Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

“Enriquecimiento de Componentes con Self-Knowledge: Técnica TFT”

TESIS DE LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

Alumno:

Mónica Graciela Silva LU 487/91 msilva@dc.uba.ar

Director:

Dr. Daniel Yankelevich

Buenos Aires, Mayo de 2007

Agradecimientos

En primer lugar les agradezco a mi mamá, Graciela, y a mi papá, Raúl, su enorme esfuerzo para que desde muy pequeña pudiera estudiar y formarme, les agradezco enormemente haberme inculcado el amor a aprender y crecer en la búsqueda del conocimiento.
Quiero agradecerle a mis profesores universitarios, en especial a Mónica Bobrowski, Victor Braberman, Marcelo Kock, Rosa Wachenchauzer, Marcelo Frias, Mario Masa, Nicolás Kicillof, Emilio Platzer, Daniel Yankelevich, Nelson Sprejer, Miguel Felder, Martina Marré, Daniel Johnson, Alfredo Vega Weiss, y Sergio Freu por haber despertado, acompañado y alimentado mi interés por la informática, su investigación y desarrollo siempre con un brillante nivel académico, profesional y humano.
También quiero agradecerles a mis amigos y compañeros de carrera Pablo, Sebastián y Flavio por acompañar, alentar y enriquecer mis horas de estudio.
Quiero agradecerle a mi director su apoyo y su confianza en mi para lograr este trabajo.
También les agradezco a mis revisores Analía, Federico, Pablo y Sebastián por sus toques finales, apoyo, aliento y confianza.
En especial quiero agradecerle al hombre de mi vida, mi amor, Fede, su soporte, su apoyo, su esfuerzo para contenerme y poner a mi disposición todo para transitar la recta final de mi carrera.
Y finalmente le agradezco mis hijos, Lucas y Gonzalo, el tiempo que me regalaron, su compañía, su amor inspirador, las tardes soleadas en el club junto a las sonrisas de Lucas, y las noches de teclear y teclear con los dulces movimientos de Gonza en la panza. A ellos les dedico este trabajo hecho con gran esfuerzo y enorme placer!

Y gracias a Dios por acompañarme siempre y por poner a todos ellos en mi camino.

Resumen

El incremento en la demanda del software, la alta disponibilidad requerida y la amplia heterogeneidad de los entornos de ejecución han forzado a la comunidad científica a desarrollar estrategias y técnicas para satisfacer los nuevos desafíos que se plantean en el desarrollo y administración de sistemas. La tecnología de componentes ha sido una de las primeras soluciones propuestas, pero problemas como los ocasionados por la falta de información, suposiciones e inconsistencias entre las partes profundizaron sus consecuencias en las tareas de aseguramiento de calidad. A principios de siglo tomó importancia la idea de desarrollar sistemas con auto-capacidades, eso es, sistemas con la habilidad de detectar, comprender y adaptarse a los diferentes comportamientos y necesidades propios y del entorno. Este trabajo presenta una técnica para enriquecer componentes con información que el mismo adquiere de sus ejecuciones. Esa información forma una base de auto-conocimiento (self-knowledge) que lo acompaña siempre, aumenta en cada ejecución y evoluciona en cada evolución del componente. El objetivo de esa base es ser utilizada en el testing y análisis del componente y de los sistemas que forma parte a fin de colaborar con sus mejoras. La repetición constante y permanente del proceso de recolección, testing y mejora sobre el componente constituye la principal contribución de la técnica, lo que esperamos que facilite la reutilización del componente, su proceso de evolución y el proceso de evolución de los sistemas.

Abstract

Increase in the demand of software, high availability required and extensive heterogeneous runtime environments and configurations have forced the scientific community to develop strategies and techniques to satisfy the new challenges presented in the systems management and development. The technology of components has been one of the first proposed solutions, but problems such as the ones caused by the lack of information, and mismatched assumptions made by reusable parts deepened their consequences in quality assurance tasks. At the beginning of the century the idea to develop systems with self-capabilities became relevant, that is, systems with the ability to detect, to understand and to adapt to either their different behaviors and needs or to those required by the environment and users. In this work we present a technique to enrich components with information collected by itself from its executions. That information forms a self-knowledge base associated to the component, that increase in each execution and evolves in each evolution of the component. The base purpose is to be utilized in testing and analysis tasks of the component and of the systems that belong to it in order to collaborate with its improvements. The permanent and constant repetition of the process of collecting, testing and improvement on the component constitutes the main contribution of the technique, what we hope will facilitate the component reuse, its process of evolution and the process of evolution of the systems.

1 INTRODUCCIÓN

En los últimos años, hemos presenciado un cambio fundamental en la forma en que el software se desarrolla y la forma en que llega a ponerse en marcha. Hace unas décadas, existían relativamente pocos sistemas de software, y estos sistemas eran a menudo hechos por encargo y corrían en un número limitado de computadoras en su mayor parte desconectadas. Después de la revolución de la computadora personal, el número de computadoras y de sistemas de software ha aumentado dramáticamente. Además, el continuo crecimiento de Internet ha incrementado significantemente la conectividad de los dispositivos informáticos y nos conduce a una situación en la que la mayor parte de estos sistemas y computadoras están interconectados.

La sociedad actual crea una carga de trabajo altamente variable e impredecible sobre sistemas conectados en red, redes que a su vez interconectan sistemas distribuidos y heterogéneos. La importancia y complejidad de estos sistemas requiere cada vez más y más habilidad de profesionales de IT para instalar, configurar, operar, ajustar y mantener este tipo de sistemas. Esto nos lleva a pensar en la necesidad de una "informática autónoma". Así como el sistema nervioso autónomo libera a la parte conciente de nuestro cerebro de la carga de lidiar con detalles vitales aunque a su vez de las funciones más básicas de nuestros sistemas, la informática autónoma debería liberar a los administradores de sistemas de la mayoría de las tareas rutinarias administrativas y operativas actuales. En la práctica se arman grupos de trabajo enormes para "sostener" asuntos que incluyen una serie de tareas, muchas de las cuales en realidad podrían ser automatizables. De ese modo las empresas podrían orientar sus capacidades de IT hacia el corazón de sus negocios, en vez de invertir cada vez más tiempo en lidiar con la complejidad de los sistemas informáticos.
Irving Wladawsky-Berger, Vicepresidente de Estrategia y Tecnología del IBM Server Group, resumió la solución en el Kennedy Consulting Summit en Noviembre del 2001: "Hay sólo una respuesta: la tecnología debe manejarse a si misma. Ahora bien, con esto no quiero decir algo lejano de proyectos de AI; lo que quiero decir es que necesitamos desarrollar el software correcto, la arquitectura correcta, los mecanismos correctos ... de modo tal que en vez de que la tecnología se comporte como lo hace habitualmente en esa forma pedante y demandante de personas que hagan todo por ella, ésta comience a comportarse más como una computadora "inteligente" como nosotros esperamos que sea, y que comience preocupándose ella misma por sus necesidades. Si no se siente bien, entonces hace algo. Si alguien la está atacando, el sistema lo reconoce y lidia con el ataque. Si necesita más poder computacional, va y lo obtiene, y no se queda buscando asuntos humanos para alcanzarlos." [GC/03].
Pero veamos, ¿la automatización de la administración de recursos informáticos es a caso un nuevo problema o una nueva necesidad para la ciencia de la computación? Definitivamente, no. Por décadas, los componentes de sistema y el software han evolucionado para lidiar con la creciente complejidad del control de sistemas, recursos compartidos y administración del correcto funcionamiento. La informática autónoma es exactamente la siguiente evolución lógica de esa tendencia del pasado dirigida a entornos informáticos actuales cada vez más complejos y más distribuidos.

Además de toparnos con el problema de la alta complejidad para administrar los sistemas actuales, vemos que cuando se ponen en marcha desarrollos de sistemas que cubren estos aspectos la aplicación de las técnicas actuales tornan a dicho desarrollo en algo muy costoso en tiempo y en dinero. Y que aún realizando las inversiones necesarias, la aplicación de dichas técnicas suelen ser más y más complejas a medida que vamos hacia sistemas de mayor escala, llegando en muchos casos de ellos hasta ser impracticables.

Por otro lado, la heterogeneidad y la velocidad de demanda de los sistemas actuales han potenciado la construcción de sistemas a partir de la unión de piezas (componentes) de diversas fuentes. Una característica importante que conlleva esta forma de desarrollo es el tema de las suposiciones (assumptions), de la intención o los fines para los que esas piezas han sido creadas en contraposición con el uso real (in-field) que luego se le da a dichas piezas [GAO/95]. Por la naturaleza misma de este tipo de sistemas ese tema es más difícil de manejar con las técnicas y metodologías de desarrollo tradicionales, e incluso muchas veces hasta imposible. Por ejemplo, consideremos tareas de aseguramiento de calidad tales como testing y análisis. Salvo poquísimas excepciones, actualmente estas actividades se realizan in-house, sobre plataformas de desarrollo, utilizando inputs proporcionados por el desarrollador, típicamente proveedor del componente del que formará parte la pieza que se está testeando. La suposición fundamental es que el software es testeado y analizado de la misma forma en que será luego utilizado realmente in-field, como verdaderamente lo utilizará el usuario del componente. Desafortunadamente, esta suposición rara vez es satisfecha. Consecuentemente, pueden malgastarse una cantidad considerable de recursos ejercitando configuraciones que no ocurren in-field y entidades de código que no serán ejercitados

por usuarios reales. Por el contrario, el testing in-house puede perder la ejercitación de configuraciones y comportamientos que sí ocurren realmente in-field, disminuyendo así nuestra confianza en el testing [OAH/03].

Parecería ser que las técnicas actuales, y la forma de encarar y desarrollar los sistemas son las que no encajan con el tipo de sistemas que esperamos. Según marca la experiencia, es una utopía pretender especificaciones 100% completas, que además de ser imposibles de relevar, requerirían un congelamiento de las necesidades de los usuarios, lo cual es totalmente irreal.

Entonces, lo que se plantea es que mientras los avances técnicos en estrechas áreas de tecnología de adaptación provean algún beneficio, los mayores beneficios vendrán desarrollando una extensa *metodología de adaptación* que abarque desde la adaptación en pequeña escala hasta la adaptación en gran escala, y a partir de allí desarrollar la tecnología que soporte el rango total de adaptaciones [OGTJMQRW/99].

Si bien estos cambios crean nuevos desafíos para el desarrollo de software, tales como acortados ciclos de desarrollo y creciente frecuencia de actualizaciones de software, ellos representan también nuevas oportunidades que, si son convenientemente explotadas, podrían proporcionar soluciones tanto a los nuevos como a los ya existentes problemas de performance y aseguramiento de calidad de software.

A partir de este contexto de la búsqueda de sistemas cada vez más autónomos y de la construcción de sistemas a partir de la unión de componentes, en el presente trabajo definimos una técnica que nos permite enriquecer componentes con información obtenida a partir de sus propias ejecuciones formando con ello su base de conocimiento. El objetivo de esa información es asistir a las tareas de ingeniería de software, en particular a las asociadas a la verificación y análisis de software en las etapas del ciclo de vida de los componentes posteriores a su primer puesta en producción, tales como su reutilización y sus actualizaciones. Para lograr ese objetivo, la técnica adopta una estrategia en particular para superar problemas como la falta de información, las suposiciones y la dependencia del proveedor del componente. La clave de la estrategia está en sacar provecho de las ejercitaciones propias y permanentes del componente cada vez que este es puesto en producción. Y las características más importantes de la estrategia son su carácter autónomo, y la noción de reuso y evolución aplicada a la base de conocimiento creada para el componente mismo.

La presentación del trabajo se desarrolla en tres partes. En la primera parte, formada por los capítulos 2, 3 y 4, presentamos una descripción general y los principales aspectos de interés, desde el punto de vista de la informática autónoma, de las aristas más importantes del presente trabajo, lo que servirá a los efectos de introducir los fundamentos necesarios al propósito de esta tesis. En particular, el capítulo 2 está dedicado a los self-healing systems, allí introducimos los términos y características asociadas a los sistemas autónomos, así como los principales problemas que buscan solucionar. El capítulo 3 está dedicado al testing y análisis, allí consideramos el rol y la perspectiva de esas tareas en el marco de sistemas autónomos y con auto-capacidades. Y el capítulo 4 está dedicado a la unión e intersección de las necesidades y las soluciones propuestas para los temas vistos en los capítulos 2 y 3. Tomando como punto de partida el desarrollo de sistemas basados en componentes, presentamos en forma general sus ventajas y los principales obstáculos que surgen al utilizar dicha metodología de desarrollo, y en particular aquellas que deben ser superadas por las tareas asociadas al testing y verificación de sistemas de software. Luego, como conclusión de ello, veremos el estado del arte de los temas expuestos mencionando las investigaciones más recientes relacionadas brindando referencias para profundizar en cada uno de ellas.
En la segunda parte, formada por los capítulos 5, 6 y 7, proponemos, definimos y describimos la técnica TFT. En primer lugar en el capítulo 5 introducimos sus principales conceptos, términos y escenarios de uso. En el capítulo 6 presentamos una implementación posible del framework de la técnica a partir de un diseño de alto nivel de sus fases principales y una descripción de los puntos estratégicos más importantes. Y en el capítulo 7 analizamos el alcance de la información obtenida con la técnica, valiéndonos de casos de usos que ayudan a mostrar la aplicación de la técnica en situaciones prácticas, y las ventajas de su uso.
En la tercera parte, formada por los capítulos 8 y 9, para cerrar con la presentación pondremos a la técnica en el contexto de las aristas presentadas al comienzo, discutiendo su relación y ubicación en el área, los trabajos relacionados, y las posibles investigaciones futuras que se abren a partir de su estudio. Finalmente en el capítulo 9 expresamos la conclusiones de la propuesta.
Los capítulos 10 y 11 contienen toda la bibliografía referenciada.

2 Self-Healing Systems

2.1 ¿A dónde apuntan los Self-Healing Systems?

Como hemos dicho, la automatización de la administración de recursos informáticos no es un nuevo problema ni una nueva necesidad para la ciencia de la computación. Por décadas, los componentes de sistema y el software han evolucionado para lidiar con la creciente complejidad del control de sistemas, recursos compartidos y administración de correcto funcionamiento. En Octubre del 2001, IBM introdujo las primeras ideas de una posible solución: desarrollar mecanismos que permitan que los sistemas sean autónomos y capaces de reconfigurarse a si mismos. De este modo surge la propuesta de la *informática autónoma* en la que los sistemas pueden administrarse a si mismos a partir de los objetivos de alto nivel de los administradores [HOR/01].
En el pasado, la adaptación de sistemas ha sido manejada en gran parte internamente a la aplicación. Por ejemplo, muchas aplicaciones utilizan típicamente mecanismos genéricos tales como manejo de excepciones o mecanismos de "heartbeat" para disparar respuestas específicas de la aplicación frente a una falla observada. Para otros aspectos de la adaptación, tal como la adaptación basada en recursos, los sistemas típicamente tejen políticas y mecanismos específicos de la aplicación internamente. Por ejemplo, una aplicación de videoconferencia puede decidir cómo reducir su fidelidad cuando el ancho de banda de transmisión es bajo utilizando alguna combinación de compresión, reducción de framesize y resolución. Mientras la adaptación interna puede ciertamente hacerse para funcionar apropiadamente, tiene varios problemas graves. Primero, cuando la adaptación está entrelazada en el código de la aplicación es difícil y costoso hacer cambios en la política y los mecanismos de adaptación. Segundo, para razones similares, es difícil reutilizar los mecanismos de adaptación de una aplicación a otra. Tercero, es difícil razonar acerca de la exactitud de un mecanismo de adaptación dado, porque uno debe considerar también toda la funcionalidad específica de la aplicación al mismo tiempo [GSC/01].
Un método alternativo es desarrollar un mecanismos de adaptación externo. En dichos sistemas, la adaptación se maneja fuera de la aplicación. Los sistemas son monitoreados para varios atributos, tales como la utilización de recursos, la confiabilidad, la calidad de servicio proporcionada, etcétera. Basados en esa información de monitoreo, los mecanismos externos deciden si la aplicación debe ser reconfigurada. En contraste con el método interno, la adaptación externa tiene muchos beneficios: los mecanismos de adaptación pueden ser extendidos más fácilmente; pueden ser estudiados y razonados independientemente de las aplicaciones monitoreadas; y permiten aprovechar la reutilización del monitoreo y de la infraestructura de adaptación.
Uno de los componentes esenciales de los mecanismos de adaptación externos en tiempo de ejecución es la habilidad de evaluar el estado del sistema en términos que le permitan a uno determinar si el sistema está operando dentro de límites aceptables. Para hacer esto efectivamente uno debe ser capaz de recolectar información acerca del sistema mientras está ejecutando e interpretar esa información en el contexto de los modelos del más alto nivel que permitan la observación de las propiedades del sistema.

2.2 ¿Qué son los Self-Healing Systems?

Existe una amplia variedad de puntos de vista acerca de la definición de *self-healing systems*, así como mucha terminología asociada: self-managing, self-configuring, self-healing, self-optimizing, self-protecting, características self-CHOP, self-adaptative, self-organizing, self-stabilizing, reflective, congnitive, homeostatic, autonomic. Tradicionalmente la comunidad científica ha dado por llamar self-healing systems a los sistemas autónomos en general, o lo que es lo mismo, a los sistemas con algún grado de autonomía. Si bien con el avance y la profundización de las investigaciones la auto-salud (self-heal) ha sido considerada como una de las tantas posibles auto-capacidades (self-capabilities) de un sistema, el término self-healing systems continúa utilizándose en su sentido más amplio englobando a los sistemas con una o más auto-capacidades. Entonces, aunque aun no hay aun una definición consensuada, la comunidad científica está de acuerdo en que intuitivamente se trata de *sistemas que son responsables de su propio comportamiento y salud* [OGTJMQRW/99] [GC/03] [GKW/02] [KOO/04].
El espectro de propuestas de auto-capacidades es amplio. Partiendo de propuestas más básicas que combinan especificaciones de la auto-capacidad con especificaciones de la aplicación, la idea es separar los asuntos concernientes a esa capacidad del software de los asuntos concernientes a la funcionalidad específica de la aplicación. Así, podemos dar un paso más en esa separación con los algoritmos on-line, y luego con algoritmos genéricos o parametrizados, y luego con selección de algoritmos, hasta llegar a la programación evolutiva. Una clara separación entre ellas facilita su análisis y evolución independiente.

La informática autónoma fue concebida como una forma de ayudar a reducir el costo y la complejidad de poseer y operar una infraestructura de IT. Entonces, en un contexto autónomo, todos los componentes de sistemas – desde el hardware como las PC´s y los mainframes, hasta el software como sistema operativo y aplicaciones de negocio – deben implementar y sustentar las características fundamentales de la autonomía y de la auto-administración (self-managing).

2.3 Elementos del espacio de problemas de los self-healing systems

En este área de los sistemas autónomos es conveniente distinguir el espacio de problemas del espacio de soluciones. Una taxonomía de los elementos del espacio de problemas de los sistemas self-healing es de suma utilidad ya que por un lado nos podría permitir entender qué elementos y en qué grado son cubiertos (self-heal) por un determinado sistema, y por otro lado permitiría que los métodos, mecanismos y técnicas puedan ser descriptos por el grupo de elementos que cubre o aspira a cubrir. Una propuesta de esta taxonomía es presentada en [KOO/03]. Las principales categorías de los aspectos del espacio de problemas de los self-healing systems son: modelo de falla, respuesta del sistema, completitud del sistema y contexto de diseño. El nombre en si de las categorías no es relevante ni estricto, lo importante es la agrupación de conceptos relacionados como elementos del espacio de problemas que deben ser atacados por los self-healing systems.

Modelo de Falla

Los self-healing systems deben tener un modelo de falla en términos de qué heridas (fallas) se espera que ellos sean capaz de curarse (self-heal). Sin un modelo de falla, no hay manera de juzgar si un sistema realmente puede curarse en las situaciones de interés. Las siguientes son las típicas características de un modelo de falla que aparecen relevantes:

- Duración de la falla
- Manifestación de la falla
- Fuente de la falla
- Granularidad
- Tipo de falla esperada

Respuesta del Sistema

El primer paso para responder a una falla es, en la mayoría de los casos, detectar realmente la falla. Más allá de eso, hay varias maneras de degradar la operación del sistema así como el intento de recuperación de la falla o la compensación de la falla. Cada dominio de aplicación tiene aspectos extra-funcionales o de calidad de servicio que son importantes, características tales como confiabilidad, safety o seguridad. Estos asuntos extra-funcionales influyen sobre las respuestas deseadas del sistema. Las características más relevantes a tener en cuenta en este grupo son:

- Detección de la falla
- Degradación
- Respuesta a la falla
- Recuperación de la falla
- Constantes de tiempo
- Garantías

Completitud del sistema

Los sistemas verdaderos rara vez están completos en todo sentido. Los métodos self-healing deben ser capaces de lidiar con la realidad de límites de conocimiento, especificaciones incompletas, y diseños incompletos. Las características más relevantes a tener en cuenta en este aspecto son:

- Completitud arquitectónica
- Conocimiento del diseñador
- Auto-conocimiento (Self-knowledge) del sistema
- Evolución del sistema

Típicamente los diseñadores tienen una total comprensión de los comportamientos típicos del sistema, pero no tienen o tiene muy poca comprensión de los comportamientos atípicos del sistema, especialmente de los comportamientos del sistema en presencia de fallas. Un aspecto vital del conocimiento del diseñador es qué tan bien el modelo de falla caracteriza al sistema y si la información de campo (field data) de las fallas retorna al diseñador del sistema.
Por otro lado, los sistemas deberían tener algún nivel de conocimiento sobre si mismos y su entorno para poder proveer auto-cura (self-heal). Este auto-conocimiento (self-knowledge) está limitado en primer lugar por los aspectos de conocimiento construidos dentro de un componente (por ejemplo, un componente puede ser capaz o no de predecir su tiempo de ejecución); en

segundo lugar por la accesibilidad de un componente al conocimiento de otro componente, y en tercer lugar por los defectos en la representación de dicho conocimiento ya sea debido a los defectos de diseño iniciales o a la caducidad del mismo ocasionada por la evolución del sistema. El concepto de reflexión a menudo es discutido en el contexto de sistemas con auto-conocimiento (systems self-knowledge); sin embargo también parece posible construir sistemas que no tengan ningún conocimiento de su estado pero que en cambio muestren la correctitud alcanzada como el resultado de la interacción de los comportamientos del componente.
Además, los self-healing systems deben lidiar, como todo sistema, con el hecho de que ellos cambian a lo largo del tiempo. Los cambios pueden estar originados por diversas razones: cambios en el modo de operación, fallas acumuladas de componentes y recursos, adaptación a entornos externos, evolución de componentes, o cambios en el uso del sistema. Utilizar la información de la dinámica del sistema podría ayudar a la auto-cura, por ejemplo ser capaz de contar con un fuera de uso planificado del sistema (o una auto-planificación de fuera de uso) para realizar las curaciones.

Contexto de diseño

Hay muchos otros factores que influyen en el alcance de las capacidades self-healing que pueden ser consideradas para armar el contexto de diseño del sistema. Muchas de ellas incluso no es posible que sean definidas por los arquitectos o diseñadores del software, sino que son consecuencia del contexto y entorno en el que el sistema debe convivir, y por ello esas capacidades puedan estar limitadas. De todos modos conocer con claridad dicho contexto ayuda a encontrar las técnicas y estrategias self-healing más apropiadas del caso y describen en cierta forma el grado de auto-cura que poseerá el sistema. Las factores más relevantes a tener en cuenta en este aspecto son:

- Nivel de abstracción
- Homogeneidad de los componentes
- Predeterminación del comportamiento
- Compromiso del usuario en la “salud” (heal) del sistema
- Linealidad del sistema
- Alcance del sistema

2.4 Características del Self-Management

La clasificación provista por IBM que reagrupa dentro del self-management las características self-CHOP (configuring, healing, optimizing y protecting), no explica claramente la relación entre estos términos, ni da lugar otras auto capacidades. El technical report de Tosi [TOS/04] es un interesante y bastante completo trabajo de investigación del estado del arte de los self-healing systems que tiene por objetivo desambiguar la terminología, identificar la principales dimensiones de la informática autónoma proveyendo una clasificación más rigurosa, y discutir tecnologías, fases y mecanismos de los sistemas autónomos, con referencias a las investigaciones más actuales. Otro aspecto interesante de ese informe en relación al trabajo que aquí se presenta, es que en cierta forma aquel está enfocado desde el área de testing, ya que relaciona varios trabajos de testing de los últimos tiempos con las distintas fases del self-management de los sistemas, lo que si bien no es el único objetivo del presente, es de suma utilidad para el área y la comunidad científica ya que en cierta forma permite algo así como trazar un mapa entre todas las investigaciones y propuestas del área.
Con respecto a la clasificación, definición y alcance de las principales características del self-management, según el análisis realizado por Tosi, de todo lo trabajado hasta el momento, las siguientes categorías son las más importantes y representativas para el área:

Self-Adaptative: esta característica denota la habilidad de un sistema para reaccionar dinámicamente en el entorno de trabajo y adaptar su comportamiento a la calidad de servicio requerida a fin de proveer y mejorar sus funcionalidades y su performance. Un sistema self-adaptative trabaja en forma proactiva. Para esto debe testear el comportamiento del código en tiempo de ejecución, a fin de modificar en tiempo de ejecución su propio comportamiento si los resultados del test son negativos, y debe incluir cosas como descripciones de las intenciones del software y un conjunto de procedimientos o algoritmos alternativos. El software self-adaptative necesita requerimientos no funcionales, o atributos de calidad, tales como confiabilidad, robustez, adaptabilidad y disponibilidad.

Self-Configuring: esta característica involucra mecanismos para modificar la interacción entre componentes, y denota la habilidad para adaptarse automáticamente a cambios dinámicos en la estructura del sistema.

Self-Healing: esta característica denota la habilidad de un sistema para examinar, encontrar, diagnosticar y reaccionar frente a funcionamientos defectuosos del sistema. Los sistemas o componentes self-healing deben ser capaces de observar fallas del sistema, evaluar restricciones impuestas por el exterior, y aplicar correcciones apropiadas. Para descubrir funcionamientos

defectuosos o posibles fallas futuras es necesario conocer el comportamiento esperado. Los sistemas autónomos deben tener conocimiento acerca de su propio comportamiento de modo que en base a dicho conocimiento puedan determinar si el comportamiento actual es consistente y es el esperado en relación a su entorno. En nuevos entornos o diferentes escenarios, nuevos comportamientos del sistema pueden ser observados y el módulo de conocimiento debe evolucionar con el entorno.

Self-Optimizing: esta característica denota la habilidad del sistema para monitorear y optimizar en forma autónoma los recursos del sistema, alocando recursos de forma apropiada a fin de proveer y garantizar un nivel de calidad aceptable.

Self-Organizing: en un sistema self-organizing, los componentes son capaces de configurar autónomamente la interacción entre ellos, garantizando las restricciones impuestas por la arquitectura de diseño del sistema. Así, la introducción de nuevos componentes no compromete las propiedades de la arquitectura ya que deben ser satisfechas las restricciones de composición de dichos componentes.

Self-Protecting: esta característica tiene como foco la seguridad del sistema. Un sistema de este tipo es capaz de observar el mundo exterior y notificar posibles ataques tales como accesos no autorizados, infecciones de virus, ataques de spam o negación de servicio, y por consiguiente tomar acciones para hacer al sistema más seguro (safe), más protegido y no vulnerable.

Self-Management: esta característica se refiere a la capacidad de un sistema digital para cambiar y adaptar dinámica y automáticamente sus comportamientos y características, a fin de mejorar su funcionalidad y confiabilidad.

Los sistemas con característica self-management han sido investigados en muchos dominios de aplicación para manejar diferentes problemas, y ellos reagrupan todos los atributos anteriores a fin de proveer un sistema siempre eficiente sin la intervención humana. Las diferentes auto-capacidades identifican mecanismos complementarios que pueden coexistir en el mismo sistema para manejar distintos problemas. Por ejemplo, un sistema de software podría manejar los requerimientos de performance con mecanismos self-adaptative, propiedades arquitectónicas con una arquitectura self-configuring, conectividad inestable con un administrador de consistencia self-healing, etcétera. Esto muestra que los atributos self-organizing, self-adaptative, self-configuring, self-protecting, self-optimizing y self-healing están fuertemente relacionados ya que cada concepto provee un aspecto de habilidad de la reconfiguración del sistema. Por ejemplo, si se necesita actualizar un componente, el sistema puede instalarlo, reconfigurarse y comenzar autónomamente un test de regresión para verificar el correcto comportamiento del sistema; si descubre un problema, el sistema debe ser capaz de comenzar con una estrategia de solución del problema o volver atrás a una configuración consistente reinstalando la vieja versión correcta.

2.5 Fases del Self-Management

Analizando los trabajos de investigación, los métodos y técnicas desarrollados, y las clasificaciones antes mencionadas se ve que los mecanismos de self-management explotan diferentes métodos ejecutando un conjunto de pasos en común, dependiendo de los objetivos y del dominio de la aplicación. En un sistema self-managed, un ciclo de reacción completo se compone de los siguientes pasos: Monitoreo, Interpretación, Diagnóstico, Adaptación y Aprendizaje, interactuando como lo indica la figura 1.

Monitoreo: ejecuta reglas definidas en una base de conocimiento (que da soporte a la toma de decisiones en otras fases) y chequea si existe alguna inconsistencia entre el conocimiento y el comportamiento actual del sistema. Provee análisis estadístico relacionado al desempeño del sistema tales como uso de CPU, uso de memoria de los procesos en ejecución, latencia de la red, etcétera. Los datos dinámicos deben ser comparados con los datos estándar para determinar si el comportamiento actual del sistema es consistente o no con el comportamiento normal del sistema. Este módulo debe capturar las excepciones levantadas por los módulos del sistema y además debe proveer análisis relacionado al entorno donde el sistema está ejecutando. Los mecanismos de monitoreo pueden observar tanto el comportamiento del sistema (monitoreo interno) como el comportamiento del entorno operativo (monitoreo externo).

