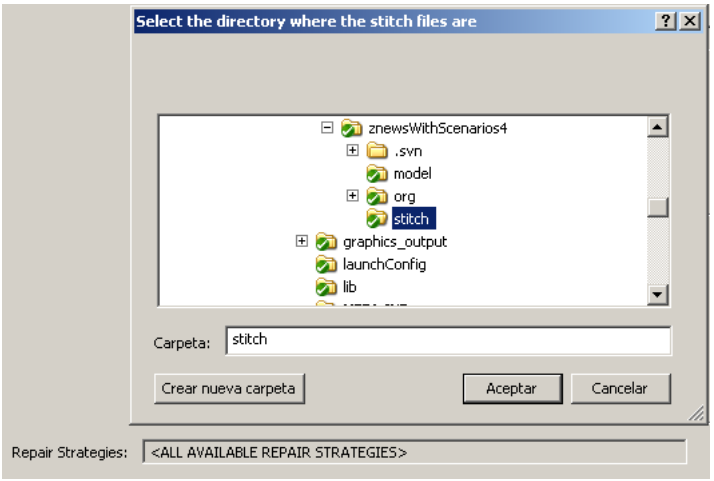


Figura 30: Diálogo de selección del archivo que contiene las estrategias

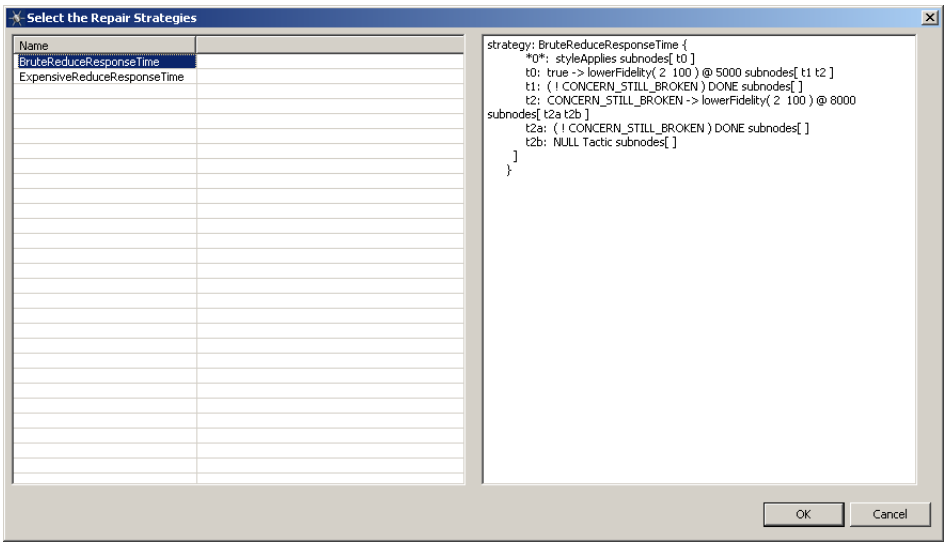


Figura 31: Diálogo de selección de la estrategia de reparación

Al momento de especificar estrategias de reparación, es importante mencionar que la aplicación también permite especificar la siguiente semántica:

"Considerar **todas** las estrategias de reparación disponibles en Arco Iris al momento de la ejecución"

Lo anterior significa ni más ni menos que el usuario que configura Arco Iris vía Arco Iris UI, no impone un conjunto específico de estrategias de reparación a ser utilizadas sino que por lo contrario, permite que cualquier estrategia disponible sea considerada por Arco Iris para la reparación del escenario subyacente.

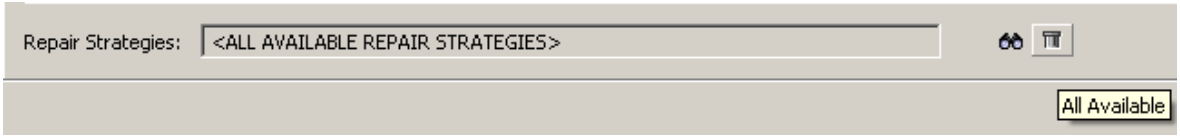


Figura 32: Todas las estrategias de reparación disponibles seleccionadas

4.6. Puntos de extensión

A continuación, enunciaremos algunos de los posibles puntos de extensión únicamente para la herramienta Arco Iris UI (en la sección 6 se cubrirán los puntos de extensión para Arco Iris):

- **Obtener artefactos y estímulos desde el archivo .acme**: Es deseable el evitar mantener en Arco Iris UI un ABM (**A**lta-**B**aja-**M**odificación) de artefactos, así también como que el usuario deba especificar estímulos de escenarios de manera no validada por la aplicación, cuando dicha información puede ser obtenida mediante un *parsing* del archivo Acme que contiene el modelo de la arquitectura. Recordemos que el artefacto de un escenario de atributos de calidad no es ni más ni menos que un componente, un conector o un sistema; y un estímulo se encuentra normalmente asociado a una **operación** provista por el artefacto. Esta funcionalidad no fue realizada en el contexto del presente trabajo ya que el *parsing* y post procesamiento del archivo `.acme` es una tarea no trivial (aunque posible, utilizando clases Java pertenecientes a Rainbow).

- **Editor de Tácticas y Estrategias**: Una funcionalidad que de seguro amenizaría la experiencia del usuario es sin duda un editor de tácticas y estrategias integrado a Arco Iris UI, con posibilidades de creación, edición y borrado. Asociado a ésto está la idea de que esta información sea guardada en el mismo archivo de configuración que el resto de la configuración producida por Arco Iris UI. Esta idea será desarrollada con un poco más de detalle en la sección 6.4.

5. Casos Prácticos

En esta sección se presentan y analizan algunos casos de prueba concretos de uso de Arco Iris a fin de evaluar su comportamiento. Se utilizará el modo simulación provisto por Rainbow (ver sección 2.4.10) para adaptar utilizando Arco Iris al sistema ficticio **Znn**, reutilizando los componentes de simulación creados para la tesis de doctorado donde dicho sistema es presentado. (ver sección 2.4.9)

La simulación permite configurar la variación de los valores de ciertas propiedades de los componentes de la arquitectura, a fin de simular diversas situaciones de carga en el sistema ficticio. Por ejemplo, se podría especificar que a los 10 segundos de haber comenzado la simulación, el ancho de banda de un servidor en particular disminuya y por otro lado que en ese mismo instante, la frecuencia de arribo de *requests* del usuario suba en una determinada proporción. Este cambio en el entorno de ejecución del sistema simulado, por ejemplo, permitiría evaluar cómo se comporta Arco Iris en un contexto de **alta carga**.

Para todos los casos de prueba presentados en esta sección, se utilizará una simulación que dura 60 segundos.

5.1. Arquitectura del Sistema Simulado

Los componentes principales de la arquitectura de Znn son clientes y servidores. Éstos no se conectan directamente entre sí, sino que lo hacen por medio de un *proxy*, al cual arriban todas las peticiones de los clientes, y es él quien conoce todos los servidores disponibles y distribuye el trabajo entre ellos. A continuación se muestra un diagrama de la arquitectura del sistema, extraída utilizando la herramienta AcmeStudio. (En el apéndice F se puede observar el código fuente Acme de dicha arquitectura)

En la arquitectura de Znn están definidos tres *concerns* aunque en el presente trabajo, a saber:

- **Tiempo de Respuesta**: tiempo de respuesta promedio experimentado por el usuario de Znn, y
- **Costo**: el cual, refleja la cantidad de servidores prestando servicio con los que Znn cuenta en un determinado momento.
- **Fidelidad del Contenido**: trata sobre la **calidad** del contenido ofrecido por un servidor de Znn. A fin de mejorar el *throughput*, un servidor podría servir contenido en un modo *full* (texto, videos, audio, animaciones, etc.), en un modo sólo texto o bien en una combinación de estos dos.

Para las pruebas realizadas en este trabajo, sólo se utilizarán las siguientes tácticas, con el fin de modificar el comportamiento del sistema en ejecución:

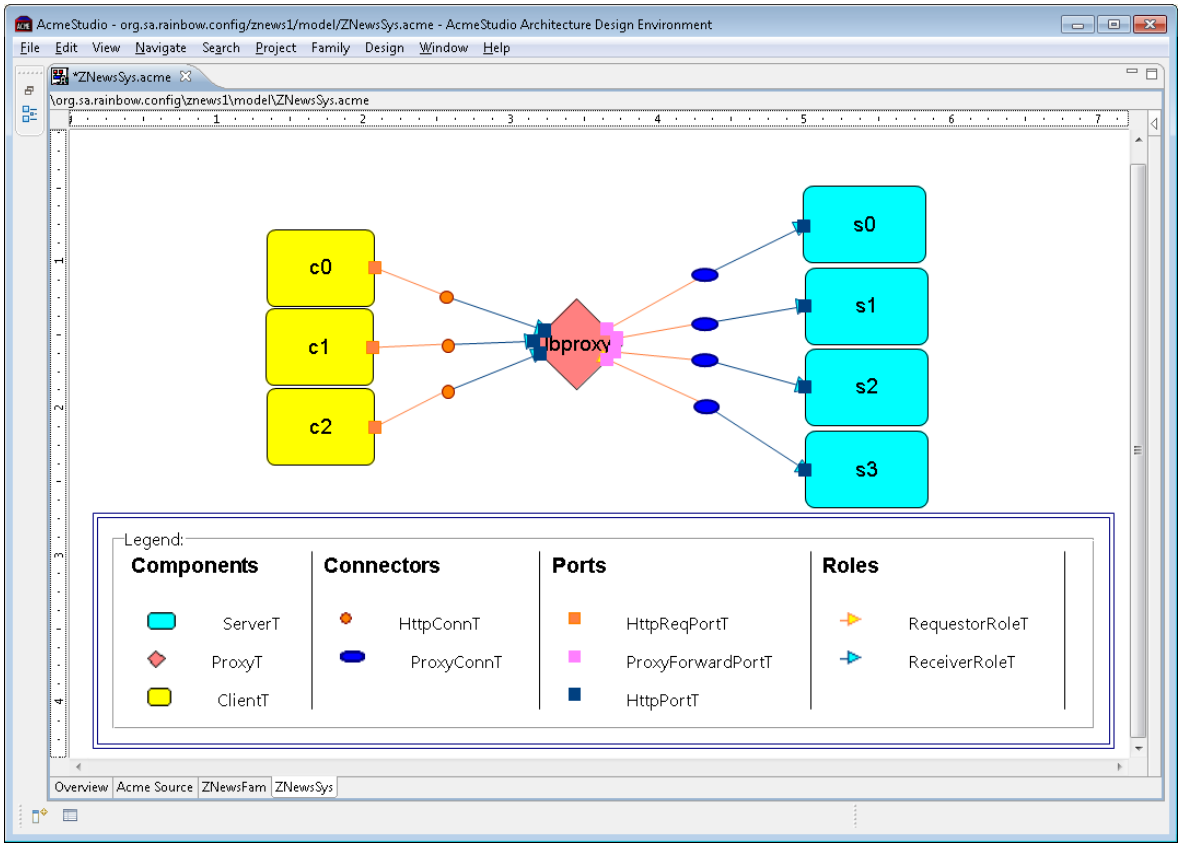


Figura 33: Arquitectura de Znn vista en Acme Studio

- Dar de alta un servidor
- Dar de baja un servidor
- Disminuir la fidelidad del contenido provisto por un servidor.

Para que estas tácticas puedan ejecutarse, Znn implementa los correspondientes *effectors*, quienes serán los responsables de efectuar los cambios propiamente dichos sobre el sistema en *runtime*. Los *effectors* serán invocados desde las tácticas, cuyas implementaciones en Znn pueden verse en el apéndice G.

5.2. Configuración Básica para Casos de Prueba

Como parte de la configuración utilizada para las pruebas que serán presentadas en esta sección, es necesario definir los siguientes conceptos:

- Entorno de carga normal.
- Entorno de alta carga.
- Escenario de tiempo de respuesta experimentado por el usuario.

- Escenario de costo de servidores del sistema.
- Estrategias asociadas a cada escenario, las cuales serán presentadas a medida que su utilización sea requerida.

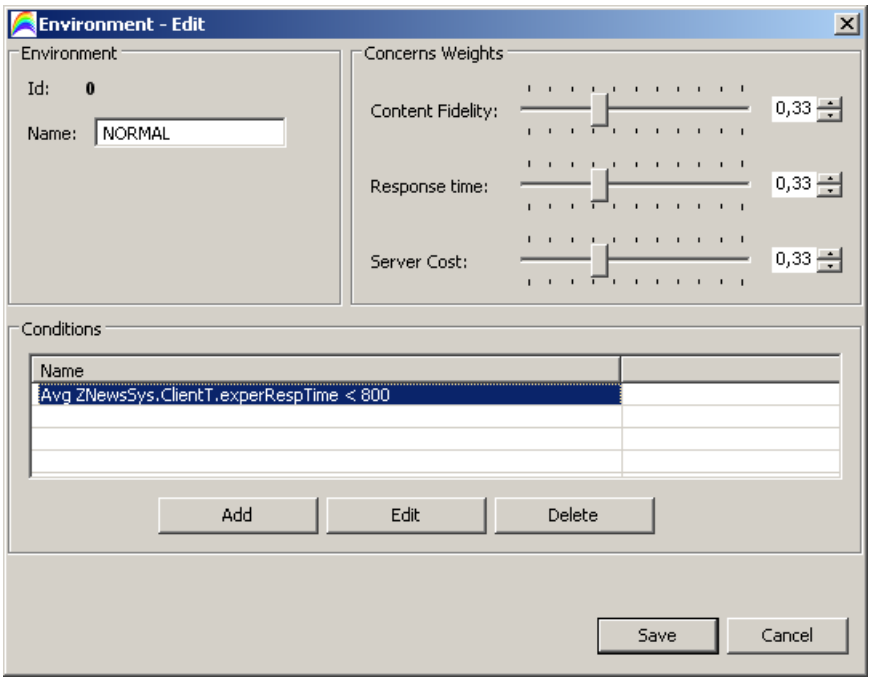


Figura 34: Entorno de ejecución de carga normal

Cabe destacar que para el entorno de carga normal, la configuración por defecto de los pesos para cada uno de los *concerns* del sistema se encuentra equidistribuida. Esta decisión convierte a las prioridades entre escenarios, cuando todos ellos pertenecen al entorno de carga normal, en el único factor influyente en la selección de una estrategia candidata para reparar el sistema (ver algoritmo de selección de estrategias en la sección 3.9.3), simplificando así los cálculos, así como también la comprensibilidad de los resultados presentados.

En la figura 35 se puede observar la configuración del entorno de alta carga, cuyos pesos no se encuentran equidistribuidos, enfatizando así la importancia del tiempo de respuesta frente a los restantes *concerns* definidos en el sistema.