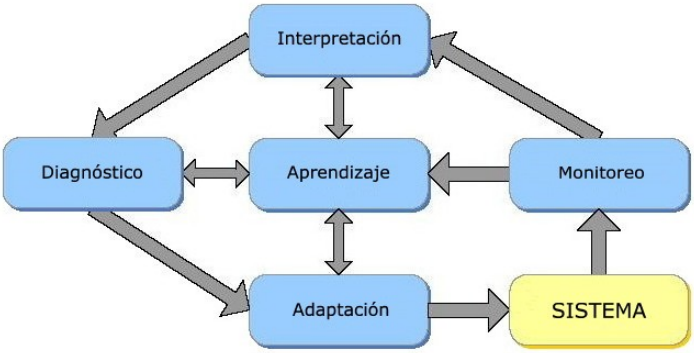

Figura 1:Ciclo de reacción en sistemas self-managed ([TOS/04])

Interpretación: analiza los datos recolectados por el monitor y verifica si existe conocimiento relacionado al problema reportado consultando el módulo de base de conocimiento. Si lo encuentra el módulo de detección trata de conseguir un registro con la solución del problema; si el módulo de base de conocimiento no tienen información sobre un problema como este, el módulo de detección actualiza el conocimiento agregando el problema reportado a través del módulo de aprendizaje.

Diagnóstico: trata de encontrar las causas del problema y verifica si las soluciones que han sido aplicadas previamente pueden resolver el problema.

Adaptación: es el módulo de resolución del problema que trata de ejecutar un ciclo de resolución de problema comenzando a partir del registro con la solución identificado por el módulo de detección. Este módulo requiere mecanismos dinámicos para planificar, poner en marcha y decretar cambios, a fin de sacar o bien las fallas diagnosticadas o bien sus efectos.

Aprendizaje: crea y actualiza la base de conocimiento adquiriendo nuevo conocimiento aprendido a través de los datos recolectados por la actividad de monitoreo.

Si bien las propuestas más completas implementan las cinco fases, algunas propuestas implementan sólo un subconjunto de ellas, incluso algunas sólo una fase. Por ejemplo el caso de los ciclos de reacción que implementan sólo la fase de adaptación como un mecanismo de prevención, sin tener capacidades para reaccionar frente a comportamientos erróneos. Otro ejemplo simple de self-management preventivo es el de los garbage collectors: administran el desalojo de memoria no utilizada por mucho tiempo por un programa, anticipándose y previniendo así fallas de overflow de memoria.

2.6 Dimensiones del Self-Management

Considerando la clasificación y las fases del self-management presentadas en las secciones anteriores se pueden encontrar en las características similitudes y diferencias, algunas incluso parecen estar sumamente relacionadas en tanto que otras parecen ocuparse de aspectos completamente diferentes, digamos ortogonales. Si bien se podría lograr una clasificación mucho más detallada y rigurosa de ello, resulta de mayor interés bosquejar una relación entre todas ellas. Tosi propone tres principales dimensiones para todas estas características: *Reparación*, *Monitoreo* y *Requerimientos*. Reconocer estas principales dimensiones es de suma utilidad para entender diferencias y similitudes entre las distintas áreas de investigación de la informática autónoma, permitiendo así clasificar y ubicar la literatura existente y futura. La relación entre estas dimensiones y las características del self-management, su ortogonalidad, y su relación con las clasificaciones más detalladas vistas anteriormente están representadas con mucha claridad en la figura 2.
La dimensión *Reparación* se refiere a los mecanismos de reparación y pone el foco en aquellos necesarios para resolver errores o problemas internos del sistema. Esta dimensión puede dividirse en: sistemas que reconfiguran la arquitectura y sistemas que reconfiguran los componentes.
La dimensión *Monitoreo* se refiere a los mecanismos de monitoreo distinguiendo entre sistemas que reaccionan frente a cambios en el entorno y sistemas que reaccionan frente a cambios dentro del sistema. Por cambios en el entorno, queremos decir cualquier cosa observable por el sistema de software, tales como inputs de usuarios o dispositivos externos o probes. En este caso, el sistema debe ser capaz de adaptarse automáticamente al cambio del entorno para proporcionar suficiente funcionalidad, mejorar el tiempo de respuesta y la performance del sistema, o incorporar comportamiento adicional en tiempo de ejecución. Aquí el punto principal está en distinguir apropiadamente cuáles son los eventos y los cambios internos o externos que necesitan ser monitoreados y cómo monitorear esos cambios.
La dimensión *Requerimientos* es otra perspectiva en la que podemos partir a los sistemas autónomos. Por ejemplo, un sistema que monitorea cambios en el ambiente puede atender los

requisitos de performance para proporcionar una calidad de servicio adecuada o un sistema que reconfigura la arquitectura puede atender los requisitos de seguridad para prevenirse de ataques externos. Así, esta dimensión es ortogonal a las otras dimensiones.

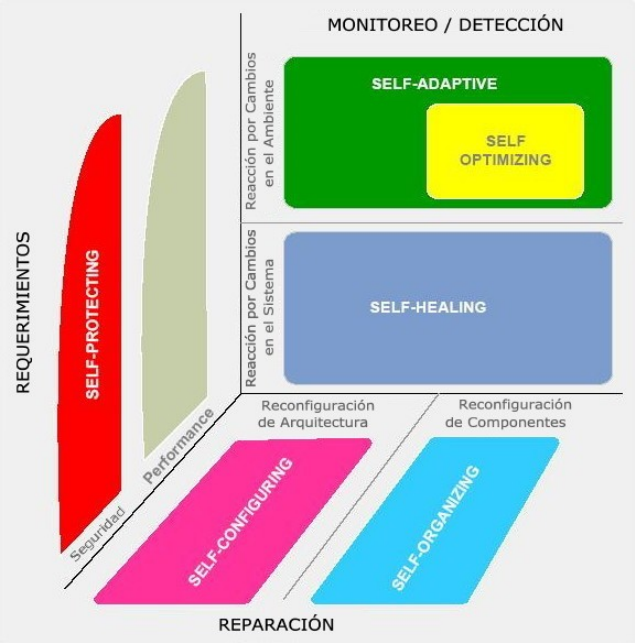

Figura 2:Clasificación detallada de Self-Management ([TOS/04])

Es obvio que hay también una relación entre las dimensiones de reparación y monitoreo ya que un sistema que pone el foco en actividades de monitoreo, proporciona también tácticas de reparación o estrategias de resolución. Por otro lado, un sistema que pone el foco en mecanismos de reparación, debe proporcionar a su vez metodologías para monitorear el entorno y detectar errores o problemas del sistema que provoquen la necesidad de una reconfiguración.

3 TESTING Y ANÁLISIS

Parecería ser que las técnicas actuales, y la forma de encarar y desarrollar los sistemas son las que no encajan con el tipo de sistemas que esperamos. Según marca la experiencia, es una utopía pretender especificaciones 100% completas, que además de ser imposibles de relevar, requerirían un congelamiento de las necesidades de los usuarios, lo cual es totalmente irrealista [SHA/02] [OGTJMQRW/99]. Desde cierta óptica, las etapas de testing y análisis parecen ser las más cercanas a la "salud" de un sistema. La mayoría de las tareas de testing apuntan a comprobar que un sistema al ejecutarse se comporte según "lo esperado", es decir que goce de "buena salud" desde su puesta en marcha y a lo largo de todo su funcionamiento. Las técnicas de testing más actuales, han reconocido muy atinadamente, que estas verificaciones deben mantenerse y garantizarse mucho más allá de la puesta en producción, debido a la heterogeneidad de los sistemas, los entornos dinámicos y cambiantes, y a la imposibilidad de considerar de antemano todos los escenarios posibles. Es así como en los últimos tiempos muchas de las técnicas de testing y análisis se han volcado hacia propuestas que proporcionan auto-capacidades en los sistemas y/o sus partes que permiten obtener, procesar o ejecutar tareas de análisis y testing ya sea luego de su puesta en marcha o al menos fuera de sus entornos de desarrollo. Este tipo de propuestas permite desarrollar sistemas más autónomos y con características propias de los self-healing systems.

El desarrollo de nuevos sistemas heterogéneos, modulares y configurables a menudo se basa en modificar un subconjunto de módulos de un sistema de la misma familia ya en uso. En la configuración general, algunos componentes son eliminados, otros son modificados, así como otros son agregados. Muy frecuentemente componentes ya existentes son reutilizados, directamente o con pequeñas modificaciones.
La integración de nuevos componentes puede resultar en fallas debido a diferencias sutiles en las interacciones provocadas por los nuevos componentes. Por ejemplo, consideremos un sistema en el que un componente es responsable de seleccionar dinámicamente el mejor catálogo para localizar ciertos artículos. Además, consideremos que este componente sustituye y actualiza un componente anterior que, en otra configuración del mismo sistema, era capaz de localizar artículos en un catálogo fijo. En este caso, el nuevo sistema puede fallar cuando los catálogos no pueden localizarse dinámicamente o cuando ellos lleguen a estar no disponibles bajo ciertas condiciones de ejecución, que nunca hubiera sido el caso en la configuración anterior. El nuevo sistema puede comportarse correctamente en muchos casos y fallar sólo en situaciones particulares.
Las cuestiones asociadas a testear y analizar software basado en componentes han sido estudiadas en los últimos años [RI/98] [PEZ/03]. Un enfoque muy interesante es el *testing y análisis perpetuo* de Pavlopoulou y Young [PY/99], por su novedosa visión del testing y el análisis no como una única etapa del proceso de desarrollo del software sino una actividad conjunta y paralela de la evolución misma del software. La describiremos con más detalle en la próxima sección. La propuesta es válida y valiosa, pero requiere de la suma e integración de muchas técnicas que se complementen entre sí, y hasta ahora las soluciones propuestas no manejan todos los aspectos de esas cuestiones. Así por ejemplo las técnicas de design-for-testability propuestas por Binder y extendida por los componentes de self-test propuestos por Martins et al., asumen que los componentes están convenientemente instrumentados con facilidades built-in para el testing [BIN/94] [MTY/01]. Los retrocomponents propuestos por Liu y Richardson requieren trabajo considerable de los diseñadores de test y por ende implica costos elevados [LR/98]. Los wrappers BIT propuestos por Edwards requieren detallado conocimiento del diseño del componente y de ese modo no siempre son aplicables a productos COTS (Commercial Off-The-Shelf) de terceras partes [EDW/01]. La técnica BCT (Behavoir Capture and Test) propuesta por Mariani et al. para testear nuevas configuraciones para sistemas basados en componentes toma las ideas subyacentes del testing perpetuo de Pavlopoulou y Young [PY/99] y aprovecha la técnica para descubrimiento de invariantes de Ernst et al. [ECGN/01] para realizar un chequeo de la equivalencia de comportamiento de los componentes [MP/04]. Esta técnica no requiere conocimiento específico de la estructura interna de los componentes ni una comprensión detallada de las especificaciones, superando de ese modo las principales limitaciones de uso práctico de trabajos anteriores, pero se limita sólo al análisis y verificación de equivalencia de comportamiento del componente. En el método Gamma propuesto por Orso et al., los datos de campo (field data) son recolectados, por un lado, para predecir el impacto de actualizaciones y, por otro, para conducir el test de regresión (selección de test de regresión, priorización de la suite de test, y aumento de la suite de test), poniendo el foco en los efectos colaterales de las actualizaciones [OAH/03].

Algunos de estos métodos se basan en la recolección y uso de información recolectada en tiempo de ejecución del sistema. La idea subyacente es beneficiar el proceso de desarrollo de software evolutivo con datos de experiencias propias y con usuarios reales del sistema y sus componentes. Esto contribuye por un lado al desarrollo de sistemas con ciertas características de auto-

conocimiento (self-knowledge), y por otro cubre tácticas y estrategias para las fases de aprendizaje y base de conocimiento del self-management.

3.1 Testing y Análisis Perpetuo: nuevo enfoque del testing

El análisis, la evaluación y el descubrimiento de errores en los sistemas son tareas típicas y principalmente asignadas a las etapas de testing del software, y en general, a la validación y verificación del software a lo largo de todas las etapas del proceso de desarrollo. El propósito de una evaluación es determinar el grado en el cual un producto de software alcanza sus requerimientos, y sugiere en lo posible, la clase de modificaciones que se esperaría que ayuden a mejorar sus habilidades para lograr esos requerimientos. El interés está en la mejora del software. De ese modo, un grupo de investigadores proponen un método de auto-evaluación (self-evaluation) llamado *análisis y testing perpetuo* (*perpetual testing and analysis*), lo que encaja perfectamente en la amplia actividad de auto-mejora (self-improvement) [OC/00].
En un proceso de mejora de software manual, o manejado por personas, las personas están a cargo del testing y la evaluación del software, las personas infieren los cambios que deben realizarse, las personas realizan los cambios, y las personas reinstalan el software modificado, punto en el cual el ciclo de mejora del software vuelve a empezar. La auto-mejora del software sugiere que algunas o todas estas tareas sean asistidas, o llevadas a cabo completamente, en forma autónoma por el software mismo, en vez de que sean realizadas únicamente por personas. Se necesita interesante y considerable investigación para reducir la dependencia de las personas en la mayoría de estas actividades. Así la base de la propuesta es una transición de la mayoría de la responsabilidad de la evaluación y el testing del software desde las personas hacia herramientas y procesos automatizados.
Mientras que el paradigma dominante actual considera al testing como una fase posterior al desarrollo y anterior a la entrega, la propuesta aquí es considerar el análisis y el testing como una actividad on-going para asegurar la calidad de mejora sin pausa a través de varias generaciones del producto, tanto en el entorno de desarrollo como en el entorno de puesta en producción.
Si bien el testing perpetuo es un área muy nueva de investigación, tiene dos características fundamentales. El testing perpetuo debe ser continuo e incremental [OST/96]. La esencia de la idea es que el software instalado debería estar continuamente bajo testing, análisis y evaluación, trabajado duramente sobre los recursos computacionales y los patrones de utilización reales encontrados en el entorno de puesta en marcha. El proceso de testing y análisis perpetuo debe ser guiado por resultados y conclusiones en forma incremental a fin de producir resultados cada vez más precisos y definitivos. Esos resultados inevitablemente conducirán a sugerencias bien informadas en forma creciente acerca de qué modificaciones del software parecerían muy probables de llevarlo hacia verdaderas mejoras. De ese modo, el testing y análisis perpetuo apunta no sólo a ayudar a reducir la necesidad de participación humana en la evaluación, sino también a asistir el esfuerzo humano en la propuesta de mejoras.

Método

La propuesta consiste en ver a un producto de software como un conjunto de diversos tipos de artefactos de software, incluyendo tipos de componentes como especificación de requerimientos, arquitectura, especificaciones de diseño de bajo nivel, código fuente, casos de test, resultados de análisis, etcétera. Se sugiere también que el producto de software incluya como un componente al proceso por el cual varios componentes del producto de software (en particular el código fuente) es modificado. De ese modo se puede asegurar que ningún componente del producto de software necesita modificarse a si mismo, en cambio un componente aparte, el proceso de modificación, es responsable de modificar otros componentes, tales como el código el fuente.
Además se propone que el producto de software esté caracterizado también por una variedad de restricciones que especifiquen la forma en que los componentes deben relacionarse unos con otros. Es necesario también que herramientas como compiladores, herramientas de diseño, monitores de casos de test, y analizadores estáticos sean considerados como parte del producto de software. Es decir, todos los dispositivos utilizados para la construcción de los componentes del producto de software deben ser considerados como parte del producto mismo.
Y finalmente, en la puesta en marcha del producto de software todos sus componentes deben ser puestos en el entorno de producción. De otro modo la modificación del código se complica sustancialmente.
Entonces el código del producto de software instalado queda ligado a todos los otros componentes del producto (incluyendo su proceso de modificación), así como las restricciones, y el conjunto de herramientas que comprenden el producto completo. De ese modo, es posible para las herramientas continuar evaluando la consistencia del código con otros artefactos y realizar las modificaciones.

Así, el método de testing perpetuo implica que el código de software es perpetuamente enriquecido por medio del acceso a un entorno que soporta su propia evaluación y mejora, y toma un rol

proactivo para asegurar que la evaluación continúa para hacer contribuciones positivas a la mejora del software. La coordinación de los numerosos y diversos tipos de artefactos de testing y análisis que existen en el entorno de desarrollo y producción es un trabajo intimidante, terriblemente costoso y propenso a errores si se lleva a cabo manualmente. Un proceso de testing y análisis altamente automatizado integra herramientas y artefactos de testing y análisis, utilizando los recursos informáticos disponibles que puede conseguir en cada momento dado. Si bien, este proceso se considera como una parte esencial del producto, no es necesario que resida con el código en el entorno de producción.

Desafíos técnicos y de investigación

Hay una gran cantidad de desafíos técnicos para llevar a cabo este tipo de método. Lo que se propone es que el testing dinámico y el análisis estático avancen en esencia interminablemente, pero que los resultados de cada una de estas actividades sean utilizados para enfocar y afinar la actividad del otro. Así, es necesaria importante investigación acerca de cuál es la mejor manera de lograr que estas dos evaluaciones complementarias trabajen en forma muy sinérgica. Se puede imaginar orquestar esta sinergia a través del uso de procesos definidos con mucha precisión, y hay mucha investigación que debe hacerse en esta área también.
Algunos de los desafíos técnicos y ramas de investigación más importantes para lograr poner en práctica este enfoque del testing y análisis perpetuo son:

- Consideraciones de instalación y puesta en producción
- Investigaciones de testing y análisis
  - Re-testing y re-análisis incrementales
  - Integración de técnicas
  - Análisis basado en especificaciones
  - Testing y análisis guiado por predicción (predictive-driven)
  - Desarrollo de un proceso de testing perpetuo

Resumen

La propuesta del testing y análisis perpetuo promete permitir la auto-evaluación continua del software a través de la sucesión de modificaciones ocurridas a lo largo de su ciclo de vida completo, y por lo tanto permitir que la auto-mejora del software pueda ser medida y evaluada. La clave para lograr esto está en ligar en forma perpetua el código del software puesto en producción con el resto de todos los componentes del producto de software, restricciones, herramientas, y el proceso de testing y análisis perpetuo mismo. Esto sin duda debería incrementar la confianza que la gente tendrá en sus productos de software. A medida que los productos de software sean cada vez más grandes y más complejos, será cada vez más difícil evaluar y mejorar, confiar y predecir a menos que un método como el testing perpetuo sea explorado.

3.2 Recolección de información in-field

Investigaciones recientes sugieren que el desarrollo de software puede beneficiarse considerablemente aumentando las tareas de análisis y medición realizadas in-house a partir de tareas de análisis y medición realizadas in-field en el software puesto en producción. Los métodos y técnicas basados en el uso de información recolectada in-field tienen dos principales ventajas: (1) el análisis depende de datos reales de campo (field data), antes que de datos ficticios, y (2) el análisis se beneficia de la amplia y variada cantidad de recursos de una comunidad de usuarios, no apenas del limitado, y a menudo homogéneo, número de recursos encontrados en un típico ambiente de desarrollo. Hay muchos escenarios en los que los métodos que utilizan datos de campo pueden ser explotados, y numerosas tareas que pueden beneficiarse de utilizar este tipo método. Cierto grupo investiga la construcción de modelos de comportamiento parciales de un sistema con información recolectada en verdaderas ejecuciones del sistema para luego realizar verificaciones de equivalencias de comportamiento, principalmente en la etapa de test de integración, en actualizaciones o sustituciones de componentes [PBDLRR/04]. Otro grupo ha definido e investigado el uso del método Gamma [OLHL/02] para recolectar información de cubrimiento de instancias de software instalado para utilizarlo en la creación de perfiles de usuario, determinar clases de usuarios de software, y calcular los costos e identificar los asuntos asociados a recolectar datos de campo [AH/02] [BOH/02]. Se ha investigado también el uso del método Gamma para la visualización de los datos de ejecuciones de programa recolectados in-field para investigar aspectos de comportamiento de software después de la puesta en producción [OJH/03]. Trabajos más recientes investigan el uso del método Gamma para dar soporte y mejorar dos tareas fundamentales para el mantenimiento de software evolutivo, típicamente realizada por los ingenieros de software: análisis de impacto y testing de regresión. Hasta donde se puede saber, todas las técnicas existentes para realizar estas tareas del proceso de desarrollo de software mencionadas anteriormente (por ejemplo, [AB/93] [BA/96] [LR/03] [RH/97] [RT/01]) dependen sólo

de datos in-house, por lo que el uso de datos de campo para estas tareas es novedoso. Los estudios realizados experimentando con algunas aplicaciones muestran por un lado que los datos de campo pueden mejorar eficientemente la calidad de los análisis dinámicos considerados, y por el otro, que las técnicas tradicionales computarían resultados que no reflejan el verdadero y real uso del sistema[OAH/03].

4 SELF-HEALING-SYSTEMS Y TESTING

4.1 Desarrollo de Sistemas Basados en Componentes. Ventajas y desventajas

La velocidad de demanda de desarrollo, los avances tecnológicos, la complejidad, la heterogeneidad, la movilidad y todas las características de los sistemas modernos han potenciado la construcción de sistemas a partir de la unión de componentes de diversas fuentes, típicamente llamados *sistemas basados en componentes* (*component-based systems*), o más brevemente CBS ´s. Un método popular para manejar este tipo de construcciones y desarrollos recae en el uso y reuso de componentes y metodologías de diseño basadas en componentes. Los componentes utilizados en el desarrollo de un CBS pueden provenir de distintos orígenes; básicamente cada componente puede: estar siendo desarrollado ad-hoc para el sistema en cuestión, ser parte de otro sistema, actualmente en producción o no, o ser un módulo desarrollado y comercializado por terceras partes. En particular estos últimos son los llamados componentes *COTS* (*Commercial Off-The Shelf*).
La tecnología de componentes se utiliza cada vez más para desarrollar complejos sistemas de hardware y software, por medio del aumento de la *composición*, del *reuso*, la *modularidad* y la *configurabilidad* [MAR/03]. Los sistemas basados en componentes se desarrollan uniendo componentes existentes, así el foco pasa de la implementación de módulos a la *composición* de unidades. La tecnología basada en componentes impulsa el *reuso* porque de ese modo la etapa de diseño se ocupa principalmente de vincular componentes existentes, y la etapa de implementación se limita al código de "pegamento". Los sistemas basados en componentes son *modulares* porque es posible sustituir, agregar o quitar componentes después de la puesta en marcha del sistema para obtener diferentes *configuraciones*.
Las tareas llevadas a cabo durante el desarrollo y el uso de un componente, y particularmente las de aseguramiento de calidad, incluyendo el testing, pueden ser consideradas según [HAR/00] desde dos perspectivas bien distinguidas. Estas perspectivas son *proveedor del componente* y *usuario del componente*. El proveedor del componente corresponde al papel del desarrollador del componente y el usuario del componente corresponde al cliente del proveedor del componente, eso es el desarrollador de un sistema que utiliza el componente.
En los últimos años, las tecnologías de software basado en componentes han sido consideradas cada vez más necesarias para crear, testear y mantener el software ampliamente más complejo del futuro. Los componentes tienen el potencial de reducir el esfuerzo de desarrollo, la velocidad del proceso de desarrollo, beneficiar otros esfuerzos de desarrollo, y reducir los costos de mantenimiento. Desafortunadamente, más allá de su importante potencial, los componentes de software deben aun mostrar su gran promesa como solución para la ingeniería de software. Sin lugar a dudas, proporcionan muchos beneficios, pero también introducen nuevos problemas, y de hecho incluso han provocado que ciertos problemas sean aun más difíciles de manejar. La presencia de componentes desarrollados externamente dentro de un sistema introduce nuevos desafíos para las actividades de ingeniería de software que las técnicas y metodologías de desarrollo tradicionales no logran superar a menos que sean reconsideradas. Los investigadores han reportado muchos de los problemas en el uso de componentes de software, incluyendo aspectos como dificultad en localizar el código responsable de determinados comportamientos del programa [COR/98], dependencias ocultas entre componentes [COR/98] [CD/99], interfaces ocultas que agravan las cuestiones de seguridad [LJ/98] [VOA/98], reducción de las posibilidades de testeo [WEY/98], y dificultades en la comprensión del programa [COR/98]. El uso de componentes en entornos distribuidos hace que todos los problemas anteriores sean aun más difíciles, debido a la naturaleza de los sistemas distribuidos. De hecho, los sistemas distribuidos por un lado generalmente utilizan un middleware, lo cual complica la interacción entre componentes, y por el otro involucran componentes que tienen una mayor complejidad inherente (por ejemplo, componentes en aplicaciones de e-commerce que encapsulan lógicas de negocio complejas y no son sólo simples botones GUI).

Una de las principales causas de los problemas del uso y reuso de componentes es la falta de información sobre los componentes que no han sido desarrollados internamente. El desarrollador de la aplicación (CBS) típicamente tiene sólo información de la interfase principal que soporta las llamadas a las funciones del componente. En particular, no tiene el código fuente, no tiene ninguna información de la confiabilidad ni de la seguridad, no tiene ninguna información relacionada a la validación, no tiene ninguna información acerca de las dependencias que podrían ayudar a evaluar el impacto de cambios, y posiblemente ni siquiera una descripción completa de la interface y aspectos del comportamiento del componente. Cuando lo que debe llevarse a cabo es la tarea de integración del componente, la información sobre la interface y las propiedades customizables del

componente pueden ser que sea todo lo que se necesite. Sin embargo, otras tareas de ingeniería del software requieren información adicional para poder ser realizadas en un sistema basado en componentes.

Otra de las principales causas de los problemas es que las partes seleccionadas no “encajan” bien juntas. En muchos casos estas incompatibilidades son causadas por problemas de bajo nivel de interoperabilidad. En otros casos, desde un nivel más abstracto, se plantea una clase de problema conocida como *inconsistencia arquitectónica* (*architectural mismatch*) [GAO/95]. Esta proviene de suposiciones inconsistentes que las partes reutilizables hacen sobre la estructura del sistema del que ellas forman parte. Estas suposiciones a menudo entran en conflicto con las suposiciones de otras partes y casi siempre son implícitas, y no explícitas, haciendo que sea extremadamente difícil analizarlas antes de construir el sistema. Otra causa importante de estas inconsistencias es que los componentes son diseñados e implementados a partir de ciertas suposiciones sobre su contexto y el fin para el que serán utilizados [UY/00] [MAR/03]. Muchas veces eso no coincide con el uso que los clientes le dan a esos componentes. El punto es que el contexto y el uso que se les dará, es decir, las suposiciones sobre el entorno de los componentes son sumamente importantes para la construcción de este tipo de sistemas.

4.2 Nuevos desafíos para el testing de CBS´s y COTS

Además de las cuestiones mencionadas en la sección anterior sobre el desarrollo de sistemas basados en componentes, surge también el problema del testing de este tipo de componentes y sistemas a lo largo de todo el proceso de desarrollo. El desarrollo independiente de componentes reutilizables y la adopción de metodologías basadas en componentes introducen nuevos desafíos de verificación que derivan de la ausencia de conocimiento acerca del sistema final mientras se están desarrollado los componentes, y acerca de las cuestiones internas de los componentes mientras se desarrolla la aplicación final [WEY/98] [BG/03]. Es importante distinguir los problemas del proveedor del componente de los problemas del usuario del componente. Los problemas y necesidades de cada uno de ellos son distintos, así las técnicas y soluciones puede que sean soluciones para uno pero no para el otro. Ciertamente, las técnicas aplicadas por cada uno de ellos puede influir en los problemas y posibles soluciones del otro. En particular, en este trabajo pondremos el foco en los problemas del usuario del componente, sus situaciones y alternativas para superarlos. Desde ese punto de vista algunos de los principales aspectos y problemas planteados por los investigadores del área son:

- garantía de la calidad del sistema final más allá de la calidad del componente,
- falta de código fuente, documentación e información en general del componente,
- propósito para el que ha sido desarrollado el componente por parte del proveedor versus el uso real dado por el usuario,
- entorno de desarrollo del proveedor versus entorno real del usuario,
- formato de la información.

Los clientes requieren sistemas de alta calidad, lo que típicamente implica una gran cantidad de tiempo para ser desarrollado y entregas en versiones claramente pautadas, fijas y estables. La tecnología de componentes puede reducir el time-to-market aumentando el reuso de componentes de terceras partes, pero la calidad del sistema dependerá fuertemente de la calidad del código desarrollado externamente y de la arquitectura del sistema. Componentes de alta calidad pueden ser obtenidos utilizando un proceso de desarrollo en particular, asegurando una documentación completa, ejecutando tests y analizando el código final. Los componentes de alta calidad son necesarios, pero no suficientes para asegurar la calidad de los sistemas derivados. La calidad de un sistema no está relacionada directamente a la cantidad de componentes compartidos, y la calidad de los componentes individuales no está relacionada directamente con la calidad del sistema basado en componentes. Aun un sistema obtenido a partir de un sistema ya existente reemplazando sólo unos pocos componentes con nuevas versiones de los mismos componentes, puede presentar nuevos problemas no revelados en el primer sistema, independientemente de la calidad de los nuevos componentes. Por lo tanto, necesitamos técnicas específicas para testear y analizar el sistema final, más allá de la calidad de los componentes utilizados. Los enfoques clásicos de testing y análisis no son aplicables directamente al software de componentes debido a las muchas diferencias que existen entre los sistemas clásicos y los sistemas basados en componentes.

Según la mayoría de los autores, la principal razón por la cual el testing de componentes es un problema y necesita ser considerado, es el limitado intercambio de información entre proveedor y usuario del componente. El desarrollo de sistemas basados en componentes generalmente requiere documentación detallada de los componentes a ensamblar. La información disponible de cada uno de ellos puede ser insuficiente para el desarrollo de este tipo de sistemas. Incluso aunque se

dispusiera de componentes con variados tipos de documentación, ésta podría ser incompatible o incompleta sintáctica o semánticamente. Esta falta de información puede deberse a diversas razones, y puede dificultar el testing tanto a proveedor como a usuario.
Las técnicas de testing de software tradicionales generalmente asumen que los testeadores tienen completo conocimiento de los requerimientos del software, de su implementación (código fuente), y del contexto de ejecución (entorno). Sin embargo, en el mundo del desarrollo de software basado en componentes COTS esto no es así. En primer lugar, los usuarios de componentes COTS no necesariamente tienen acceso al código fuente de estos componentes. En segundo lugar, el usuario debe testearlo completamente, ya que debería asegurarse que el uso del componente es consistente con el uso para el que fue creado. Y en tercer lugar, los productores de los componentes COTS no tienen un conocimiento completo y acabado del contexto de ejecución final del componente que están escribiendo.
Así, para sobreponerse a ello, los tipos de soluciones propuestas hasta ahora por la comunidad científica son: proveer componentes con una especificación asociada [BIN/94], o desarrollar juntos el componente y su suite de test [LR/98], o desarrollar componentes con facilidades de testing [EDW/01]. Estos tipos de métodos pueden clasificarse en dos categorías [BG/03]: 1) los enfoques que apuntan a evitar la falta de información, es decir, manejan la causa de la falta de modo tal que las dificultades para testear los componentes no aparezcan [BIN/94] [LR/98] [OHR/00]; 2) los enfoques que apuntan a combatir los problemas causados por la falta de información; es decir no manejan la causa sino que atacan las dificultades potenciales que quizás se encuentren al testear los componentes [EDW/01] [BGST/03] [MTY/01]. Una desventaja importante que se repite en estos métodos es que todos estos trabajos funcionan sobre la base de hipótesis específicas de los componentes y su integración, y así inevitablemente restringen el uso de componentes y pueden fallar cuando las hipótesis de verificación son violadas. Muchos de ellos incluso están diseñados para ciertas tecnologías en particular, perdiendo así generalidad de aplicación de la técnica o método.

Las diferencias que pueden existir entre los fines e intenciones para los cuales un componente es desarrollado y los fines e intenciones para los cuales es utilizado suelen ser considerables en este tipo de sistemas. Lo mismo sucede con las diferencias de contexto y recursos disponibles en entornos de desarrollo, típicamente finitos, comparados con los entornos de usos reales, típicamente impredecibles, e incalculables en su totalidad anticipadamente. Estas diferencias plantean a lo largo de todo el proceso de desarrollo de los CBS´s escenarios y situaciones muy distintas comparado con el desarrollo de sistemas tradicionales. Salvo poquísimas excepciones, las tareas de aseguramiento de calidad tales como testing y análisis, suelen realizarse in-house, sobre plataformas de desarrollo, utilizando inputs proporcionados por el desarrollador, proveedor del componente. La suposición fundamental es que el software es testeado y analizado de la misma forma en que será luego utilizado realmente in-field, como verdaderamente lo utilizará el usuario del componente. Desafortunadamente, esta suposición rara vez es satisfecha. Así, las nuevas técnicas deberían evitar malgastar recursos ejercitando configuraciones que no ocurren in-field y entidades de código que no serán ejercitados por usuarios reales. A su vez, sería interesante que esas mismas técnicas ayudarán al testing in-house a no perder ejercitar configuraciones y comportamientos que sí ocurren realmente in-field.

Asociado al problema de la falta de información intercambiada entre proveedor y usuario del componente, hay un tema que requiere investigación en particular. Es el tema del formato de la información. Las técnicas utilizadas para sobreponerse a dicha falta de información obtienen o generan esa información faltante, ya sea estática o dinámicamente, ya sea antes o luego del uso del componente. Pero en cualquier caso la información obtenida tiene un formato específico, y lo deseable es que ese formato sea lo suficientemente genérico como para poder ser utilizado por las diversas técnicas que podrían aplicarse en el desarrollo de los diferentes CBS´s que incorporarán al componente. La idea de proporcionar datos adicionales - en forma de *metadato* o *metamétodos* que devuelven el metadato - junto con un componente no es nueva. Las propiedades asociadas con un componente JavaBean [JAV/00] es una forma de metadato utilizada para parametrizar el componente dentro de una aplicación. El objeto BeanInfo asociado a un componente JavaBean encapsula tipos adicionales de metadato acerca del componente, incluyendo el nombre del componente, una descripción textual de su funcionalidad, descripciones textuales de sus propiedades, etcétera. Análogamente, la interfase IUnknown para un componente DCOM [DCOM/98] permite obtener información (es decir, metadato) acerca de la interfase del componente. Ejemplos adicionales de metadato y metamétodos pueden encontrarse en otros modelos de componente y en la literatura [NZ/99] [XOTCL/00] [HUN/98]. Si bien estas soluciones para el problema de cómo proporcionar datos adicionales acerca de un componente son buenas para los asuntos específicos a los que están dirigidos, ellas carecen de generalidad. En los modelos de componentes existentes, el metadato típicamente es utilizado sólo para proporcionar información de uso genérico acerca de un componente (por ejemplo, el nombre de su clase, los nombres de sus métodos, los tipos de los parámetros de sus métodos) o información de apariencia acerca del componente del GUI (por ejemplo, sus colores de fondo y de primer plano, su tamaño, su tipo de letra si es un componente de texto). El trabajo de Orso et. al. [OHR/00] presenta un formato para intercambiar información poniendo énfasis en la generalidad del mismo, y es el primero en

explorar el metadato como un mecanismo general para ayudar a las tareas de ingeniería de software, tales como el análisis y el testing, en presencia de componentes.

Por otro lado, para proveer técnicas y métodos que aseguren la calidad del software y en particular de los sistemas basados en componentes es fundamental como punto de partida tener una taxonomía coherente de los tipos de fallas que pueden surgir en el desarrollo de esta clase de sistemas. Mariani [MAR/03] proponen una taxonomía de esas fallas para ser utilizadas en el desarrollo y la evaluación de técnicas de testing y análisis de software basado en componentes. Esta clasificación pone el foco en las fallas de integración que caracterizan la tecnología de componentes. Las fallas se clasifican de acuerdo a sus causas y sus efectos. Las causas están relacionadas a la tecnología o a un escenario en particular, por ejemplo, mantenimiento del sistema. Los efectos son las fallas causadas en el sistema. Las principales clases de fallas propuestas son:

Clases	Categoría Principal	Subcategorías
Relacionado a Servicios	Sintáctico	• Violación de Interfase
	Semántico	• Malentendido en el Comportamiento
		• Malentendido en los Parámetros
		• Malentendido en los Eventos
		• Malentendido en el Protocolo de Interacción
	No Funcional	• Performances
		• Calidad de Servicio
Relacionado a Estructura	Conectores	• Desacuerdo en el Protocolo
		• Modelo de Datos Incompatible
	Infraestructura	• Servicios Subyacentes
		• Sistema Subyacentes
	Topología	• Callback
		• Re-entrada
		• Recursión
Otros		• Multi-thread
		• Lenguajes Heterogéneos
		• Persistencia
		• Eventos Inconsistentes

Figura 3:Taxonomía de fallas en CBS´s ([MAR/03])

4.3 Mapa de Tecnologías e Investigaciones

En los últimos años desde muchas áreas de investigación, tanto en laboratorios como en productos comerciales, se han comenzando a incluir y considerar muchas de las características que aumentan y proporcionan de una forma u otra mayor autonomía a los sistemas.
Los trabajos u aspectos relacionados con los self-healing systems, los sistemas autónomos en general, están siendo tratados desde áreas muy diversas y en diversos niveles de aplicación y abstracción. Incluso algunas de las principales áreas asociadas, como tolerancia a fallas o sistemas móviles entre otros, son más amplias y muy anteriores a esta nueva visión de los sistemas autónomos y llevan muchos años de investigación, pero este amplia área de los self-healing systems aprovecha algunas facetas y variantes de todos ellos alcanzando varios puntos en común y/o complementarios en la búsqueda de lograr las principales características del self-management presentadas en el capítulo ***2*** *Self-Healing Systems*. Así, existen propuestas desde áreas como la biología, y propuestas informáticas que van desde de meta-procesos para el desarrollo de sistemas autónomos de la escala más pequeña a la más grande, hasta herramientas o algoritmos específicos para tareas muy puntuales. En su mayoría cada uno de ellos ataca algunas características en particular, con procesos, métodos, técnicas e implementaciones ad-hoc. Sin bien no es la aspiración de máxima, es un camino a recorrer, el uso y las costumbres de desarrollo utilizando estas nuevas tendencias, todas aportarán en su medida al avance del área en general.
En el cuadro que sigue a continuación proponemos un esquema de clasificación de las principales áreas de investigación relacionadas con los self-healing systems, si bien no es exhaustivo ni excluyente permite ver los diferentes enfoques del tema y los diversos niveles de aplicación que

mencionábamos, y de ese modo sirve de referencia para la ubicación y alcance de diferentes propuestas:

General	• Teoría de Control	
	• Tolerancia a fallas y Confiabilidad	• Técnicas clásicas
	• Inmunología	
	• Informática Autónoma	
Dominio de Aplicación	• Redes, Sistemas Distribuidos, Middleware	• Middleware adaptativo • Protocolos de red adaptativos • Chroma y otros trabajos de SO CMU • Sistemas de multi-fidelidad
	• Sistemas Móviles • Informática Omnipresente	• Informática orientada-a-tareas • Informática conciente-del-contexto • Informática conciente-de-recursos (incluyendo conciente-de-energía, asignación de recursos adaptativa, etcétera) • Sistemas basados-en-agentes
	• Simulación Biológica	• Modelos Biológicos: Instituto Santa Fe
	• Interfaces de Usuario	• Modelado de Usuarios • Aprendizaje por observación • Sistemas guiados inteligentemente • Interfaces de usuario inteligentes, Interfaces de usuario atentas
	• Informática Cooperativa	
	• Juegos	
Herramientas, Mecanismos y Técnicas	• Modelos de Arquitecturas	• Adaptación basada-en-arquitecturas
	• Algoritmos/basado-en-código	• Algoritmos de auto-estabilización • Hot swapping • Código móvil seguro
	• Modelos Formales	• Lenguajes de Coordinación • Razonamiento formal sobre sistemas móviles
	• Algoritmos genéricos / modelos alternativos	• Métodos de IA, incluyendo Redes Neuronales • Informática Amorfa • Autómata Multicelular
	• Agentes	
	• Teoría Económica	
Objetivos	• Mejora de performance del sistema / uso de recursos	
	• Mejora en la experiencia de usuario, reducción de distracción de usuario	
	• Mejora de confiabilidad	

Hasta acá hemos presentado una descripción general y los principales aspectos de interés desde el punto de vista de la informática autónoma de las aristas más importantes del presente trabajo: Self-Healing Systems, Testing y Desarrollo de Sistemas Basados en Componentes. A continuación y como cierre de esta parte de la presentación veremos el estado del arte de ellas mencionando las investigaciones más recientes relacionadas con todos estos temas, basados en el trabajo de relevamiento de Tosi [TOS/04]. Para mayor claridad, se clasifican de acuerdo a la fase del self-management (vistas en el capítulo *2.5 Fases del Self-Management*) a la que apuntan principalmente: Monitoreo, Interpretación, Diagnóstico, Adaptación y Aprendizaje. Las referencias bibliográficas de esta sección se presentan en el Apéndice A.

Monitoreo

El monitoreo puede mirar el estado interno del sistema o el entorno operativo del mismo. Las propuestas de monitoreo interno instrumentan el programa para recolectar información sobre los cambios en el estado del mismo. Insertar código adicional a la aplicación original (*instrumentación*

estática) puede ser útil para observar parámetros y eventos específicos. Utilizar instrumentos como gauges o probes (*instrumentación dinámica*) permite adjuntar monitores dinámicamente a la aplicación [GSC/01].

También las técnicas de *design-for-testability*, explotadas exitosamente en otros dominios, pueden mejorar sustancialmente la efectividad del monitoreo interno. Mientras que la instrumentación del software funciona en módulos desarrollados completamente, design-for-testability se aplica en la fase de diseño, pautando dispositivos adicionales built-in al software que facilitan el testing [BIN/94] [MTY/01] [TDJ/99]. Ejemplos de dispositivos de test built-in son interfaces especiales para acceder a variables de estado privadas sólo para fines de testing, y suites de test embebidas en los componentes. Los métodos de design-for-testability manejan los efectos de la falta de información con componentes que se administran automáticamente y en muchos casos pueden ser fácilmente explotados para poner en práctica los requisitos de monitoreo de los sistemas self-managed.

Varias de las soluciones del área de la *informática conciente-del-contexto* (*context-aware computing*), estudiadas para aplicaciones móviles, pueden ayudar a monitorear el entorno de los sistemas self-managed. En la informática distribuida moderna, los entornos altamente dinámicos presentan nuevas oportunidades pero a su vez también presentan limitaciones y dificultades para administrar los continuos cambios de entorno. Muchos investigadores han sostenido que la informática conciente-del-contexto es la habilidad de las aplicaciones para detectar cambios en su entorno y adaptar sus comportamientos en respuesta a estos cambios de contexto [SAW/94]. La informática conciente-del-contexto introduce una nueva clase de aplicaciones que son concientes del contexto en el cual están ejecutando. Los tres aspectos importantes del contexto son: dónde estás, con quién estás, y qué recursos están cercanos. En esta dirección, los primeros trabajos de investigación en el área, se enfocaron en la conciencia de la ubicación (location-awareness) y en la habilidad para adaptar sus comportamientos en respuesta a movimientos físicos [HH/94]. Más recientemente, los sistemas concientes-del-contexto involucran como parte del contexto las facetas más variadas del entorno. En [SDA/99] se presenta un paquete de herramientas para asistir el desarrollo de aplicaciones context-enabled. Existen muchos sistemas para recolectar y distribuir información relacionada a un contexto. Algunos de ellos, tienen esas características construidas dentro (built in) [BRO/96] [AAHLKP/97], mientras que otras soluciones ayudan a definir un middleware capaz de distribuir información del contexto utilizando políticas del tipo publicar-suscribir (publish-suscribe) [CNF/01] o coordinaciones de interacción de estados (state interaction coordinations) [JR/02]. En [RJP/04] Roman explora y define un modelo formal, llamado Context UNITY que puede ser utilizado para capturar explícitamente *conocimiento-del-contexto* (*context-awareness*). Las Context UNITY heredan muchas de las características de las Mobile UNITY, incluyendo su notación y su lógica de prueba. Este trabajo provee una nueva interpretación de la definición de contexto. El aspecto más importante es que las aplicaciones individuales solicitan distintas cosas de sus entornos, así resulta interesante definir contexto desde la perspectiva de un único componente. Context UNITY define un modelo formal para explorar contexto y comportamiento.

Interpretación

Las primeras investigaciones y muchos de los trabajos actuales utilizan *assertions*, ya sean pre y post-condiciones, o invariantes de clases. Comenzando desde los trabajos de Floyd [FLO/67] [ILL/75] [GVKMOPSL/79] por un lado y B. Meyer por otro con *Design by Contract* [MEY/97], varias son las propuestas que se basan en los mismos principios. Hoy en día las capacidades de assertions están disponibles para muchos lenguajes de programación incluyendo C++ [CL/90] y Java [KRA/98]. La herramienta Jass (Java con assertions) [BFMW/01] permite las verificaciones de programas más comunes como pre y post-condiciones e invariantes, pero además define *refinement checks* y *trace assertions*. Estos últimos se utilizan para monitorear el comportamiento dinámico de un objeto, el orden de invocación y llamadas a lo largo del tiempo.

Mecanismos de detección más complejos monitorean patrones de eventos que pueden ser chequeados por la validez de propiedades temporales como verificaciones en tiempo de ejecución. *Runtime verfication* intenta cubrir la brecha entre los métodos formales y el testing de software, monitoreando un programa mientras está ejecutando y chequeando contra propiedades relevantes. Dado que en los self-healing systems es importante monitorear y chequear el comportamiento en tiempo de ejecución, runtime verificación puede proveer nuevas e importantes soluciones y mecanismos para generar oráculos de test de forma automatizada y utilizar estos oráculos para monitorear el programa durante su ejecución. Esto cubre varios campos técnicos: análisis dinámico de programa, lógicas y lenguajes de especificación, e instrumentación de programas.
Existen varias técnicas para recolectar información durante la ejecución de programas y algoritmos para detección de errores en “execution traces” [FS/01] [HR/03] [KKLSV/01] [KKLSV/02] [KMSF/01] [GRMD/01].

Detección dinámica y verificación estática son dos técnicas complementarias para tratar invariantes. La primera obtiene información de las ejecuciones del programa, pero no se garantiza

que las propiedades sean ciertas para todas las posibles ejecuciones. La segunda examina el código fuente del programa y chequea que esas propiedades sean ciertas siempre, pero generalmente es una solución muy cara. En [NE/01] encontramos una combinación de ambas y sus ventajas.

Considerando lógicas y lenguajes tenemos propuesta como [FSS/02].

Considerando instrumentación de programa tenemos propuesta como [AK/02].

Una propuesta diferente para la detección de fallas son los *sistemas tolerante a fallas* (*fault tolerant systems*), abocados a la replicación de bloques de programa [HLMR/74] [KIM/84] [SGM/85] [AK/88] [BDX/93] [AC/77].

Software *self-checking* no es una metodología rigurosa, pero es un nuevo término que se utiliza para identificar un método ad-hoc para sistemas safety-critical y fault-tolerant.

Diagnóstico

Hasta la actualidad la fase de diagnóstico ha sido manejada con estrategias muy simples, como por ejemplo propuestas basadas en mapeos fijos entre comportamientos erróneos y métodos de adaptación. Estrategias apenas más sofisticadas seleccionan procedimientos de adaptación por medio de la aplicación iterativa de tácticas de solución posibles hasta que el problema es resuelto [GS/02]. Otro mecanismo de diagnóstico de falla se basa en el método de inferencia por abducción [PR/90], tratando de encontrar la mejor relación entre síntoma y causa de la falla.

El diagnóstico es probablemente la fase que requiere mayor atención en los estudios futuros, y la que puede llegar a brindar mejoras sustanciales en los sistemas self-managed. La capacidad de localizar con precisión la fuente del comportamiento erróneo puede mejorar ampliamente la identificación de estrategias de adaptación que encajen mejor.

Adaptación

La fase de adaptación es el corazón de los sistemas self-managed. La mayoría de las propuestas operan a nivel de arquitectura explotando estrategias de reconfiguración de arquitectura, basados en *arquitecturas reconfigurables* (*reconfigurable architectures*), es decir arquitecturas de software y estilos de arquitecturas concebidas con capacidades para reponer dinámicamente la coordinación y la interacción entre componentes, a fin de garantizar las propiedades arquitectónicas. Estudios recientes indican que estas propuestas funcionan bien para garantizar atributos de calidad del sistema tales como disponibilidad y performance. Algunas de las propuestas son [COWL/02] [COWCL/01] [DF/02] [GS/02][1] [GMK/02] [MK/96][2] [MIN/03] [OGTHJMQRW/99] [RTLB/020] [RTB/02] [SG/02] [VK/02] [VK/03] [YCSSSM/02] [AHSWDKS/02] [KCL/02] [DR/02] [DR/03].

Otras propuestas trabajan en los niveles más bajos de abstracción. Las propuestas de *rollback and resume* permiten recuperar el sistema en tiempo de ejecución luego de anunciarse una falla [BBGHO/98] [SY/85] [KT/87].

Estrategias de enmascaramiento de fallas se aplican a nivel de estado de ejecución [DM/02].

Estrategias de *realocación de código* (*code relocation*) se aplican a nivel de código [FPV/98] [PIC/98] [AHSWDKS/02].

Aprendizaje

La actividad de aprendizaje es el proceso de creación de una nueva base de conocimiento a partir de los fenómenos experimentados. Aplicaciones de procedimientos de aprendizaje pueden encontrarse en investigaciones de *componentes self-testing* y *detección de invariante de componente*.
Algunos de los trabajos más importantes sobre componentes con capacidades self-testing son [LR/98] [MP/04] [OHR/00] [BEY/04] [PY/99].
Todos estos estudios sientan una base científica importante para aumentar los sistemas de software con capacidades self-testing que aprenden y utilizan casos de test en tiempo de ejecución. Esto puede ser aprovechado por los sistemas self-managed de dos formas: primero, pueden proveer un medio para hacer un chequeo funcional de regresión dinámicamente en

[1] Este trabajo puede ubicarse tanto en el dominio de sistemas con capacidades Self-Healing como Self-Adaptative.
[2] Según la clasificación presentada en las primeras secciones, estos dos últimos trabajos se ubican en el área de sistemas Self-Configuring.

condiciones operativas cambiantes; luego, pueden ser útiles en casos particulares al final del ciclo de reacción para chequear que el sistema adaptado funciona apropiadamente.

Basada en técnicas de detección de invariantes, otra solución interesante es inferir pre y post-condiciones de un sistema [EGN/99] [MAR/04][ECN/01].

4.4 Conclusión

El software es propenso a errores debido a su creciente complejidad, decreciente tiempo de desarrollo, y a las limitaciones de las técnicas actuales de testing y análisis. Por lo tanto, se necesitan técnicas que monitoreen el comportamiento del software mientras están en producción, y que permitan a los administradores de software encontrar y reparar eficientemente los problemas después que el software es puesto en producción. Los problemas incluyen temas como errores, incompatibilidades con el entorno de ejecución, huecos de seguridad, performance pobres, usabilidad pobre, o fallas del sistema para satisfacer las necesidades del usuario. Dichas técnicas le permitirían a los desarrolladores prevenir problemas o al menos reaccionar eficientemente cuando ellos ocurren.

5 Propuesta de la Técnica TFT

5.1 Objetivo

En este contexto, la propuesta es investigar y trabajar en el enriquecimiento de componentes utilizados en el desarrollo de sistemas basados en componentes (CBS´s), con el objetivo de superar obstáculos como la falta de información, y las causas y consecuencias de las inconsistencias por suposiciones, principalmente aquellas asociadas al entorno. En particular, pondremos el foco en los problemas con los que se puede topar el usuario del componente, sus situaciones y necesidades, y sus posibilidades y alternativas para superarlos independientemente del proveedor del componente.
Las técnicas tradicionales de testing y análisis hacen poco uso de la información de la calidad de los componentes y subsistemas a la hora de testear el sistema completo. Así, la reutilización para la evaluación de la calidad y la reducción de costos relacionado a la calidad no han sido explotados completamente. Como hemos visto, la verificación de CBS´s está típicamente entorpecida por la frecuente falta de información acerca de los componentes que son proporcionados por terceras partes sin el código fuente y con documentación incompleta. Este tipo de contexto reduce la aplicabilidad de muchas técnicas tradicionales de testing y análisis para CBS´s. Por lo tanto, en este trabajo definiremos una técnica que permita testear y analizar eficientemente CBS´s en presencia de información limitada reutilizando información que puede ser recolectada en usos previos de los componentes.
La contribución en el enriquecimiento de componentes a desarrollar se basará y apuntará principalmente a las características de autonomía, testing y evolución. Por un lado, dado un componente, se espera brindar mecanismos que permitan articularlo externamente para ponerlo en funcionamiento y que allí el mismo componente recabe información sobre sí a partir de sus propias experiencias (ejecuciones), logrando así construir una base de conocimiento en forma autónoma. Por otro lado, todo ese conocimiento se espera que sea utilizado en tareas de testing y análisis del componente y de los sistemas que formará parte, contribuyendo así a sus mejoras. La repetición constante y permanente del proceso de recolección, testing y mejora sobre el componente constituye la principal contribución de la técnica a proponer, ya que de esa forma esperamos brindar aportes para la reutilización del componente, brindar aportes al proceso de evolución del componente mismo, y a la evolución del sistema del que forma parte.
Con el uso de la técnica propuesta se espera facilitar el reuso de componentes, contribuyendo al análisis y al desarrollo de sistemas a partir de partes existentes, brindando información que permita anticiparse a los problemas y a la detección de errores. Información que a su vez podría ser muy costosa de conseguir de otra forma, y en muchos casos hasta imposible, ya sea por omisión, en el caso de las suposiciones implícitas, o ya sea por inexistencia, en el caso de verdaderos usos futuros.

5.2 Motivación: información necesaria, útil y posible

Como ya dijimos, los asuntos que surgen en el contexto de sistemas basados en componentes pueden verse desde dos perspectivas: la perspectiva del proveedor (desarrollador) del componente y la perspectiva del usuario del componente, a su vez también desarrollador de una aplicación que se basa en dichos componentes. Estos dos actores tienen diferente conocimiento, comprensión y visibilidad del componente. El proveedor del componente sabe acerca de los detalles de implementación, y ve al componente como una caja transparente. El usuario del componente, quien integra uno o más componentes para construir una aplicación entera, típicamente no es conciente de las cuestiones internas del componente y lo trata como una caja negra. Consecuentemente, proveedor y usuarios de un componente tienen diferentes necesidades y expectativas, y se ocupan de problemas diferentes.
El proveedor de componentes implementa un componente que podría ser utilizado en variados, y posiblemente impredecibles, contextos. Por lo tanto, debe proporcionar suficiente información para lograr que el componente sea utilizable tan extensamente como sea posible. En particular, extendiendo levemente la clasificación propuesta en [OHR/00], la siguiente información podría ser o necesitada o requerida por un usuario genérico de un componente:

1) **Información para evaluar el componente**: por ejemplo, información de métricas estáticas y dinámicas computadas sobre los componentes, tales como la complejidad de ciclomática y nivel de cubrimiento alcanzado en el testing.
2) **Información para integrar el componente**: por ejemplo, información relacionada a los servicios, información relacionada a las características estructurales de la arquitectura para

la que fue desarrollado el componente, protocolos utilizados, descripción y características del entorno esperado, y suposiciones (assumptions) en general.

3) **Información para poner en producción el componente**: por ejemplo, información adicional de la interfase del componente, como precondiciones, postcondiciones, e invariantes.
4) **Información para testear y depurar el componente**: por ejemplo, una representación de máquina de estado finita del componente, las suites de test de regresión juntos con los datos de cubrimiento, e información acerca de las dependencias entre inputs y outputs.
5) **Información para analizar el componente**: por ejemplo, información de flujo de datos resumida, representaciones de grafos de flujo de control de partes del componente, e información de dependencia de control.
6) **Información sobre cómo customizar / personalizar o extender el componente**: por ejemplo, una lista de las propiedades del componente, un conjunto de restricciones de sus valores, y los métodos que deben ser utilizados para modificarlos.

Mientras que es obvio que el usuario del componente podría requerir la información anterior, es menos obvio por qué el proveedor del componente desearía poner esfuerzo en computar y adjuntar dicha información. Desde el punto de vista del proveedor del componente, sin embargo, la capacidad de proporcionar esta clase de información puede hacer la diferencia para determinar si el componente es o puede ser seleccionado por un usuario de componente que desarrolla una aplicación, y así, si el componente es viable como producto. Además, a veces, la provisión de respuestas puede aún ser requerido por organizaciones de estándares - por ejemplo donde concierne la seguridad de software crítico. En tales casos, la motivación para el proveedor del componente puede surgir a partir de la motivación del usuario del componente. Ahora bien, esta podría ser una buena motivación para el proveedor, pero no necesariamente tendría que ser siempre el caso general, es decir, es necesario considerar aquellos casos en que los componentes nuevos o ya existentes no sean proporcionados con información brindada por su proveedor.

Por otro lado, teniendo en cuenta que desde el punto de vista del desarrollo de sistemas basados en componentes, el reuso de componentes es un aspecto casi fundamental, es interesante plantear cuál es el tipo de información que podría ser o necesitada o requerida por un usuario genérico de un componente a la hora de reutilizarlo. Esta inquietud no necesariamente nos lleva a una diferencia sustancial en la información requerida sino más bien a una posible diferencia del "proveedor" de dicha información.
Veamos, cuando el componente pasa por primera vez desde el proveedor al usuario, digamos que en principio el proveedor es el único que podría proporcionar todo este tipo de información, pero a medida que el componente es utilizado y reutilizado por el usuario, es posible pensar en la posibilidad de que parte de esa información, e incluso un nuevo tipo de información, pueda obtenerla el usuario mismo. Es decir, el proveedor de la información del componente ya no necesariamente debe ser el mismo proveedor del componente. En la actualidad, en muchos casos, cuando cierta información es necesaria y no viene dada por el proveedor, el usuario la construye al menos parcialmente, y aunque parcial y a veces hasta errónea, le es de gran utilidad. Si a eso le sumamos la reutilización del componente, la parcialidad y la inexactitud podrían ir disminuyendo si el usuario va aprehendiendo esa información a partir del comportamiento real que va experimentando el componente y de cada nueva utilización que le da. Así, del mismo modo que por ejemplo una empresa que adquiere un generador de reportes, componente que desea que sea utilizado en más de uno de sus sistemas, típicamente lo integra en primer lugar en un sistema, recolectando en esa experiencia toda la información posible y relevante referida a la integración, de modo tal de facilitar y evitar problemas por falta de información en la integración de dicho componente a los demás sistemas de la empresa, también podría recolectar información de las ejecuciones y utilizarla en futuras tareas de evaluación y integración.
Es importante notar que cierta información del componente, no provista por el proveedor, podría ser irrecuperable, como por ejemplo el mejor criterio de testing aplicable al componente teniendo en cuenta su diseño interno. Pero sin embargo, y como contrapartida, mucha otra información justamente sólo es adquirible con el uso real del componente, dicho en términos muy generales digamos que su comportamiento real frente a cada contexto real, y sus características, sólo son adquiribles a partir del momento en que el componente pasa del lado del usuario. Un ejemplo claro de esto podría ser información sobre el propósito real y final para el que es utilizado el componente más allá del propósito para el que fue desarrollado; otro ejemplo podría ser información sobre características del contexto en el cual el componente final y efectivamente se ha ejecutado.