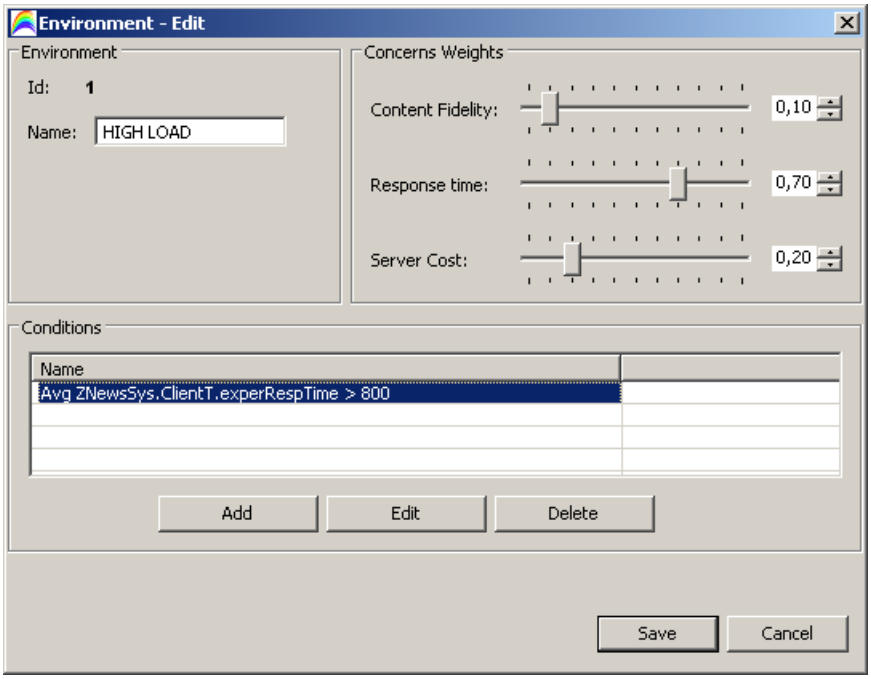


Figura 35: Entorno de ejecución de alta carga

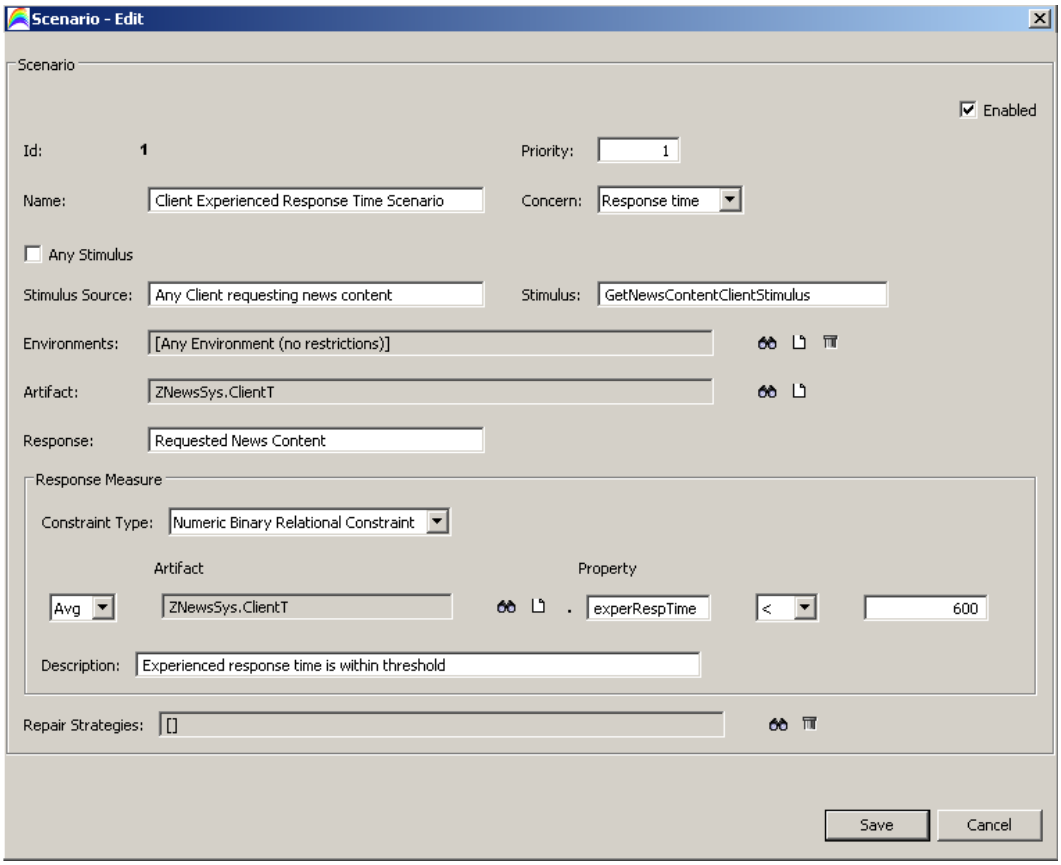


Figura 36: Escenario de tiempo de respuesta experimentado por el usuario

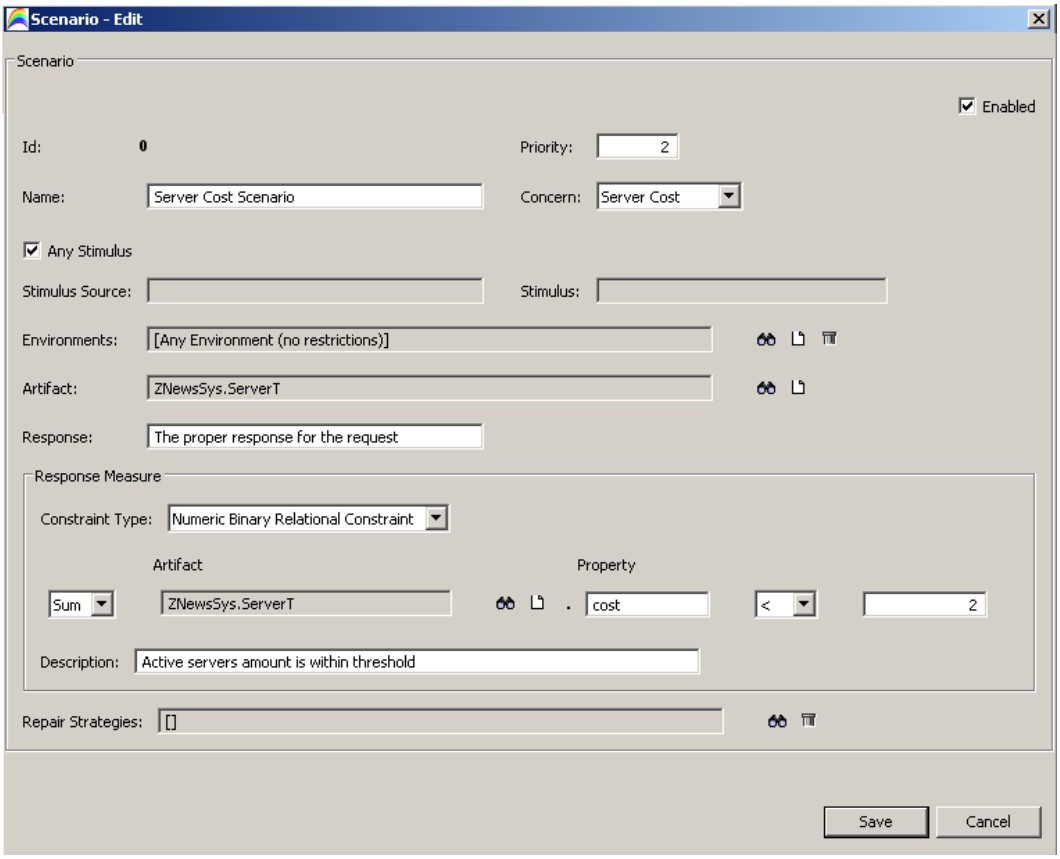


Figura 37: Escenario de costo de servidores del sistema

En el apéndice H se muestra a modo de ejemplo la representación en XML del escenario de tiempo de respuesta definido anteriormente.

Con respecto a la configuración por defecto de los escenarios aquí definidos, notar lo siguiente:

- el escenario de tiempo de respuesta posee mayor prioridad que el de costo.
- ambos escenarios aplican para cualquier entorno en que se encuentre el sistema en ejecución (ver definición del pseudo entorno "ANY" en la sección 3.4.3);
- ambos escenarios no poseen estrategias de reparación configuradas.

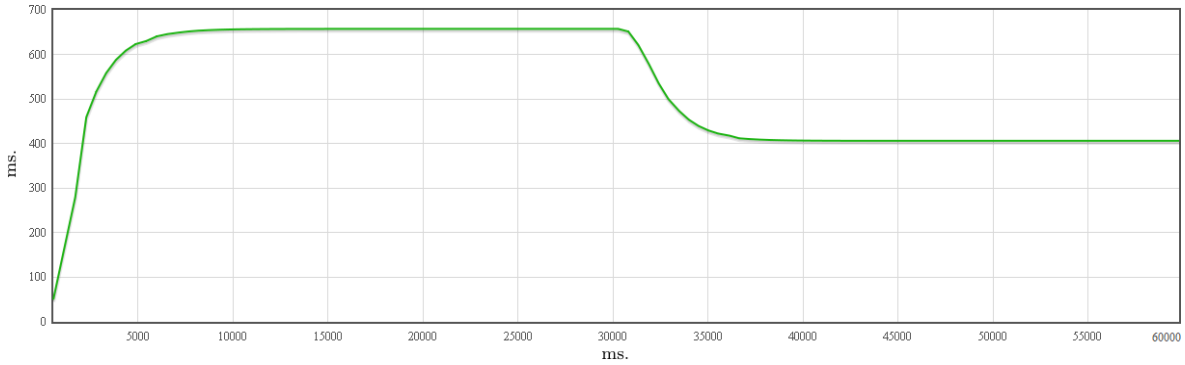
La configuración básica expuesta hasta aquí será la utilizada por todos los casos de prueba a desarrollar en el presente trabajo. Al avanzar con las pruebas, y de acuerdo a las necesidades particulares de configuración de cada una, será necesario efectuar algunos ajustes menores que impactarán sobre los valores de los siguientes atributos:

- Prioridad de cada escenario.
- Estrategias asociadas a cada escenario.
- Pesos de los *concerns* para cada entorno.

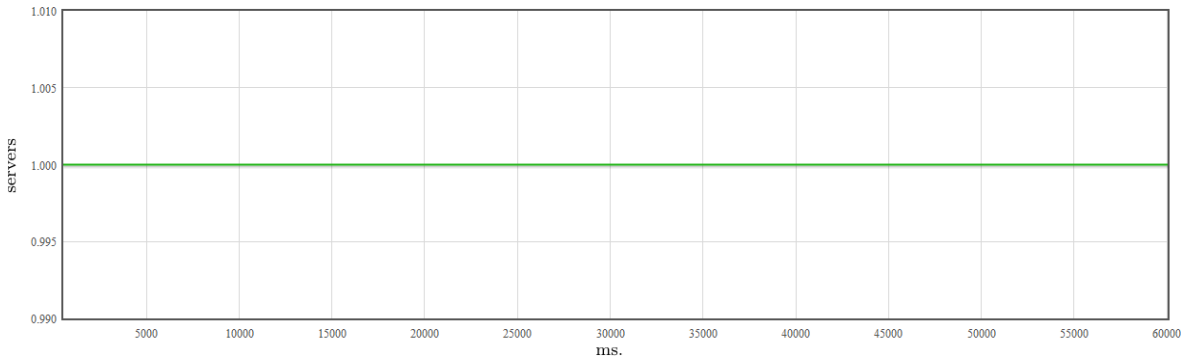
5.3. Caso 0: Comportamiento del Sistema sin Auto Reparación

Para comenzar, se presenta el comportamiento del sistema de no existir escenarios ni estrategias. Esto sentará las bases para luego poder comparar y evaluar el comportamiento de Arco Iris a medida que se vayan agregando escenarios y/o estrategias en los siguientes casos de prueba a considerar.

En la figura 38 se puede observar el comportamiento del sistema sin escenarios, es decir, sin mecanismo de auto reparación alguno.



(a) Tiempo de Respuesta



(b) Costo de Servidores

Figura 38: Comportamiento del sistema sin escenarios

Como se puede observar, la simulación ha sido configurada explícitamente para que Znn se comporte de la siguiente manera: el tiempo de respuesta crece hasta superar los 600 ms., manteniéndose allí hasta 30 segundos después de haber comenzado la simulación, para luego ir bajando paulatinamente hasta estacionarse cerca de los 400 ms. Notar que el costo de los servidores se mantiene inmutable frente a los cambios en el tiempo de respuesta, es decir que el sistema, de no mediar un usuario administrador o un *framework* de auto reparación como Arco Iris, trabaja siempre con un único servidor. Es importante tener en cuenta que la merma en el tiempo de respuesta no se debe a ninguna acción propia de la auto reparación, sino a cambios en el ambiente, externos al sistema, como por ejemplo el ancho de banda de la conexión de cada uno de sus clientes.

5.4. Caso 1: Comportamiento con un Escenario, Sin Estrategias

Para el presente caso de prueba se utiliza el escenario de tiempo de respuesta definido anteriormente (ver figura 36), el cual determina un umbral máximo aceptado de 600 ms. para el tiempo de respuesta experimentado por el usuario.

El objetivo de esta prueba es visualizar cómo, al no haberse definido aún ninguna estrategia, Arco Iris detectará que existe un escenario que no se satisface aunque no efectuará reparación alguna sobre el sistema.

Dado que Arco Iris no ejecuta estrategia alguna, el costo de servidores no se verá modificado, manteniéndose constante en 1, tal como se ha visto en la figura 38(b).

Por otro lado, en la figura 39 se puede observar que Arco Iris detecta que, a partir de cierto instante, el tiempo de respuesta alcanza y supera el umbral predefinido en la cuantificación de la respuesta del único escenario del sistema.

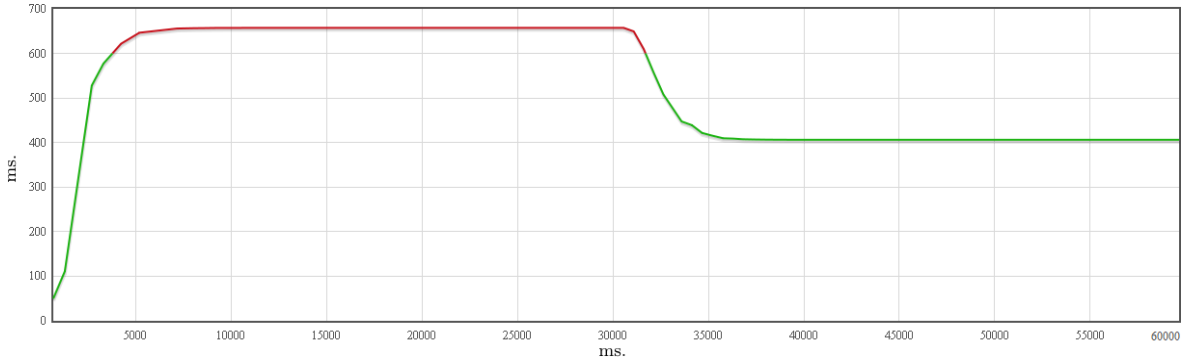


Figura 39: El umbral definido para el tiempo de respuesta es superado

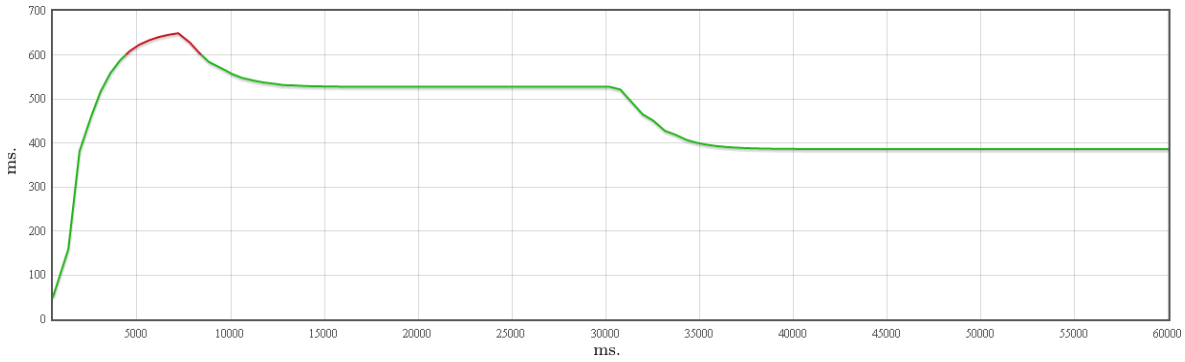
5.5. Caso 2: Comportamiento con un Escenario y una Estrategia

En el presente caso se intenta reflejar cómo Arco Iris repara el sistema al encontrar una estrategia candidata adecuada para el escenario de tiempo de respuesta anteriormente presentado. Para tal fin, se define una estrategia que consiste simplemente en agregar un servidor, siempre y cuando existan servidores disponibles. La estrategia, definida en Stitch, posee la siguiente lógica:

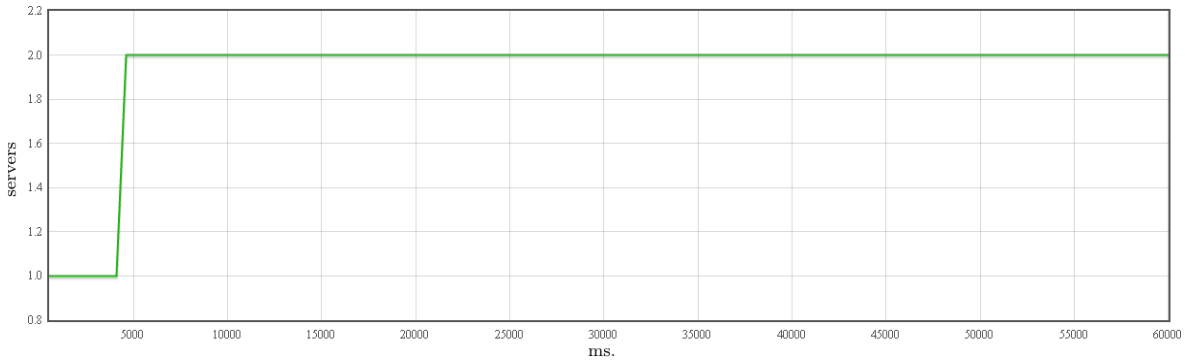
```
strategy EnlistServerResponseTime {
  t0: (true) -> enlistServers(1) @[5000 /*ms*/] {
    t1: (!RESP_TIME_STILL_BROKEN) -> done;
    t2: (default) -> TNULL;
  }
}
```

Figura 40: Estrategia que agrega un servidor más al sistema

Al agregar esta estrategia al escenario, se observa en la figura 41(a) que el tiempo de respuesta experimentado por el usuario mejora (i.e. desciende) rápidamente. De manera simultánea a esta mejora, el costo de servidores aumenta a 2, producto de la ejecución de la estrategia. Esto puede observarse en la figura 41(b).



(a) Mejora en el tiempo de respuesta debido a la ejecución de una estrategia



(b) Impacto de la estrategia sobre el costo de servidores

Figura 41: Impacto del agregado de una estrategia

En resumen, se ha visto hasta aquí el comportamiento del sistema en las siguientes circunstancias:

1. no existe información alguna sobre auto reparación.
2. se ha definido un escenario pero sin estrategias que lo puedan reparar.
3. se ha definido un escenario con una estrategia asociada.

Antes de proseguir con casos de prueba más complejos, cabe mencionar que los *logs* generados por Arco Iris ofrecen la información necesaria para analizar en detalle los casos de pruebas presentados en este informe. Dada la extensión de dichos archivos, es inviable mostrarlos todos para cada caso de prueba, por lo cual, a modo de ejemplo, en el apéndice I se presenta un extracto del *log* generado por Arco Iris para el caso que se acaba de desarrollar en esta sección.

5.6. Caso 3: *Tradeoff* entre Estrategias

El presente caso intenta mostrar cómo Arco Iris escoge, entre varias estrategias candidatas para un mismo escenario, aquella que maximiza la utilidad del sistema.

En particular, este caso presenta dos escenarios: uno relacionado con el costo de servidores, sin estrategias de reparación definidas; y otro cuyo *concern* es el tiempo de respuesta, configurado con la estrategia `EnlistServerResponseTime` antes definida y la introducción de una nueva estrategia:

```
strategy LowerFidelityReduceResponseTime {
  t0: (true) -> lowerFidelity(2, 100) @[5000 /*ms*/] {
    t1: (!RESP_TIME_STILL_BROKEN) -> done;
    t2: (RESP_TIME_STILL_BROKEN) -> lowerFidelity(2, 100) @[8000 /*ms*/] {
      t2a: (!RESP_TIME_STILL_BROKEN) -> done;
      t2b: (default) -> TNULL;  // in this case, we have no more steps to take
    }
  }
}
```

En concreto, en este caso de prueba se puede observar de qué manera (mediante el uso del concepto de **Utilidad del Sistema**), Arco Iris - dentro de las estrategias que reparan al escenario en cuestión - otorga más valor a aquellas estrategias que no "rompen" otro escenario, es decir, las que dejan al sistema en una situación más estable.

Es de notar que, al "competir" estrategias relacionadas con el mismo *concern* y reparando ellas al mismo escenario, las prioridades configuradas para cada escenario, en este caso, carecen de importancia.

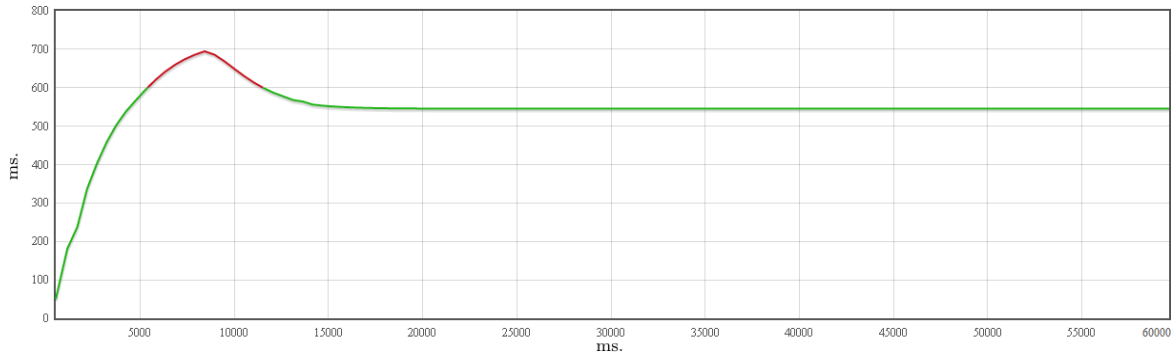
Cabe mencionar que, para que el escenario de costo deje de cumplirse, debe ser un escenario que aplique al entorno actual del sistema. Esta condición se satisface trivialmente, considerando que el escenario de costo fue definido para aplicar en cualquier entorno.

En el siguiente extracto de *log* se puede observar cómo Arco Iris considera las estrategias mencionadas, escogiendo a `LowerFidelityReduceResponseTime` por sobre `EnlistServerResponseTime`, ya que si bien ambas reparan al escenario de tiempo de respuesta, la última rompe al escenario de costo de servidores:

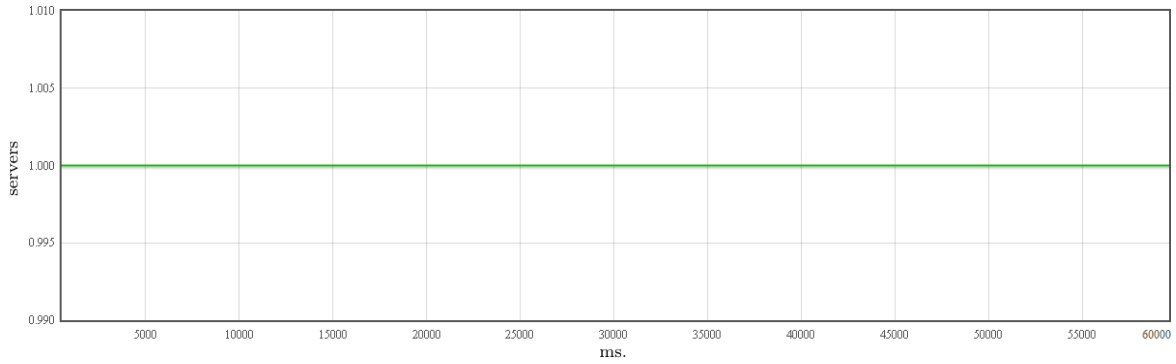
```
...
Evaluating strategy EnlistServerResponseTime...
Scoring EnlistServerResponseTime...
  Server Cost Scenario broken after simulation for Server Cost ([ESum] 2.0)? true
  Experienced Response Time Scenario broken after simulation for Response time ([EAvg] 457.81)? false
  Score for strategy EnlistServerResponseTime: 0.333
  Current best strategy EnlistServerResponseTime
  Evaluating strategy LowerFidelityReduceResponseTime...
Scoring LowerFidelityReduceResponseTime...
  Server Cost Scenario broken after simulation for Server Cost ([ESum] 1.0)? false
```

```
  Experienced Response Time Scenario broken after simulation for Response time ([EAvg] 481.81)? false
  Score for strategy LowerFidelityReduceResponseTime: 0.49
  Current best strategy: LowerFidelityReduceResponseTime
Selected strategy!: LowerFidelityReduceResponseTime
EXECUTING STRATEGY LowerFidelityReduceResponseTime...
...
```

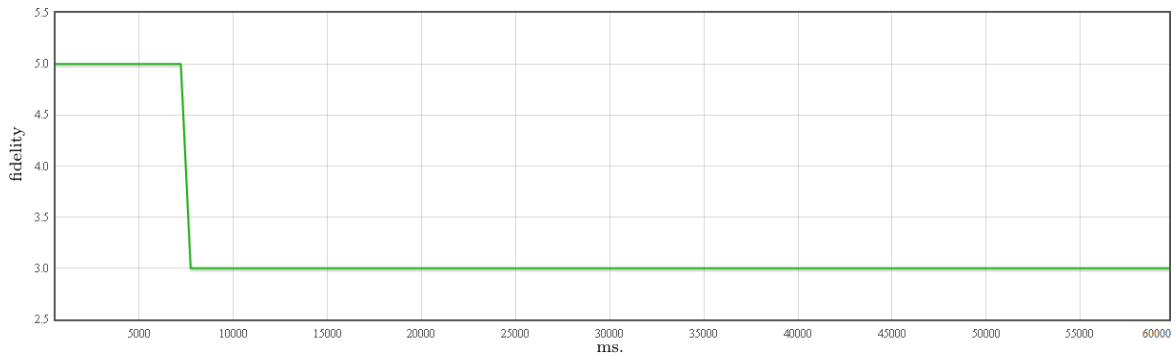
Por último, en la figura 42 se pueden observar las variaciones de los *concerns* tiempo de respuesta, costo de servidores y fidelidad, para este caso de prueba.



(a) Reparación del tiempo de respuesta usando la mejor estrategia



(b) El costo de servidores se mantiene intacto



(c) Desciende la fidelidad de la información

Figura 42: Variaciones de los tres *concerns* involucrados

5.7. Caso 4: *Tradeoff* entre Escenarios según Prioridades

El objetivo de esta prueba consiste en evaluar el comportamiento de Arco Iris ante la existencia de escenarios con distintas prioridades.

Para el presente caso de prueba, se toma como base la configuración del caso anterior con las siguientes modificaciones:

- El escenario de tiempo de respuesta contará ahora solamente con la estrategia `EnlistServerResponseTime`,
- Al escenario de costo se le agrega una estrategia de reparación, cuya lógica puede verse a continuación:

```
strategy ReduceOverallCost {
  t0: (true) -> dischargeServers(1) @[2000 /*ms*/] {
    t1: (!COST_STILL_BROKEN) -> done;
    t3: (default) -> TNULL;
  }
}
```

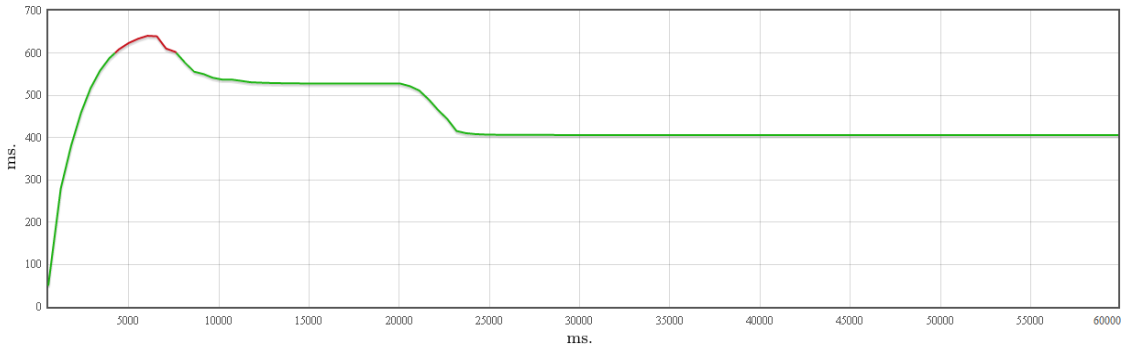
Como ya se ha mencionado en la introducción de la presente sección (ver sección 5.2), el escenario de tiempo de respuesta es más prioritario que el de costo. Esta configuración es crucial en este caso de prueba, ya que determinará el rumbo de la auto reparación llevada a cabo por Arco Iris.

En las figuras 43(a) y 43(b) se puede observar el comportamiento del tiempo de respuesta y del costo, respectivamente, para la configuración actual.

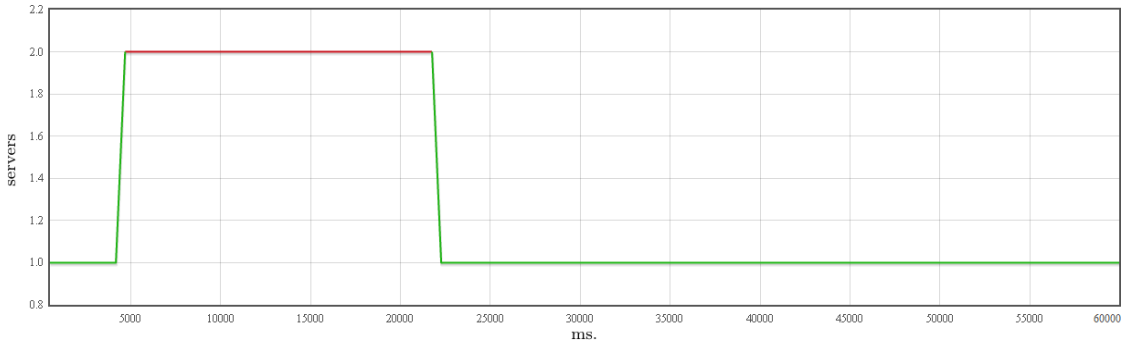
Tal cual se ha visto en los casos básicos anteriores, el escenario de tiempo de respuesta es el primero en dejar de cumplirse. Considerando que sólo se ha configurado una única estrategia de reparación para dicho escenario, la misma es ejecutada exitosamente, ya que cerca de los 9 segundos, el tiempo de respuesta vuelve a ubicarse en valores aceptables.

Ahora bien, la estrategia ejecutada consiste ni más ni menos que en agregar un servidor más al sistema, con lo cual se observa en la figura 43(b) de qué manera, a partir de los 5 segundos, el escenario relacionado con el costo de servidores deja de cumplirse, puesto que la cantidad máxima de servidores allí especificados es 1.

Arco Iris debe decidir si repara o no a este nuevo escenario que se ha "roto". Es aquí donde las prioridades entre escenarios juegan un papel determinante: dado que el escenario relaciondo con el tiempo de respuesta es más prioritario que aquel relacionado con el costo de servidores, Arco Iris decide no efectuar reparación alguna sobre este último, ya que detecta (mediante la heurística explicada en la sección 3.9.3) que el arreglar el escenario de costo, potencialmente llevaría a "romper" el escenario de tiempo de respuesta, que es más prioritario. En otras palabras, la auto reparación provista por Arco Iris no intentará reparar el escenario



(a) Uso de prioridades favoreciendo al escenario de eficiencia por sobre el de costo



(b) Reducción del costo como consecuencia de un cambio en el entorno

Figura 43: Comportamiento del sistema respetando prioridades entre escenarios

de costo de servidores mientras que la utilidad del sistema en el estado actual sea mayor a la prevista en caso de repararlo.

En la figura 43(b), aproximadamente a partir de los 33 segundos, se puede observar cómo Arco Iris decide desactivar un servidor en el preciso momento en que el entorno de ejecución de la aplicación cambia y el tiempo de respuesta mejora por razones ajenas a la auto reparación. Aprovechando esto, Arco Iris logra satisfacer así al escenario que estaba sin cumplirse, sin perjudicar al otro escenario (más prioritario) que se estaba cumpliendo hasta ese momento.

Finalmente, se arriba a un estado de estabilidad donde ambos escenarios se satisfacen simultáneamente. Si bien el tiempo de respuesta sufre un pequeño detrimento al trabajar el sistema con un servidor menos, los valores de las propiedades relacionadas con los *concerns* de interés siguen siendo lo suficientemente aceptables como para no violar ninguna de las restricciones definidas en la cuantificación de la respuesta de los escenarios aquí configurados.

En resumen, se ha mostrado por un lado la potencia y conveniencia de agregar el concepto de prioridad entre escenarios como un elemento de relevancia para configurar al *framework* y por otro, cómo distintos escenarios con distinta prioridad pueden convivir en Arco Iris, tomando éste decisiones inteligentemente sobre qué escenario(s) reparar, considerando siempre como factor crucial la utilidad que el sistema exhibiría de ejecutar, o no, determinada estrategia de reparación.

5.8. Caso 5: *Tradeoff* entre Escenarios según *Concerns*

El objetivo de este caso de prueba es analizar cómo Arco Iris, al tener que escoger entre favorecer dos escenarios con igual prioridad, elige favorecer a aquel cuyo *concern* posee un peso mayor para el entorno de ejecución actual.

Los escenarios utilizados en este caso de prueba son idénticos a los del caso anterior, excepto que ahora pasan a tener ambos igual prioridad y el escenario de costo no posee estrategias de reparación asociadas. Por otro lado, el entorno de carga normal pasa a asignar mayor peso al *concern* costo de servidores:

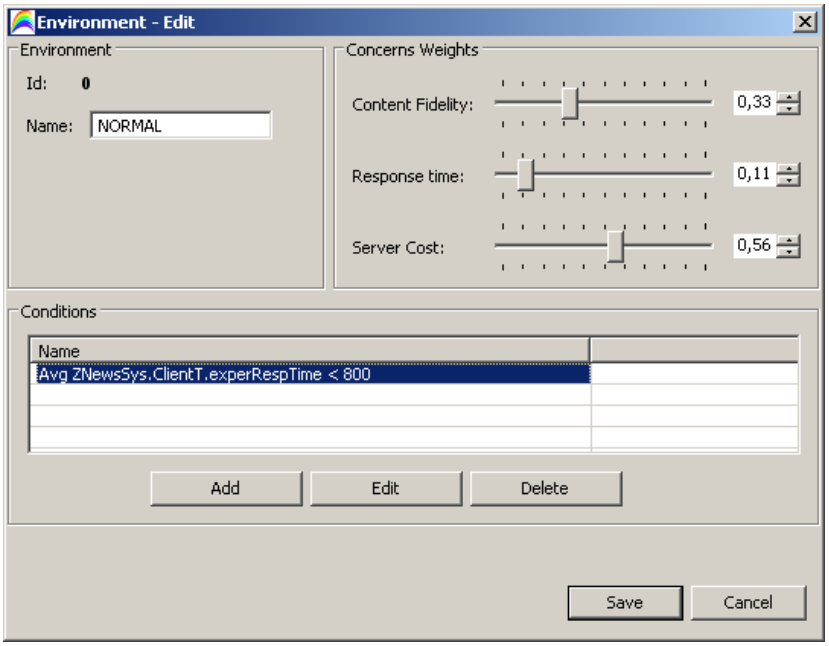


Figura 44: Nueva distribución de pesos para el entorno "normal"

En el extracto del *log* generado por Arco Iris se observa cómo, al igual que en la mayoría de los casos de pruebas ya presentados aquí, el escenario de tiempo de respuesta deja de cumplirse y Arco Iris debe decidir qué acción llevar a cabo ante dicha situación. En la figura 45 puede apreciarse que Arco Iris opta por no reparar el escenario de tiempo de respuesta ya que esto perjudicaría al escenario de costo, puesto que si bien ambos poseen idéntica prioridad, el último está relacionado al *concern* **costo de servidores**, el cuál en el entorno de ejecución actual (normal) tiene más peso que el *concern* **tiempo de respuesta**.

En otras palabras, al igual que en el caso anterior, Arco Iris vuelve a utilizar la Utilidad del sistema como una medida para estimar el estado en que quedaría el sistema, de ejecutar o no una determinada estrategia; concluyendo que, en el caso de intentar reparar el escenario de tiempo de respuesta "roto", el sistema brindaría menos utilidad que en el estado actual. Vale reiterar que el factor determinante en el cálculo de dicha utilidad simulada del sistema es, ni más ni menos, que el peso del *concern* costo de servidores posee en el entorno de ejecución actual.

```
Current environment: NORMAL
Computing Current System Utility...
Server Cost Scenario broken for [ESum] 1.0? false
Experienced Response Time Scenario broken for [EAvg] 602.25? true
Current System Utility (Score to improve): 0.555
Evaluating strategy EnlistServerResponseTime...
  Scoring EnlistServerResponseTime...
  Server Cost Scenario broken after simulation for Server Cost ([ESum] 2.0)? true
  Experienced Response Time Scenario broken after simulation for Response time ([EAvg] 458.25)? false
  Score for strategy EnlistServerResponseTime: 0.111
NO applicable strategy, adaptation cycle ended.
```

Figura 45: *log* de Arco Iris para el caso de prueba 5

Cabe mencionar que, avanzada ya la ejecución del sistema, el tiempo de respuesta vuelve a estar por debajo del umbral máximo definido en el escenario correspondiente. Esto, nuevamente, se debe a condiciones cambiantes en el entorno de ejecución y no a una acción activa llevada a cabo por el *framework* para lograr tal efecto. Esta situación, junto con las variaciones del tiempo de respuesta durante toda la ejecución de este caso de prueba, pueden visualizarse en la figura siguiente:

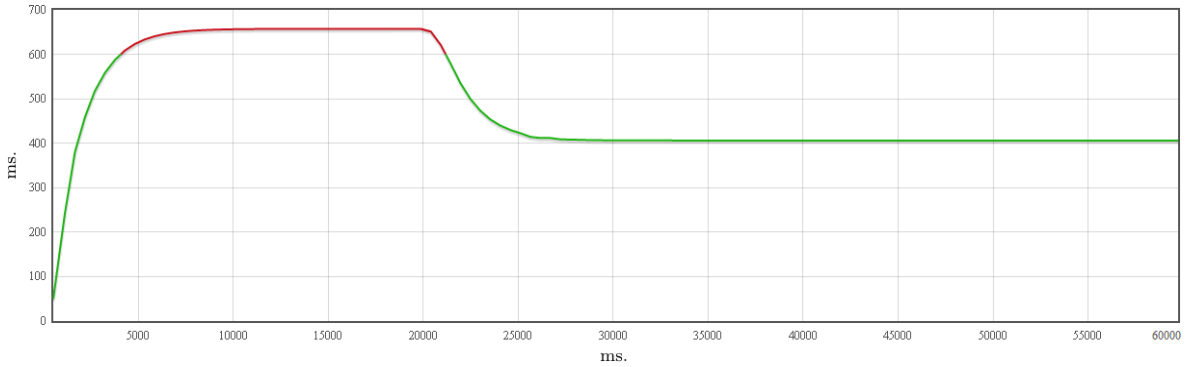


Figura 46: El tiempo de respuesta no es reparado por Arco Iris, arreglandose solo luego.