Sin lugar a dudas, hay dos aspectos importantes para analizar al respecto. Por un lado si tiene utilidad o no, y cuál sería la utilidad de este tipo de información. Y por otro lado, quién, cómo, cuándo y dónde esta información puede conseguirse. En los próximos capítulos iremos analizando y tratando de dar respuesta a estas cuestiones.

5.3 Enriquecimiento de componentes: componentes self-knowledge

Como hemos dicho, en el desarrollo de sistemas basados en componentes, estos últimos son piezas que se intercambian, se reutilizan. El mismo componente puede ser utilizado en varios sistemas, y el mismo sistema puede ponerse en producción en muchas configuraciones que difieren para algunos componentes. En cierto modo, a lo largo del tiempo los componentes van viajando de un sistema a otro, a veces tal como estaban definidos en el primer momento, a veces con modificaciones, y todos esos pasos pueden verse como su evolución. Los sucesos y la información que pudiera relevarse de todos esos sucesos a lo largo del tiempo pueden verse como el diario de viaje del componente. En el mundo de los CBS´s donde la reutilización es una actividad intrínseca, nuestra propuesta es que ese diario de viaje del componente sea considerado como parte del componente mismo. Es decir, contar con componentes que sean más que su funcionalidad (código binario) y alguna descripción de alto nivel de las características del componente o algunas pautas de uso o aplicación. De ese modo, un componente de la era de la informática autónoma es un conjunto de funcionalidades más un conjunto de información referida a su vida, que el componente lleva consigo como su base de conocimiento.
En términos generales, esa información consiste por un lado en todo aquello que el proveedor pudo o quiso dejar documentado y asociado al componente, y por otro lado todo aquello que el usuario del componente puede o quiere registrar y mantener asociado al componente a lo largo de todos los usos y aplicaciones del mismo. Notas de su "gestación", eso es durante su desarrollo; notas de su "nacimiento" y "educación", primeras experiencias y aprendizajes, eso es durante su beta-test y los primeros usos del usuario; y notas de su "crecimiento", sus experiencias posteriores que lo van consolidando en un componente con historia y conocimiento propio, eso es durante sus ejecuciones y reusos. Es decir, contar con "notas" sobre todo lo que le va pasando al componente, no contado como una historia a posteriori sino una narración lo más cercana posible al momento en que le van sucediendo realmente las cosas al componente. Si pensamos en todos esos usos y sistemas por los que va pasando el componente como el viaje que este hace a lo largo de todo su ciclo de vida, podríamos decir que esas "notas" adjuntadas al componente forman el *diario de viaje* del componente. Entonces definimos un ***componente knowledge*** como un conjunto de funcionalidades más una base de conocimiento propia definida por su diario de viaje.
Para que estos *componentes knowledge* sean propios y dignos de la era de la informática autónoma, es esperable que todo ese conocimiento lo adquieran principalmente por sí mismos. Es decir, la parte de información correspondiente al componente, la base de conocimiento, el diario de viaje, podría ser proporcionado por el proveedor o por el usuario del componente, pero lo interesante y novedoso es que ese diario lo vaya construyendo el componente mismo (o la tecnología asociada al componente) en forma autónoma, eso es sin la intervención del ser humano. Así las propias experiencias y usos del componente pueden ser las "proveedoras" del tipo de información necesaria o requerida por un usuario del componente. Entonces, dado que vamos a considerar el caso en que el mismo *componente knowledge* construye su propia base de conocimiento lo llamaremos ***componente self-knowledge***.

Esta idea de proveer información con los componentes de software está muy relacionada a lo que los ingenieros electrónicos hacen con los componentes de hardware: del mismo modo que una resistencia no es útil sin sus características esenciales como valor de resistencia, tolerancia y embalaje, así un componente de software necesita proveer alguna información sobre si mismo para ser útil en distintos contextos. Mientras más información esté disponible acerca de un componente, menores serán las restricciones sobre las tareas que pueden ser ejecutadas por el usuario del componente, tales como técnicas de análisis de programa aplicables, model checking o simulaciones. En este sentido, la posibilidad de disponer de información de un componente puede ser percibido como un "indicador de calidad" por un desarrollador de CBS que está seleccionando los componentes a incluir en su sistema.

En este trabajo presentamos un framework que le permite al usuario del componente enriquecer un componente con distinto tipo de información recolectada a partir de sus propias ejecuciones, dependiendo del contexto y las necesidades específicas. Más aun, el desarrollador del componente puede si lo desea, entregar el componente enriquecido con datos de sus ejecuciones preliminares en etapas de testing, como por ejemplo beta-test, o bien entregarlo sin ellos, o bien entregarlo con sugerencias de qué datos serían de utilidad recolectar en las ejecuciones del componente acorde con la intención con la que fue desarrollado el componente. Por otro lado, como no estamos interesados simplemente en tener un tipo específico de información del componente, como podría ser su especificación o la performance de sus servicios, necesitamos considerar una forma de proveer diferentes tipos de información. Así, el framework se basa en presentar esta información en un formato genérico que permite representar cualquier tipo de información, permite describir aspectos estáticos y dinámicos del componente. Este formato está concebido para que pueda ser fácilmente accedido por el usuario y sus herramientas, con el fin de asistir las diferentes tareas del desarrollo de software en presencia de componentes en un sistema.

La idea de proporcionar datos adicionales junto con el componente no es nueva, es una característica común de muchos modelos de componentes existentes, aunque generalmente es una característica que proporciona funcionalidad relativamente limitada, y en la mayoría de los casos se trata de información estática. De hecho, las soluciones proporcionadas hasta ahora por los modelos existentes de componentes están hechas a medida para una clase específica de información y carecen de generalidad. La propuesta aquí es explorar el uso de información recolectada en las ejecuciones como un mecanismo general para ayudar a las tareas de ingeniería de software, tal como el análisis y el testing, en presencia de componentes. Y sobre la base de un formato genérico, permitirle al usuario del componente definir propiedades a relevar durante la ejecución del componente, información que le será de utilidad por un lado para el análisis y mantenimiento de aplicaciones ya puestas en producción, y por otro lado principalmente para la evolución de esas aplicaciones, desarrollo de nuevas versiones o reutilización de dichos componentes en otros sistemas o entornos. Nótese así que el mayor provecho de la información recolectada sucede en un tiempo distinto del que es obtenida, es decir, la información es recolectada "hoy" para ser utilizada "mañana", o lo que es lo mismo, la propuesta es recolectar información "*hoy para mañana*" ("***T**oday **F**or **T**omorow*"). Entonces en el presente trabajo definimos una técnica en particular para crear la base de conocimiento de los componentes self-knowledge a partir de esa información de "hoy para mañana", de este modo la llamamos ***técnica TFT*** y definimos los ***componentes self-knowledge TFT*** como los componentes self-knowledge cuya base de conocimiento ha sido creada acorde a la definición de la técnica TFT.

Como en este trabajo estamos analizando los problemas y soluciones desde el punto del vista del usuario del componente, independientemente del proveedor del componente, debemos pensar que ese diario de viaje comienza a escribirse al llegar el componente a manos del usuario. Claramente, contar con información brindada por el proveedor es muy útil, incluso como dijimos antes porque cierta información sólo él puede proveerla, pero hoy por hoy el caso más común en CBS´s es que los proveedores no la proporcionan. Además también queremos considerar el caso de componentes que ya existen, es decir, presentar una propuesta que pueda ser aplicable incluso a componentes que ya han sido adquiridos por los usuarios. Es por eso que en el framework propuesto, en el caso más amplio, el primer "usuario" del componente que puede comenzar a ayudar a escribir el diario de viaje es el proveedor del componente, ya sea con información del desarrollo como con información de los versiones beta que serían de suma utilidad, pero esta no será una condición indispensable para la aplicación de la técnica, ni lo que brindará las principales ventajas que buscamos alcanzar con ella.

Una situación similar se plantea con la información que puede brindar manualmente el usuario del componente versus la información que puede conseguir el componente por sí mismo. Claramente, es factible que el usuario adicione manualmente información al componente, pero dado el costo que ello implica en muy pocos casos es realizado, y mucho menos sistemáticamente. Es por eso que en esta propuesta, en el caso más amplio, el usuario del componente podría escribir datos complementarios en el diario de viaje, principalmente aquella información que no fuera alcanzables por herramientas automáticas, pero esto no será una condición indispensable para la aplicación de la técnica, ni lo que brindará las principales ventajas que buscamos alcanzar. Por el contrario, la técnica propuesta pondrá el foco en la información que pueda conseguirse en forma autónoma dándole al usuario un rol fundamental pero sólo en las dos siguientes tareas: 1) definir qué datos considera que son útiles y relevantes recolectar para él acerca de lo que le va sucediendo al componente, y 2) utilizar el diario de viaje construido por el componente.

5.4 Componentes Self-Knowledge TFT

Para las descripciones y consideraciones que veremos en las próximas secciones llamaremos ***C*** al componente self-knowledge TFT al que deseamos construirle el diario de viaje. Definimos al componente self-knowledge TFT ***C*** como la tupla <F^C, D^C> donde F^C es su *funcionalidad* (código binario), y D^C es su *diario de viaje* o base de conocimiento.
Siendo ***S*** un CBS en el que es utilizado ***C***. Diremos que el *contexto o entorno de ejecución* de ***C*** es la tupla ***X*** = <***S*** - ***C***, ***R***, ***U***, ***F***>. ***S*** - ***C*** representa al sistema ***S*** sin considerar a ***C***, es decir todo el sistema ***S*** menos el componente ***C***. ***R*** representa el conjunto de recursos presentes en el entorno donde ejecuta ***S*** tales como otros software y recursos físicos, por ejemplo, sistema operativo, motor de base de datos, memorias, procesadores, censores, etcétera. ***U*** representa el conjunto de usuarios de ***S***. Y ***F*** representa la funcionalidad que tiene ***C*** al formar parte de ***S***, es decir ***F*** = F^C. De este modo modelamos el contexto de ejecución del componente como el resto del sistema del que forma parte el componente más el entorno operativo de dicho sistema, también llamado ambiente. Así, si el componente pasa a ser utilizado en otro sistema, su contexto de ejecución habrá cambiado, del mismo modo que si cambia su funcionalidad entonces su contexto de ejecución habrá cambiado. Veremos un poco más adelante cuáles son todos los escenarios de cambios posibles y cómo ellos afectan o se ven reflejados en un componente self-knowledge TFT.

Las componentes de ***X*** , sus partes y sus elementos tienen propiedades. Esas propiedades son muy variadas, y a su vez son objeto de estudio de diferentes áreas. El valor que han tenido, esas propiedades durante las ejecuciones de ***C***, principalmente las que tengan alguna relación o interés desde el punto de vista de análisis del comportamiento de ***C***, son las que nos interesará registrar ya que ellas nos darán información sobre "cómo le ha ido", "qué ha visto", y "qué le ha sucedido" a ***C*** mientras ejecutaba en dicho contexto de ejecución. Entonces, al conjunto de todas las propiedades posibles de un entorno de ejecución de un componente lo llamaremos ***M*** y diremos que sus elementos son *metadatos* que representan a dichas propiedades. Llamaremos ***V*** al conjunto de los *valores de esos metadatos*. Teniendo en cuenta que la información que recolectará la técnica TFT será utilizada para amar el diario de viaje de un componente en particular, las ejecuciones serán consideradas desde el punto de vista de dicho componente. Así, la información contenida en el diario de un componente self-knowledge TFT, serán los *metadatos* de ese componente, es decir datos que hablan o predican sobre características o propiedades del componente y su entorno, y que han sido recolectados en las ejecuciones de dicho componente.

Ahora bien, veamos más claramente cómo se relacionan los metadatos, las ejecuciones y el diario de viaje de un componente self-knowledge TFT.
Dado un contexto de ejecución ***X***, y los valores de metadato obtenidos para ***C*** hasta un instante determinado, cada próxima ejecución de ***C*** determina nuevos valores de metadato de ***C*** . Cada uno de estos valores puede ser igual o distinto al obtenido en las ejecuciones anteriores, pero en términos generales y a medida que aumenta el número de ejecuciones considerado, esos valores van cambiando. Por ejemplo, supongamos el metadato "rango de I/O" de alguno de los servicios de ***C*** , es decir una propiedad de $\boldsymbol{F^C}$. En las primeras ejecuciones de ***C*** puede que dicho servicio no sea utilizado y por ende el valor del metadato considerado (valor de "rango de I/O") será el conjunto vacío (asumiendo que representamos rangos con conjuntos). Pero cuando se alcanza una ejecución que utiliza el servicio en cuestión tenemos un nuevo valor para "rango de I/O" igual al conjunto formado por los valores de los parámetros de entrada y de salida de dicha ejecución. En las siguientes ejecuciones de ***C*** en que el servicio es utilizado tenemos otro nuevo valor para "rango de I/O" que puede ser igual al anterior, si es que se utiliza con los mismos parámetros, o puede no ser igual, si se utiliza con otros parámetros, en cuyo caso el nuevo valor para "rango de I/O" es el conjunto formado por los rangos comprendidos entre los parámetros anteriores que ya teníamos y los parámetros de esta última ejecución. De este modo, vemos que cada ejecución produce una transformación en el valor del metadato, y en cada sucesiva ejecución ese valor continúa transformándose. Entonces podemos extender esta misma idea para todos los metadatos y pensar que cada ejecución produce una transformación en sus valores, y en cada sucesiva ejecución esos valores continúan transformándose. Eso es, dado un metadato ***m***, ***k*** ejecuciones y $\boldsymbol{v_j}$ valores de metadato con $0 \leq j \leq k$, tenemos la siguiente sucesión de transformaciones:

$$(m, v_0) \overset{exec_1}{\Rightarrow} (m, v_1) \overset{exec_2}{\Rightarrow} (m, v_2) \overset{exec_3}{\Rightarrow} \dots \overset{exec_k}{\Rightarrow} (m, v_k)$$

donde $exec_j$, $0 \leq j \leq k$, representan las ejecuciones del componente.
Teniendo en cuenta que con la técnica TFT estamos interesados en llevar registro de los metadatos de ***C***, representaremos esto con las funciones $\boldsymbol{RegTFT_j} : \boldsymbol{M} \times \boldsymbol{V} \to \boldsymbol{M} \times \boldsymbol{V}$, con $0 \leq j \leq k$, donde dado un metadato y un valor alcanzado, cada función $\boldsymbol{RegTFT_j}$ captura la transformación de la j-ésima ejecución de ***C***, es decir:

$$\boldsymbol{RegTFT_j}\,(<m, v_{j-1}>) = <m, v_j>.$$

De este modo podemos representar el resumen o las conclusiones de las vivencias de un componente ***C*** a lo largo de ***k*** ejecuciones en un contexto ***X*** como las funciones $\boldsymbol{e^C_k} : \boldsymbol{M} \to \boldsymbol{V}$ que dado un metadato devuelve el último valor alcanzado en la k-ésima vivencia, es decir, para todo ***m*** ϵ ***M*** tenemos:

$$\boldsymbol{e^C_k}\,(m) = \boldsymbol{RegTFT_k} \circ \boldsymbol{RegTFT_{k-1}} \circ \dots \circ \boldsymbol{RegTFT_2} \circ \boldsymbol{RegTFT_1}\,(<m, v_0>)$$

o lo que es lo mismo:

$$\boldsymbol{e^C_k}\,(m) = \boldsymbol{RegTFT_k}\,(<m, \boldsymbol{e^C_{k-1}}\,(m)>)$$

es decir que la última vivencia de ***C*** es un registro de ejecución más sobre la vivencia última anterior. Por tener $\boldsymbol{e^C_k}$ el resumen de todas las vivencias la llamaremos *experiencia* de ***C*** en ***X***. Así, la última experiencia $\boldsymbol{e^C_k}$ de ***C***, o resumidamente $\boldsymbol{e^C}$, son los metadatos recolectados para ***C*** a partir de las ejecuciones in-field del software puesto en producción. Nótese que de esta forma, y tal como sucede en la realidad, la experiencia de ***C*** depende del contexto de ejecución, es decir, tanto del sistema en el que ejecuta el componente ***C***, como de los recursos disponibles, de los usuarios y de la propia funcionalidad de ***C***. Y la experiencia también depende del uso del sistema, es decir, de cómo y para qué es utilizado el sistema, y ello queda reflejado en cada transformación $\boldsymbol{exec_j}$.

Como hemos dicho, en el desarrollo de sistemas basados en componentes, estos últimos son piezas que se intercambian, se reutilizan. El mismo componente puede ser utilizado en varios sistemas, y el mismo sistema puede ponerse en producción en muchas configuraciones que difieren para algunos componentes. En cierto modo, a lo largo del tiempo los componentes van viajando de un sistema a otro, a veces tal como estaban definidos en el primer momento, a veces con modificaciones, y todos esos pasos pueden verse como su evolución. Cada uno de esos pasos son a su vez cambios en el entorno de ejecución del componente C. Para cada uno de esos entornos podemos construir u obtener la experiencia como vimos antes. De este modo, a lo largo de toda la vida de C podemos ir obteniendo sucesivas experiencias, cada una asociada al contexto de ejecución de C, o lo que es lo mismo, cada cambio de contexto de ejecución de C produce un *hito* en su evolución que lo representamos comenzando una nueva experiencia que se agrega a la secuencia de las experiencias que reflejan la evolución de las ejecuciones de C. Entonces finalmente diremos que el *diario de viaje* D^C es la secuencia de todas las experiencias de C obtenidas a partir de cada uno de los entornos en los que ha ejecutado C a lo largo de toda su vida.
Puesto en términos, sea S_1 el CBS en el que C es utilizado por primera vez, y sea $X_1 = <S_1 - C, R_1, U_1, F_1>$ el entorno al poner en producción a S_1, y por ende el primer entorno de C. Sean $X_2, X_3 \ldots X_n$ los sucesivos entornos en los que C es reutilizado con $X_i = <S_i - C, R_i, U_i, F_i>$ para $1 \leq i \leq n$. Por un lado, para un i dado cualquiera, con $1 \leq i \leq n$, $F^C{}_i$ es la funcionalidad de C en el sistema S_i. Si al pasar el componente de un contexto X_{i-1} a otro X_i su funcionalidad no cambia entonces $F^C{}_i$ será igual a $F^C{}_{i-1}$. Por otro lado, al considerar el primer uso del componente, eso es su primer puesta en producción, D^C estará vacío. Si el proveedor entrega a C con datos de sus ejecuciones preliminares por ejemplo de beta tests, D^C no estará vacío. Para mayor claridad supondremos que el proveedor entrega sólo F^C y que D^C es construido totalmente del lado del usuario. Luego puede pensarse simplemente al proveedor como el usuario de primera vez quien ya entregó a C con parte de ese diario ya construido; esto desde el punto de vista teórico no plantea diferencias, y desde el punto de vista de utilidad y ventajas puede verse como un beneficio para el usuario. Luego, de acuerdo a como TFT va armando el diario de C, D^C va cambiando en cada ejecución de cada uno de los sistemas y contextos de los que forma parte. Llamaremos $D^C{}_{Xi}$, o resumidamente $D^C{}_i$, al diario construido para C en las ejecuciones de S_i en el contexto X_i. Durante el desarrollo de S_i y hasta la 1er ejecución en producción de S_i, $D^C{}_i$ es igual a $D^C{}_{i-1}$, es decir, el diario de viaje del componente no cambia durante la etapa de desarrollo, sólo cambia en ejecuciones en producción (en el caso general, en ejecuciones de beta test o similares que deseen dejarse registradas). Cuando S_i se pone en producción, surge un nuevo contexto en el que ejecutará C, eso es $X_i = <S_i - C, R_i, U_i, F_i>$. Dado que el contexto de ejecución de C pasará de ser X_{i-1} a ser X_i entonces comenzaremos una nueva experiencia para C por lo que agregaremos un nuevo elemento al diario de viaje de C en la que comenzará a registrarse todo lo que se pueda observar en las ejecuciones de S_i en X_i. Entonces, al poner en producción a S_i tendremos que $C = <F^C{}_i, D^C{}_i>$ donde $D^C{}_i = D^C{}_{i-1} + e^C{}_i$, y $e^C{}_i = \emptyset$, es decir, que el nuevo diario de viaje de C es el viejo diario de viaje más una experiencia vacía (la función vacía) pues aún no ha sido ejecutado en el nuevo contexto X_i [3]. Luego en cada ejecución de C, $e^C{}_i$ irá modificándose (aumentándose) con la información recolectada en esas ejecuciones como vimos en la definición de la función e^C. De este modo, en el caso general, siendo X_n el último contexto en el que ha sido utilizado el componente self-knowledge TFT C, su diario de viaje $D^C = D^C{}_n = D^C{}_{n-1} + e^C{}_n$, o lo que es lo mismo $D^C = e^C{}_1 + e^C{}_2 + \ldots + e^C{}_n$, es decir que el diario de viaje es la secuencia de todas las experiencias de C obtenidas a partir de cada uno de los entornos en los que ha ejecutado C a lo largo de toda su vida.
Entonces digamos que cuando un sistema S_i tiene a C como una de sus piezas, C tiene un diario o base de conocimiento $D^C{}_i$ formado por la información sobre "cómo le ha ido", "qué ha visto", y "qué le ha sucedido" cuando fue utilizado en los sistemas anteriores ($S_1, \ldots, S_{i-1}$) gracias a su $D^C{}_{i-1}$ con el que empezó en S_i, y más la información de "cómo le ha ido", "qué ha visto", y "qué le ha sucedido" mientras está siendo utilizado en S_i.
Nótese que para que un componente cualquiera empiece a ser, o se transforme en, un componente self-knowledge TFT basta construir una tupla con la funcionalidad del componente y un diario de viaje vacío, eso es, una secuencia vacía ***nil***, es decir, $C = <F^C, \textbf{nil}>$.

Esta forma de construir el diario de viaje nos permite ver que para todo i, con $1 \leq i \leq n$, $D^C{}_i$ es siempre incremental y consecutivo.

Cabe aclarar que para lograr los objetivos buscados con la técnica TFT no será necesario llevar registro de los contextos de ejecución X_i para todo $1 \leq i \leq n$, sino sólo lo que parcialmente pueda saberse de ellos a través de las propiedades capturadas en tiempo de ejecución en los metadatos de C. Así mismo, tampoco será necesario llevar registro de todas las experiencias intermedias que va atravesando un componente en las ejecuciones en un mismo contexto de ejecución sino sólo de la última alcanzada en cada contexto.

[3] En la definición de la experiencia e^C el subíndice k ha sido utilizado para representar el número de ejecución del componente en un contexto, notesé que aquí el subíndice i de la experiencia $e^C{}_i$ ha sido utilizado para representar el número de contexto del componente; de aquí en adelante este último será el significado de dicho índice.

De aquí en adelante consideraremos que $X_n = <S_n - C, R_n, U_n, F_n>$ es el último contexto en el que fue utilizado ***C***. Y usaremos los nombres ***F***, ***D***, ***S***, ***X*** y ***e*** para referirnos a alguna funcionalidad F^C_i, un diario de viaje D^C_i, un sistema S_i, un contexto de ejecución X_i y una experiencia e^C_i de un componente ***C*** con $1 \leq i \leq n$ cuando no sea relevante el índice *i* considerado.

5.5 Escenarios

Un tema importante a analizar en esta propuesta es el de *la información*. Cuestiones como qué información sería útil recolectar, qué información es posible recolectar en tiempo de ejecución, cómo puede ser recolectada dicha información, cuál es el formato más apropiado para esa información, es posible recolectar toda y cualquier información deseada, etcétera. En los próximos capítulos iremos analizando estos puntos, pero antes queremos presentar aquí cuáles son a nuestro entender los principales escenarios en los que esperamos que sea de utilidad la información recolectada durante las ejecuciones del componente en todos los sistemas y entornos de los que va formando parte.

Para clarificar la descripción de cada escenario consideraremos, y sugerimos tener en mente, algunos de lo tantos posibles metadatos a recolectar en las ejecuciones como podrían ser: ejecuciones notables (un subconjunto de datos de input y output efectivamente ejecutados), rango de valores o invariantes de I/O, recursos solicitados, recursos no encontrados, y recursos más y menos solicitados. No entraremos en detalles ni justificaciones de cada uno de ellos ya que eso lo veremos en detalle más adelante.

La propuesta es que este tipo de información recolectada en las ejecuciones puede ser utilizada para verificar la calidad de CBS´s en varios casos interesantes:

(a) cuando un componente ***A*** que forma parte de un sistema en producción ***S*** se añade a un nuevo sistema ***S'***.
(b) cuando un nuevo componente ***B*** reemplaza a un componente ***A*** en un sistema ***S***.
(c) cuando un sistema ***S*** que incluye a un componente ***A*** es instalado en un nuevo ambiente de producción.
(d) cuando un componente ***B*** que forma parte de un sistema ***S'*** reemplaza a un componente ***A*** en un sistema ***S***.

El caso (a) ocurre cuando un sistema es extendido agregando nuevos componentes self-knowledge TFT que se toman de una librería de componentes ya en uso en otros sistemas. Digamos que hasta el momento de uso de ***A*** en ***S*** el contexto de ejecución del componente self-knowledge TFT ***A*** ha sido $X = <S - A, R, U, F^A>$ con $A = <F^A, D^A>$. A partir de la puesta en producción de ***S'*** el nuevo contexto de ejecución de ***A*** será $X' = <S' - A, R', U', F^A>$ con $A = <F^A, D^A + Ø>$, es decir que su contexto cambiará totalmente excepto su funcionalidad que seguirá siendo la misma, y en su diario de viaje se producirá un hito y comenzará una nueva experiencia ya que el contexto ha cambiado. En este caso, la información obtenida cuando el componente self-knowledge TFT fue utilizado en sistemas existentes, eso es D^A, puede ser utilizada para verificar la compatibilidad del componente en los nuevos sistemas como ***S'***. En particular por ejemplo, las ejecuciones notables pueden utilizarse para testear la integración del componente como parte del nuevo sistema, mientras que los invariantes pueden utilizarse como oráculos para identificar las desviaciones en el comportamiento esperado. Los rangos e invariantes pueden utilizarse en monitores que verifiquen la consistencia del comportamiento del componente cuando este está siendo ejecutado en el nuevo sistema, es decir en tiempo de ejecución. Cuando un invariante es violado, el monitor generaría una advertencia que podría corresponder o bien a una falla, o bien a un comportamiento del componente no explorado previamente. Un análisis en tiempo de ejecución de nuevos recursos utilizados puede alertar sobre nuevas necesidades del componente no tenidas en cuenta hasta el momento.
El caso (b) ocurre cuando un componente en uso es sustituido por una versión más nueva. Digamos que hasta el momento de uso de ***A*** en ***S*** el contexto de ejecución del componente self-knowledge TFT ***A*** ha sido $X = <S - A, R, U, F^A>$ con $A = <F^A, D^A>$. A partir de la puesta en producción de la nueva versión de ***S***, en la que ***A*** es sustituido por ***B***, llamémosla ***S'***, el nuevo contexto de ejecución de ***B*** será $X' = <S' - B, R, U, F^B>$ con $B = <F^B, D^A + Ø>$, es decir que desde el punto de vista de lo que era ***A***, casi todo su contexto será el mismo excepto su funcionalidad que pasará a ser la del nuevo componente ***B***, y en su diario de viaje se producirá un hito y comenzará una nueva experiencia ya que el contexto ha cambiado. En este caso, la información del "viejo" componente ***A***, es decir, el diario de viaje de ***A***, pasa a ser el diario de viaje del "nuevo" componente ***B*** con una experiencia más en la que comenzarán a registrarse las vivencias en el nuevo entorno. Así, la información computada para el "viejo" componente ***A*** ejecutado dentro de la vieja versión de ***S*** puede ser utilizado para testear y monitorear el "nuevo" componente ***B*** y la nueva versión de ***S***. Por ejemplo, un subconjunto de las ejecuciones notables representan casos del test de regresión e integración como en el caso anterior. También un subconjunto de los rangos e invariantes pueden

ser utilizado en monitores que verifiquen que el comportamiento de la "nueva" versión ***B*** es compatible con la "vieja" versión ***A***. Una violación de un invariante puede revelar o bien una falla en la nueva versión, o bien un comportamiento correcto de la nueva versión que nunca se vio con la vieja versión. Del mismo modo, un análisis en tiempo de ejecución de los recursos utilizados puede alertar necesidades no exploradas hasta el momento.
El caso (c) ocurre cuando un sistema en uso, junto con todos sus componentes, es instalado en un nuevo ambiente, como podría ser por ejemplo en otro equipo. Digamos que hasta antes de la reinstalación el contexto de ejecución del componente self-knowledge TFT ***A*** ha sido ***X*** = <***S*** - ***A***, ***R***, ***U***, F^A> con ***A*** = <F^A, D^A>. A partir de la puesta en producción de ***S*** en el nuevo ambiente, el nuevo contexto de ejecución de ***A*** será ***X'*** = <***S*** - ***A***, ***R'***, ***U'***, F^A> con ***A*** = <F^A, D^A + ***Ø***>, es decir que su contexto cambiará principalmente en el entorno operativo, recursos disponibles y usuarios, y el resto del sistema y su funcionalidad seguirán siendo los mismos, y en su diario de viaje se producirá un hito y comenzará una nueva experiencia ya que el contexto ha cambiado. En este caso, la información obtenida cuando el componente self-knowledge TFT fue utilizado en el entorno anterior puede ser utilizada para verificar la compatibilidad del sistema con el nuevo entorno. Por ejemplo, si se deseara testear el sistema en el nuevo ambiente, las ejecuciones notables representan casos del test de regresión como en el caso anterior. También los rangos e invariantes pueden ser utilizado en monitores que verifiquen que el comportamiento en el "nuevo" entorno es compatible con el "viejo" entorno. Un análisis en tiempo de ejecución de los recursos utilizados puede alertar necesidades no satisfechas o incompatibles entre el sistema y el nuevo entorno, o bien necesidades no exploradas hasta el momento. Del mismo modo, una violación de un invariante puede revelar o bien una falla en el nuevo entorno, o bien un comportamiento correcto que nunca se vio en el viejo entorno.
El caso (d) ocurre cuando un componente en uso en otro sistema reemplaza un componente en uso en el sistema de interés para agregar y/o modificar las funcionalidades implementadas. Si el componente ***A*** no fuera un componente self-knowledge TFT y el componente ***B*** sí lo fuera la situación sería similar a la del caso (a) aplicada a ***B***, si el componente ***B*** no fuera un componente self-knowledge TFT y el componente ***A*** sí lo fuera la situación sería similar a la del caso (b), y si ninguno de los componentes fueran componentes self-knowledge TFT, estaríamos frente al mismo caso de cualquier componente que puede comenzar a ser un componente self-knowledge TFT con sólo asignarle un diario de viaje vacío y en sus posteriores cambios se aplicarían algunos de todos los casos aquí presentados. El caso interesante es aquel en el que ambos son componentes self-knowledge TFT. Entonces, digamos que hasta el momento de uso de ***A*** en ***S*** el contexto de ejecución de los componente self-knowledge TFT ***A*** y ***B*** han sido ***X*** = <***S*** - ***A***, ***R***, ***U***, F^A> con ***A*** = <F^A, D^A>, y ***X'*** = <***S'*** - ***B***, ***R'***, ***U'***, F^B> con ***B*** = <F^B, D^B>, respectivamente. A partir de la puesta en producción de la nueva versión de ***S***, en la que ***A*** es sustituido por ***B***, llamémosla ***S''***, el nuevo contexto de ejecución de ***B*** será ***X''*** = <***S''*** - ***B***, ***R***, ***U***, F^B> con ***B*** = <F^B, D^B + ***Ø***>, es decir que desde el punto de vista de lo que era ***A***, casi todo su contexto será el mismo excepto su funcionalidad que pasará a ser la del nuevo componente ***B***, y desde el punto de vista de lo que era ***B***, todo su contexto cambiará totalmente excepto su funcionalidad que seguirá siendo la misma, y en su diario de viaje se producirá un hito y comenzará una nueva experiencia ya que el contexto ha cambiado. Nótese que si bien este caso parece similar al caso (a), aquí dado que ambos componentes, reemplazante y reemplazado, son componentes self-knowledge TFT se dispone de la información contenida en los dos diarios de viaje que podrían ser cruzadas. Así por ejemplo las ejecuciones notables computadas para ellos, aunque en diferentes contextos de ejecución, pueden utilizarse para testear el componente que se está reemplazando en el nuevo contexto. Los rangos, los invariantes y la información de recursos computados, y toda la información recolectada para los componentes reemplazante y reemplazado puede ser mapeada para compararlos, asistiendo a tareas tempranas como el análisis de factibilidad del reemplazo contando con datos muy concretos y reales. Dependiendo del tipo de información recolectada, las diferencias de contexto de cada uno podrían ponerse en evidencia en esa comparación, lo que a su vez pondría en claro las diferencias y consecuencias que pudieran surgir al sustituir un componente por otro. Así, el mapeo de esta información puede identificar posibles incompatibilidades dando incluso pautas del origen de las mismas, y por ende brindando también posibles guías o pautas para su solución. Toda esa misma información también puede utilizarse en monitores para el componente que se reemplaza de la misma forma que en los casos anteriores.