Cabe destacar que el costo de servidores, por su parte, se mantiene constante en 1 durante toda la ejecución.

5.9. Caso 6: Comportamiento Ante Los Cambios en el Entorno de Ejecución

Este caso de prueba pretende demostrar cómo Arco Iris varía su comportamiento dependiendo del entorno en el cual se encuentra ejecutando el sistema en un momento dado.

Para este caso es necesario contar con tres escenarios similares a los utilizados hasta aquí pero con las siguientes variaciones:

- Un escenario de **prioridad 3** cuya cuantificación de la respuesta acota el tiempo de respuesta a 600 ms. como máximo, cuando el sistema se encuentra en **carga normal**.
- Un escenario de **prioridad 1** que define que el tiempo de respuesta no debe superar los 900 ms. cuando el sistema se encuentra bajo **alta carga**.
- Un escenario de **prioridad 2** que predica sobre el costo de servidores, **limitando al sistema a utilizar como máximo un servidor**, cuando se encuentre en un entorno de **carga normal**.

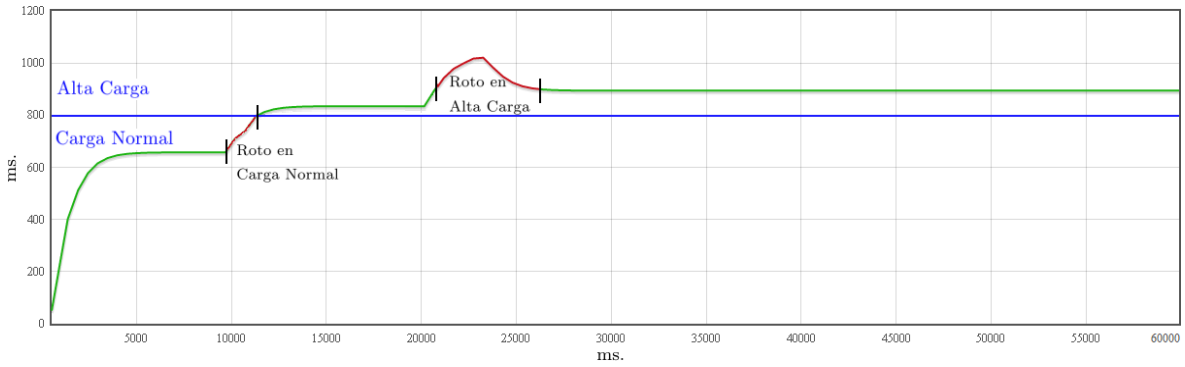
En otras palabras, mientras el sistema se encuentre dentro de parámetros normales de carga (i.e. carga normal), se prefiere que el tiempo de respuesta suba por encima de su umbral máximo (600 ms.) antes que agregar un servidor más al sistema. Ahora, cuando el entorno de ejecución es de alta carga, el tiempo de respuesta pasa a tener más importancia que el costo de servidores. Esto, en teoría, habilitaría a Arco Iris a agregar uno o más servidores en pos de que el tiempo de respuesta no supere su umbral máximo (900 ms.) para un entorno de alta carga. Este es el objetivo primordial de este caso de prueba.

Para lograr que el sistema en un momento determinado de la ejecución pase a estar en alta carga, ha sido necesario modificar los parámetros de la simulación, generando así condiciones que emulan un incremento en la carga de pedidos al único servidor disponible en el sistema. Recordar que la condición que especifica cuándo el sistema se encuentra en alta carga, está configurada en el entorno definido en la sección 5.2 (ver figura 35).

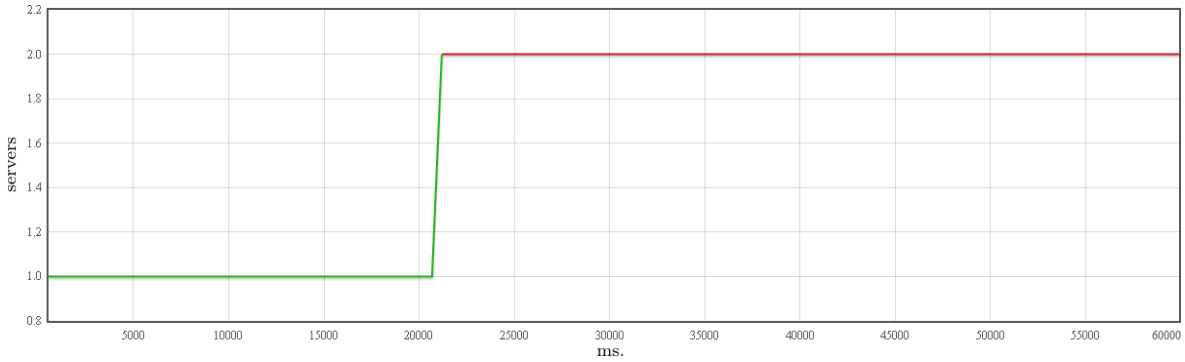
Los escenarios de tiempo de respuesta tendrán asociada la estrategia de auto reparación `EnlistServerResponseTime`, utilizada en pruebas anteriores, mientras que el escenario de costo no poseerá estrategias asociadas.

Tal como se esperaba, Arco Iris no toma acción alguna mientras el sistema se encuentra en carga normal, puesto que el escenario de costo posee más prioridad que el escenario de tiempo de respuesta para dicho entorno. Esto se puede apreciar en la primera porción del gráfico de la figura 47(a).

Luego de superados los 800 ms. el sistema pasa a estar en alta carga, por lo cual dos de los tres escenarios pasan a satisfacerse trivialmente. Consecuentemente, Arco Iris deberá procurar sólamente que se cumpla el escenario de tiempo de respuesta que impide que se superen los 900 ms. Puede observarse en la figura 47(b) que la auto reparación decide agregar un servidor aproximadamente a partir de los 23 segundos de iniciada la simulación, reparando efectivamente el escenario en cuestión.



(a) Arco Iris sólo arregla el tiempo de respuesta con el sistema en alta carga.



(b) En carga normal, el costo es priorizado y se mantiene en 1.

Figura 47: Comportamiento del sistema en distintos entornos de ejecución

6. Trabajo Futuro

Si bien con la introducción de las extensiones presentes en Arco Iris se han presentado considerables mejoras con respecto al uso básico de Rainbow, también se han suscitado numerosos puntos de extensión posibles que si bien no pudieron formar parte del presente trabajo, es importante que sean detallados en pos de incentivar la investigación futura. A continuación, citaremos los puntos de extensión posibles a Arco Iris y/o Rainbow.

6.1. Arco Iris: un *plug-in* de Rainbow

Una característica deseable para Arco Iris es, sin duda, que la extensión se inserte en el *framework* (conceptualmente hablando) de manera similar a cómo un *plug-in* trabaja en cualquier otro sistema de *software*, pudiendo el usuario, de manera relativamente simple, elegir entre utilizar los mecanismos de auto reparación provistos por Arco Iris o los de Rainbow.

Para que esto sea posible, es necesario que Rainbow provea una arquitectura abierta a la extensión de sus componentes claves, como ser el `AdaptationManager`, el `RainbowModel` y `GaugeCoordinator`, entre otros.

A continuación, se enumeran algunos de los motivos por los cuales Rainbow, en la actualidad, no permite extensiones de tipo *plug-in*:

1. Rainbow no trabaja orientado a interfaces, tal como Erich Gamma explica en [GAM/94], observando así un alto acoplamiento entre las clases que conforman su diseño y la consecuente imposibilidad de cambiar las implementaciones por defecto por otras nuevas.

2. El mecanismo utilizado por los componentes de Rainbow para obtener referencias a otros componentes con los que desean interactuar, implementado en una clase Java denominada `Oracle`), es insuficiente para permitir la inyección de diferentes implementaciones del mismo componente. Esto es un error de diseño íntimamente relacionado con el punto anterior.

3. Muchos de los componentes básicos de Rainbow, los cuales se modelan como clases Java, impiden explícitamente su extensión (i.e. herencia) mediante modificadores de acceso restrictivos.

A modo de conclusión, sería muy positiva una reestructuración de Rainbow para permitir extensiones o agregados al mismo de una manera más prolija y ordenada que la que tuvo que utilizarse para la creación de Arco Iris.

6.2. Análisis y Aprendizaje de la Auto Reparación

6.2.1. Herramientas de análisis y visualización

Supongamos que los *stakeholders* ya han definido todos los escenarios que utilizará Arco Iris para gestionar la auto adaptación de un sistema dado. Ahora bien, imaginemos que existe un escenario de muy alta prioridad y a su vez las condiciones necesarias para que no se cumpla se dan con mucha frecuencia. Esto posiblemente resulte en la auto reparación repetida del mencionado escenario. Pero, ¿qué sucedería si la única estrategia que repara dicho escenario "rompe" constantemente otros, digamos tres, escenarios de menor prioridad? A priori pareciera que este es el comportamiento esperado, pero... ¿podrían los *stakeholders* convalidar este comportamiento? Claramente la implementación actual de Arco Iris carece de una serie de **estadísticas y vistas** que permitan analizar lo que está siendo reparado, así también como el impacto sobre los escenarios perjudicados y sobre cada uno de los *concerns* del sistema.

También sería útil proveer una herramienta que permita analizar qué estrategias se fueron ejecutando históricamente, y de qué manera se llegó a decidir que cada una de ellas era la adecuada para reparar el sistema en cada momento.

6.2.2. Más Visibilidad Sobre Las Estrategias Fallidas

Tal como se comentó en la sección 2.4.8, Rainbow provee un mecanismo que considera la historia de ejecución de las estrategias con el objetivo de evitar elegir aquellas con un alto porcentaje histórico de fallas, esto es, estrategias que no resolvieron anteriormente las anomalías que se suponía debían resolver.

Si bien este mecanismo es bastante aceptable, también se reconocen posibilidades de mejora a futuro en dos aspectos:

- El porcentaje de fallas máximo permitido para que una estrategia pueda ser considerada debe ser configurado por el usuario final. La forma más simple de hacer esto es sencillamente externalizando este valor al archivo de configuración de la aplicación ya existente (`rainbow.properties`). Este trabajo no se ha encarado en el presente trabajo, puesto que implica la modificación (en contraposición a una extensión) de una clase específica de Rainbow. Es conveniente que las modificaciones al núcleo de Rainbow sean realizadas por el grupo de gente a cargo de su mantenimiento evolutivo.
- Este mecanismo provee poca visibilidad al usuario final. Una posible idea para analizar la configuración de la auto reparación del sistema podría ser un ***ranking* de estrategias**, en el que se presente el porcentaje de éxito y fallas de cada una así como la cantidad de veces que se ejecutó, cuál es la utilidad del sistema promedio luego de ejecutar la estrategia, etc.

- Otra posible idea podría ser el proveer una suerte de alarma al usuario administrador (vía e-mail por ejemplo) que, previamente a desactivar una estrategia o luego de haber desactivado reiteradas veces la misma estrategia, avise sobre este tipo de situaciones, haciendo explícito este tipo de decisiones cruciales y permitiendo así la rectificación o ratificación de la configuración de *self healing* que está siendo utilizada.

6.3. Ampliación de la Recarga Dinámica de Configuración

En la sección 3.10.3 se explicó el trabajo hecho en Arco Iris en materia de recarga automática de la configuración. Recordemos que se implementó un mecanismo que refresca la configuración relativa a Escenarios, Entornos, *Artifacts* y las referencias a las estrategias asociadas a cada escenario. Si bien tal mecanismo representa una mejora sustancial con respecto a Rainbow (el cual recordemos que no provee ningún tipo de recarga "en caliente" de la configuración) entendemos que se puede dar un paso más en la materia, a fin de que el *framework* resulte más útil para su uso en un ambiente tan dinámico como el de la industria del *software*; dónde el reiniciar la aplicación para aplicar un cambio en la configuración resulta muchas veces sencillamente inaceptable. En consecuencia, es altamente deseable tender a **recargar dinámicamente el 100 % de la configuración relacionada con auto reparación**. Los cambios necesarios para lograr ésto deben llevarse a cabo principalmente en Rainbow, dónde reside el grueso de la configuración estáticamente cargada.

Rainbow actualmente maneja (a grandes rasgos) los siguientes puntos de configuración:

- el archivo `rainbow.properties`, un típico archivo `.properties` de Java, el cual es leído una única vez al inicializarse el *framework* y que sirve para externalizar propiedades tales como el nivel de *log* deseado para la aplicación, el *path* en dónde buscar estrategias y tácticas, el archivo `.acme` con el modelo de la arquitectura del sistema a adaptar, etc. En este caso, sería relativamente sencillo implementar un mecanismo de recarga dinámica idéntico al explicado en detalle en la sección 3.10.3 para recargar la configuración de Arco Iris. Dicho mecanismo tendría que ser realizado en el constructor de la clase `Rainbow)`.

- `utilities.yml`, dónde se configuran las curvas de utilidad para el sistema. Al igual que en el caso anterior, estas propiedades se leen en el constructor de la clase `Rainbow` utilizando una clase *helper* llamada `YamlUtil`. Utilizando el mismo mecanismo descrito anteriormente se podría dinamizar también la recarga dinámica de este archivo.

- archivos Stitch (con extensión `.s`) de tácticas y estrategias. La lógica de lectura de tácticas y estrategias está ubicada en el `AdaptationManager`. Puesto que en Arco Iris se necesitó extender esta clase, se aprovechó la oportunidad para abstraer esta lógica de lectura desde un archivo, a una clase específica llamada `StitchLoader`, la cual tiene la

doble responsabilidad de cargar los datos desde un archivo y de proveer también acceso dichos datos. Sería una buena idea el incorporar esta clase al código base de Rainbow puesto que dicha abstracción provee la posibilidad de incorporar rápidamente el mecanismo de refresco mencionado ya reiteradas veces. Actualmente dicho comportamiento tampoco está incorporado en Arco Iris, no porque sea un trabajo difícil sino debido a que dicha tarea se encuentra afuera del alcance del presente trabajo.

Como podemos observar, el mecanismo explicado en detalle en la sección 3.10.3 es fácilmente reutilizable y también se observan posibilidades de mejora para simplificar aún más su uso extendido en todo el *framework*.

6.4. Ampliación de la Configuración Existente

A lo largo del desarrollo del presente trabajo, hemos notado que existen algunas posibilidades de ampliar la configuración existente en Arco Iris. A continuación, mencionaremos algunos de ellos.

6.4.1. Toda la configuración en un solo archivo

A fin de simplificar el uso del *framework* para el usuario final, es altamente deseable poder reducir la cantidad de archivos de configuración que deben ser creados, modificados y/o mantenidos para que el tándem Rainbow / Arco Iris funcione. Sería deseable realizar modificaciones en ambos *frameworks* para que la configuración se encuentre centralizada en uno (idealmente) o dos archivos de configuración; conglomerando allí información referente a tácticas, estrategias, el modelo de la arquitectura y todo el modelo existente de Arco Iris, actualmente configurable via XML.

6.4.2. Mas tipos de restricciones por defecto

En Arco Iris los tipos de restricciones - debido a su inherentemente compleja lógica - están codificados en clases Java que, en conjunto, configuran el abanico de restricciones soportadas por el *framework*. Actualmente se provee soporte para un tipo solo de restricción: aquella que modela una restricción sobre el valor de una propiedad de un artefacto, con respecto a una función binaria numérica como por ejemplo los operadores $<$, $\leq$, $>$, $\geq$ y $=$.
Si bien este tipo de restricciones son altamente representativas del tipo de cosas sobre las que un usuario promedio normalmente desea predicar, es de notar que el poder expresivo está claramente acotado. Se propone para un trabajo futuro extender el esquema de restricciones modelado en Arco Iris, mediante nuevas implementaciones de la interfaz `Constraint`.

6.5. Atributos de Calidad y *Concerns* configurables por el usuario

Actualmente Arco Iris maneja un conjuntos fijo de Atributos de Calidad y *Concerns*, los cuales se encuentran embebidos en clases Java a las cuales el usuario final no puede acceder. En otras palabras, el usuario no puede agregar o eliminar libremente Atributos de Calidad o *Concerns*. Claramente, esta rigidez restringe la usabilidad del *framework* y debe ser superada.

Puesto que en Arco Iris los *concerns* juegan un papel más preponderante que los Atributos de Calidad, en este apartado nos referiremos únicamente al problema de permitir que el usuario final de Arco Iris pueda agregar o eliminar *Concerns* a una configuración de *Self Healing*.

Existen básicamente dos formas de compensar la falencia antes mencionada: una es sencilla de implementar pero incompleta y la otra es difícil de implementar aunque definitiva.

6.5.1. Implementación sencilla e incompleta

El agregado de una amplia y variada gama de *Concerns* en el código fuente de Arco Iris representa una buena solución de compromiso entre máxima flexibilidad para el usuario y facilidad de implementación, ya que sólo se trata de recabar una lista de los *Concerns* más conocidos y usualmente utilizados y agregarlos a la clase enumerada que modela los tipos posibles de *Concerns* reconocidos por Arco Iris.

6.5.2. Implementación difícil y definitiva

El problema de permitir el libre agregado y eliminación de *Concerns* por parte del usuario posee varios aristas, algunas de las cuáles no son fáciles de resolver, a saber:

- **Actualización del mapa <Concern,Weight> en todos los Entornos**

  Al agregar un concern al sistema, se suscita la duda sobre qué hacer con el mapa `<Concern,Weight>` de cada uno de los Entornos ya existentes en el sistema.

  Una posible opción sería cambiar la semántica actual de dicho mapa, pasando a almacenar únicamente aquellos pesos que son distintos de cero, en otras palabras, al no encontrarse en el mapa un valor para un determinado *Concern*, se sobreentiende que su peso relativo es 0 (cero).

  Existe otra opción que no cambia la semántica de los mapas (i.e. todos los *Concerns* seguirían siendo enumerados por extensión) y para eso se vale de la herramienta Arco Iris UI. Básicamente, al agregar un nuevo *Concern*, Arco Iris UI automáticamente lo agregaría también en los mapas de todos los Entornos existentes en el sistema, pero... ¿con qué peso? ¿con peso cero? ¿con un peso por defecto configurable al crear el concern? ¿se obligaría al usuario a configurar el peso en todos los Entornos? En el

caso de eliminación, ¿qué ocurre con el peso del *Concern* eliminado? ¿se reparte entre los restantes *concerns*? Pensemos además en lo engorroso que serían todo este tipo de reajustes en la configuración para un usuario que no utiliza Arco Iris UI...

Éstas y otras preguntas son las que alguien encargado de flexibilizar la configuración de *Concerns* debería responder y, por ese motivo, preferimos sólamente plantearlas y dejar su resolución abierta.

- **Actualización del *Concern* de Escenarios**

  En el caso de dar de baja un *Concern* lo más razonable sería que Arco Iris UI lo elimine automáticamente de todos los Entornos, aunque cabe preguntarse: ¿qué ocurre con los escenarios que estaban relacionados con este *Concern*? ¿deben darse de baja? ¿debe forzarse su actualización? Nuevamente, si el usuario no usara Scenarios UI, el mantenimiento de la configuración podría tornarse dificultoso.

- **Asignación equidistribuida de pesos para un Entorno nuevo**

  En Arco Iris, al construir una instancia de `Environment`, se invoca a una función llamada `createMapWithEquallyDistributedWeights` que inicializa el mapa de pesos del Entorno de manera equidistribuida entre todos los entornos (e.g. si hay 3 *Concerns* en el sistema, asigna $0.\widehat{33}$ a cada uno).

  Esta decisión de diseño si bien es razonable, es también mejorable: esto podría ser configurable por el usuario del *framework* para poder adaptar mejor la configuración por defecto a los intereses particulares de cada sistema. Por ejemplo: es probable que un sistema financiero le otorgue más prioridad a los *concerns* relacionados con la eficiencia, mientras que una aplicación de *e-commerce* es probable que le de más importancia a la seguridad.

  En el supuesto de que los *Concerns* sean configurables por el usuario final, la función `createMapWithEquallyDistributedWeights` debería reubicarse en una clase del tipo *helper* que tenga acceso de alguna manera al `SelfHealingConfigurationManager` que es el objeto que tendría acceso a la información presente en el archivo de configuración de *self healing*.

- **Acceso del *ArcoIrisAdaptationManager* a los *Concerns***

  Una situación similar a la anterior (el acceso al universo de *Concerns* definidos en el sistema) ocurre también en el `ArcoIrisAdaptationManager`:

```
public static boolean isConcernStillBroken(String concernString) {
        try {
                Concern concern = Concern.valueOf(concernString);
                ...
```

El `ArcoIrisAdaptationManager` actualmente posee como colaborador interno a `RainbowModelWithScenarios`, el cuál accede al `SelfHealingConfigurationManager`, quién es el que puede proveer el acceso deseado a todos los *Concerns* definidos en el sistema en un momento dado.

6.6. Flexibilización del Entorno

Actualmente, el algoritmo que decide en qué entorno de ejecución se encuentra el sistema a adaptar en un determinado instante, presupone que las restricciones de todos los Entornos del sistema son mutualmente excluyentes entre sí. Esto tiene como principal consecuencia el hecho de que, al momento, no se pueden configurar Entornos que posean algún tipo de intersección en sus condiciones de aplicabilidad. En particular y a modo de ejemplo, el usuario querría poder especificar un escenario de "Alta Carga" y otro de "Extrema Alta Carga", cuyas condiciones de aplicabilidad podrían tener una intersección no nula pues probablemente las condiciones necesarias para que el entorno de "Alta Carga" se cumpla están incluidas en las condiciones el entorno de "Extrema Alta Carga".

Es deseable que en el caso de entornos con intersección no nula en sus condiciones de aplicabilidad, Arco Iris seleccione a aquel que sea más adecuado, de acuerdo al caso en particular. En el ejemplo anterior, "Extrema Alta Carga" sería el elegido para representar más fielmente el estado actual del sistema. Esto sin duda es altamente relevante ya que los pesos configurados en cada uno de los escenarios pueden determinar el curso de la auto reparación puesto que afectan directamente al algoritmo de selección de estrategias.

Una posible solución para este problema podría ser el explicitar relaciones de intersección (o de inclusión) entre restricciones y reconfigurar el algoritmo de detección del Entorno actual de ejecución existente en el `ArcoIrisAdaptationManager` para que elija aquel Entorno con condiciones más generales. A continuación, un pseudo-código del posible algoritmo que contempla **inclusión entre Entornos**:

```
función detectCurrentSystemEnvironment
        candidate = null;
        para cada ent en todos los entornos
                si ent aplica en las actuales condiciones de ejecución entonces
                        si candidate == null entonces
                                candidate = ent;
                                siguiente iteración;
                        fin si
                si ent está incluido en candidate entonces
                        candidate = ent;
                fin si
            is
        done
```

6.7. Configuración de Escenarios en AcmeStudio

Actualmente existe una herramienta de creación y edición de arquitecturas modeladas en Acme, llamada **AcmeStudio** [15] la cual se encuentra integrada en la popular herramienta de desarrollo Eclipse[16] 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 y demás conceptos introducidos en Arco Iris, utilizando la misma herramienta.

Dependiendo del grado de profundidad en el que se avance en la integración de Arco Iris con Acme Studio, se podría incluso especular con un reemplazo completo de la herramienta Arco Iris UI. Esta tarea no fue realizada en el contexto del presente trabajo debido a que el encarar el desarrollo de un *plug-in* de Eclipse con estas características es una tarea compleja que no agrega valor al esfuerzo de investigación.

6.8. Optimización en la Selección de la Estrategia

En la sección 3.9 se describe en detalle el mecanismo de selección de la mejor estrategia para resolver una situación anómala en el sistema que se está adaptando. Este problema posee muchas aristas complejas y por lo tanto, una solución sofisticada requiere un trabajo de investigación que excede lo que se puede realizar en este trabajo. Esta idea fue además compartida por los investigadores de Carnegie Mellon que fueron consultados al inicio del presente trabajo.

La solución propuesta aquí al problema de selección de la mejor estrategia de reparación, es una heurística simple basada en el uso del concepto de **utilidad del sistema**, el cual, si bien es útil para reducir la dimensión del problema y poder avanzar en otros aspectos, es claramente una solución limitada, debido a que para el cálculo de esta utilidad se realizan muchos supuestos y simplificaciones sobre el contexto de ejecución del sistema y su correlato con el modelo de su arquitectura. Las oportunidades de mejora avisoradas por los autores de este trabajo pasan principalmente por implementar mecanismos de aprendizaje sobre qué es lo que funcionó y qué no en reparaciones anteriores, por mecanismos dinámicos de modificación de los pesos de los *concerns* y prioridades de los escenarios en función del estado del sistema en un momento dado y finalmente por ajustes a las tácticas y estrategias que se aplican.

[15] Para más información acerca de AcmeStudio, visitar http://www.cs.cmu.edu/~acme/AcmeStudio
[16] Para más información acerca de Eclipse, visitar http://www.eclipse.org

6.9. Ausencia u Obsolescencia del Modelo de la Arquitectura

Tanto Rainbow como Arco Iris hacen foco en disponer de un modelo (actualizado) de la arquitectura del sistema a adaptar pero en realidad, no siempre es posible disponer de dicho modelo: puede no haber existido nunca o bien haber quedado desactualizado. Existen líneas de investigación que intentan derivar el modelo de la arquitectura a partir de un sistema, obteniendo información del mismo en *runtime*. Uno de los proyectos tiene lugar en el contexto del grupo de investigación ABLE, el responsable de Rainbow y se llama **DiscoTect**. Para más información sobre dicho proyecto, visitar http://www.cs.cmu.edu/~able/research/discotect.html.

6.10. Adaptación de un Sistema Sobre el Cual se Tiene Poco Control

Otro de los tópicos sobre los cuales es necesario también avanzar es en desarrollar técnicas y herramientas que permitan poder alterar el comportamiento de un sistema sobre el cual se posee muy poco control. Tal es el caso, por ejemplo, de sistemas desarrollados por terceras partes, de los cuales no se posee ni siquiera el código fuente. Otro ejemplo podría ser el de sistemas pobremente modularizados o codificados con un lenguaje de programación antiguo sobre los cuales parece poco probable que se puedan establecer puntos de comunicación con un *framework* de auto adaptación como Arco Iris o Rainbow.

Este tema es crítico para el desarrollo de la auto adaptación de sistemas de *software* considerando que existen en el mundo numerosas aplicaciones que se encuentran en las condiciones descritas anteriormente.

6.11. Mecanismo de Reparación "declarativo"

Hasta el momento, en Rainbow y en Arco Iris, las estrategias de reparación se encuentran configuradas de una manera imperativa, es decir, el arquitecto o una persona con ese rol, configura el *framework* especificando una serie de estrategias de reparación, que no son más que algoritmos que intentan resolver o paliar una anomalía en el sistema basandose en observaciones sobre sus propiedades en *runtime* y en un conjunto de tácticas (e.g. levantar un servidor extra, disminuir el nivel de *logging*, apagar la encripción, etc...) también provistas por el usuario del *framework*.

Otra posibilidad menos programática y más declarativa podría consistir en que el usuario únicamente especifique, para un escenario, las tácticas que podrían ejecutarse en el caso en que el escenario no se cumpla; dejando al *framework* la tarea de combinar "inteligentemente" dichas tácticas de la mejor manera posible. Es decir, el *framework* debería poder ser capaz de manejar heurísticamente los recursos que posee para interactuar con el sistema en *runtime*, de

una manera similar a cómo se opera actualmente con estrategias estáticamente configuradas por el usuario.

Lo antedicho agregaría flexibilidad puesto que el *framework* no estaría acotado a un conjunto fijo de estrategias pre configuradas sino que podría intentar otras opciones dinámicamente, considerando la historia de sus ejecuciones pasadas, entre otros tantos aspectos posibles. Por otro lado, siempre que se agrega flexibilidad, normalmente se pierde en predictibilidad, ya que las acciones del *framework* serían más difíciles de seguir y predecir.

7. Conclusiones

7.1. Resumen del trabajo realizado

En resumidas palabras, el presente trabajo añade una serie de características a Rainbow, todas ellas tendientes a disponer de una herramienta genérica de auto reparación más poderosa y flexible y que, al mismo tiempo, los actores funcionales del sistema se vean involucrados en el proceso de definición de requerimientos de atributos de calidad, puesto que este tipo de actores son los encargados de determinar (junto a arquitectos, diseñadores, etc.) cómo se debe comportar el sistema ante determinadas situaciones de operación. Las características añadidas en este trabajo son básicamente las siguientes:

- 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.
- Definición de entornos de ejecución para el sistema, con ponderaciones particulares para cada uno de los *concerns* existentes, los cuales juegan un papel determinante en el algoritmo de elección de estrategias de reparación.
- Posibilidad de asociar estrategias de reparación a escenarios.
- Modelado de un nuevo algoritmo de elección de la mejor estrategia, utilizando todos los conceptos introducidos en Arco Iris.
- Implementación de cambios en diversos módulos de Rainbow (como por ejemplo, el *RainbowModel* y el *AdaptationManager* e implementación de algunos casos prácticos que permitan mostrar cómo esta estrategia puede funcionar y llevar a un *framework* de adaptación más flexible y poderoso.
- Implementación de un mecanismo de recarga dinámica de la configuración de Arco Iris, la cuál en un futuro podría ser fácilmente integrada a Rainbow en pos de dinamizar el uso del *framework*, minimizando su *downtime* debido a cambios en la configuración.
- Implementación de una herramienta visual de escritorio que permita al usuario de Arco Iris, configurar el *framework* de una manera sencilla, amena y eficiente.

Todo lo antedicho tiene como consecuencia que los usuarios tendrán más herramientas para influir 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, dificultando el seguimiento de las decisiones tomadas por el *framework*.

7.2. Arco Iris comparado con Rainbow

En la presente sección se repasarán las ventajas que posee Arco Iris por sobre Rainbow. Se profundizará en los puntos básicos, sumarizados en la figura 48.

Problema	**Implementación en Rainbow**	**Implementación en Arco Iris**	**Ventajas**
Rapidez en el cambio de configuración	La configuración se lee una vez al principio y no puede ser cambiada dinámicamente mientras Rainbow está funcionando.	Los escenarios se actualizan dinámicamente en *runtime* al ser modificados.	Una gran parte de los cambios de configuración pasan a impactarse instantáneamente, evitando reiniciar el sistema para cambiar la configuración
Información sobre restricciones	Guardadas en el modelo de la arquitectura (en Acme)	Mediante escenarios de QAW editados por *stakeholders*	Fácil y transparente agregado y edición de restricciones.
Decisiones sobre que reparaciones realizar	También se encuentran en archivos de configuración	Se evaluán los escenarios cargados y como se afectan entre ellos. Luego se invoca a un módulo que decide de qué reparaciones pueden llevarse a cabo en ese contexto.	Posibilidad de extensión de dicho módulo para incluír complejas heurísticas que aprendan de los efectos de reparaciones pasadas, etc.
Entorno de ejecución (e.g. Alta Carga)	Configurado estáticamente. Para modificarlo es necesario el reinicio del *framework*.	Se permite especificar distintos entornos de ejecución. El sistema cambia dinámicamente de entorno sin necesidad de intervención humana ni reinicio del *framework*.	Se agrega una nueva variable a considerar para la elección de una estrategia, aumentando la probabilidad de que dicha estrategia sea más adecuada para la situación actual del sistema.

Figura 48: Ventajas de Arco Iris por sobre Rainbow

7.2.1. Rapidez en el cambio de configuración

Como se ha mencionado anteriormente, Arco Iris (es decir, Rainbow con las extensiones realizadas en este trabajo) trabaja utilizando:

- por un lado, una configuración de *Self Healing* en formato XML, producida ya sea por la herramientas Arco Iris UI o bien manualmente; la cual incluye: escenarios de QAW, entornos de ejecución y *artifacts*.

- por otro, una batería de archivos de configuración heredados de Rainbow, en distintos formatos: stitch, Acme, *properties files*, etc.

Una de las mejoras que agrega este trabajo a Rainbow es el de proveer un mecanismo dinámico de actualización de la configuración de *Self Healing* mediante el cual cualquier modificación en el archivo XML de configuración cargada al iniciar Arco Iris, conlleva un "refresco" de la configuración que está siendo usada en tiempo de ejecución por el *framework*, sin necesidad de intervención alguna de un operador.

Este dinamismo agregado en lo que respecta a los cambios de configuración, trae aparejado una considerable serie de ventajas, a saber:

- ya no es necesario reiniciar el *framework* de auto reparación para modificar cualquier característica relacionada con los escenarios, entornos de ejecución y *artifacts* utilizados en el sistema. Esto evita tiempos de no-servicio, recursos involucrados en la reconfiguración y reinicio del sistema, etc.
- se agrega flexibilidad al uso de la herramienta: se podrían agregar nuevos escenarios, nuevos entornos de ejecución que (re)definan situaciones de ejecución concretas, cambiar pesos de *concerns* y/o prioridades relativas entre escenarios de acuerdo a necesidades puntuales del negocio e infinidad de otros cambios de configuración relevantes a la auto reparación quedarían impactados en el sistema automáticamente.

7.2.2. Información sobre restricciones

Como se ha visto anteriormente, en Rainbow las restricciones impuestas sobre el funcionamiento del sistema (e.g. "el tiempo de respuesta para cualquier tipo de *request* no debe superar los 5 mílisegundos") son guardadas en el modelo de la arquitectura, el cual está expresado en el lenguaje de descripción de arquitecturas Acme. Esto claramente representa un impedimento para que los *stakeholders* no técnicos puedan participar activamente en la definición de requerimientos de auto reparación del sistema; ya que de intentarlo, deberían poder comprender y modificar correctamente un diagrama de arquitectura, con sus restricciones escritas en un lenguaje no coloquial sino especialmente técnico.

En Arco Iris, los *stakeholders* usan una interfaz visual intuitiva y fácil de utilizar (ver sección 4) para expresar restricciones del sistema en formato de escenarios de QAW; los cuales a su vez fueron pensados originalmente para facilitar la participación de personas con roles funcionales.

La información incorporada de esta manera, es analizada por Arco Iris para tomar decisiones en tiempo de ejecución sobre las auto reparaciones a realizar, considerando una variedad de aspectos, todos ellos configurados en un único lugar: los escenarios de QAW.

Como vemos, Arco Iris provee una manera simplificada de agregar y/o editar restricciones sobre el sistema, sin necesidad de modificar su modelo de arquitectura, sino por medio de los escenarios de QAW. Esto representa una avance sobre como funciona Rainbow hoy día en ese aspecto.

7.2.3. Decisiones sobre que reparaciones realizar

Con respecto a este punto, la diferencia principal entre Rainbow y Arco Iris es el enfoque: al momento de decidir qué estrategia es la más adecuada para ser ejecutada en el contexto de una o más *constraints* no cumpliendose, en Rainbow se calcula un *score* para cada una de las estrategias existentes en el sistema, mientras Arco Iris busca aquellos escenarios de QAW "rotos" y sólo asigna un *score* a las estrategias definidas en dicho subconjunto de escenarios. Este enfoque, además de ser más eficiente, es considerablemente más sencillo de configurar y tiene más sentido a nivel funcional, ya que en el caso de Rainbow no parece haber una manera sencilla de intuir de antemano qué estrategia eligiría en cada caso, hecho que complica en análisis de las reparaciones efectuadas por Rainbow durante el transcurso de una ejecución.