5.6 Caso de uso

Veamos a continuación un breve y simple ejemplo para describir cómo funciona la técnica en la práctica, y para ver un poco más claramente alguno de sus aportes y beneficios. Los módulos aquí utilizados y el funcionamiento completo de la técnica TFT serán descriptos en detalle en los próximos capítulos.

Sea un componente llamado "*Calculador*" desarrollado por terceras partes que brinda servicios para diferentes tipos de cálculos (aritméticos, financieros, lógicos, etcétera), donde uno de ellos, el servicio *CalcExpresion* en particular resuelve el tipo de cálculo más genérico que consiste en

calcular el resultado de una expresión cualquiera computable por un lenguaje procedural interpretado. Este servicio se implementa con un variable de entrada de tipo texto, y entre los detalles conocidos por el proveedor del componente y desconocidos por los usuarios, tenemos que internamente esa variable es enviada a un intérprete de una librería interna de la plataforma quien la evalúa en tiempo de ejecución y devuelve su resultado. Debido al tipo de manipulación necesario sobre el texto para poder ser evaluado, y por estar involucrado el intérprete en la evaluación, la performance total del servicio *CalcExpresion* es baja. Pero para el proveedor del componente esa performance es suficientemente buena ya que ese servicio se incluye en el componente sólo para dar el mayor alcance funcional posible al componente en su totalidad, y con la intención de que sea utilizado excepcionalmente.
Por su parte, una cierta empresa decide incluir el componente "*Calculador*" en un sistema de cálculo de comisiones que está desarrollando. Dicho sistema es instalado en una de las áreas de la empresa para el cálculo de ciertas comisiones de su grupo de trabajo. En particular, la performance del funcionamiento del sistema es muy buena de acuerdo a las expectativas de los usuarios, tanto en la puesta en marcha como en su utilización durante algo más de un año. Tal es así que al surgir en otro área también la necesidad de un sistema de cálculo de comisiones, el mismo sistema es replicado e instalado sobre otro grupo de usuarios. De este modo, al poner en marcha esta réplica, todo comienza bien y normal pero al lanzarse los cálculos propiamente dichos se detecta una performance muy mala respecto de las expectativas. Tan mala que por momentos el sistema da la sensación de haber entrado en loop ya que parecería no avanzar, lo que torna impracticable el procesamiento, ya que lo que se esperaba calcular en horas resulta tardar días enteros.

¿Cómo podría ayudar en un caso como este la técnica TFT? Para ello supongamos que la empresa, usuario del componente "*Calculador*", decide insertarlo en su primer sistema de comisiones como un componente self-knowledge TFT, y veamos cómo funciona la técnica. En su primer uso en el primer sistema de comisiones, el componente va registrando sucesos de su entorno, en este caso, entre otros metadatos, asumamos que se registra información estadística sobre la cantidad de veces que son utilizados los servicios del componente y lo requerido por este. Así, a largo de las ejecuciones la técnica va registrando en el diario de viaje del componente la cantidad de veces que los servicios del componente son utilizados. En particular en este ejemplo si se consultara el valor de este metadato para el servicio *CalcExpresion* se vería que en este primer sistema el número de invocaciones es bajo y mucho menor al del resto de los servicios del componente, para darle un valor numérico digamos que en este ejemplo la relación es del 2%.
Luego el componente, junto con su diario, que tiene los valores de metadatos recolectados en esa experiencia anterior, es instalado en el segundo área, y como ha cambiado su entorno, la técnica registra un hito, una nueva experiencia, en el diario de viaje del componente. Nuevamente, en esta segunda experiencia, el componente va registrando sucesos de su entorno, y en particular supongamos que otra vez el usuario decidió registra información sobre la cantidad de veces que son utilizados los servicios del componente y lo requerido por este. También ahora, a largo de las ejecuciones, la técnica va registrando en el diario de viaje del componente la cantidad de veces que los servicios del componente son utilizados pero ahora dentro de la segunda experiencia del diario, sin modificar los datos de la primer experiencia. Ahora bien, si se consultara el valor de este metadato para el servicio *CalcExpresion* en esta segunda experiencia se vería que en este segundo sistema el valor del número de invocaciones es mayor (y mucho mayor) que el del resto de los servicios del componente, para darle un valor numérico digamos que en el ejemplo que estamos considerando ahora la relación es del 80%.
TFT tiene un módulo analizador que detecta estas diferencias importantes en tiempo de ejecución y emite un alerta para dar aviso de ello a los analistas. Nótese que esa alerta en principio es preventiva, ya que el módulo no detecta ni avisa de ninguna falla sino de una experiencia distinta a sus experiencias anteriores, avisando que algo inexplorado está ocurriendo. Por ser un alerta, e incluso varias alertas del mismo tipo pudieron haber sucedido, supongamos que no se hace nada inmediatamente, en principio digamos que es un dato más. Pero seguramente en paralelo a ello o poco tiempo después se haya manifestado la degradación en la performance del sistema, es decir, un comportamiento inesperado reclamado por el usuario. Entonces, al analizar la situación, los posibles problemas y sus posibles causas y soluciones, el componente self-knowledge TFT "*Calculador*" puede ser consultado por "los últimos sucesos llamativos" lo que resultará en un informe sobre la experimentación de ese cambio importante en la relación de uso de los servicios del componente, sumado al registro reciente del cambio de entorno. Es decir, la técnica brinda información concreta y real sobre el uso del componente mostrando que el servicio *CalcExpresion* en usos anterior del componente era ejecutado sólo el 2% de las veces respecto de los demás servicios del componente en tanto que en este último sistema está siendo utilizado el 80% de las veces. Dado que el componente está siendo utilizado de una forma distinta a la que ha sido utilizado hasta ese momento, es posible que también este siendo utilizado de una forma distinta a la que sus diseñadores esperaban que fuera utilizado. Analizando conjuntamente con esa información su funcionalidad, así como el uso de otros componentes del sistema, ya sea utilizando para ellos los datos recolectados con la técnica TFT aplicada a este u otros componente, más otras técnicas de análisis cualesquiera que se quisieran utilizar puede llegar a saberse que: a) en la práctica resulta que en este segundo uso internamente, y de forma inadvertida e imprevista para los ingenieros de software que decidieron el reuso del sistema, el 80% de los cálculos se resuelven

a través del cálculo genérico (*CalcExpresion*), en tanto que en el uso anterior sólo el 2% de los cálculos se resolvían a través de él; b) el uso de los servicios del componente "*Calculador*" depende de las políticas de premios, cada una de ellas utiliza sólo un cierto conjunto de servicios de ese componente, lo que indica que las políticas de premios utilizadas en este segundo área son distintas de las primeras, y si bien las estrategias y el tipo de operaciones son del mismo estilo, ciertos casos o cálculos que antes eran excepcionales ahora son los casos más típicos; c) entonces, lo que sucede en particular en este caso, y que no fue tenido en cuenta, es que si bien ambas áreas utilizan el mecanismo de comisiones para incentivar a su grupo, cada una de ellas pone el foco en distintos objetivos, lo que se traduce en requerimientos distintos que si bien en un caso son coincidentes con las intenciones para la que ha sido desarrollado el componente, en el otro caso no.
De ese modo, la técnica ayuda a lograr un análisis rápido y fácil guiando al analista, dando pautas claras y precisas sobre la fuente y el origen del problema, y hasta brindando alguna pauta de posibles soluciones (optimización de función o sustitución del componente por un eficiente en dichos cálculos, o redefinición de las políticas de premios que involucran este tipo de cálculo, etcétera).

En un caso como este la técnica ha ayudado al usuario del componente, sin necesidad de recurrir al proveedor por un problema habitual de diferencias entre el propósito para el que fue creado un componente vs el propósito para el que es utilizado. Típicamente en los primeros usos esos propósitos coinciden, pero con el correr del tiempo, y alentado incluso por el éxito en esos usos, los componentes son reutilizados en nuevos sistemas o entornos o en casos tan simples como este sólo con nuevos usuarios que hasta parecen tener el mismo propósito. Pero resulta que en la práctica esa similitud no es tal en lo que al componente en si mismo se refiere, y así surgen problemas de lo más inesperados, y por ello muy difíciles de analizar.

6 IMPLEMENTACIÓN DEL FRAMEWORK TFT

En este capítulo mostraremos una posible implementación del framework de TFT. La implementación propuesta proporciona una forma genérica de recolectar información de un componente y su contexto en tiempo de ejecución, verificar desvíos importantes en ella en tiempo de ejecución, y la posibilidad de recuperarla luego de las ejecuciones. Todo ello sin estar atado a ningún modelo específico de componentes.

6.1 Descripción

La técnica TFT recolecta información del comportamiento, de las interacciones y del entorno de un componentes monitoreando la ejecución del CBS del que forma parte, cuando es posible resume esa información escogiendo datos relevantes y destilando información, principalmente estadística. Cuando el componente es reutilizado en otro sistema su información lo acompaña, y esa misma información es aumentada con la información monitoreada en la ejecución del nuevo CBS del que ahora forma parte el componente. Como esta información va acompañando al componente en su "viaje" a través de los distintos sistemas, y ella contiene datos y conclusiones de cada experiencia, llamamos a esta información *diario de viaje del componente*. Además, en tiempo de ejecución, la técnica incluye la realización de chequeos comparativos de desvíos importantes para dar alertas, y fuera de línea, permite consultar la información recolectada, puntual y comparativamente, para asistir a tareas de ingeniería de software como análisis, testing e integración, tanto para el mantenimiento de un CBS como para el desarrollo de otros CBS´s que reutilicen el componente. TFT se basa en siete fases principales que se repiten por cada entorno de ejecución en que el componente es utilizado:

- *Fase (1) Selección de los registradores (durante la pre-puesta en producción)*. TFT permite seleccionar qué clase de información será recolectada para el componente durante las ejecuciones del sistema que forma parte actualmente, es decir, permite definir cuáles serán las metadatos que conformarán la experiencia de la parte del diario de viaje del componente correspondiente al sistema en el que está siendo instalado el componente.
- *Fase (2) Generación e instalación de los registradores (durante la puesta en producción)*. TFT genera e instala pequeños módulos de software para el componente monitoreado a modo de wrapper formado de tres grupos principales: el registrador de estímulos, encargado de interceptar los servicios solicitados al componente, el registrador de interacciones, encargado de interceptar los servicios solicitados por el componente, y los registradores de metadatos para monitorear e interpretar la información a recolectar. Estos grupos, principalmente el tercero, están formados por varias entidades y módulos, y su número y capacidades están determinados por la selección de la fase anterior.
- *Fase (3) Captura de las ejecuciones (en tiempo de ejecución)*. Los registradores de estímulos, interacciones y metadatos capturan las interacciones entre el sistema y el componente monitoreado en tiempo de ejecución, capturando así datos de campo (field data) en verdaderos usos y verdaderos contextos del componente. El alcance, tamaño y comportamiento de estos registradores está en función de la selección hecha en la primer fase.
- *Fase (4) Filtros de comportamientos (en tiempo de ejecución)*. El monitoreo de la ejecuciones produce demasiados comportamientos, muchos de los cuales incluso no son especialmente interesantes. El submódulo filtro de cada registrador de metadato de TFT, en cierta forma y cuando es posible, escoge un subconjunto de ejecuciones notables para ser considerado y eventualmente almacenado por ejemplo filtrando algunos valores de entrada como casos de test de regresión.
- *Fase (5) Destilación de información de viaje (en tiempo de ejecución)*. El comportamiento del componente en cuestión y el de su entorno es destilado en formato de metadato. El submódulo destilación de cada registrador de metadato de TFT produce y deja en el diario de viaje del componente metadatos con información resumida o estadística como por ejemplo invariantes de I/O, invariantes de interacción, estadísticos de desempeño, recursos y contexto, es decir, datos que resumen las relaciones entre los pedidos del sistema y los resultados del componente, los patrones de interacción entre el componente y el sistema, y las cualidades funcionales y extra funcionales de la relación entre el componente y su contexto.
- *Fase (6) Verificación de desvíos y novedades (en tiempo de ejecución)*. TFT utiliza la información ya contenida en el diario de viaje para verificar el comportamiento del componente en tiempo de ejecución realizando comparaciones y análisis de desvíos entre

los nuevos valores de metadatos detectados en la ejecución actual y los valores de metadatos registrados en el diario de viaje del componente hasta el momento.

- *Fase (7) Consulta, análisis y verificación (durante el seguimiento, regresión o reutilización)*. TFT permite consultar el diario del componente en sus valores generales como así también en los valores de cada una de sus evoluciones, es decir por cada vez que fue reutilizado. Así permite realizar evaluaciones comparativas de los diferentes usos del componente a lo largo del tiempo y disponer de esta información para analizar y verificar las nuevas versiones o aplicaciones del componente o del sistema mismo.

La figura 4 muestra los principales módulos utilizados en cada fase de TFT y las diferentes etapas en las que la información del diario de viaje es utilizada.

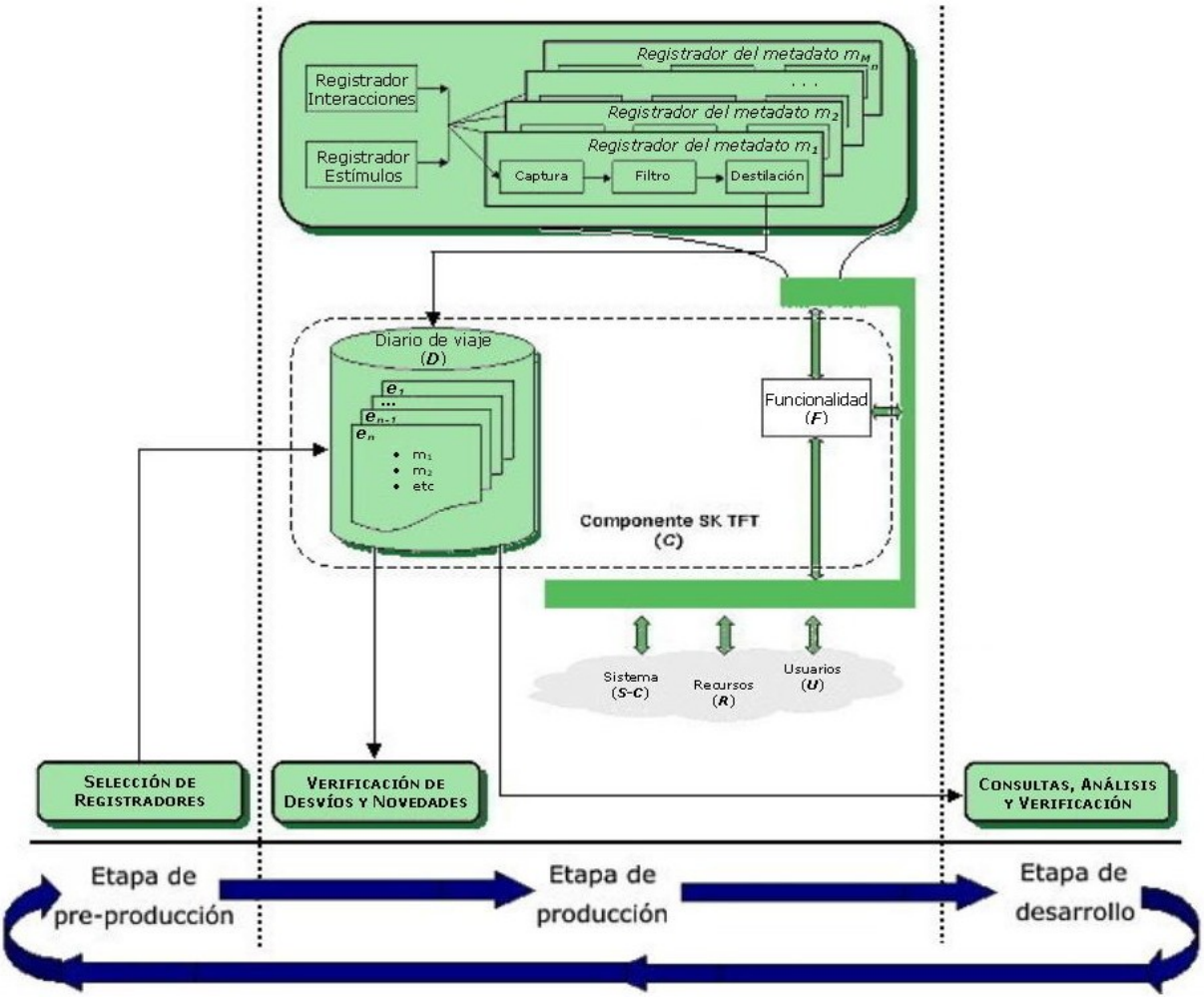

Figura 4:TFT: Selección y recolección de la información, y uso del diario del componente

Los módulos “Selección de registradores”, “Verificación de desvíos y novedades” y “Consultas, análisis y verificación” son las herramientas de TFT con las que interactúa y utiliza el usuario del componente. La primera, utilizada en la fase (1), le permite al usuario definir el subconjunto de metadatos $\boldsymbol{M_i}$, con $\boldsymbol{M_i} \subseteq \boldsymbol{M}$, para los que desea registrar valores durante las ejecuciones del componente en el nuevo entorno de ejecución $\boldsymbol{X_i}$ en el que estará éste cuando el sistema sea puesto en producción. Las dos últimas herramientas, utilizadas en las fases (6) y (7), le permiten al usuario consultar la información registrada en el diario de viaje. Describiremos más sobre estas tres fases en una algunas de las siguientes secciones de este capítulo.

Los módulos y submódulos que forman el wrapper del componente en producción son parte de TFT y son los responsables de monitorear, capturar, filtrar y destilar la información y los datos de campo hasta el diario de viaje del componente. El propósito y beneficio de utilizar un wrapper es que esos módulos pueden ser agregados o sacados en el entorno del usuario del componente sin necesidad de acceder al código fuente del componente monitoreado. Estos módulos son autónomos y no requieren, o más bien, han de implementarse de forma tal que no requieran, intervención del usuario del componente. En la fase (2), un módulo propio e interno del framework de TFT es el responsable de generarlos, instalarlos y acomodarlos a medida del componente y de la plataforma de producción, de acuerdo a los metadatos seleccionados en la fase (1). Para cada metadato seleccionado se pone en producción un registrador con los módulos y las entidades

necesarias para registrar información sólo sobre ellos, evitando así sobrecargas de instrumentación e información no solicitada por el usuario. Esencialmente, cada registrador de metadato está formado por tres submódulos: captura de la información (fase (3)), filtro (fase (4)), y destilación al diario (fase (5)). El objetivo de esta modularización por metadato tiene dos partes principales. Por un lado busca reducir el tamaño del wrapper, tanto en instrumentación como en la sobrecarga del componente en tiempo de ejecución, ya que sólo estarán presentes y funcionando las piezas necesarias para poder obtener la información que se solicitó registrar, y los módulos necesarios para registrar aquella información que no haya solicitada no estarán. Por otro lado, permite armar un diseño abierto de modo que la técnica pueda extenderse fácilmente en la clase y número de metadatos que podrían registrarse utilizando TFT ya que la capacidad de registrar un metadato no depende de otro.

Otro aspecto importante de las fases asociadas a los submódulos de los registradores de metadatos es cómo estos pueden partir de valores de campo monitoreados y llegar a valores de metadatos, es decir, propiedades de mayor nivel de abstracción. Veremos algunas posibilidades en la siguiente sección de este capítulo.

Así mismo, en las próximas secciones describiremos algunos aspectos importantes de la técnica necesarios para implementar este framework: (1) ¿cómo puede ser recolectada la información?, (2) ¿es posible recolectar toda y cualquier información deseada?, (3) ¿cuál es el formato más apropiado para la información recolectada?, y (4) ¿cómo adjuntar al componente la información recolectada, de modo tal que el usuario del componente pueda consultar el tipo de información disponible y recuperarlo de una forma conveniente?.

6.2 Monitoreo e interpretación

La creación y la autonomía de una base de conocimiento para un componentes en tiempo de ejecución, en particular, y en general la adaptación de sistemas en tiempo de ejecución, dependen de dos actividades principales: monitoreo e interpretación.

Monitoreo: debe haber alguna manera de observar el comportamiento de un sistema mientras está en ejecución. Sin esa información, no será posible determinar qué está sucediendo realmente y si se necesita dar una alerta o realizar un cambio. La infraestructura debe soportar una gran variedad de tecnologías de monitoreo. Dado que nuestro deseo es monitorear una gran variedad de aplicaciones, y dado que es probable que estas aplicaciones sean escritas en una variedad de lenguajes, es probable que se utilicen diferentes tecnologías de monitoreo. Por ejemplo debería ser posible para la infraestructura aprovecharse de tecnologías de monitoreo aplicables a componentes desarrollados por terceras partes tales como servicios que dan el estado del entorno en el cual un programa está ejecutándose. Los mecanismos de monitoreo pueden observar tanto el comportamiento del componente (monitoreo interno) como el comportamiento del entorno operativo (monitoreo externo). Según esta clasificación la técnica TFT incluye mecanismo de “monitoreo externo” y parcialmente de “monitoreo interno”, pues de las cuestiones internas del componente sólo observa cómo se comporta este visto desde afuera, es decir, respecto de su entorno.

Interpretación: La información recolectada debe ser aumentada e interpretada en el contexto de las propiedades y características del sistema del más alto nivel posible de uno o más modelos de alto nivel. Por ejemplo, un valor monitoreado que mide el ancho de banda entre dos nodos en una red no puede tener un significado apropiado hasta que su efecto en el sistema pueda ser determinado (por ejemplo, en términos de terminación o tiempo de respuesta).

Desde el punto de vista de ingeniería de sistemas y software, para llevar a cabo estas actividades hay diversas cuestiones que deben ser consideradas. Entre las que se plantean sobre el monitoreo tenemos:

- ¿Cómo agregamos capacidades de monitoreo a un sistema en forma no- intrusiva?
- ¿Qué tipo de cosas podemos monitorear?
- ¿Cuál es la tecnología apropiada para proporcionar monitoreo a una aplicación?
- ¿Cómo se puede dar soporte a una gran variedad de técnicas de monitoreo?
- ¿Cómo se puede agregar monitoreo a sistemas existentes de una manera que minimice el efecto sobre el sistema observado?
- ¿Es posible construir mecanismos de monitoreo reutilizables que puedan ser agregados fácilmente a sistemas existentes?
- ¿Cómo deberíamos diseñar los sistemas y los componentes para que puedan ser monitoreados más fácilmente?

Y algunas de las cuestiones que se plantean sobre la interpretación son:

- ¿Cómo le damos sentido a la información de bajo nivel monitoreada?

- ¿Qué clase de modelos de alto nivel deben ser soportados?
- ¿Cómo y cuándo debe comenzar esta interpretación, y cuán a menudo debe ser hecha?
- ¿Cómo podemos localizar con toda precisión la fuente de un problema?
- ¿Qué modelos están mejor apareados con atributos de calidad y sistemas específicos?

Uno de los puntos centrales para contestar estas preguntas es determinar qué clase de modelos debe utilizarse para interpretar la conducta observada. En principio muchas clases de modelos podrían ser utilizados, incluyendo modelos de comportamiento, de performance, de estructura, y de tiempo. Dado que la técnica se aplica a componentes, los modelos de arquitecturas y todos aquellos compatibles, coincidentes o equivalentes con las nociones básicas de los modelos de arquitecturas pueden, en primer instancia, ser considerados asociados a esta técnica, como por ejemplos modelos y técnicas de testing y verificación que manejen la noción de componente. Más aun, la elección del modelo no es restrictiva para el uso de la técnica, la información recolectada por el monitoreo puede ser útil para uno o más modelos, o para ninguno en particular sino que tiene valor informativo en sí mismo. La intención detrás de la idea de la elección de un modelo es por un lado tomar provecho de las nociones que ese modelo proporciona para la presentación de la información recolectada, brindando así mayor estandarización, utilidad, homogeneidad y claridad, y por otro lado poder elegir tecnologías de monitoreo ya existentes para determinados modelos. Además, en el caso de la técnica TFT el nivel de la información (propiedades) almacenadas en el diario de viaje debería ser tan alto como el nivel de los modelos que desea utilizar el usuario para la verificación y análisis del componente en los distintos escenarios que espera que sea de utilidad la técnica.

Así, para proporcionar un puente entre los valores observados a nivel de ejecución del sistema y los valores de mayor nivel de abstracción a registrar, una posibilidad es utilizar en TFT un método como el método de tres capas propuesto por Garlan [GSC/01], ilustrado en la figura 5.

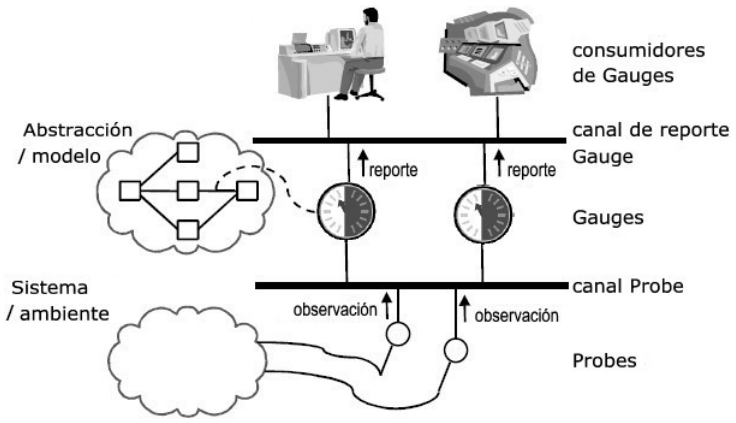

Figura 5:La infraestructura Gauge ([GSC/01])

En el nivel más bajo hay un conjunto de probes, los cuales están instalados junto con el componente o en el mismo ambiente físico, y estos anuncian las observaciones del sistema real vía un *canal probe* (*probe bus*). En el segundo nivel un conjunto de gauges consume e interpreta las mediciones de las probes de bajo nivel en término de las propiedades de modelos de un mayor nivel de abstracción. Al igual que los probes, los gauges difunden información vía un *canal de reporte gauge*·(*gauge reporting bus*). Las entidades de más alto nivel del gráfico son consumidores gauge, quienes consumen la información difundida por los gauges. Tal información puede ser utilizada, por ejemplo, para actualizar una abstracción o uno de esos modelos, para tomar decisiones de reparación del sistema, para mostrar advertencias y alarmas a los usuarios del sistema, o para mostrar el estado actual del sistema que está ejecutando; así por ejemplo las entidades de los registradores de metadatos de TFT formarían la capa del conjunto de consumidores gauge.

¿Qué son exactamente los gauges? Los gauges son entidades de software que recolectan, agregan, computan, analizan, difunden y/o representan información monitoreada en sistemas de software. Los agentes o herramientas de software, los ingenieros de software, y los operadores del sistema consumen dicha información, la utilizan para evaluar el estado del sistema y dinámicamente tomar decisiones. En su forma más pura, un gauge no cambia su modelo asociado ni controla al sistema de software directamente. Sin embargo, los outputs de un gauge pueden ser utilizadas por otras entidades para realizar tales cambios. Varios principios o suposiciones subyacen esta noción de gauges y han sido utilizados para guiar el diseño de la especificación de los gauges y las APIs gauge. Estas suposiciones incluyen:

1. El valor (o los valores) informado por un gauge puede tener múltiples consumidores.
2. Diferentes entidades / empresas pueden desarrollar diferentes tipos de gauges.
3. El conjunto de consumidores gauge puede cambiar dinámicamente.
4. Cada gauge tiene un tipo, el cual describe los requerimientos de setup y configuración del gauge, y los tipos de los valores que este reporta.
5. Los gauges están asociados a modelos.

Se identifica también la necesidad de ciertas entidades administradoras de gauges – llamadas gauge managers – desarrolladas para facilitar el control, la administración, y las consultas de meta-información de gauges. En el caso de TFT esto correspondería a los módulos de las fases (1) y (2), selección y generación de los registradores.

Esta infraestructura tiene varias características claves:

• los gauges están desconectados del sistema implementado (en virtud de la capa probes). Esta desconexión permite ejecutar los gauges en forma distribuida, de modo que ellos no afecten el desempeño del sistema que se esta midiendo, y también da la oportunidad de utilizar los informes de los probes de diversas maneras para modelos diferentes. Por ejemplo, los mismos outputs de un probe pueden ser utilizadas por gauges conectados a otros modelos o tipos de modelos, tales como Meta-H o UML o incluso, si fuera el caso, pueden ser utilizados por más de un componente.

• los gauges pueden mezclarse y pueden mapearse, dando soporte a la interoperabilidad entre gauges que evalúen propiedades bastante diferentes y que hayan sido desarrollados por organizaciones diferentes.

• los gauges están aislados de los mecanismos de transporte de más bajo nivel, permitiendo que los gauges sean instalados tanto en canales basados en RPC como en canales con mecanismos de publish-subscribe.

• los gauges pueden incorporarse a descripciones arquitectónicas, permitiendo la generación y la ejecución automáticas de gauges.

Esta es una descripción muy general y abstracta de la infraestructura propuesta por Garlan, poniendo énfasis en la generalidad del método para monitorear e interpretar los distintos metadatos tal como pretende cubrir la técnica TFT, puede encontrarse una descripción detallada de esta infraestructura de monitoreo en el trabajo [GSC/01].

Este tipo de método y todas las diversas cuestiones planteadas para el monitoreo y la interpretación que mencionamos más arriba, están siendo investigadas y aplicadas en profundidad y con la misma generalidad necesaria para TFT en trabajos como [GS/02] [CHGSS/04] y [MP/04]. En particular, el método Gamma de Orso et al. [OLHL/02] define una tecnología llamada "*tomografía del software*" (*software tomography*) que también puede ser aprovechada en la implementación de TFT ya que se alinea perfectamente con sus objetivos. Esa tecnología está organizada en tres fases, primero se dividen las tareas de monitoreo del software y recolección de la información de ejecución necesaria en un conjunto de subtareas que requieren sólo una mínima instrumentación, luego se asignan las subtareas a diferentes instancias del software de modo tal que ninguna de las instancias experimente una degradación significativa en su performance a causa de la instrumentación, y finalmente se integra la información devuelta por las diferentes instancias del software para recolectar la información global monitoreada. De este modo, la técnica permite recolectar información monitoreada utilizando una muy ligera instrumentación aprovechando el uso de muchos usuarios conectados a través de una red.

6.3 Información a recolectar

Cuando un componente desarrollado externamente se integra en un sistema, puede ser necesario realizar diversas tareas. Estas suelen requerir más que el código binario con alguna descripción de alto nivel. Más aun, el interés no está simplemente en tener un tipo específico de información del componente, como podría ser su especificación, sino contar con diferentes tipos de información dependiendo del contexto y las necesidades. Desde el punto de vista de las necesidades esta información podría ir desde modelos de máquina de estado finito del componente, hasta información relacionada a QoS, hasta documentación plana. De hecho, en esta propuesta consideramos que cualquier artefacto de ingeniería de software puede ser un *metadato*, es decir un dato del diario de viaje, la base de conocimiento, de un componente dado, siempre y cuando (1) sea obtenido durante la ejecución del componente o bien represente información del contexto de ejecución del mismo, (2) sea empaquetado con el componente en forma estándar, y (3) sea procesable por herramientas de desarrollo y entornos automáticos (incluyendo la posibilidad de presentaciones visualizables por usuarios humanos).

Sin embargo, sea cual sea la información que se recolecte en ***D*** para un componente ***C***, y sea cual sea el sistema ***S*** y ***X*** la definición de contexto, entorno o ambiente que consideremos, hoy por hoy

en la práctica no es factible recolectar todos los datos ni todos sus valores [PY/99] [OAH/03]. Entonces la propuesta es que la técnica TFT provea un mecanismo de selección respecto de qué información recolectar en las ejecuciones del componente, es decir, qué metadatos (propiedades) recolectará en sus ejecuciones. Esa selección puede tener una preselección hecha por el proveedor del componente ***C***. Esta preselección no será obligatoria ni determinante para la aplicación de la técnica, pero dar la posibilidad de que sea hecha permite obtener algún tipo de sugerencias del diseñador del componente sobre qué información podría ser relevante considerar en las ejecuciones de ***C*** teniendo en cuenta su conocimiento interno de ***C*** y las intenciones para las que fue creado ***C***. De todos modos, la selección final de los metadatos a recolectar durante la ejecución del componente siempre quedará en manos del usuario del componente, y ésta podrá ser redefinida cada vez que el componente ***C*** sea utilizado en nuevo contexto.
Entonces como dijimos anteriormente, toda la información recolectada en las ejecuciones de cada sistema S_i para un componente ***C***, y todo lo que pudiera resumirse o concluirse de todas esas ejecuciones puede ser pensado como una experiencia, a la que llamamos e_i con $1 \leq i \leq n$. Por otro lado, tal como estamos proponiendo, existen distintos posibles metadatos a recolectar, ese es el conjunto llamado ***M***, y de ese conjunto, para cada contexto X_i en el que el componente será utilizado, el usuario del mismo elige un subconjunto M_i, es decir $M_i \subseteq M$, que será el subconjunto de metadatos para el que se recolectará información durante las ejecuciones de ***C*** en X_i luego de ser puesto en producción. Así el usuario es quien elige qué metadatos tendrán valor en dicha experiencia, o lo que es lo mismo, cuáles serán los metadatos de esa experiencia. De esta forma, siendo el diario de viaje del componente $D = e_1 + e_2 + ... + e_n$, , la secuencia de experiencias e_i, con $1 \leq i \leq n$, y dado que las experiencias son funciones $e : M \rightarrow V$, tenemos que el dominio de cada una esas experiencias del diario de viaje son los conjuntos de metadatos M_i, con $1 \leq i \leq n$, respectivamente. Y diremos que las imágenes de esas experiencias son los conjuntos de valores de metadato V_i, respectivamente, es decir, los V_i son los valores adquiridos por los metadatos seleccionados en las experiencias correspondientes.

La decisión de cuáles metadatos recolectar y cuáles no puede tener diferentes motivaciones. Por ejemplo podrían agregarse propiedades a recolectar en el caso en que se hayan comenzado a manifestar situaciones que para su análisis requieran de información que aun no estaba disponible. O contrariamente, podrían dejarse de recolectar ciertas propiedades o porque ya se tiene suficiente información, o porque esa información ya no es necesaria para los futuros análisis previstos. En cualquier caso, siempre debe tenerse en cuenta que la disponibilidad de esos datos para futuras tareas dependerán de estas decisiones. En este punto la técnica tiene cierta flexibilidad que si bien consideramos necesaria para lograr la generalidad a la que aspira, entendemos que es una puerta abierta que podría debilitar las ventajas y los objetivos a los que aspira ya que pone en riesgo la característica de continuidad requerida por el Testing Perpetuo. Es decir, dejando la posibilidad de modificar los metadatos a recolectar en cada sistema, e incluso que la elección de ellos pueda modificarse en cualquier momento puede llevar a que la información recolectada no llegue a conformar un diario de viaje en el sentido esperado. Por ejemplo, si los invariantes de I/O se recolectan sólo en algún período de tiempo y luego no, o viceversa, puede que estos no reflejen el comportamiento del componente, y lo que es peor es que se utilicen como si sí lo reflejaran. De ese modo, este tipo de situaciones podría empeorar el resultado de las tareas que esperábamos beneficiar con esta información llevándolas a falsas conclusiones.
Otra forma completamente distinta de atacar el punto de qué metadatos conviene recolectar es proponer como parte de la técnica TFT un conjunto fijo de propiedades a recolectar en tiempo de ejecución justificando la conveniencia de disponer siempre de esa información. Pero, si bien esto puede resolver el problema de la continuidad antes mencionado, e incluso llegar con esto a una técnica mucho más acotada, y hasta en cierto sentido, hacerla parecer mucho más implementable, intuimos que sea cual sea el conjunto elegido no será posible justificar que ese sea el mejor conjunto para todo tipo de sistema y todo tipo de situación. Del mismo modo que ciertos tipos de sistemas requieren o sacan provecho de ciertos tipos de análisis o modelos, cada tipo de sistema requerirá o sacará provecho de cierto tipo de información recolectada en tanto que otra información recolectada no le será de utilidad, y en consecuencia el costo de conseguirla puede ser un precio que el usuario no esté dispuesto a pagar.
Entonces, lo que proponemos es que la técnica incluya varios conjuntos de metadatos a recolectar a modo de sugerencia para el usuario. Es decir, el usuario del componente self-knowledge TFT realiza la selección de qué metadatos recolectará la técnica durante las ejecuciones de su sistema, y para armar esa selección a aplicar a dicho componente parte de alguno de los conjuntos sugeridos pudiendo extenderlos con más metadatos, reducirlo o unirlo con otros conjuntos sugeridos.

De este modo, hasta donde estamos definiendo en este trabajo, la elección entonces queda en manos del usuario del componente. Este debe tomar solo la decisión de qué información recolectar. Pero se espera que esa decisión esté basada, y tome en consideración, principalmente y tanto como le sea posible (1) los objetivos y fines para los que ha sido concebida la técnica TFT (vistos en el capítulo ***5***. *Propuesta de la Técnica TFT*), (2) el tipo del componente, y la familia o estilo de sistemas de los que puede formar parte, y con ello los tipos de análisis que en el futuro querrán hacerse sobre el componente, (3) proveer suficiente información para hacer usable al componente

tan ampliamente como sea posible, ya que mientras más información esté disponible en el futuro menores serán las limitaciones de análisis posibles, y (4) aplicar una cota al punto anterior para hallar una relación costo-beneficio “good enough”.

El presente trabajo no incluye una descripción detallada y exhaustiva de los conjuntos de sugerencias de metadatos incluidos en la técnica. Consideramos que es necesario un trabajo de investigación importante e interesante para elaborar estos conjuntos, analizar sus utilidades, teóricas y prácticas, en profundidad. Investigar por ejemplo un conjunto mínimo de metadatos que pueda ser utilizado para ejecutar las tareas tradicionales de ingeniería de software, tales como testing, análisis, computación de métricas estáticas y dinámicas, y debugging. Más aun, hallar incluso relaciones entre esos conjuntos y taxonomías de componentes, o sistemas, por ejemplo podrían definirse conjuntos por estilos de arquitectura, conjuntos por tecnologías, conjuntos por tipo de errores o análisis, etcétera. De todos modos, en el próximo capítulo mencionaremos a modo de ejemplo algunos metadatos, propiedades, que podrían recolectarse y los casos en que esos podrían ser de utilidad.
También puede ser interesante investigar la posibilidad de brindar junto con la clase de metadato a recolectar algún tipo de información sobre el costo del mismo, y por qué no algún valor del beneficio, de modo de asistir al usuario en la decisión de la elección de los elementos que conformarán el diario de viaje del componente.

6.4 Formato de la información recolectada

La información recolectada y almacenada como base de conocimiento del componente es deseable que esté en un formato que sea lo suficientemente genérico como para poder ser utilizada por las diversas técnicas que podrían aplicarse en el desarrollo de los diferentes CBS´s que incorporarán al componente. Elegir un formato específico ajustable a todos los posibles tipos de datos es difícil. Como hemos dicho antes, de ninguna manera queremos restringir los datos posibles. Queremos tener la posibilidad de presentar cualquier clase de dato posible - desde especificaciones textuales de una funcionalidad, hasta un grafo de dependencias del componente, o alguna clase de información de tipos - en la forma de metadato. Por lo tanto, para cada clase de metadato, queremos (1) tener la posibilidad de usar el formato más conveniente, y (2) ser coherentes, de modo tal que el usuario (o la herramienta) que utiliza una clase específica de metadato sepa cómo manejarlo.
Esto es muy parecido a lo que pasa en Internet con los adjuntos del correo electrónico o con los archivos que se bajan a través del browser. Por esa razón la decisión ha sido aplicar la misma idea que hay detrás de los tipos MIME (Multi-purpose Internet Mail Extensions) para la definición del formato más apropiado para la información recolectada. Entonces, se define el tipo metadato como un tag compuesto de dos partes: un tipo y un subtipo, separados por una barra slash. Así como el tipo MIME “application/zip” le indica a un browser el tipo del archivo que se está bajando de una forma no ambigua, el tipo metadato “analysis/control-flow” puede decirle al usuario del componente (o a una herramienta) la clase de metadato recuperada, y por ende cómo manejarla. La información real contenida dentro del metadato puede entonces representarse de cualquier forma específica siempre y cuando seamos coherentes, eso es, siempre y cuando haya una relación uno-a-uno entre el formato de la información y el tipo de metadato.
Siguiendo este esquema, podemos definir un conjunto abierto de tipos que permita agregar nuevos tipos e identificar unívocamente la clase de metadatos disponibles. Así un metadato está compuesto por una cabecera, que contiene el tag que identifica su tipo y subtipo, y un cuerpo que contiene la información real, es decir, el valor del metadato.
Este formato se basa en el propuesto en [OHR/00]. Como se menciona allí, sería interesante investigar la posibilidad de utilizar XML [XML/00] para representar la información real contenida dentro de un metadato. Por medio de la asociación de un DTD (Document Type Definition) único a cada tipo de metadato, se podría brindar información acerca del formato del cuerpo del metadato en forma genérica y estándar.

En la técnica que estamos proponiendo definimos un formato único y un tag único para cada tipo de metadato provisto. De modo tal que el metadato pueda ser extendido fácilmente si se necesitan proveer nuevos tipos de metadato. Los tags le permiten al usuario del componente tanto tratar de forma correcta la información provista como metadato, así como consultar por una parte específica de la información. Esta es la idea detrás del concepto de metadato: definir una infraestructura que le permita agregar al componente y recuperar del componente, los diferentes tipos de datos que se necesitan en un contexto dado y una tarea dada.

Una característica adicional interesante es que el metadato puede producirse también para los componentes desarrollados internamente, de ese modo todos los componentes utilizados para construir una aplicación pueden manejarse de una forma homogénea.

6.5 Consulta de la información recolectada

Al igual que en la elección del formato del metadato, aquí también la idea es proporcionar una solución general que no restrinja los metadatos que se puedan manejar. En particular, queremos ser tan flexibles como sea posible con respecto a la forma en que puedan agregársele nuevos metadatos a registrar (propiedades) a un componente y en la forma en que el usuario del componente pueda recuperar esa información. Esto puede lograrse proporcionándole al módulo administrador de la base de conocimiento del componente self-knowledge TFT dos métodos: uno para preguntar acerca de las experiencias y los metadatos disponibles en ellas, y otro para recuperar un metadato específico. Así, dicho módulo de TFT del componente se encargaría de implementar estos dos métodos adicionales en una manera conveniente. Cuando el usuario del componente desee realizar alguna tarea que involucre uno o más componentes self-knowledge TFT, entonces puede determinar qué clase de datos adicionales necesita, consultar la base de conocimiento de cada componente, y recuperar el valor del metadato apropiado si ellos están disponibles.

A su vez, la propuesta es desarrollar una herramienta de análisis construida sobre la base de conocimiento creada por TFT que permita hacer evaluaciones globales y comparativas sobre las experiencias de un componente. Es decir, una herramienta que permita observar el diario de viaje del componente construido hasta el momento en su totalidad y desde diferentes ángulos. Supongamos un componente self-knowledge TFT utilizado en 4 contextos, eso es $n = 4$, y por lo tanto $\boldsymbol{D} = \boldsymbol{e_1} + \boldsymbol{e_2} + \boldsymbol{e_3} + \boldsymbol{e_4}$, y todas esas experiencias con el mismo dominio $\boldsymbol{M_1} = \boldsymbol{M_2} = \boldsymbol{M_3} = \boldsymbol{M_4} = \{m_1, m_2, m_3, m_4, m_5, m_6\}$, es decir, habiendo seleccionado los mismos 6 metadatos para las ejecuciones de los 4 contextos. Una representación gráfica del diario brindada por la herramienta podría tener el aspecto de la figura 6.

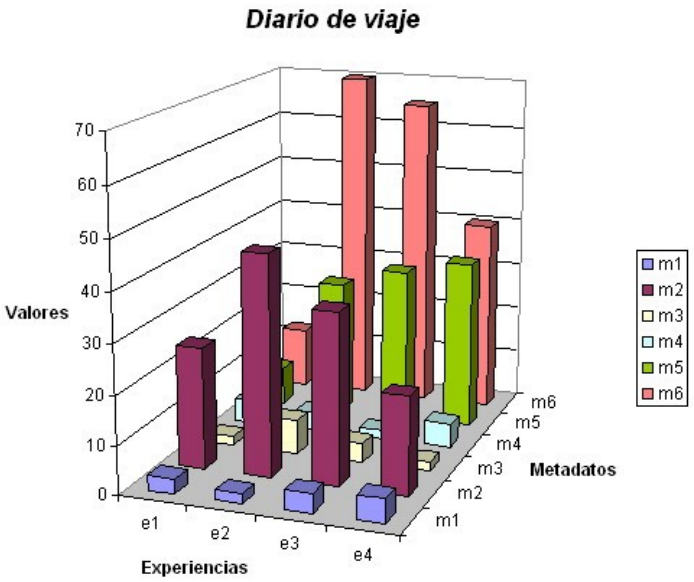

Figura 6:TFT: Relación de metadatos y experiencias según el diario del componente self-knowledge TFT

Nótese que en el gráfico se asume que de alguna forma los valores de todos los metadatos han sido transformados a algún dominio numérico y comparable. Esto no necesariamente debe ser así, el gráfico sólo busca dar una idea general del tipo de cruce de información que espera poder hacerse sobre el diario de un componente self-knowledge TFT. Es decir, visto de esta forma el gráfico podría ser consultado a largo o a lo ancho como lo muestra la figura 7. Un consulta *a lo ancho* corresponde a preguntarse: ¿qué metadatos se recolectaron en la experiencia $\boldsymbol{e_i}$ para algún $1 \leq i \leq n$?; una consulta *a lo largo* corresponde a preguntarse: ¿cuáles fueron todos los valores experimentados por el componente en el metadato m_j para algún $1 \leq j \leq \#\boldsymbol{M}$?, como por ejemplo: ¿cuáles fueron todos los sistemas operativos en los que fue ejecutado el componente?.

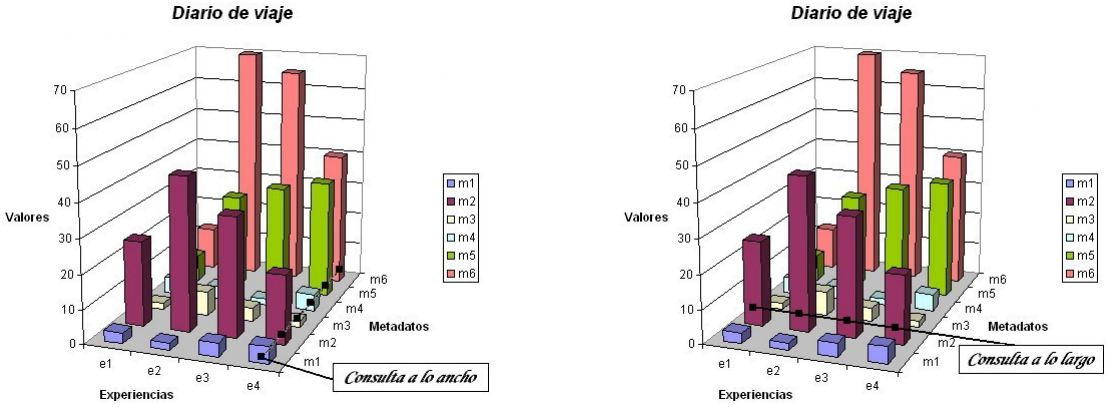

Figura 7:TFT: Consultas del diario de viaje del componente self-knowledge TFT

Veamos, un ejemplo de metadato podría ser: "los invariantes de I/O construidos a partir de usos reales del componente", y sería interesante hacer una consulta a lo largo de un metadato como este para poder ver los invariantes de cada experiencia, dicho de otra forma, esto equivale a ver los dominios en los que ha sido ejercitado realmente el componente. Este tipo de información puede ser de suma utilidad a la hora de reutilizar el componente. Por otro lado, como decíamos anteriormente, no parece tener sentido ni ser necesario (al menos hasta donde llega el interés del presente trabajo) lograr alguna comparación puntual o detallada de este tipo de metadato con otros tipos de metadato como por ejemplo el de "los sistemas operativos en los que ha ejecutado el componente". Nótese que aun así, la consulta a lo ancho también es de utilidad, ya que permite hacer algún análisis, aunque sea manual e intuitivamente, sobre alguna relación entre diferentes metadatos, como por ejemplo ver en qué sistemas operativos logró, experimentó, mejores o peores desempeños, y ello puesto en consideración por ejemplo del tiempo o la cantidad de veces que ha sido utilizado el componente, o el tamaño del sistema, o la cantidad de usuarios simultáneos, etcétera.
Entonces todos estos tipos de consultas son de utilidad principalmente a la hora de reutilizar el componente, es decir, evaluando qué se espera de él de acuerdo a cómo ha sido su comportamiento en entornos anteriores, o comparando varios componentes para decidir cuál se acerca más a lo esperado en un nuevo sistema.

Otra forma de consultar la información recolectada en las ejecuciones es solicitar las últimas novedades o situaciones notorias experimentadas por el componente self-knowledge TFT durante la ejecución en el contexto actual. Es decir, cómo funciona la técnica en este punto: el componente va registrando sucesos de su entorno, y en cualquier momento puede ser consultado por los "*últimos sucesos más llamativos*", eso es, tomar los metadatos recolectados en el contexto actual, de alguna forma, compararlos con los de experiencias anteriores, y mostrar aquellos con "diferencias" "significativas". Así, para analizar una falla o un comportamiento inesperado en un sistema, podrían consultarse los componentes self-knowledge TFT sobre sus "*últimos sucesos más llamativos*", lo que en cierta forma equivaldría a preguntarle a los habitantes de una casa donde ha ocurrido un incidente o un hecho delictivo "*¿usted notó algo diferente en los últimos días?*". Aunque no se espera que sus respuestas resuelvan el caso inmediatamente, son de suma utilidad para la investigación las pistas que sus respuestas pueden darle a los investigadores, quienes no sólo consultarán a los habitantes de la casa sino incluso a los vecinos, o lo que en la técnica sería, consultar el diario de viaje de más de un componente.
Así, al consultar por las últimas novedades, la técnica puede brindar información como por ejemplo: "el servicio ***s*** que ha sido muy poco utilizado en experiencias anteriores ahora es muy utilizado", o "el tiempo de respuesta del componente ***B*** (usado por el componente ***C***) ha decaído un 50% comparado con experiencias anteriores", o "el número de réplicas del componente es 80 veces más grande que en las experiencias anteriores", etcétera.
Este tipo de consultas puede ser de utilidad en la puesta en marcha de sistemas que reutilizan componentes, y en las tareas de mantenimiento y administración de esos sistemas, como puede ser el análisis de fallas o degradaciones o comportamientos inesperados. En etapas tempranas de la puesta en marcha de un sistema, los resultados de estas consultas pueden ayudar a entender y analizar diferencias entre el entorno de desarrollo y el de producción, y al descubrimiento de suposiciones (assumptions) y diferencias entre los propósitos para los que ha sido desarrollado el componente y aquel para el que es utilizado que no estuvieran documentados o que no hubieran sido tenidos en cuenta al momento de reutilizar el componente. En etapas más avanzadas de la puesta en producción de un sistema, los resultados de estas consultas pueden ayudar a entender y analizar cambios en el entorno o nuevos usos o formas de uso inesperadas.

6.6 Cuestiones técnicas

Desde el punto de vista teórico, en lo que respecta a la ubicación de ***D***, el diario de viaje o base de conocimiento de un componente self-knowledge TFT ***C***, podría estar físicamente junto con el componente, o estar almacenado remotamente para que el componente lo recupere cuando le sea solicitado, o sea evaluado por demanda. En principio consideramos que el lugar de almacenamiento de ***D*** será un aspecto que tendrá que ver con decisiones de implementación de la técnica TFT y decisiones de implementación asociadas a las cuestiones técnicas y tecnológicas de ***C*** y su entorno, pero desde el lado de la definición teórica de TFT no debería haber ninguna restricción al respecto. Sí debe tenerse en cuenta que a los efectos de la técnica, cualquiera sea la ubicación física de ***D***, la semántica de la información contenida en él siempre será desde el punto de vista de ***C***, es decir, desde la ubicación real que tuvo ***C*** mientras era ejecutado y la información era recolectada, tanto en lo temporal como en lo espacial.

En lo que respecta al espacio y la performance para utilizar datos de campo (field data), la técnica debe restringir tanto la instrumentación requerida para recolectar los metadatos como los metadatos recolectados para cada ejecución [OAH/03]. Sería deseable entonces que la técnica requiriera sólo una ligera instrumentación, lo que dependerá de la implementación de la técnica y las tecnologías asociadas. Del mismo modo sería deseable que la técnica TFT recolectara metadatos en el orden de unos pocos kilobytes para cada ejecución, lo que dependerá en parte de la implementación y de las tecnologías asociadas al monitoreo e interpretación (probes, gauges, gauges consumers TFT), y dependerá de la selección de los metadatos a recolectar que efectúe el usuario del componente en la fase (1) de cada nuevo entorno de ejecución de dicho componente.

En lo que respecta a las cuestiones técnicas y tecnológicas de las fases, su implementación, la implementación del wrapper, el proceso de selección, la implementación del monitoreo, la interpretación y abstracción de la información, su manipulación, escalabilidad, etcétera, muchos son los desafíos que se plantean y que requieren ser resueltos para poner en práctica técnicas como la técnica TFT. Mencionaremos algunos de los desafíos más sobresalientes unos capítulos más adelante.

6.7 Contribución y ventajas

La técnica TFT combina varias tecnologías para formar conocimiento a partir de ejecuciones reales de los componentes para la verificación y análisis de sistemas basados en componentes. A nuestro entender, es el primer trabajo que combina estas tecnologías con este objetivo. Por otro lado, la técnica TFT utiliza ideas y técnicas propuestas en trabajos anteriores.
En su aspecto más general, la técnica desarrollada toma las ideas subyacentes en el testing perpetuo de Pavlopoulou y Young acerca de extender el testing hasta la operación normal del software [PY/99], compartiendo y delimitando así el área de interés, el marco de investigación y la motivación. Las dos características fundamentales del testing perpetuo es que debe ser continuo e incremental [OST/96]. La técnica TFT recolecta información de experiencias reales, de contextos reales con usuarios reales, y esa información crece en cada evolución del componente, por lo tanto esta técnicas contribuye muy bien como un mecanismo para sostener el testing perpetuo: el sistema está en testing permanente, a lo largo de toda su vida útil, y el conocimiento adquirido en ella es siempre acumulativo.
El diseño e implementación de la técnica toma ideas y métodos de otras propuestas, ya sea parcialmente o extendiéndolas o combinándolos para lograr los objetivos de TFT. Así, la técnica TFT aprovecha las propuestas de monitoreo genéricas de Garlan [GSC/01]. Además extiende los fines del framework de metadatos de Orso et.al. [OHR/00], compartiendo sus utilidades y reutilizando el formato genérico para la representación de la información del componente. La técnica TFT comparte la idea de utilizar los datos recolectados in-field para testear y verificar sistemas de software con otras investigaciones: testing remoto, datos de campo (field-data) y equivalencia de comportamientos. El Testing Remoto ANTS dependen del uso real de un grupo de usuarios seleccionados para testear la navegabilidad de sitios web [ROD/02]. El método Gamma de Orso et al. recolecta datos de campo del software puesto en producción, para darle soporte tanto al análisis de impacto como al test de regresión [OAH/03]. El método BCT de Mariani utiliza datos de campo para construir y obtener información del comportamiento de los componentes a fin de testear y verificar automáticamente equivalencia de comportamientos en sistemas de software basados en componentes [MP/04]. A diferencia de otros métodos, TFT permite enriquecer componentes en forma autónoma con información recolectada a partir de datos de campo, formando una base de auto-conocimiento (self-knowledge) que evoluciona en cada nuevo entorno o escenario del componente para dar soporte a las tareas de ingeniería de software principalmente a las vinculadas con la evolución del componente, como reutilización y actualizaciones.

En resumen, las principales contribuciones de la técnica TFT son:

- la combinación de monitoreo, recolección, verificación y testing en tiempo de ejecución para verificar comportamiento de sistemas basados en componentes,
- la aplicación de información computada en ejecuciones en contextos anteriores para identificar diferencias de comportamiento y entorno entre diferentes versiones de un sistema de software, revelando así posibles inconsistencias y defectos tanto en el uso como en la reutilización de componentes,
- una forma de generar información del lado del usuario del componente disminuyendo los problemas asociados a la falta de información en el desarrollo de CBS´s,
- una forma de disminuir del lado del usuario del componente los problemas asociados a la dependencia del proveedor del componente en el desarrollo de CBS´s,
- la combinación de conceptos de self-healing systems e informática autónoma con desarrollo de sistemas basados en componentes enfocados en la construcción de una base de conocimiento y su proceso de aprendizaje,
- la definición de componente self-knowledge que incluye información de sus ejecuciones como parte de si mismo, y la introducción de la noción de *reuso de sus ejecuciones* al ser reutilizado el componente,
- una noción posible de base de conocimiento, definiendo qué la constituye y dándole un carácter evolutivo que contribuye y a su vez se alimenta de la evolución misma del componente.