7.2.4. Entorno de ejecución

En Rainbow, el concepto de entorno de ejecución es estáticamente configurado en un archivo de configuración y no se modifica de acuerdo al estado dinámico del sistema que se intenta reparar. Para modificar tal estático y limitado valor, es necesario el reinicio de Rainbow.

El modelo de Arco Iris incluye el concepto de Entorno de Ejecución tal cual es descripto en la metodología ATAM[KAZ/00], el cual permite al usuario especificar distintos entornos de ejecución posibles para el sistema, los cuales poseen restricciones asociadas que se chequean continuamente y que, de cumplirse, hacen que se considere que el sistema se encuentra en otro entorno de ejecución, afectando así al algoritmo de decisión de estrategias de reparación; ya que dicha decisión está condicionada al *scoring* de estrategias de reparación, el cual se realiza considerando los pesos relativos que poseen los *concerns* arquitecturales en el entorno de ejecución actual.

Lo antedicho agrega una nueva arista a las variables consideradas al momento de elegir la mejor estrategia de reparación, aumentando las probabilidades de que la estrategia seleccionada sea más precisa de acuerdo a la situación del sistema en tiempo de ejecución.

Si bien en Rainbow el concepto de "entorno de ejecución" no existe como tal, uno podría encontrarlo indirectamente dentro de ciertas condiciones de aplicabilidad embebidas dentro del código de las estrategias de reparación.

Ahora bien, observemos que Rainbow no tiene forma de determinar a priori si alguna de las estrategias configuradas por el usuario van a aplicar en el contexto actual de ejecución: dada

alguna *constraint* violada, necesita inspeccionar la totalidad de las estrategias para verificar si son aplicables en la situación actual de *runtime*. Esto último es una notable diferencia a favor de Arco Iris, ya que en este caso, al estar las estrategias de reparación candidatas embebidas en el escenario, aquellos escenarios cuyo entorno no se corresponda con el entorno actual de ejecución, no se considerarán "rotos", evitando cómputos innecesarios y refinando así el método original provisto por Rainbow.

7.3. Compatibilidad hacia atrás: una empresa sin sentido

Uno de los objetivos de diseño más deseables en una extensión a un *framework* ya existente como Rainbow, es sin dudas que la extensión sea "compatible hacia atrás" con *Rainbow*, entendiendose ésto como la capacidad de Arco Iris de agregar nuevas características manteniendo al 100 % la funcionalidad original.

En principio, si se considera únicamente el diseño de alto nivel de Rainbow, la idea de una extensión *backwards compatible* parece factible. Pero, lamentablemente, si se consideran las diferencias conceptuales de ambos modelos, no es difícil notar que el intento de mantener a ambos conviviendo no sólo no es una tarea sencilla sino que también carece de sentido, ya que, en algunos puntos claves, son enfoques diametralmente opuestos. Veamos algunos de los motivos que apoyan esta idea:

- **Cómo se dispara la auto reparación:** El enfoque de Arco Iris hace centro en el concepto de Escenario como medio para expresar requerimientos de atributos de calidad para un sistema. Con lo cual, el inicio de todo el proceso de auto reparación se da de una manera sincrónica con respecto al Estímulo de algún escenario(s) ocurrido en el sistema en ejecución. Esto se contrapone con el enfoque de Rainbow, en el cual, periódicamente se verifican todas las restricciones e invariantes de la arquitectura. Observamos que, si quisieramos mantener esta última característica de Rainbow, ya el foco absoluto de la auto reparación no estaría puesto en los Escenarios de atributos de calidad sino que paralelamente existirían dos flujos distintos de auto reparación disputando entre sí. Esto llevaría a un comportamiento excesivamente complejo y poco predecible, lo cual resulta poco deseable para un *framework* de auto reparación.

- **Cómo se decide cuándo adaptar:** Arco Iris sólo intenta efectuar adaptaciones acotadas al escenario o los escenarios que se dejan de cumplir en un determinado momento, mientras que Rainbow contempla todo el abanico de estrategias disponibles en el caso de que detecte que una violación a una restricción o invariante ha tenido lugar. Evidentemente, los enfoques son distintos, sin mencionar que los algoritmos de *scoring* de estrategias de ambos *frameworks* difieren considerablemente, agregando el algoritmo de Arco Iris nuevas variables a la fórmula, tal como se explica en la sección 3.9.3.

7.4. Aplicabilidad en sistemas reales

Luego de haber repasado las características principales de Arco Iris y de haber entendido su mecánica, es muy probable que al lector se le suscite la siguiente pregunta: ¿Qué aplicabilidad tiene esta extensión de Rainbow en un sistema real?

Consideramos que Arco Iris se encuentra un paso más cerca de ser utilizado en un sistema de *software* real (i.e. en un ámbito no académico) que lo que su antecesor, Rainbow, se encontraba. Esta consideración se basa en el hecho de que Arco Iris hace foco en la accesibilidad del usuario final para configurar el *framework* de una manera simple y más flexible que la provista originalmente por Rainbow, incluyendo una interfaz de usuario visual que facilita dicha tarea así también como el mantenimiento y evolución de configuración existente.

Habiendo dicho lo anterior, también reconocemos que Arco Iris todavía puede no ser la herramienta más sólida y madura que los sistemas de *software* de la industria necesitan para confiar la compleja y crucial tarea de agregar auto reparación a un sistema. Esto tiene como causas diversos motivos, algunos de los más importantes son:

- Actualmente, no todas las organizaciones que desarrollan *software* poseen un modelo formal de la arquitectura, tal como es requerido por Rainbow o Arco Iris. Esto complica la adopción de *frameworks* que centren la auto reparación de un sistema en el modelo de su arquitectura. Sin embargo, como hemos mencionado en la sección 6.9, existen líneas de investigación intentando atacar este problema, derivando el modelo de la arquitectura de un sistema a partir de información obtenida durante su ejecución, lo cual representa un avance importante en pos de facilitar el uso de herramientas como Arco Iris.
- El trabajo de creación de *Gauges* y *Probes* (los cuales usualmente no son reutilizables entre distintas aplicaciones) sigue siendo una tarea de complejidad no trivial a cargo del usuario del *framework*.
- La información necesaria para que el *framework* funcione, no obstante las mejoras introducidas en Arco Iris descritas en la sección 3.10.3, sigue estando dispersa en diversos archivos de configuración, en archivos separados de estrategias y tácticas, en el modelo de la arquitectura, etc. Esto, sumado a una incompleta documentación de Rainbow, configura una curva de aprendizaje del *framework* un tanto pronunciada para un usuario nuevo. Es necesario seguir trabajando en la centralización de la configuración (tal cual fue descrito en la sección 6.4.1) y también en el incremento de la documentación para facilitar el uso, tanto de Rainbow como de Arco Iris.

7.5. Soluciones Dinámicas a Entornos de Ejecución Cambiantes

Arco Iris, la extensión a Raibow desarrollada en este trabajo, aporta un ladrillo más a la construcción de una solución de auto reparación para sistemas de *software* que permita que el

sistema, en lugar de **implementar soluciones** para satisfacer requerimientos de atributos de calidad, directamente **conozca dichos requerimientos**, disponiendo así de un *framework* genérico y personalizable que provea diversas soluciones que se **adaptan dinámicamente** al cambio de requerimientos producto de los **cambios en el entorno** de ejecución del sistema.

Este dinamismo en la auto reparación de sistemas, junto a la flexibilidad y facilidad de configuración agregadas al modelo de selección de soluciones, hacen de Arco Iris una opción interesante a ser considerada para su estudio y extensión, a fin de **continuar avanzando en la flexibilización de herramientas de auto reparación** para sistemas de *software* y consecuentemente, tender a la adopción progresiva de este tipo de herramientas por parte de la industria del *software*.

8. Bibliografía

[BAR/95] Barbacci M., Klein M., et al., "Quality Attributes", Technical Report, Software Engineering Institute (SEI), Diciembre 1995.

[BAR/03] Barbacci M., Ellison R., et al., "Quality Attributes Workshops (QAWs)", Technical Report, Software Engineering Institute (SEI), Agosto 2003.

[BAS/03] Bass L., Clements P., Kazman R. "Software Architecture in Practice". Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 2003.

[CAS/05] Casuscelli, F. "Arquitecturas de Software para Sistemas Autónomos". Trabajo Final del posgrado "Carrera de Especialización en Ingeniería del Software" (CEIS), Universidad Católica Argentina, 2005, pp. 16-30

[GAM/94] Gamma E., Helm R., et al. "Design patterns: elements of reusable object-oriented software". Addison-Wesley, USA, Introduction, Chapter pp 18, 1994.

[GAN/03] Ganek, Alan G., Corbi, Thomas A. "The dawning of the autonomic computing era". IBM Syst. J., 42(1):5-18, 2003. http://www.cs.cmu.edu/~garlan/17811/Readings/ganek.pdf

[GAR/02] Garlan, D., Schmerl, B. "Model-based adaptation for self-healing systems", Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina, USA, 18-19 Noviembre, 2002. http://portal.acm.org/citation.cfm?id=582134

[GAR/00] Garlan, D., Monroe R. T., Wile D. "Acme: Architectural Description of Component-Based Systems", Foundations of Component-Based Systems, Cambridge University Press, pp 47-68, 2000. http://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf

[HOR/01] Horn, P. "Autonomic Computing: IBM's Perspective on the State of Information Technology", IBM Corporation, Octubre 2001. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[KAZ/96] Kazman, R., Abowd, G., Bass, L., Clements, P. "Scenario-Based Analysis of Software Architecture", IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. http://eprints.kfupm.edu.sa/63611/1/63611.pdf

[KAZ/00] Kazman R., Klein, M., Clements P. "ATAM: Method for Architecture Evaluation", Software Engineering Institute (SEI), Agosto 2000. http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm

[ORI/99] Oriezy P. et al. “An Architecture-Based Approach to Self-Adaptive Software”, IEEE Intelligent Systems, vol. 14, no. 3, 1999, pp. 54-62. http://www.ics.uci.edu/~peymano/papers/ieee-is99.pdf

[PAN/10] Pandey R. “Architectural description languages (ADLs) vs UML: a review”. University Institute of Computer Science and Applications (UICSA) and R. D. University, Jabalpur (M.P.) India, 2010.

[SHA/08] Shang-Wen C. Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Mayo 2008. http://tinyurl.com/Owen-2008-0510-FinalThesis

[PAN/10] Clements P. et al., “Documenting Software Architecture: Views and Beyond”, Addison-Wesley, 2003.

Apéndices

A. Implementación de NumericBinaryRelationalConstraint

```
@XStreamAlias("numericBinaryRelationalConstraint")
public class NumericBinaryRelationalConstraint
                extends BaseSinglePropertyInvolvedConstraint {

        private NumericBinaryOperator binaryOperator;

        private Number constantToCompareThePropertyWith;

        @XStreamOmitField
        private String exponentialPropertyName;

        public NumericBinaryRelationalConstraint(Quantifier quantifier, Artifact artifact,
                        String property,NumericBinaryOperator binaryOperator,
                        Number constantToCompareThePropertyWith) {
                super(artifact, property);
                this.quantifier = quantifier;
                this.binaryOperator = binaryOperator;
                this.constantToCompareThePropertyWith = constantToCompareThePropertyWith;
        }

        public boolean holds(Number expValue) {
                boolean holds = this.binaryOperator.performOperation(expValue,
                                this.constantToCompareThePropertyWith);
                return holds;
        }
        ...
}
```

B. Implementación de Probe y extensión de Arco Iris

```
public class ClientProxyProbe {
        ...
        public void run() {
                byte[] bytes = new byte[Util.MAX_BYTES];
                URL url = new URL(urlStr);
                HttpURLConnection httpConn = (HttpURLConnection) url.openConnection();
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                int cnter = 0;
                long startTime = System.currentTimeMillis();
                int length = httpConn.getContentLength();
                BufferedInputStream in = new BufferedInputStream(httpConn.getInputStream());
                while (in.available() > 0 || cnter < length) {
                        int cnt = in.read(bytes);
                        baos.write(bytes, 0, cnt);
                        cnter += cnt;
                }
                long endTime = System.currentTimeMillis();
                String rpt = "[" + Util.probeLogTimestamp() + "]<" + id() + "> " +
                                        url.getHost() + ":" + (endTime - startTime) + "ms";
                reportData(rpt);
                try {
                        Thread.sleep(sleepTime());
                } catch (InterruptedException e) {
                        // intentional ignore
                }
        }
        ...
}
```

```
public class ClientProxyProbeWithStimulus {
        ...
        public void run() {
                byte[] bytes = new byte[Util.MAX_BYTES];
                URL url = new URL(urlStr);
                HttpURLConnection httpConn = (HttpURLConnection) url.openConnection();
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                int cnter = 0;
                long startTime = System.currentTimeMillis();
                int length = httpConn.getContentLength();
                BufferedInputStream in = new BufferedInputStream(httpConn.getInputStream());
                while (in.available() > 0 || cnter < length) {
                        int cnt = in.read(bytes);
                        baos.write(bytes, 0, cnt);
                        cnter += cnt;
                }
                long endTime = System.currentTimeMillis();
                String rpt = "[" + Util.probeLogTimestamp() + "]<" + id() + "> " +
                                url.getHost() + "<stimulus:" + stimulusName + ">:"
                                + (endTime - startTime) + "ms";
                reportData(rpt);
                httpConn.disconnect();
                try {
                        Thread.sleep(sleepTime());
                } catch (InterruptedException e) {
                        // intentional ignore
                }
        }
        ...
```

C. Configuración de Arco Iris en Representación XML

```
<selfHealingConfiguration description="test">
  <artifacts>
    <artifact id="0" name="ClientT" systemName="ZNewsSys"/>
    <artifact id="1" name="ServerT" systemName="ZNewsSys"/>
  </artifacts>
  <environments>
    <environment id="0" name="NORMAL">
      <conditions>
        <numericBinaryRelationalConstraint>
          <artifact reference="../../../../../artifacts/artifact"/>
          <property>experRespTime</property>
          <quantifier>IN_AVERAGE</quantifier>
          <binaryOperator>LESS_THAN</binaryOperator>
          <constantToCompareThePropertyWith class="int">800</constantToCompareThePropertyWith>
        </numericBinaryRelationalConstraint>
      </conditions>
      <weights class="tree-map">
        <no-comparator/>
        <entry>
          <concern>RESPONSE_TIME</concern>
          <double>0.333</double>
        </entry>
        <entry>
          <concern>SERVER_COST</concern>
          <double>0.333</double>
        </entry>
        <entry>
          <concern>CONTENT_FIDELITY</concern>
          <double>0.333</double>
        </entry>
      </weights>
    </environment>
    <environment id="1" name="HIGH LOAD">
      <conditions>
        <numericBinaryRelationalConstraint>
          <artifact reference="../../../../../artifacts/artifact"/>
          <property>experRespTime</property>
          <quantifier>IN_AVERAGE</quantifier>
          <binaryOperator>GREATER_THAN</binaryOperator>
          <constantToCompareThePropertyWith class="int">800</constantToCompareThePropertyWith>
        </numericBinaryRelationalConstraint>
      </conditions>
      <weights class="tree-map">
        <no-comparator/>
        <entry>
          <concern>RESPONSE_TIME</concern>
          <double>0.7</double>
        </entry>
        <entry>
          <concern>SERVER_COST</concern>
          <double>0.2</double>
        </entry>
        <entry>
```

```
          <concern>CONTENT_FIDELITY</concern>
          <double>0.1</double>
        </entry>
      </weights>
    </environment>
  </environments>
  <scenarios>
    <selfHealingScenario id="0" name="Client Experienced Response Time Scenario"
           enabled="true" priority="1">
      <concern>RESPONSE_TIME</concern>
      <stimulus name="GetNewsContentClientStimulus" source="Any Client requesting news content"
             any="false"/>
      <environments>
        <anyEnvironment></anyEnvironment>
      </environments>
      <artifact reference="../../../artifacts/artifact"/>
      <response>Requested News Content</response>
      <responseMeasure>
        <description>Experienced response time is within threshold</description>
        <constraint class="numericBinaryRelationalConstraint">
          <artifact reference="../../../../../artifacts/artifact"/>
          <property>experRespTime</property>
          <quantifier>IN_AVERAGE</quantifier>
          <binaryOperator>LESS_THAN</binaryOperator>
          <constantToCompareThePropertyWith class="int">500</constantToCompareThePropertyWith>
        </constraint>
      </responseMeasure>
      <repairStrategies class="specificRepairStrategies">
        <repairStrategy>VariedReduceResponseTime</repairStrategy>
      </repairStrategies>
    </selfHealingScenario>
    <selfHealingScenario id="1" name="Server Cost Scenario" enabled="true" priority="2">
      <concern>SERVER_COST</concern>
      <stimulus any="true"/>
      <environments>
        <anyEnvironment reference="../../../selfHealingScenario/environments/anyEnvironment"/>
      </environments>
      <artifact reference="../../../artifacts/artifact[2]"/>
      <response>The proper response for the request</response>
      <responseMeasure>
        <description>Active servers amount is within threshold</description>
        <constraint class="numericBinaryRelationalConstraint">
          <artifact reference="../../../../../artifacts/artifact[2]"/>
          <property>cost</property>
          <quantifier>SUM</quantifier>
          <binaryOperator>LESS_THAN</binaryOperator>
          <constantToCompareThePropertyWith class="int">4</constantToCompareThePropertyWith>
        </constraint>
      </responseMeasure>
      <repairStrategies class="specificRepairStrategies">
        <repairStrategy>ReduceOverallCost</repairStrategy>
      </repairStrategies>
    </selfHealingScenario>
  </scenarios>
</selfHealingConfiguration>
```