Y las principales ventajas de la técnica TFT son:

- es autónoma,
- permite reducir los costos para obtener información,
- permite obtener comportamiento de usos reales, y por lo tanto comportamientos reales,
- brinda la posibilidad de llevar a cabo tareas de análisis y testing aun con componentes desarrollados por terceras partes,
- brinda la posibilidad de descubrir suposiciones implícitas (assumptions),
- en algunos casos brinda la posibilidad de acotar tareas de análisis y testing, lo que produce una reducción en los costos de las mismas,
- se basa en un formato genérico que puede ser extendido fácilmente para soportar nuevos metadatos,
- se basa en un formato genérico que puede ser fácilmente utilizado por herramientas automatizadas para tareas de desarrollo, testing y análisis.

7 INFORMACIÓN

Un tema importante a analizar en esta propuesta es el de *la información*, es decir, qué metadatos conforman la base de conocimiento de un componente self-knowledge TFT. En el capítulo anterior hemos discutido desde el punto de vista de la propuesta de la técnica TFT los asuntos relacionados a cómo puede ser recolectada dicha información, qué técnicas y tecnologías existen al respecto, cuál es el formato más apropiado para esa información, es posible recolectar toda y cualquier información deseada. En este capítulo, para definir qué metadatos conforman la base de conocimiento de un componente self-knowledge TFT, nos centraremos en la pregunta qué información sería útil recolectar y consideraremos la pregunta qué información es posible recolectar en tiempo de ejecución en el contexto de la respuesta a la pregunta anterior.

En primer lugar, para responder a la pregunta "qué información sería útil recolectar" debemos recordar "para qué deseamos utilizar" esa información, es decir considerar qué metadatos, propiedades, recolectar desde el punto de vista de su utilidad, para qué pueden servir, y no recolectar por el solo hecho de recolectar. En términos generales vimos en la sección ***5.2*** Motivación: información necesaria, útil y posible que a un usuario genérico de un componente le puede ser de utilidad la siguiente información:

1) información para evaluar el componente
2) información para integrar el componente
3) información para poner en producción el componente
4) información para testear y depurar el componente
5) información para analizar el componente
6) información sobre cómo customizar / personalizar o extender el componente

Y en el contexto del presente trabajo donde en particular estamos definiendo la técnica TFT, hemos dicho que deseamos recolectar información de las ejecuciones del componente en los distintos sistemas y entornos de los que este va formando parte para verificar la calidad de los CBS´s, centrándonos principalmente en tareas relacionadas al testing, análisis, reuso, integración, suposiciones y entorno. Más aun, en la sección ***5.5*** Escenarios hemos propuesto los principales escenarios en los que esperamos que la información recolectada con la técnica sea de utilidad. A saber:

(a) cuando un componente ***A*** que forma parte de un sistema en producción ***S*** se añade a un nuevo sistema ***S'***.
(b) cuando un nuevo componente ***B*** reemplaza a un componente ***A*** en un sistema ***S***.
(c) cuando un sistema ***S*** que incluye a un componente ***A*** es instalado en un nuevo ambiente de producción.
(d) cuando un componente ***B*** que forma parte de un sistema ***S'*** reemplaza a un componente ***A*** en un sistema ***S***.

Así, tenemos que, será de utilidad toda aquella información que contribuya, aporte, ayude o facilite las tareas asociadas a la verificación de calidad de los CBS´s, y la del componente mismo, en cualquiera de los escenarios anteriores.
Esta definición es sumamente amplia, y mucha información puede satisfacer estas necesidades. Incluso muchas técnicas existentes ya nos proporcionan una buena cantidad de ella. Pero en el marco de la técnica TFT ese conjunto está más acotado. Por un lado está acotado siguiendo los objetivos propios de TFT: superar obstáculos como la falta de información, y las causas y consecuencias de las inconsistencias por suposiciones, principalmente aquellas asociadas al entorno, poniendo el foco en los problemas con los que puede toparse el usuario del componente, sus situaciones y necesidades, y sus posibilidades y alternativas para superarlos independientemente del proveedor del componente. Por otro lado está acotado considerando información recolectable en tiempo de ejecución y sin intervención humana, es decir información que fundamentalmente se alinee con la característica de autonomía de la técnica. Y por otro lado está acotado considerando información acumulable que permita generar una base de conocimiento (know how) a lo largo de la vida útil del componente a fin de sacar el mayor provecho posible de la evolución misma del componente, y que a su vez colabore con esa evolución. Teniendo en cuenta estas consideraciones, el conjunto puede que sea aun muy amplio, pero nos da una pauta más clara de qué información será útil recolectar utilizando la técnica TFT.

Respecto del carácter evolutivo de la información, justamente Tosi menciona en [TOS/04] como parte de la característica self-healing de los sistemas self-managed: "En nuevos entornos o diferentes escenarios, nuevos comportamientos del sistema pueden ser observados y el módulo de conocimiento debe evolucionar con el entorno". De este modo, desde el punto de vista de utilidad,

otra arista interesante del carácter evolutivo es que la información así recolectada puede que a lo largo de una y otra y otra experiencia pase a ser información bien conocida por el componente, algo así como obvia, entonces claramente la información a recolectar también debe evolucionar, y también a su vez atendiendo a nuevos problemas en estás áreas, nuevas propiedades deben recolectarse y utilizarse. Por ejemplo, muy posiblemente en sistemas y/o componentes cada vez más autónomos y con capacidades self-healing como se esperan que sean los sistemas en los próximos años, ya será posible, y hasta necesario, recolectar datos como porcentaje de veces recuperado autónomamente de una falla, porcentaje de veces en que la estrategia auto-propuesta sirvió como solución, porcentaje de veces que requirió intervención humana, etcétera. Por lo tanto, esos podrán ser algunos de los nuevos metadatos que formarán parte del diario de viaje del componente. A su vez esos metadatos serán consultados y utilizados en los análisis para testear o reutilizar el componente en los escenarios subsiguientes, lo que a su vez hará evolucionar al componente nuevamente, y alimentará y aumentará su base de conocimiento y en consecuencia habrá evolucionado el conocimiento también. Cabe destacar que gracias al diseño del framework de la técnica TFT, en lo que se refiere tanto al formato elegido para representar la información, como a la generalidad e independencia de los módulos respecto de la información particular que se esté recolectando, es totalmente factible evolucionar la información posible de recolectar con la técnica. Más aun, es posible evolucionar la información de un componente self-knowledge TFT, es decir, un componente que ya tenga su diario de viaje construido.

Ahora bien, en la búsqueda de definiciones más concretas del tipo de información a recolectar, diremos que estamos interesados en capturar comportamientos, características, información tanto del componente como de su entorno, donde por entorno pensamos tanto en el resto del sistema del que forma parte el componente como en el entorno operativo del sistema, también llamado contexto o ambiente. Entonces, en primer lugar podremos distinguir la información en metadato “interno” o metadato “externo” según haga referencia al componente en si mismo o al ambiente en el que está ejecutando, respectivamente.
Otra visión de esa misma información, teniendo en cuenta que ésta es recolectada por monitoreo en tiempo de ejecución, es distinguir lo que “sí” es utilizado en las ejecuciones de lo que “no” es utilizado en las ejecuciones. Dependiendo del mecanismo empleado para ver esta información puede ser que lo “no” utilizado en las ejecuciones no llegue a distinguirse (justamente por no haber sido usado) pero veremos más adelante con algunos ejemplos que aun así puede ser provechoso al menos conocer lo que “sí” fue utilizado ya que eso, en cierta forma, equivale a saber con exactitud qué fue necesario hasta el momento, o desde cierto punto de vista qué fue “lo importante” (ya que fue usado), o qué fue probado en producción, etcétera. Además es habitual que el universo completo sea posible conocerlo empleando otros medios, pero no es habitual conocer el subconjunto verdaderamente utilizado en tiempo de ejecución hasta el momento. Aplicado a metadatos “internos” un ejemplo posible es que típicamente se tiene la lista completa de los servicios de un componente pero no siempre se tiene el subconjunto de servicios efectivamente utilizados en tiempo de ejecución. Y aplicado a metadatos “externos” un ejemplo puede ser tener la lista completa de los periféricos conectados o activos en el ambiente, pero no tener el subconjunto de periféricos efectivamente utilizados por un componente en sus ejecuciones.
Y por último, independientemente de si la información se refiere a un metadato “interno” o “externo”, o si refiere a lo “sí” utilizado o a lo “no” utilizado, y teniendo en cuenta que comúnmente la información se referirá a algún elemento de interés, otra visión de esa misma información es considerar tres aspectos distintos de un elemento: 1) “qué es” (asociado a “qué hay”), es decir la definición del elemento, por ejemplo “es un componente”, o “es un base de datos”; 2) “cómo es”, es decir la descripción de sus características, típicamente más bien información estática, por ejemplo “es un componente con los servicios s_1 y s_2”, o “es una base de datos Oracle 10*g* Release 2”; y 3) “cómo se comporta”, es decir la descripción de su comportamiento, cómo interactúa el componente o con el componente en tiempo de ejecución, típicamente aspectos dinámicos, por ejemplo patrones de comportamiento de cada servicio del componente, o “el tiempo de respuesta de la base de datos siempre estuvo entre t_1 y t_2”. Este tercer aspecto parece ser el más interesante o útil, pero los dos anteriores por un lado son de utilidad para describir a este último, y por otro lado pueden ser de mucha utilidad en sí mismos en los escenarios planteados. Un poco más adelante veremos algunos ejemplos de ello.

Hasta acá hemos visto qué tipo de información sería útil recolectar y qué queremos recolectar durante las ejecuciones de un componente. También hemos visto algunas formas de clasificar o distinguir esa información. A continuación presentaremos algunos ejemplos concretos de información que puede recolectarse y mostraremos casos de uso, es decir situaciones habituales en el desarrollo de CBS's, incluso algunas tomadas de experiencias reales, en las que describiremos un problema o necesidad proponiendo qué metadatos registrar y viendo cómo esos metadatos recolectados con la técnica TFT pueden asistir a las tareas de verificación involucradas.

7.1 Casos de uso de la técnica TFT

Para la presentación de los casos de uso de la técnica TFT hemos agrupado los ejemplos en *clases* que reúnen tipos de información o metadatos afines entre si, de modo que todos los ejemplos de una clase están asociados a un tipo de propiedad. Las clases llevan el nombre de ese tipo de propiedad y no intentan ser taxonómicas sino ayudar a la claridad de la presentación. Los distintos metadatos recolectados en cada clase difieren por ejemplo en el nivel de detalle, o en la forma de describir la propiedad, o el nivel de abstracción, o el elemento sobre el que se pone el foco, etcétera.
Tanto las clases como los ejemplos de metadatos a registrar son sólo algunos de todos los posibles, no son exhaustivos ni sería acorde a la técnica buscar algún conjunto completo. Todos ellos fueron elegidos por su carácter ilustrativo del uso de la técnica TFT y para mostrar en la práctica algunas de las ventajas de la técnica TFT.
También cabe destacar que todos los metadatos recolectados pueden ser de utilidad en todos o casi todos los escenarios de la técnica. Teniendo en cuenta que el objetivo central de este capítulo es mostrar qué información puede recolectarse en tiempo de ejecución con la técnica TFT y cómo los datos de campo pueden ser de utilidad, los casos elegidos muestran la utilidad de los metadatos sólo en algún escenario en particular, y si bien en algunos casos se dan las pautas para plantear fácilmente situaciones análogas en otros escenarios queda a cargo del lector desarrollarlo en forma completa.
Entonces estos son sólo algunos ejemplos de toda la información que puede ser útil recolectar. Los ejemplos son generales y no tienen un análisis riguroso formal. No pretenden mostrar el mejor uso de la información recolectada, posiblemente uno o más metadatos recolectados sirvan para análisis similares, o diferentes modelos, pero no exploraremos aquí todas las posibilidades ni las ventajas entre unos y otros, ya que ello excede el alcance del presente trabajo.

Clase: Performance

¿Qué metadatos recolectar? Registrar estadísticos e indicadores de performance como tiempo de respuesta mínimo, tiempo de respuesta máximo y tiempo de respuesta promedio en el sistema actual para:

- Servicios propios
- Servicios requeridos a otros componentes
- Recursos accedidos
- Servicios requeridos al entorno operativo

Pautas de cómo conseguir en tiempo de ejecución esa información: Una implementación muy sencilla es un timer en el wrapper de TFT que se activa al ser invocado el servicio y se desactiva justo antes de terminar. Desde el punto de vista técnico, como hemos dicho, es necesario tener en cuenta especialmente la sobrecarga del instrumento de monitoreo y registración (wrapper) en sí mismo. Pero si los valores son utilizados para compararlos entre sí, y no como valores absolutos, como es el caso de la comparación entre experiencias del mismo componente, esta sobrecarga no afectaría el significado del resultado de la comparación si todos son tomados con el impacto del wrapper de TFT incluido.

Caso de uso: *Cambio de entorno operativo*
En una empresa de servicios, uno de los sistemas de información (por ejemplo, el sistema de facturación o el sistema de cobranzas) tiene un módulo donde uno de sus servicios permite realizar la impresión de un comprobante. Internamente, para llevar a cabo esa impresión, el módulo realiza la búsqueda de un archivo de fonts a través de un servicio del sistema operativo, y si este responde que el archivo no se encuentra, el servicio del módulo resuelve su funcionalidad con una alternativa emitiendo una respuesta correcta desde el punto de vista de la necesidad del usuario, por lo que dicho fracaso de búsqueda es totalmente transparente para el usuario, tan transparente que ni usuarios ni administradores técnicos del sistema saben de la búsqueda de ese archivo y su fracaso.

Por razones de mantenimiento, en el equipo en el que se encuentra instalado ese sistema de información se instalará una versión más actualizada del sistema operativo. Este es el típico caso de actualización del sistema operativo o de algún componente del entorno, en el que se espera que nada cambie en su comportamiento, sino que en todo caso mejore la performance o la confiabilidad o lo que el proveedor haya mencionado que mejorará, pero eso no sucede así y en cambio comienzan a surgir “problemas” donde antes todo funcionaba perfecto, y sin saber por qué exactamente. Las respuestas en muchos casos suelen estar en las suposiciones que el sistema y cada uno de sus componentes hacen de ese entorno.
Lo que sucede en particular en este caso es que el servicio de búsqueda de archivo de la versión actual del sistema operativo en el que se encuentra instalado el sistema, y en todas las versiones anteriores en las que ha ejecutado este módulo, cuando no encuentra el archivo solicitado responde inmediatamente al módulo que no lo puede encontrar, y así el módulo internamente resuelve su funcionalidad con la alternativa propia antes mencionada. En la versión más

actualizada del sistema operativo, dicho servicio de búsqueda al no encontrar el archivo, en vez de "darse por vencido", realiza otros intentos disparando otros procesos de búsqueda, que casualmente en este equipo (entorno) tampoco encuentran nada en este caso en particular, y luego de varios intentos terminan avisando que no puede hallar lo solicitado y el módulo en cuestión nuevamente resuelve su funcionalidad con la alternativa propia como lo hacía hasta ahora. Así, el resultado final es que al instalar el sistema en esta nueva versión del sistema operativo la nueva forma de comportarse del servicio de búsqueda provoca por un lado una degradación en la performance del componente, ocasionada por una demora en la respuesta del sistema operativo a lo requerido por él, y por otro lado una degradación de todos los procesos del equipo (con quienes comparte recursos como el procesador). Entonces, la actualización tiene un impacto negativo considerable ocasionando problemas en la operatoria general de la empresa.

¿Cómo podría ayudar en un caso como este la técnica TFT? Recordemos cómo funciona la técnica. El componente va registrando sucesos de su entorno, en este caso, entre otros metadatos, estamos considerando que se registra información sobre la performance de los servicios del componente y lo requerido por este. Además, dado que ha sido instalado en un nuevo sistema operativo, y por ende ha cambiado su entorno (escenario c), se ha registrado un hito, nueva experiencia, en el diario de viaje del componente. En este nuevo contexto consideramos que el componente también va registrando, entre otros metadatos, información sobre la performance de los servicios del componente y lo requerido por este. Además, en cualquier momento el componente puede ser consultado por los "últimos sucesos más llamativos", en este caso, frente a la degradación del equipo o la degradación del sistema, cualquiera haya sido lo que el usuario haya detectado primero, el módulo en cuestión será consultado por sus últimos sucesos más llamativos, lo que resultará en un informe sobre la experimentación de un cambio importante en la performance del servicio de impresión del módulo así como en la performance del tiempo de respuesta del servicio utilizado del sistema operativo, sumado al registro reciente del cambio del entorno (donde el entorno registrado podemos pensarlo como los componentes típicamente conocidos como influyentes en el sistema (sistema operativo, base de datos, memoria, etcétera), o como los componentes conectados, o como recursos disponibles – veremos más sobre estos posibles metadatos en otros casos de uso -). De ese modo, gracias a la técnica surgen pautas claras y precisas para encontrar la fuente del problema.
Nótese también que si bien el caso no está aplicado explícitamente a un escenario de reuso en otro sistema o testing, esta situación puede darse en forma totalmente análoga si el componente es reutilizado en otro sistema bajo el mismo sistema operativo pero en su nueva versión, cosa que en tiempo de desarrollo no se hubiera esperado que provocara diferencias en el comportamiento de dicho componente.
Así, en un caso como este la técnica ha ayudado al usuario del componente, sin necesidad de recurrir al proveedor, con un problema habitual de suposiciones (assumptions) y falta de información: el componente supone determinado tiempo de respuesta del sistema operativo que típicamente no está documentado. También tiene que ver con cambios de entorno. E incluso podría llegar a plantearse como un caso de diferencias entre el entorno del proveedor del componente y el entorno real donde el usuario instala el componente, pues podría haber sucedido exactamente lo mismo si el módulo se hubiera desarrollado y testeado en una versión del sistema operativo y luego hubiera sido incluido en un sistema instalado en la versión actualizada del sistema operativo. Cabe destacar que en este último caso habría una diferencia importante, para que la técnica diera la información mencionada más arriba, el proveedor debería haber entregado el componente con lo registrado por TFT (o al menos un diario de viaje con el formato esperado) en las pruebas de testing o beta test, ya que de lo contrario el módulo de verificación no tendría información previa contra qué comparar como para distinguir estos "sucesos llamativos".

Ventajas: Vemos acá cómo la técnica TFT ayuda a resolver un problema que no es propio o interno del componente pero que lo afecta ciento por ciento, como es típicamente el caso de las suposiciones (assumptions) del entorno. Nótese además que no sería el caso general que el proveedor hubiera informado y documentado esta suposición sobre el comportamiento del servicio del sistema operativo ya que muy posiblemente el desarrollador también haya asumido el comportamiento del sistema operativo sin considerar que en nuevas versiones ello cambiaría. Aun así, si fuera conocido por el proveedor en la generalidad de los casos eso no suele llegar al usuario del componente provocando un típico problema a causa de la falta de información.

Clase: Uso

¿Qué metadatos recolectar? Registrar estadísticos e indicadores de "desuso", "bajo uso", "uso" y "alto uso", o número de invocaciones para:

- Servicios propios
- Servicios requeridos a otros componentes
- Recursos accedidos
- Servicios requeridos al entorno operativo

Pautas de cómo conseguir en tiempo de ejecución esa información: Una implementación muy sencilla son contadores en el wrapper de TFT que se acumulan al ser requerido un servicio propio, y al requerir servicios a otros.

Ejemplo de la información recolectada: el componente self-knowledge TFT ***C*** ha sido invocado 1800 veces; su servicio GetMap ha sido invocado 1200 veces; su servicio SetMap ha sido invocado 10 veces; y el servicio Draw del componente ***B*** ha sido invocado por el componente ***C*** 12 veces.

Caso de uso**:** *Optimización*
En un sistema, que entre otras funcionalidades, administra mapas a través del componente self-knowledge TFT ***C*** del ejemplo, desea hacerse una optimización de dicho componente para mejorar su performance. La optimización puede llevarse a cabo en principio tanto cambiando el componente por otro componente tomado de una librería propia o de terceras partes (escenarios a o d), o con un componente a desarrollarse internamente (escenario b). En cualquiera de estos casos los metadatos recolectados en esta clase son de suma utilidad. Por ejemplo ayudan a saber con certeza que si se mejora la performance del servicio que brinda los mapas (GetMap) se obtiene una mejora considerable ya que es el servicio más utilizado. A su vez también se puede ver que el servicio para setear un mapa (SetMap) es poco frecuente por lo que en el caso de desarrollarse un nuevo componente (escenario b) no es necesariamente aquí donde convendría poner las principales mejoras.
Por otro lado, si se toman otros componentes self-knowledge TFT de librerías o de otros sistemas en funcionamiento (escenarios a o d), por ejemplo un componente ***C´***, los metadatos de esta misma clase y los metadatos de la clase Performance, si es que están disponibles para ***C´***, podrían compararse con los del componente ***C*** para analizar sus equivalencias. Por ejemplo, si los valores de usos de cada servicio son similares en ***C*** y en ***C´*** puede pensarse en una verdadera similitud entre ambos como para comparar sus metadatos de performance, en tanto que si los valores de usos de cada servicio fueran menores en ***C´*** no deberían considerarse como definitivas las comparaciones entre sus performances ya que al tener menos usos puede que en esos pocos casos fueran bien performante y no en “los más variados” que ha sido usado ***C***. Nótese que toda esta información de metadatos ayuda a realizar análisis y a hacer consideraciones, incluso cada vez más complejas al ir cruzando información de varios metadatos. La precisión y contundencia de los resultados del análisis dependerán en gran medida de los modelos y de las herramientas utilizadas para dicho análisis, lo que logra TFT es poner a disposición del usuario del componente información que típicamente no tiene para hacer evaluaciones como las necesarias a la hora de reutilizar componentes de terceras partes o sustituciones como muestra este caso.
Así mismo, si decide reimplementarse el componente (escenario b) todo el diario de viaje de ***C*** pasará a ser el diario del nuevo componente, y la información contenida en él ayudará a la evaluación del nuevo componente, por ejemplo, guiando sobre que servicios requieren ser testeados más a fondo ya que han sido los más utilizados, y si hubiera servicios nunca utilizados tal vez su test podría postergarse para luego de la puesta en producción, si es que los costos de tiempo así lo consideraran necesario teniendo en cuenta que es poco probable su uso. Más aun si esto último cambiara, es decir si el servicio en cuestión comenzara a ser utilizado, el diario lo registraría y lo notificaría en los “últimos sucesos más llamativos”, ya que al reinstalarse el componente tendría un hito, nueva experiencia, en su diario que le permite comparar las dos experiencias distintas. Además si consideramos también los metadatos registrados en la clase anterior, los objetivos de mejora de performance que esperaban lograrse con la optimización también pueden ir evaluándose mientras el nuevo componente está puesto en producción (o desarrollo) haciendo consultas comparativas de experiencias como pueden hacérsele a los componentes self-knowledge TFT.

Ventajas: Vemos acá cómo la técnica TFT permite contar con información que sería interesante que el proveedor la brinde al usuario junto con el componente pero que típicamente no lo hace, e incluso en su sentido más estricto sólo es conseguible en campo, es decir en las ejecuciones reales del usuario. Más aun, en casos como los planteados le brinda al usuario cotas importantes para tareas costosas en tiempo y recursos permitiéndole así evolucionar sus componentes y sus sistemas con menores costos que si no contara con la información provista por TFT, tanto en el desarrollo de la evolución del componente como luego de la puesta en marcha del componente evolucionado, por un lado evitando posibles problemas (“sorpresas”) con las que podría encontrarse, y por otro lado poniendo a disposición información de experiencias anteriores para analizar los cambios en los resultados.

Caso de uso**:** *Análisis de impacto y Test de regresión*
Un caso particular de este tipo de metadato sería aquel que sólo lleve registro del conjunto de servicios utilizados en cada ejecución. Supongamos que se está planeando realizar una modificación al componente y con este metadato queremos asistir dos tareas importantes del desarrollo de CBS's: análisis de impacto y test de regresión.
¿Cómo podría ayudar en un caso como este la técnica TFT? La información así recolectada sería muy similar a la recolectada por el método descrito en [OAH/03] que también recolecta información de uso en tiempo de ejecución de "entidades", donde las entidades son partes de código o métodos

completos (servicios), y que aplicados sólo a estos últimos daría resultados equivalentes a los de TFT. Dicho trabajo muestra en detalle con un sencillo algoritmo los pasos a través de los cuales este metadato puede ser utilizado para asistir a esas tareas. Más aun, presenta datos experimentales sobre algunas aplicaciones mostrando por un lado que los datos de campo pueden mejorar eficientemente la calidad de los análisis dinámicos considerados, y por el otro, que las técnicas tradicionales computarían resultados que no reflejan el verdadero y real uso del sistema. La diferencia entre TFT y el método de [OAH/03] es que en este último parte de la instrumentación para la recolección es interna al componente, pero si bien es una diferencia de implementación importante con TFT lo que respecta a las conclusiones de los experimentos son igualmente válidas para ambos.

¿Qué metadatos recolectar? Junto con los metadatos anteriores de uso de servicios registrar tiempo de uso efectivo del componente.

Ejemplo de la información recolectada: el componente self-knowledge TFT ***C*** se encuentra en producción desde hace 3 años, y ha sido utilizado efectivamente durante 2 días y 4 horas; o dicho en otros términos, de las 254600 corridas del sistema, el componente ***C*** ha sido invocado 136 veces.

Caso de uso**:** *Costos y prioridades sobre procesos globales*
Un cierto componente está involucrado en un procesos de optimización, como puede ser una optimización de accesos a base de datos, o una optimización de uso de objetos por referencia en vez de objetos pasados por valor (copia), etcétera.
¿Cómo podría ayudar en un caso como este la técnica TFT? Supongamos que el metadato recolectado por la técnica nos dice que el componente sólo ejecutó 5 veces en 3 años, habiendo de 4 a 6 corridas mensuales del sistema. En una planificación de tareas y etapas este componente tiene baja o nula prioridad para ser adaptado o testeado basando la decisión en la información del uso real y efectivo de dicho componente respecto del uso global del sistema. Incluso el mismo metadato puede ser utilizado para compararlo entre diferentes componentes o diferentes servicios del mismo componente, y lograr así algún tipo de prioridad para llevar a cabo tareas como las mencionadas anteriormente u otras.

¿Qué metadatos recolectar? En sistemas del tipo kiosco-visitante (software context-aware con agentes móviles), registrar disponibilidad de variables presentadas entre los componentes del "visitante" y los componentes del "puesto" o "kiosco" al que llega.

Ejemplo de la información recolectada: el componente self-knowledge TFT ***A*** llega frente al componente ***B***, ***A*** solicita la variable v_1 obligatoriamente y la variable v_2 opcional, a lo que ***B*** responde con v_1, v_2 y v_3, entonces posibles metadatos a registrar en ***A***: 1) porcentaje de componentes a los que llega ***A*** y tienen las variables solicitadas; 2) porcentaje de componentes que tienen las variables opcionales requeridas por ***A***; 3) qué otros datos no solicitados por ***A*** (y porcentajes) tienen los componentes con los que habitualmente se encuentra ***A***. Análogamente, si ***B*** fuera un componente self-knowledge TFT, sería posible registrar metadatos como: 1) la variable v_3 fue ofrecida siempre pero sólo el 0,5% de las veces fue aceptada; 2) porcentaje de veces que fueron solicitadas variables no disponibles.

Caso de uso**:**
Casos de uso simples de estos metadatos pueden darse en tareas de análisis de usabilidad en proceso de optimizaciones o actualizaciones para saber cuán útil es una variable opcional, o bien cuan útil es que sea considerada como tal, por ejemplo para aumentar capacidades del componente ***A*** que requirieran variables como v3 sabiendo el porcentaje de componentes que la van a poder presentar; o teniendo un alto porcentaje de presentación de una variable no requerida podría decidirse aumentar la capacidad del componente sabiendo que ésta podría ser aprovechada en una mayoría de usos del componente.

¿Qué metadatos recolectar? Registrar invariantes de entrada/salida (I/O), es decir, dominios y codominos efectivamente alcanzados en tiempo de ejecución (usos reales) para:

- Servicios propios
- Servicios requeridos a otros componentes
- Recursos accedidos
- Servicios requeridos al entorno operativo

Pautas de cómo conseguir en tiempo de ejecución esa información: La implementación para obtener este tipo de metadato debe lidiar con temas no triviales como el descubrimiento de invariantes y la manipulación de objetos complejos. Entre muchas otras investigaciones, por ejemplo, la técnica BCT desarrollada por Mariani en [MP/04] ataca este mismo tipo de problemática y brinda algunas soluciones posibles utilizando una extensión de Daikon [ECGN/01] para inferir

invariantes que representan propiedades de los servicios del componente aplicado sobre objetos "aplastados" convenientemente.

***Caso de uso*:**
Este tipo de metadato puede ser utilizado para verificar la compatibilidad del componente en nuevas experiencias. Por ejemplo un componente con un servicio utilizado siempre en el rango $b \leq x \leq c$ para su único parámetro de entrada x, y que se espera que en un reuso sea utilizado en un rango $a \leq x \leq d$, con $a \leq b \leq c \leq d$ es información útil y valedera para decidir testear con alta prioridad dicho servicio en los rangos $[a; b]$ y $[c; d]$ ya que sin información del proveedor no hay razones para creer que hayan sido testeados o siquiera si son parte del dominio. Además los invariantes de salida pueden utilizarse como oráculos para identificar las desviaciones en el comportamiento esperado. A su vez en tiempo de ejecución de nuevas experiencias pueden utilizarse para dar alertas de nuevos dominio o codominios alcanzados, lo que en cierta forma puede verse como una extensión del testing sobre la etapa de ejecución complementaria, por decisión o por omisión, a los testeos del desarrollo. Desde el punto de vista teórico, y en lo que respecta fundamentalmente a este metadato, esto sería similar al método BCT de Mariani [MP/04] con la diferencia de que TFT permite la posibilidad de ser aplicado componente por componente, en tanto que BCT en principio se aplica a todo el sistema.