D. Implementación de FileSelgHealingConfigurationDao

```
public class FileSelfHealingConfigurationDao implements SelfHealingConfigurationDao {

	private static final long CONFIG_RELOAD_INTERVAL_MS = Long.valueOf(Rainbow
			.property("customize.scenarios.reloadInterval"));

	private static final String SELF_HEALING_CONFIG_FILE_NAME =
							Rainbow.property("customize.scenarios.path");

	private static final File SELF_HEALING_CONFIG_FILE =
	Util.getRelativeToPath(Rainbow.instance().getTargetPath(), SELF_HEALING_CONFIG_FILE_NAME);

	private static final Log logger = LogFactory.getLog(FileSelfHealingConfigurationDao.class);

	private SelfHealingConfiguration scenariosConfig;

	private final Collection<SelfHealingConfigurationChangeListener> listeners;

	public FileSelfHealingConfigurationDao() {
		super();
		this.listeners = new HashSet<SelfHealingConfigurationChangeListener>();
		this.loadSelfHealingConfigurationFromFile();

		TimerTask task = new FileChangeDetector(SELF_HEALING_CONFIG_FILE) {
			@Override
			protected void onChange(File file) {
				logger.info(SELF_HEALING_CONFIG_FILE_NAME + " has just changed,
							reloading Self Healing Configuration!");
				loadSelfHealingConfigurationFromFile();
				notifyListeners();
			}
		};

		Timer timer = new Timer();
		timer.schedule(task, new Date(), CONFIG_RELOAD_INTERVAL_MS);
	}

	public List<SelfHealingScenario> getAllScenarios() {
		return this.scenariosConfig.getScenarios();
	}

	public List<Environment> getAllEnvironments() {
		return this.scenariosConfig.getEnvironments();
	}

	public Environment getEnvironment(String name) {
		for (Environment environment : this.getAllEnvironments()) {
			if (environment.getName().equals(name)) {
				return environment;
			}
		}
		throw new RuntimeException("Environment " + name + " not defined.");
	}
```

```
        public List<Artifact> getAllArtifacts() {
                return this.scenariosConfig.getArtifacts();
        }

        public void register(SelfHealingConfigurationChangeListener listener) {
                this.listeners.add(listener);
        }

        protected void notifyListeners() {
                for (SelfHealingConfigurationChangeListener listener : this.listeners) {
                        listener.selfHealingConfigurationHasChanged();
                }
        }

        private void loadSelfHealingConfigurationFromFile() {
                SelfHealingConfigurationPersister persister = new SelfHealingConfigurationPersister();
                this.scenariosConfig =
                                persister.readFromFile(SELF_HEALING_CONFIG_FILE.getAbsolutePath());
        }
}
```

E. Instalación y Ejecución de Arco Iris y Arco Iris UI

E.1. Aclaración importante

En el DVD que acompaña al presente trabajo, se incluye el código fuente de Rainbow, Arco Iris y Arco Iris UI, así también como la totalidad de las herramientas necesarias para visualizar, compilar y ejecutar dichas aplicaciones. De cualquier manera, describiremos brevemente los requisitos y pasos necesarios para que cualquier persona interesada pueda instalar y ejecutar las aplicaciones producto de este trabajo.

E.2. Prerrequisitos

E.2.1. Requisitos para la Ejecución de las Aplicaciones

Java 6.0 o superior

Tanto Rainbow como Arco Iris y Arco Iris UI son aplicaciones Java que necesitan de una máquina virtual Java instalada en la computadora dónde se desee ejecutar estas aplicaciones, la cual se encuentra disponible para la gran mayoría de las plataformas utilizadas comúnmente (Windows, Unix, Solaris, MAC OS, etc.) y puede ser descargada gratuitamente desde http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Luego de instalar la máquina virtual de Java, es necesario configurar la variable de entorno `JAVA_HOME` para que apunte al directorio de instalación. Ejemplos:

En Windows:

```
JAVA_HOME = %ProgramFiles%\Java\jdk1.6.0_23
```

En Linux:

```
JAVA_HOME = /usr/lib/jvm/java-6-sun
```

Luego, es necesario que el ejecutable de Java se encuentre en el *path* del sistema operativo. Para lograr esto, es necesario hacer lo siguiente:

En Windows:

```
PATH = %PATH%;%JAVA_HOME%\bin
```

En Linux:

```
PATH = $PATH:$JAVA_HOME/bin
```

Para verificar que el ejecutable de Java va a poder ser utilizado normalmente, ejecutar el siguiente comando:

```
> java -version
```

Se debería obtener una respuesta de este tipo:

```
java version "1.6.0_21"
Java(TM) SE Runtime Environment (build 1.6.0_21-b06)
Java HotSpot(TM) 64-Bit Server VM (build 17.0-b16, mixed mode)
```

E.2.2. Requisitos para la Inspección y Edición del Código Fuente

Eclipse 3.6 o superior

En el caso en que se desee inspeccionar o editar el código fuente de Rainbow, Arco Iris y/o Arco Iris UI, una herramienta no imprescindible aunque muy recomendada es **Eclipse**. Eclipse es un entorno de desarrollo integrado que permite visualizar, editar, compilar y ejecutar aplicaciones Java (entre muchas otras cosas). Se descarga gratuitamente desde http://www.eclipse.org/downloads y la versión "for Java Developers" es suficiente a los fines de trabajar con las aplicaciones que conciernen a este trabajo.

El código fuente incluído incluye metadatos de Eclipse, con lo cual, si se utilizan los correspondientes *plugins* detallados a continuación, debería ser relativamente sencillo poder disponer rápidamente de todos los proyectos que conforman el tandem Rainbow-Arco Iris. los *plugins* de Eclipse requeridos son:

- **Maven plugin**: El plugin para Eclipse de Maven, el cuál puede ser obtenido utilizando el *Marketplace* de Eclipse o utilizando la siguiente URI http://m2eclipse.sonatype.org/sites/m2e.[17]
- **Subclipse**: Se trata del plugin para Eclipse que provee soporte para interactuar con repositorios Subversion[18]. Tanto el código de Rainbow como el de Arco Iris se encuentran alojados en servidores Subversion (SVN).

Maven 2 o superior

A fin de compilar y empaquetar el código fuente de la aplicación mediante la línea de comandos del sistema operativo, es necesario utilizar Maven, la popular herramienta de construcción de aplicaciones Java. Maven es una aplicación multi plataforma con una abundante

[17] Se presupone que el lector está familiarizado con el mecanismo de instalación de un *plugin* de Eclipse, caso contrario, ver una explicación en la siguiente página web: http://wiki.eclipse.org/FAQ_How_do_I_install_new_plug-ins%3F

[18] Para más información sobre Subversion, visitar http://subversion.tigris.org/

cantidad de documentación de calidad, la cual puede ser consultada visitando su página web: http://maven.apache.org/, desde la cual se pueden descargar también los binarios de dicha aplicación.

Para instalar Maven, basta con descomprimir el archivo `.zip` (o `.tar.gz`) en un directorio a elección y luego configurar la variables `M2_HOME` para que apunte a dicho directorio:

```
M2_HOME = ~/apache-maven-3.0.3
```

Luego, es necesario que el ejecutable de Maven se encuentre en el *path* del sistema operativo. Para ello, es necesario hacer lo siguiente:

```
PATH = $PATH:$M2_HOME/bin
```

Para chequear si Maven va a poder ser utilizado normalmente, ejecutar el siguiente comando:

```
> mvn --version
```

Se debería obtener una respuesta de este tipo:

```
Apache Maven 3.0.3 (r1075438; 2011-02-28 14:31:09-0300)
Maven home: ~/apache-maven-3.0.3
Java version: 1.6.0_21, vendor: Sun Microsystems Inc.
Java home: /usr/lib/jvm/java-6-sun-1.6.0.21/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "2.6.38-8-generic", arch: "amd64", family: "unix"
```

E.3. Obtención del código fuente de Rainbow, Arco Iris y Arco Iris UI

El grupo ABLE posee un repositorio SVN privado, el cuál es sólo accesible mediante el otorgamiento de un usuario (normalmente de sólo lectura) con contraseña, a pedido explícito del interesado. A fin de obtener dichas credenciales de acceso, es necesario redactar un e-mail a David Garlan <garlan@cs.cmu.edu>. Sin embargo, no es necesario obtener estas credenciales puesto que en el repositorio SVN de Arco Iris (http://code.google.com/p/arco-iris) se puede encontrar todo el código necesario a los fines del presente trabajo.

Tanto Arco Iris como Arco Iris UI son de código abierto (*open source*), esto significa que cualquier persona puede acceder libremente al código fuente y examinarlo.

E.4. Estructura de Directorios del Presente Trabajo

A continuación, se observa la estructura de directorios del código fuente de las aplicaciones creadas/modificadas en el presente trabajo.

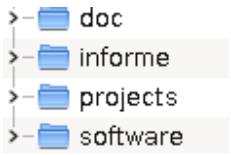


Figura 49: Estructura de directorios de este trabajo

doc	Aquí se incluye un subconjunto de la bibliografía utilizada para la realización del presente trabajo, a modo de referencia rápida en caso de estar leyendo el presente informe en forma digital.
informe	Incluye la propuesta de tesis y este documento en formato PDF.
projects	Aquí se encuentra la totalidad del código fuente correspondiente a las aplicaciones objeto de este trabajo: Arco Iris, Arco Iris UI, y algunos módulos de Rainbow que fueron ligeramente modificados.
software	Aquí se incluyen los prerequisitos para compilar y ejecutar las aplicaciones. (ver apéndice E.2)

Hilando más fino en el directorio más importante de los anteriormente mencionados, observamos los proyectos contenidos en el directorio `projects`:

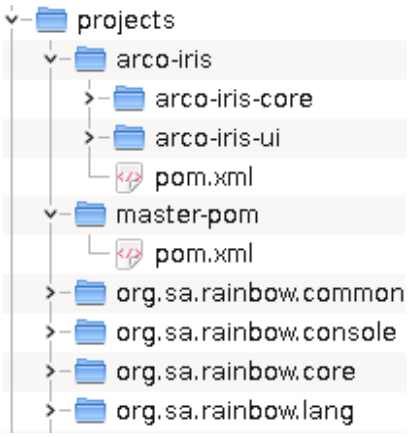


Figura 50: Estructura de directorios del directorio “projects”

arco-iris	La totalidad del código nuevo desarrollado para este trabajo, dividido en `arco-iris-core`, Arco Iris propiamente dicho; y `arco-iris-ui`.
master-pom	Este POM[19] de Maven contiene descripciones de características comunes a los proyectos relacionados con Arco Iris: `arco-iris-core`, y `arco-iris-ui`.

[19] Para más información sobre los **P**roject **O**bject **M**odel de Maven, referirse a http://maven.apache.org/pom.html

org.sa.rainbow.* Aquellos proyectos Java cuyo nombre comienza con el prefijo `org.sa.rainbow` son básicamente **copias** de submódulos de Rainbow que fueron modificados ligeramente principalmente a fines de mejorar el *logging* de los componentes *core* de Rainbow. A fines prácticos, se podría prescindir de estos proyectos y Arco Iris debería seguir funcionando de la misma manera, aunque el *log* de la aplicación sería bastante más verboso.

E.5. Compilación de Arco Iris y Arco Iris UI

A fines de compilar y construir los artefactos `.jar` que luego serán utilizados para ejecutar Arco Iris y Arco Iris UI, es necesario:

- construir el `master-pom`, el cual luego será utilizado tanto por Arco Iris como por Arco Iris UI.
- instalar las dependencias de `arco-iris-core` en el repositorio local de Maven.
- construir `arco-iris-core` y `arco-iris-ui` de una sola vez, utilizando el *reactor-pom* ubicado dentro del directorio `arco-iris`.

En los próximos apartados se describirán en detalle estos 3 simples pasos.

E.5.1. Construcción del pom maestro

El primer paso es muy simple: simplemente es necesario posicionarse en el directorio `master-pom` e instalar el pom que allí se encuentra ubicado:

```
cd master-pom
mvn clean install
```

Esto instalará el pom en el repositorio local de Maven. Este pom será utilizado por el resto de los módulos a compilar.

E.5.2. Instalación Manual de Dependencias del Core de Arco Iris

Maven es un sistema de construcción de aplicaciones Java que se basa en el supuesto de que todas las dependencias de una aplicación deben ser encontrables en un repositorio de artefactos Maven (ya sea en Internet, o en un repositorio *cache* local en la computadora del usuario). La gran mayoría de las dependencias de Arco Iris cumplen con esta condición y se pueden encontrar en el repositorio oficial de Maven (http://repo1.maven.org/maven2/).

Lamentablemente, las dependencias de Arco Iris relacionadas con Rainbow no se encuentran alojadas en ningún repositorio Maven público, y mucho menos aquellos proyectos que se han modificado ligeramente en el curso de este trabajo. Es por eso que es necesario, **instalar manualmente** tales dependencias (archivos `.jar`) a fin de que, durante el proceso de construcción, sean encontradas por Maven.

Afortunadamente, Maven ofrece una forma sencilla de instalar manualmente artefactos en el repositorio local. Los comandos a ejecutar están especificados en el *script* (el cuál debería funcionar tanto en Linux como en Windows) ubicado en la siguiente ruta: `projects/arco-iris/arco-iris-core/lib/install_missing_artifacts.cmd`.

El mismo procedimiento debe ser reiterado para aquellas dependencias de `arco-iris-ui` que no se encuentran en repositorios Maven públicos. Para ello, es necesario correr el *script* `projects/arco-iris/arco-iris-ui/lib/install_missing_artifacts.cmd`.

Cabe mencionar que para los 2 proyectos Java aquí presentados (`arco-iris-core` y `arco-iris-ui`) se incluye un directorio `lib` con las dependencias de cada proyecto, de modo que cada uno pueda compilar y correr sin que el usuario necesite disponer de una conexión a internet.

E.5.3. Construcción de los Proyectos con Maven

Generar los binarios de las dos aplicaciones anteriormente mencionadas, es tan simple como ejecutar los siguientes dos comandos:

```
cd arco-iris
mvn clean install
```

Esto debería generar:

1. el archivo `arco-iris-ui-0.0.1-SNAPSHOT.jar` con los binarios de **Arco Iris UI**, compilados utilizando las librerías nativas del sistema operativo subyacente (Windows y Linux están actualmente soportados, para sus versiones de 32 y 64 bits)

2. el archivo `arco-iris-ui-0.0.1-SNAPSHOT-distribution.zip` que contiene el `.jar` anteriormente mencionado de **Arco Iris UI** junto con un directorio `lib` que posee todas las dependencias de dicha aplicación.

3. un archivo `arco-iris-core-0.0.1-SNAPSHOT-distribution.zip` que contiene el `.jar` del *core* de **Arco Iris** junto con un directorio `lib` que posee todas sus dependencias y un directorio `configs` el cual contiene los casos de prueba realizados en el presente trabajo y detallados en la sección 5.

E.6. Ejecución de las aplicaciones

E.6.1. Arco Iris

Arco Iris (al igual que Rainbow) toma como parámetros de entrada la ubicación de archivos de configuración que le indican a la aplicación la ubicación del sistema al cual adaptar (o datos sobre la simulación de tal sistema), ubicación del modelo de la arquitectura de dicho sistema, etc...

Supongamos que se desea invocar a Arco Iris utilizando el caso de prueba denominado como "TestCase4". La forma de llevar a cabo ésto es la siguiente:

```
cd arco-iris/arco-iris-core/target
unzip arco-iris-core-0.0.1-SNAPSHOT-distribution.zip
java -Drainbow.config=configs
     -Drainbow.target=TestCase4
     -jar arco-iris-core-0.0.1-SNAPSHOT.jar
```

E.6.2. Arco Iris UI

Ejecutar Arco Iris UI es similar a ejecutar Arco Iris, sólo que la aplicación recibe no recibe parámetros:

```
cd arco-iris/arco-iris-ui/target
unzip arco-iris-ui-0.0.1-SNAPSHOT-distribution.zip
java -jar arco-iris-ui-0.0.1-SNAPSHOT.jar
```

F. Arquitectura de Znn modelada con Acme

```
import TargetEnvType.acme;

Family ZNewsFam extends EnvType with {

        Port Type HttpPortT extends ArchPortT with {

        }
        Role Type RequestorRoleT extends ArchRoleT with {

        }
        Component Type ProxyT extends ArchElementT with {

                Property deploymentLocation : string <<  default : string = "localhost"; >> ;

                Property load : float <<  default : float = 0.0; >> ;

        }
        Port Type ProxyForwardPortT extends ArchPortT with {
```

```
}
Component Type ServerT extends ArchElementT with {

        Property deploymentLocation : string <<  default : string = "localhost"; >> ;

        Property load : float <<  default : float = 0.0; >> ;

        Property reqServiceRate : float <<  default : float = 0.0; >> ;

        Property byteServiceRate : float <<  default : float = 0.0; >> ;

        Property fidelity : int <<  HIGH : int = 5; LOW : int = 1; default : int = 5; >> ;

        Property cost : float <<  default : float = 1.0; >> ;

        Property lastPageHit : Record [uri : string; cnt : int; kbytes : float; ];

        Property anotherConstraint : string
                        <<  default : string = "heuristic self.load <= MAX_UTIL;"; >> ;
}
Role Type ReceiverRoleT extends ArchRoleT with {

}
Connector Type ProxyConnT extends ArchConnT with {
        Role req : RequestorRoleT = new RequestorRoleT extended with {

        }
        Role rec : ReceiverRoleT = new ReceiverRoleT extended with {

        }
}
Component Type ClientT extends ArchElementT with {

        Property deploymentLocation : string <<  default : string = "localhost"; >> ;

        Property experRespTime : float <<  default : float = 100.0; >> ;

        Property reqRate : float <<  default : float = 0.0; >> ;
        rule primaryConstraint = invariant self.experRespTime <= MAX_RESPTIME;
        rule reverseConstraint = heuristic self.experRespTime >= MIN_RESPTIME;

}
Port Type HttpReqPortT extends ArchPortT with {

}
Connector Type HttpConnT extends ArchConnT with {

        Property bandwidth : float <<  default : float = 0.0; >> ;

        Property latency : float <<  default : float = 0.0; >> ;

        Property numReqsSuccess : int <<  default : int = 0; >> ;

        Property numReqsRedirect : int <<  default : int = 0; >> ;
```

```
                Property numReqsClientError : int <<  default : int = 0; >> ;

                Property numReqsServerError : int <<  default : int = 0; >> ;

                Property latencyRate : float;
                Role req : RequestorRoleT = new RequestorRoleT extended with {

                }
                Role rec : ReceiverRoleT = new ReceiverRoleT extended with {

                }
        }

        Property MIN_RESPTIME : float;

        Property MAX_RESPTIME : float;

        Property TOLERABLE_PERCENT_UNHAPPY : float;

        Property UNHAPPY_GRADIENT_1 : float;

        Property UNHAPPY_GRADIENT_2 : float;

        Property UNHAPPY_GRADIENT_3 : float;

        Property FRACTION_GRADIENT_1 : float;

        Property FRACTION_GRADIENT_2 : float;

        Property FRACTION_GRADIENT_3 : float;

        Property MIN_UTIL : float;

        Property MAX_UTIL : float;

        Property MAX_FIDELITY_LEVEL : int;

        Property THRESHOLD_FIDELITY : int;

        Property THRESHOLD_COST : float;

        Property SUPPORT_FRACTION_GRADIENT : boolean;
}

System ZNewsSys : ZNewsFam = {

        Property MIN_RESPTIME : float = 100.0;

        Property MAX_RESPTIME : float = 1000.0;

        Property UNHAPPY_GRADIENT_1 : float = 0.1;

        Property UNHAPPY_GRADIENT_2 : float = 0.2;
```

```
Property UNHAPPY_GRADIENT_3 : float = 0.5;

Property FRACTION_GRADIENT_1 : float = 0.2;

Property FRACTION_GRADIENT_2 : float = 0.4;

Property FRACTION_GRADIENT_3 : float = 1.0;

Property TOLERABLE_PERCENT_UNHAPPY : float = 0.4;

Property MIN_UTIL : float = 0.1;

Property MAX_UTIL : float = 0.75;

Property MAX_FIDELITY_LEVEL : int = 5;

Property THRESHOLD_FIDELITY : int = 2;

Property THRESHOLD_COST : float = 4.0;

Property SUPPORT_FRACTION_GRADIENT : boolean = false;

Component s1 : ServerT, ArchElementT = new ServerT, ArchElementT extended with {

        Property deploymentLocation = "phoenix";

        Property isArchEnabled = true;

        Property cost = 1.0;

        Property fidelity = 3;

        Property load = 0.891;
        Port http0 : HttpPortT, ArchPortT = new HttpPortT, ArchPortT extended with {

        }

}

Component lbproxy : ProxyT = new ProxyT extended with {

        Property deploymentLocation = "127.0.0.1";

        Property isArchEnabled = true;

        Property load = 0.01;

        Port fwd0 : ProxyForwardPortT = new ProxyForwardPortT extended with {

                        Property isArchEnabled = true;
        }
        Port fwd1 : ProxyForwardPortT = new ProxyForwardPortT extended with {

        }
        Port fwd2 : ProxyForwardPortT = new ProxyForwardPortT extended with {
```

```
        }
        Port fwd3 : ProxyForwardPortT = new ProxyForwardPortT extended with {

        }
        Port http0 : HttpPortT = new HttpPortT extended with {

                Property isArchEnabled = true;
        }
        Port http1 : HttpPortT = new HttpPortT extended with {

                Property isArchEnabled = true;
        }
        Port http2 : HttpPortT = new HttpPortT extended with {

                Property isArchEnabled = true;
        }
}

Component s2 : ServerT, ArchElementT = new ServerT, ArchElementT extended with {

        Property deploymentLocation = "127.0.0.3";

        Property isArchEnabled = true;

        Property fidelity = 5;

        Property load = 0.99;

        Property cost = 1.0;

        Port http0 : HttpPortT, ArchPortT = new HttpPortT, ArchPortT extended with {

                Property isArchEnabled = false;
        }

}

Component s3 : ServerT, ArchElementT = new ServerT, ArchElementT extended with {

        Property deploymentLocation = "127.0.0.4";

        Property isArchEnabled = true;

        Property cost = 1.0;

        Property fidelity = 3;

        Property load = 0.891;

        Port http0 : HttpPortT, ArchPortT = new HttpPortT, ArchPortT extended with {

        }
}
```

```
Component s0 : ServerT, ArchElementT = new ServerT, ArchElementT extended with {

        Property deploymentLocation = "oracle";

        Property cost = 0.9;

        Property fidelity = 3;

        Property load = 0.594;

        Property isArchEnabled = true;

        Port http0 : HttpPortT = new HttpPortT extended with {

                        Property isArchEnabled = true;
        }
}
Component c1 : ClientT = new ClientT extended with {

        Property deploymentLocation = "127.0.0.1";

        Property isArchEnabled = true;

        Property experRespTime = 433.36273;

        Port p0 : HttpReqPortT = new HttpReqPortT extended with {

                        Property isArchEnabled = true;
        }
}

Component c2 : ClientT = new ClientT extended with {

        Property deploymentLocation = "127.0.0.1";

        Property isArchEnabled = true;

        Property experRespTime = 344.6827;

        Port p0 : HttpReqPortT = new HttpReqPortT extended with {

                        Property isArchEnabled = true;
        }
}

Component c0 : ClientT = new ClientT extended with {

        Property deploymentLocation = "127.0.0.1";

        Property isArchEnabled = true;

        Property experRespTime = 414.76843;

        Port p0 : HttpReqPortT = new HttpReqPortT extended with {
```

```
                        Property isArchEnabled = true;
        }
}

Connector conn0 : HttpConnT = new HttpConnT extended with {

        Property latencyRate = 0.0;

        Property isArchEnabled = true;

        Role req  = {

                        Property isArchEnabled = true;
        }
        Role rec  = {

                        Property isArchEnabled = true;
        }
}

Connector proxyconn0 : ProxyConnT = new ProxyConnT extended with {

        Property isArchEnabled = true;

        Role req  = {

                        Property isArchEnabled = true;

        }

        Role rec  = {

                        Property isArchEnabled = true;
        }
}

Connector proxyconn1 : ProxyConnT, ArchConnT = new ProxyConnT, ArchConnT extended with {

}
Connector proxyconn3 : ProxyConnT, ArchConnT = new ProxyConnT, ArchConnT extended with {

}

Connector proxyconn2 : ProxyConnT, ArchConnT = new ProxyConnT, ArchConnT extended with {

}

Connector conn : HttpConnT = new HttpConnT extended with {

        Property latencyRate = 0.0;

        Property isArchEnabled = true;

        Role req  = {
```

```
                                Property isArchEnabled = true;
                }
                Role rec  = {

                                Property isArchEnabled = true;
                }
        }

        Connector conn1 : HttpConnT = new HttpConnT extended with {

                Property latencyRate = 0.0;

                Property isArchEnabled = true;

                Role req  = {

                                Property isArchEnabled = true;
                }
                Role rec  = {

                                Property isArchEnabled = true;
                }
        }

        Attachment lbproxy.fwd0 to proxyconn0.req;
        Attachment s0.http0 to proxyconn0.rec;
        Attachment lbproxy.http0 to conn0.rec;
        Attachment c1.p0 to conn.req;
        Attachment c2.p0 to conn1.req;
        Attachment lbproxy.http2 to conn1.rec;
        Attachment lbproxy.http1 to conn.rec;
        Attachment c0.p0 to conn0.req;
        Attachment s2.http0 to proxyconn2.rec;
        Attachment lbproxy.fwd1 to proxyconn1.req;
        Attachment s1.http0 to proxyconn1.rec;
        Attachment s3.http0 to proxyconn3.rec;
        Attachment lbproxy.fwd3 to proxyconn3.req;
        Attachment lbproxy.fwd2 to proxyconn2.req;
}
```

G. Tácticas de Znn

```
module newssite.tactics;

import model "ZNewsSys.acme" { ZNewsSys as M, ZNewsFam as T };
import op "znews0.operator.EffectOp" { EffectOp as S };
import op "org.sa.rainbow.stitch.lib.*";


/**
* Enlist n free servers into service pool.
* Utility: [v] R; [^] C; [<>] F
*/
tactic enlistServers (int n) {
        condition {
                // some client should be experiencing high response time
                exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
                // there should be enough available server resources
                Model.availableServices(T.ServerT) >= n;
        }
        action {
                set servers = Set.randomSubset(Model.findServices(T.ServerT), n);
                for (T.ServerT freeSvr : servers) {
                        S.activateServer(freeSvr);
                }
        }
        effect {
                // response time decreasing below threshold should result
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
        }
}

/**
* Deactivate n servers from service pool into free pool.
* Utility: [^] R; [v] C; [<>] F
*/
tactic dischargeServers (int n) {
        condition {
                // there should be NO client with high response time
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
                // there should be enough servers to discharge
                Set.size({ select s : T.ServerT in M.components | s.load < M.MIN_UTIL }) >= n;
        }
        action {
                set lowUtilSvrs = { select s : T.ServerT in M.components | s.load < M.MIN_UTIL };
                set subLowUtilSvrs = Set.randomSubset(lowUtilSvrs, n);
                for (T.ServerT s : subLowUtilSvrs) {
                        S.deactivateServer(s);
                }
        }
        effect {
                // still NO client with high response time
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
        }
}
```

```
/**
* Lowers fidelity by integral steps for percent of requests.
* Utility: [v] R; [v] C; [v] F
*/
tactic lowerFidelity (int step, float fracReq) {
        condition {
                // some client should be experiencing high response time
                exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
                // exists server with fidelity to lower
                exists s : T.ServerT in M.components | s.fidelity > step;
        }
        action {
                // retrieve set of servers who still have enough fidelity grade to lower
                set servers = { select s : T.ServerT in M.components | s.fidelity > step };
                for (T.ServerT s : servers) {
                        S.setFidelity(s, s.fidelity - step);
                }
        }
        effect {
                // response time decreasing below threshold should result
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
        }
}

/**
* Raises fidelity by integral steps for percent of requests.
* Utility: [^] R; [^] C; [^] F
*/
tactic raiseFidelity (int step, float fracReq) {
        condition {
                // there should be NO client with high response time
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
                // there exists some client with below low-threshold response time
                exists c : T.ClientT in M.components | c.experRespTime < M.MIN_RESPTIME;
        }
        action {
                // first find the lowest fidelity set
                set servers = { select s :
                                T.ServerT in M.components | s.fidelity <= M.MAX_FIDELITY_LEVEL - step};
        for (T.ServerT s : servers) {
                        S.setFidelity(s, java.lang.Math.min(s.fidelity + step, M.MAX_FIDELITY_LEVEL));
                }
        }
        effect {
                // still NO client with high response time
                forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
        }
}
```

H. Representación en XML del Escenario de Tiempo de Respuesta

```
<selfHealingScenario id="0" name="Client Experienced Response Time Scenario" enabled="true"
        priority="1">
        <concern>RESPONSE_TIME</concern>
        <stimulusSource>Any Client requesting news content</stimulusSource>
        <stimulus>GetNewsContentClientStimulus</stimulus>
        <environments>
                <defaultEnvironment></defaultEnvironment>
        </environments>
        <artifact reference="../../../artifacts/artifact"/>
        <response>Requested News Content</response>
        <responseMeasure>
                <description>Experienced response time is within threshold</description>
                <constraint class="numericBinaryRelationalConstraint" sum="false">
                        <artifact reference="../../../../../artifacts/artifact"/>
                        <property>experRespTime</property>
                        <binaryOperator>LESS_THAN</binaryOperator>
                        <constantToCompareThePropertyWith class="int">600
                                </constantToCompareThePropertyWith>
                </constraint>
        </responseMeasure>
        <architecturalDecisions/>
        <repairStrategy></repairStrategy>
</selfHealingScenario>
```

I. Extracto del *Log* de un Caso de Prueba de Arco Iris

A continuación, se incluye un extracto significativo del *log* generado por Arco Iris para el caso de prueba número 2, presentado anteriormente en la sección 5.5. Dada la longitud del *log* original, el mismo ha sido recortado a fin de mostrar las partes más importantes del procesamiento que efectúa Arco Iris para el caso de prueba de referencia.

Observese que Arco Iris al iniciar, toma desde el directorio `/home/user/arco-iris/arco-iris-core/configs/TestCase2` toda la configuración requerida, tanto por Rainbow como por Arco Iris (archivo `.sim` de simulación, tácticas y estrategias, el modelo de la arquitectura en Acme, el archivo XML de configuración de Arco Iris, etc.)

Nótese que luego Arco Iris, de acuerdo a la configuración en el archivo de simulación, efectúa la modificación de varias propiedades del sistema de acuerdo a lo que allí se indica.

Finalmente, para esta porción de *log*, se buscan escenarios rotos, no encontrando ninguno en esta etapa de la ejecución.

```
Rainbow config path: /home/user/arco-iris/arco-iris-core/configs/TestCase2
Loading Rainbow config file: [rainbow.properties]
/========================/
|  Trial run 1 of 1...
\========================\
Loading simulation file from/home/user/arco-iris/arco-iris-core/configs/TestCase2/TestCase.sim
(...)
Change prop: ZNewsSys.c1.perf.service.time = 180
Change prop: ZNewsSys.s0.isArchEnabled = true
Change prop: ZNewsSys.conn2.bandwidth.up = 512
Change prop: sim.size = 6
Change prop: ZNewsSys.c1.experRespTime = 50
Change prop: ZNewsSys.s1.fidelity = 5
Change prop: ZNewsSys.s1.cost = 1.0
Change prop: ZNewsSys.lbproxy.load = 0.01
Change prop: ZNewsSys.conn1.bandwidth.up = 256
Change prop: ZNewsSys.conn2.bandwidth.down = 512
Change prop: ZNewsSys.s0.load = 0.01
(...)
Self Healing Configuration successfully loaded from /home/.../TestCase2/selfHealingConfiguration.xml
Arco Iris Model initialized!
Arco Iris Model started
Arco Iris Adaptation Manager started.
Finding broken scenarios...
Is scenario Client Experienced Response Time Scenario broken?
Client Experienced Response Time Scenario broken for [EAvg] 50.0? false
Client Experienced Response Time Scenario --> PASSED!
END Finding broken scenarios!
Finding broken scenarios...
                <several iterations without finding any broken scenarios...>
```

Luego de un tiempo, se vuelve a ejecutar la rutina de búsqueda de escenarios que no se cumplen y se detecta que el escenario de tiempo de respuesta se encuentra “roto” (el tiempo

de respuesta ha subido a 602,25 ms. cuando el umbral máximo era 600 ms.). Se evalúa la única estrategia existente para dicho escenario y se decide ejecutarla debido a que se estima que el tiempo de respuesta luego de la ejecución de dicha estrategia, será de 458,25 ms.

Nótese que los resultados de ejecutar la táctica `enlistServers` (que agrega un servidor más al sistema), difícilmente sean instantáneos, es decir, con lo cual, junto a la ejecución de dicha táctica, se determina una demora (*delay*) de 2,5 segundos (2529,55 ms.) para esperar a que los resultados de la táctica sean visibles en el sistema en *runtime*.

```
Finding broken scenarios...
Is scenario Client Experienced Response Time Scenario broken?
Client Experienced Response Time Scenario broken for [EAvg] 602.25? true
Client Experienced Response Time Scenario --> BROKEN!
END Finding broken scenarios!
Adaptation triggered, let's begin!
Current environment: NORMAL
Computing Current System Utility...
Client Experienced Response Time Scenario broken for [EAvg] 602.25? true
Current System Utility (Score to improve): 0.0
Evaluating strategy EnlistServerResponseTime...
Scoring EnlistServerResponseTime...
Client Experienced Response Time Scenario broken after simulation for Resp.time ([EAvg] 458.25)? false
Score for strategy EnlistServerResponseTime: 0.333
Current best strategy: EnlistServerResponseTime
Selected strategy: EnlistServerResponseTime!!!
--> EXECUTING STRATEGY EnlistServerResponseTime...
t0: true? true
Tactic action! enlistServers
Effector time delay to simulate resource limitations: 2529.55
```

Finalmente, luego de varios segundos de ejecución y habiendo esperado el tiempo prudencial para determinar si el haber levantado un servidor más en el sistema tuvo el efecto esperado o no, cerca del segundo 8 de ejecución (8012 ms.) se arriba a un estado dónde el sistema baja su tiempo de respuesta promedio a 583,73, ligeramente por debajo del umbral máximo: 600 ms. Así, el sistema decide que el resultado (*outcome*) de la estrategia es exitoso y determina que en ese momento ya no hay más escenarios sin cumplirse.

Alrededor de los 40 segundos (valor configurado en el archivo de simulación), la ejecución del sistema simulado termina y se indica a al tándem Rainbow/Arco Iris que termine también, finalizando así el ciclo de adaptación.

```
------------ Running ZNewsSim at 8012------------
Is Concern Response time Still Broken?
Client Experienced Response Time Scenario broken for [EAvg] 583.73? false
Concern Response time Not Broken Anymore!
t1: !? true
t2: DEFAULT condition!
DONE action!
Outcome: SUCCESS
```

```
------------- Running ZNewsSim at 8512------------
Change prop: ZNewsSys.s0.load = 0.33
Change prop: ZNewsSys.s1.load = 0.45
Change prop: ZNewsSys.c0.experRespTime = 475.57
Change prop: ZNewsSys.c1.experRespTime = 687.08
Change prop: ZNewsSys.c2.experRespTime = 429.82
Finding broken scenarios...
Is scenario Client Experienced Response Time Scenario broken?
Client Experienced Response Time Scenario broken for [EAvg] 578.51? false
Client Experienced Response Time Scenario --> PASSED!
END Finding broken scenarios!
(...)
------------- Running ZNewsSim at 40087------------
*** Signalling Terminate ***
TargetSystem Simulation Runner terminated.
****** Rainbow Runtime Infrastructure - Master terminated! ******
```