¿Qué metadatos recolectar? Registrar invariantes de interacción, es decir, patrones de comportamiento efectivamente alcanzados en tiempo de ejecución (usos reales) para:

- Servicios propios
- Servicios requeridos a otros componentes
- Recursos accedidos
- Servicios requeridos al entorno operativo

***Caso de uso*:**
Este metadato, al igual que el del caso anterior, permite en primer lugar obtener componentes con cierto grado de especificación, característica no siempre presente en el desarrollo de CBS's. Además también puede ser utilizado para verificar la compatibilidad del componente en nuevas experiencias, principalmente en lo que se refiere a la relación entre el comportamiento esperado y el verdadero comportamiento logrado en las experiencias anteriores del componente. Es decir, nuevamente desde el punto de vista teórico, y en lo que respecta fundamentalmente a este metadato, esto sería similar al método BCT de Mariani [MP/04] incluso presentando las mismas diferencias con TFT que en el metadato anterior. Así mismo, este metadato podría ser utilizado para filtrar o acotar tareas asociadas a test de integración o análisis de impacto, del mismo modo que la información recolectada en tiempo de ejecución es utilizada en [OAH/03] para filtrar el test de regresión y el análisis de impacto en el desarrollo de CBS's.

Clase: Respuestas exitosas y no exitosas

¿Qué metadatos recolectar? Registrar estadísticos e indicadores de requerimientos (invocaciones) no respondidos, es decir requerimientos con fallas que incluyan una interrupción de la ejecución (tanto por un "abort" como por un "loop"), computando por separado aquellos casos en que la interrupción se produjo mientras se esperaba respuesta de un servicio requerido a otro componente o recurso, de aquellos casos en que la interrupción se produjo mientras se realizaban cálculos internos al componente. Así se puede saber el número de veces (o porcentaje) que el componente y cada uno de sus servicios falla durante procesos internos, "falla interna", y cuando falla esperando procesos externos, "falla externa", y sabiendo el uso total de ellos se puede saber también el número (o porcentaje) de éxitos. Nótese que por simplicidad aquí, y en el resto de este capítulo, por "éxito" nos referimos sólo a ejecuciones (invocaciones) que llegan a ser respondidas sin interrupción, y por "falla" nos referimos a ejecuciones interrumpidas , lo que es un abuso de generalización, ya que en rigor existen fallas que no necesariamente incluyen una interrupción en la ejecución, y por ende el hecho de que no sea interrumpido no necesariamente indicaría éxito [ALR/01].

Pautas de cómo conseguir en tiempo de ejecución esa información: Del mismo modo que en los metadatos de uso, una implementación sencilla, son contadores agregando aquí una cierta lógica para incrementar sólo si el requerimiento finaliza, y en esos casos, para distinguir entre internas o externas las corridas no finalizadas, bastaría por ejemplo un flag indicando si mientras se espera la finalización del servicio éste requiere a otro externo o no.

Ejemplo de la información recolectada: el componente self-knowledge TFT ***C*** invocado 30000 veces, falló sin aviso 255 veces, de las cuales el 80% de la veces fueron "fallas internas" y el 20% fueron "fallas externas".

***Caso de uso*:**
Un análisis del comportamiento de este metadato podría darnos pautas de la robustez del componente, y más aun si se lo mira a lo largo de las distintas experiencias de su diario de viaje.

En casos en que la técnica TFT, o alguna compatible, haya sido aplicada a todos o casi todos los componentes del sistema, y con unas pequeñas consideraciones más en la implementación de la recolección de este metadato puede ser posible construir el "camino" de la última falla (o todas), lo cual puede ser muy útil en tareas de análisis de fallas como por ejemplo: 1) mapeos de esta información con modelos de caminos críticos; 2) mapeos de esta información con caminos testeados o no. Nótese que en casos como estos la información recolectada es de utilidad no sólo para el análisis del componente sino también para el análisis de propiedades de todo o casi todo el sistema, es decir, más allá del componente en sí mismo.

Ventajas: Vemos acá cómo la técnica TFT permite obtener información para casos de falla que pudieron haber quedado fuera del testing del sistema. Si esos casos se le escaparon al testing (en la etapa de desarrollo) sin la intención de que así fuera, TFT ayuda a compensar dicho testing, y si bien la falla llega a producirse, su análisis es más rápido que si no se contara con el metadato. Y si esos casos quedaron fuera del testing intencionalmente, por decisión del diseñador del test, TFT permite justamente tomar esas decisiones más relajadas reduciendo costos de tests exhaustivos (típicamente muy caros) brindando mecanismos e información en tiempo de ejecución que permitirán encontrar y analizar las fallas. Esta última estrategia puede ser de mucha utilidad en sistemas no críticos.

Variante: Si se cuenta con componentes como los Containment Units [COWL/02], módulos que proveen las bases para construir sistemas auto-adaptables que en una de sus interfases indican las fallas que el componente puede dar como resultado, justamente por no poder manejarlas, podríamos pensar en recolectar un metadato que registre estadísticos sobre las veces que el componente responde con esas fallas y su relación con los casos en que finaliza sin ellas.
Y nuevamente aquí la información recolectada con TFT ayudaría al análisis de la robustez del componente. Si bien este metadato, tal como ha sido planteado en esta variante, carece de la generalidad deseada para la técnica, ello no impide que se pueda contar con él y que el usuario lo use si es que tiene la posibilidad de utilizar un componente como los Containment Units, simplemente bastaría seleccionar o no el metadato en la fase (1). Además, nos permite ver que contando con metadatos de mayor nivel de abstracción, ya sea por el tipo de componente en el que se utiliza la técnica, o por la interpretación y abstracción lograda por el manipulador TFT del metadato, más útiles e interesantes son esos metadatos para las tareas de análisis, evaluación y verificación del uso y reuso de los componentes.

¿Qué metadatos recolectar? Registrar valores de los parámetros en invocaciones a servicios del componente, e invocaciones desde el componente a otros componentes para ejecuciones con fallas que incluyan una interrupción, es decir, los parámetros de entrada de los requerimientos que el componente no terminó de responder.

Pautas de cómo conseguir en tiempo de ejecución esa información: Aquí, al igual que algunos metadatos anteriores, nuevamente se debe abordar el tema de objetos completos, estados, tratamiento de elementos como bases de datos, etcétera. Podrían utilizarse técnicas ya existentes o en investigación, o bien establecer y definir casos de aplicabilidad.

***Caso de uso*:**
Este metadato no indica necesariamente que esos valores estén fuera del dominio de los servicios del componente mismo, pero es posible que alguno de ellos sí. Por ejemplo, si en un parámetro índice de un arreglo se tiene registrado varias veces que al tener el valor 0 el componente no termino exitosamente, eso estaría indicando con alta probabilidad que el índice no comienza en 0. Esta información puede ser interesante verla y analizarla con el metadato “rango de dominio”, pues por ejemplo si su valor es [1, 70] apoyaría la idea de que 0 no es parte del dominio. En términos generales este metadato puede ser útil para análisis de fallas en la etapa de mantenimiento, casos de borde en futuros test al reutilizar el componente o al desarrollar nueva versiones.

Ventajas: Vemos acá cómo la técnica TFT permite obtener información que por un lado sólo es conseguible en tiempo de ejecución, y principalmente nos da información que facilita mucho el análisis del una falla al punto tal que podría hacer totalmente transparente la fuente del problema.

¿Qué metadatos recolectar? Junto al metadato anterior, registrar algún caso de éxito para valores de casos no terminados exitosamente. Es decir, para los casos guardados en el punto anterior, si luego alguna ejecución futura sí termina exitosamente entonces registrarlo como tal.

Pautas de cómo conseguir en tiempo de ejecución esa información: Este metadato puede ser pesado de calcular principalmente si las fallas registradas han sido muchas, o bien si los parámetros son complejos, como objetos muy grandes, o matrices, etcétera. Pero en general para cada tipo de variable podría buscarse una buena relación costo-beneficio acorde a la interesante utilidad de este metadato.

***Caso de uso*:**
Esta información cumplimenta la información del caso anterior para la interpretación que pueda

darse al problema. De este modo, se tiene información para ver que los valores en cuestión, que han sido parte de una falla, o bien pertenecen a una falla no permanente, es decir a una falla intermitente, o bien no tienen relación con la misma. Queda claro que faltaría más información para conocer la fuente de la falla pero esta información puede ser al menos de utilidad en el análisis.

Clase: Arquitectura parcial

Datos útiles para la evaluación o el análisis de la arquitectura.
Dado un sistema, en su arquitectura ciertos componentes son los más importantes, y para ellos ciertas características son las más relevantes tanto para la verificación y testing como para su análisis en tiempo de ejecución, por ejemplo para análisis de comportamientos inesperados, como para el mantenimiento y adaptación o evolución de dicho sistema. Entonces sería deseable que los componentes que implementan a dichos componentes de la arquitectura contaran con información propia para llevar a cabo esas tareas. Y esa información podría obtenerse de sus propias ejecuciones, y reutilizarse al reutilizar el componente en otros sistemas, o bien cuando sea sustituirlo por otro. Eso es justamente lo que nos brindan los componentes self-knowledge TFT. Entonces por ejemplo podría ser deseable registrar componentes con los que interactúa verdaderamente, y servicios utilizados, es decir, la sub-arquitectura, o arquitectura parcial, vista desde el componente y con uso real. Otra posibilidad podría ser registrar propiedades del componente respecto de la arquitectura, por ejemplo en un sistema cliente-servidor, el componente servidor de la arquitectura se espera que tenga o cumpla determinadas propiedades sobre su ancho de banda o la cantidad de clientes que atiende al mismo tiempo. Por lo tanto, los componentes que los implementan podrían setearse para recolectar información sobre ancho de banda y número de clientes atendidos. Así, por un lado podría conocerse y evaluarse en tiempo de ejecución si se logran los valores esperados, y por otro lado si se desea sustituir o adaptar a dichos componentes se cuenta con datos reales sobre los valores que deben satisfacerse o alcanzarse. También otros datos de la arquitectura son de utilidad tanto para el caso en que es sustituido el componente, como para el caso en que es reutilizado, por ejemplo información de componentes y conectores con los que interactúa, como “nombre”, “marca” y “versión”. Un ejemplo concreto de ello podría ser que un componente que utiliza una base de datos lleve registrado en su diario algo como "48 meses utilizando un base Oracle versión 8, luego 20 meses utilizando una base SQL Microsoft 7.1, y actualmente 2 meses utilizando una base Oracle 10g". Información de este tipo es de suma utilidad a la hora de analizar el componente, el hecho de que ya haya interactuado con determinado motor de base de datos no garantiza un comportamiento libre de errores con dicho motor, pero sí eleva la confianza y la probabilidad de buen funcionamiento, reduciendo la cantidad y tipo de errores que pueden surgir en el testing, como fallas de protocolo, que si ya fue utilizado seguramente ya pudieron haber sido superadas. Claramente, la idea sería implementar esta clase de metadatos conjuntamente con metadatos como los vistos en otras clases y otros similares. Otro dato útil referido a la arquitectura es el número de réplicas de un componente, es decir, llevar registrado en el diario del componente información de las cantidades mínimas y máximas de réplicas utilizadas en cada evolución (sistema). Así, esto podría ayudar a anticiparse a algunos de los problema de suposiciones como por ejemplo si quisiéramos utilizar el componente X-Windows para implementar las celdas de una planilla de cálculo [UY/00], ya que nos sería de utilidad ver que en los usos anteriores el número máximo de réplicas es muy lejano al que esperamos alcanzar en una planilla de cálculo.
Un aspecto importante a tener en cuenta en esta clase de metadatos asociados a arquitecturas es que estos metadatos requieren mayor nivel de abstracción entre la información monitoreada en tiempo de ejecución y su valor registrado en el diario. La técnica debe lograr esto o bien en tiempo de ejecución para ya almacenar datos del más alto nivel posible, o bien con interpretación posterior a la registración y anterior a ser consultado el diario. Seguramente aquí sea necesaria una evaluación del tipo costo-beneficio de varios asuntos tales como el costo de interpretación on-line vs el costo en volumen de información que necesita ser registrada para una post-interpretación. Y otro aspecto mucho más crucial asociado a definir metadatos para tareas e información a nivel de arquitecturas es el problema de la relación entre la arquitectura de un sistema y su implementación. Actualmente esta problemática forma un área de investigación en sí misma, sin lugar a dudas avances en ese área pueden dar pautas claras y concretas sobre qué información y cómo esa información puede ser interpretada desde el monitoreo en tiempo de ejecución (implementación) hacia el nivel de arquitectura.

Clase: Contexto

Para describir esta clase es importante primero poner en claro qué estamos interpretando por contexto. En principio la definición de contexto, y principalmente la de recursos, dependerá del tipo de sistema y de cómo esté planteada la arquitectura de un sistema, el estilo, el lenguaje, el gusto, etcétera. Por ejemplo, una base de datos puede ser vista como un componente o como un recurso de almacenamiento. Tomando nociones básicas y generales a todas las áreas informáticas, sin restringirnos a ningún sub-área en particular, aunque ello no esté limitado o impedido por la técnica, en esta sección pensaremos en el contexto y los recursos como todo aquello posible

circundante al componente y al sistema del que forma parte el componente, incluso el sistema mismo puede pensarse como contexto del componente. De ese modo, ejemplos de contexto y recursos pueden ser la plataforma donde ejecuta el componente, el sistema operativo, los periféricos, el computador, su procesador, memorias, otros procesos que ejecutan antes, durante o después de la ejecución del componente, la ubicación, los usuarios, etcétera. Independientemente de ello, cualquiera sea el elemento que consideremos como definición o parte del contexto de un componente (además del sistema) o el recurso, en primer lugar y en términos generales, podemos pensar en registrar metadatos con información como la que presentamos en los casos anteriores como indicadores de uso, relaciones o patrones de uso respecto del componente que reflejen la dinámica con el componente en tiempo de ejecución, o datos informativos como "nombre", "marca", "versión", etcétera, o más específicos conociendo el tipo de recurso.
A continuación presentamos algunos otros metadatos de información asociada al contexto y sus elementos que podrían recolectarse en tiempo de ejecución.

¿Qué metadatos recolectar? Registrar disponibilidad de recursos como por ejemplo estadísticos e indicadores de disponibilidad de memoria antes y después del uso del componente. Registrar estadísticos e indicadores de aplicaciones o procesos concurrentes o paralelos.

***Caso de uso*:**
Para casos de reutilización, sustitución, cambio de ambiente (reinstalaciones), contar con un diario con información del estilo "siempre que ejecutó el componente había por lo menos *x* MB libres de memoria" o "mientras el componente ejecutaba: Word estaba también ejecutando el 80% de las veces, Block de Notas 10%, Eudora 50%, y Explorer 99% de las veces" puede ser de suma utilidad. Incluso pueden ser tenidos en cuenta casi como requisitos del componente, y aunque las cotas fueran muy grandes pueden ser útiles como punto a partir del cual llevar a cabo los análisis, las evaluaciones o las consideraciones.

¿Qué metadatos recolectar? Registrar estadísticos e indicadores de usuarios concurrentes.

***Caso de uso*:**
Supongamos que se tiene especificado o documentado que el componente tiene una tolerancia de hasta 10000 usuarios, y según lo recolectado en tiempo de ejecución el número real de usuarios nunca fue mayor a 200.
¿Cómo podría ayudar en un caso como este la técnica TFT? 1) si con esos 200 usuarios la tolerancia / performance del componente no se correspondiera con la esperada, se tendría rápidamente una cota o margen de tolerancia real; 2) si esos 200 usuarios fueron bien tolerados al reutilizar el componente se sabría que el uso de 200 usuarios está "garantizado", o mejor "avalado" o "más" avalado por la experiencia continua (considerando el dato tiempo de vida y uso del sistema y componente), y 10000 sigue siendo un dato teórico; 3) si se deseara hacer un test de stress para una reimplementación del sistema que no implicara un aumento en el número de usuarios actuales, gracias a la técnica se sabría que no sería necesario probar con un número más grande que 200 usuarios y consecuentemente conseguir así una reducción de costo confiable de dicho test.

¿Qué metadatos recolectar? Registrar a nivel estadístico información de los usuarios que usan efectivamente el componente, en grupos pequeños pueden ser por individuo, o en grupos grandes pueden ser por clases de usuarios como perfiles, niveles de acceso, áreas, puntos de acceso, etcétera, para:

- Servicios propios
- Servicios requeridos a otros componentes
- Recursos accedidos
- Servicios requeridos al entorno operativo

Ejemplo de la información recolectada: el uso del componente self-knowledge TFT ***C*** se distribuye de la siguiente forma: 60% son usuarios que acceden a través del site, 30% son usuarios acceden a través de la red de área local, y 10% son usuarios que acceden vía FTP.

***Caso de uso*:** *Seguridad*
Supongamos el caso de un sistema que es modificado en alguna de sus partes pero no en el componente self-knowledge TFT en cuestión. Supongamos también que por un lado el metadato recolectado para dicho componente contienen información sobre el punto de acceso de los usuarios, y por el otro que, de acuerdo a las modificaciones realizadas, no se espera que cambien los puntos de acceso permitidos a cada usuario.
¿Cómo podría ayudar en un caso como este la técnica TFT? En su primer uso el componente va registrando, entre otros metadatos, información sobre los puntos de acceso, y en particular digamos que podría ser el caso que desde cierto punto de acceso *P* el componente nunca fue accedido. Luego el componente, junto con su diario de viaje, es puesto en producción nuevamente

luego de haber sido modificado el resto del sistema, y como ha cambiado su entorno, se registra un hito, nueva experiencia, en el diario de viaje del componente. Nuevamente, en esta segunda experiencia, el componente va registrando información sobre los puntos de acceso. Si en algún momento el componente es accedido a través del punto de acceso *P* surgirá un alerta del módulo analizador de TFT mencionando esta diferencia importante. Nótese que en este caso esta alerta no es preventiva, como en otros casos, sino que está indicando una violación de la seguridad del sistema no esperada, es decir se trata de una falla. Este es también un caso que nos permite ver que activar el módulo de verificación en tiempo de ejecución de TFT ("on-line") puede brindarle beneficios al usuario que justifiquen el costo de la sobrecarga de la técnica, e incluso que es conveniente tener grados de desvío y grados de alerta asociados a cada tipo de metadato.

¿Qué metadatos recolectar? Datos del sistema en el que ejecuta el componente self-knowledge TFT.
Sobre el sistema del que forma parte el componente self-knowledge TFT en principio podemos pensar en mucha información a recolectar pero que por un lado habría que ver que esa información sea automáticamente conseguible, y por otro lado que verdaderamente sea de utilidad. Por ejemplo se podría registrar el tamaño del sistema (del que forma parte) en número de componentes y conexiones, y/o número de instancias replicadas en tiempo de ejecución (ver Clase: Arquitectura parcial), y/o número de usuarios.
También se podrían registrar metadatos con información sobre fecha y hora de instalación y actualizaciones del sistema, algo así como la historia evolutiva de los sistemas a los que va perteneciendo.
También se podrían registrar, en los casos en que fuera posible, metadatos con información sobre tiempo y lapso en el que sistema se encuentra levantado, pensando en sistemas que arrancan y detienen sus ejecuciones, no en aquellos del tipo 24 x 7 x 365, sino sistemas para los cuales puede ser útil el uso efectivo de un componente sobre el uso efectivo del sistema. Este metadato estaría entre los datos que lleva el componente del sistema pero como ya hemos dicho antes siempre desde el punto de vista del componente.

7.2 Conclusión

La información contenida en el diario de viaje de un componente self-knowledge TFT, es decir, la base de conocimiento del componente puede ser pensada como el curriculum vitae del componente. Cuando uno toma y reutiliza ese componente sabe exactamente con quién está tratando, ya que no es una caja totalmente negra y desconocida, y esto es útil del mismo modo que el curriculum vitae de una persona es útil para contratarlo: no sólo nos dice la casa de estudios y los títulos obtenidos sino también las experiencias, los proyectos, y la gente con la que ha trabajado, y cómo se ha desempeñado en ello, información que en su totalidad, y analizándola de forma apropiada, nos ayuda a realizar mejores elecciones, o al menos, elecciones mejor fundamentadas.

8 TFT EN CONTEXTO

Una vez presentada la técnica TFT, habiendo descrito sus objetivos y sus partes, y habiendo mostrado algunos casos de uso de la misma, a continuación queremos describir someramente cómo, a nuestro entender, ella se relaciona, vincula o enmarca en las áreas y motivaciones mencionadas en los primeros capítulos de este trabajo, así como también presentar los trabajos relacionados y los temas abiertos pendientes de investigación.

8.1 Discusión

Acorde a los elementos del espacio de problemas de los self-healing systems descrito en [KOO/03] y presentado en la sección ***2.3***, a nuestro entender la técnica TFT es una propuesta que ataca los problemas de completitud de los sistemas. Cubre los problemas de completitud arquitectónica ya que por un lado ayuda a afrontar los problemas de lidiar con la composición de componentes desarrollados por terceras partes, por otro lado ayuda a superar los problemas de la falta de conocimiento del desarrollador del componente acerca del contexto final y el propósito final con el que es utilizado el componente, y por otro, ayuda a entender los problemas que surgen en la administración de configuraciones. Además, permite construir y aumentar el conocimiento que los componentes tienen de si mismos, y en consecuencia el conocimiento que los sistemas tiene de si mismos. Y finalmente, ataca el problema de la evolución de los sistemas asumiendo la evolución como una parte misma de los componentes, por un lado evolucionando su conocimiento con ellos, y por el otro brindando información y herramientas para llevar a cabo las principales tareas asociadas a la evolución de los componentes y de los sistemas de los que forman parte dichos componentes.

En el capítulo ***2***.*Self-Healing Systems* mostramos el diagrama de ciclo de reacción de los sistemas self-managed. Como hemos dicho, diferentes técnicas y métodos implementan una o más fases del ciclo. La figura 8 muestra las fases e interacciones implementadas por la técnica TFT resaltándolas en ese mismo diagrama.

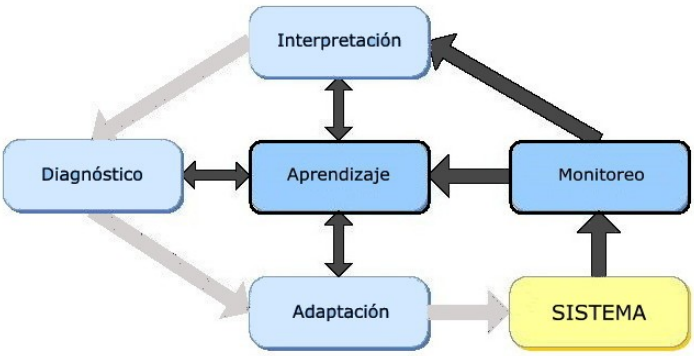

Figura 8:TFT dentro del ciclo de reacción de sistemas self-managed

Es decir, la técnica TFT implementa principalmente la fase de aprendizaje, su alimentación a partir de información de la fase de monitoreo, así como también las consultas sobre el conocimiento adquirido (interacciones). Hay una parte importante de la fase de aprendizaje no cubierta por TFT que es aquella referida principalmente a la relación entre la información recolectada y un problema o falla real a tratar, e incluso la relación con su solución. TFT no cubre estos últimos aspectos ni busca cubrirlos. Aun así, cabe destacar que TFT ha sido diseñado con la intención de permitir complementar las fases no cubiertas por la técnica con otras técnicas o métodos distintos a fin de lograr el ciclo completo.

También hemos visto que Tosi propone tres principales dimensiones para las características del self-management: *Reparación*, *Monitoreo* y *Requerimientos*. Reconocer estas principales dimensiones es de suma utilidad para entender diferencias y similitudes entre las distintas áreas de investigación de la informática autónoma, permitiendo así clasificar y ubicar la literatura existente y futura. En particular puede ayudarnos a ver dónde se ubica la técnica TFT propuesta en el presente trabajo. Para ello, en la figura 9, retomamos el gráfico de la figura 2 que nos muestra la relación entre las dimensiones y las características del self-management, su ortogonalidad, y su relación con las clasificaciones más detalladas vistas al comienzo.

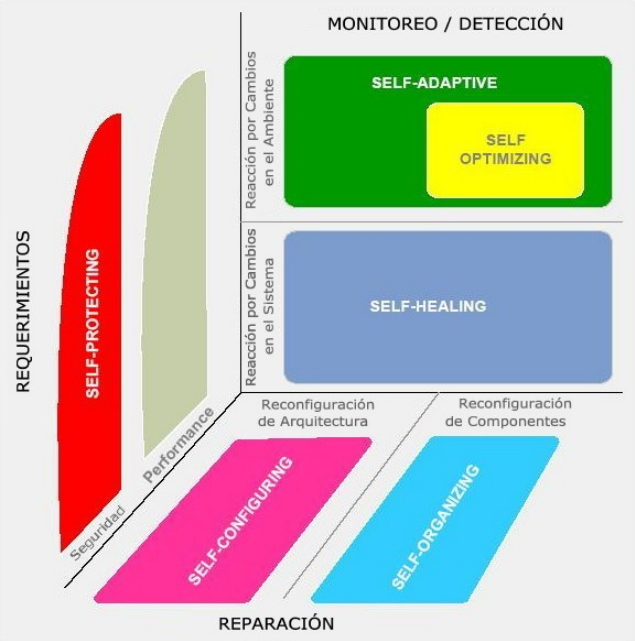

Figura 9:Clasificación detallada de Self-Management ([TOS/04])

Si analizamos dicho gráfico podemos ver que la técnica TFT se ubica en la dimensión de Monitoreo / Detección, sin incluir los mecanismos de auto-reparación / adaptación dejando que eso pueda ser implementado manualmente o con alguna técnica complementaria. Es importante notar también que las técnicas y métodos ubicados en las áreas (zonas) del gráfico no cubiertas por TFT podrían sacar provecho de la base de conocimiento, el diario de viaje construido a partir de la utilización de la técnica TFT.

Desde cierto punto de vista, al igual que todas las investigaciones de componentes self-testing, TFT también forma parte del grupo de estudios que conforman una base científica importante que apuntan a aumentar los sistemas de software con auto-capacidades que aprenden y usan casos de test en tiempo de ejecución. Como vimos, esto puede ser aprovechado por los sistemas self-managed de dos formas: por un lado, brindar los medios para realizar test de regresión dinámicos de funcionalidades frente a cambios en el entorno operativo; por otro lado, las herramientas y la información recolectada podría ser útil en casos particulares al final del ciclo de reacción para verificar que el sistema así adaptado funciona apropiadamente.

Adicionalmente, la información recolectada también podría ser utilizada para enviársela al desarrollador del componente periódicamente bajo un acuerdo entre desarrolladores (proveedores) de software y usuarios. No ahondaremos en este tipo de uso en este trabajo, pero sin lugar a dudas este intercambio podría ser muy valioso para los proveedores de componentes y podría llegar a ser parte de una práctica exhaustiva de testing de software perpetuo.

A lo largo de toda la definición de la propuesta de la técnica TFT hemos hecho hincapié en su característica de autonomía, tanto en sus objetivos y en su diseño como en la implementación final que se haga de ella. Esta característica de autonomía permite obtener con TFT las siguientes ventajas principales:

- obtener información que de otra forma no se conseguiría porque nunca llegara a hacerse lo necesario. Es decir, toda la información recolectada tal vez pueda conseguirse manualmente o utilizando diversos mecanismos, técnicas y herramientas ejecutadas ad hoc, pero como ello requiere recursos adicionales, principalmente tiempo y personas, típicamente en la práctica no llega a hacerse. En cambio si puede conseguirse en forma autónoma y automática, requiriendo sólo recursos informáticos de almacenamiento (cada vez más baratos), y con un costo de degradación en la performance relativamente pequeño respecto del beneficio, la técnica logra ser realmente útil, factible y practicable;
- desde el punto de vista de “calidad de dato” podríamos decir que la información recolectada por medio de esta técnica es más confiable y mejor que si se consiguiera manualmente o con instrumentación interna ad-hoc, ya que la generalidad de la técnica y su desarrollo totalmente independiente del desarrollo del componente garantiza objetividad y transparencia.
- unificación del formato de la información recolectada de diversos componentes, lo que permite una estandarización de uso, y todos los beneficios que ello conlleva en el desarrollo de los subsiguientes sistemas basados en dichos componentes, facilitando el análisis, la comparación y evaluación de los mismos.

- considerando uno de los puntos anteriores, una ventaja importante es que TFT permite abaratar costos desde el punto de vista de recursos humanos, no sólo en tiempo sino también eliminando la necesidad del knowhow funcionales y técnicos requeridos para la tarea. De ese modo las empresas pueden lograr que sus capacidades de IT dejen de invertir cada vez más tiempo en lidiar con la complejidad de los sistemas informáticos y tareas rutinarias.

Una contribución importante de la técnica TFT para afrontar los nuevos desafíos para el testing de CBS´s y COTS es el uso de monitoreo para ello, es decir, brinda otro tipo de técnica para afrontar los problemas asociados al testing de este tipo de sistema. Dicho de otra forma: ¿cómo solucionamos el problema o parte de los problemas de reuso? con experiencia, recolectando y utilizando información de las propias vivencias de los componentes, ¿cómo solucionamos el problema o parte de los problemas de suposiciones (assumptions)? con experiencia, recolectando y utilizando información de las propias vivencias de los componentes, ¿cómo solucionamos el problema o parte de los problemas de diferencias de entornos? con experiencia, recolectando y utilizando información de las propias vivencias de los componentes.

Con esto también queremos dejar en claro que lo novedoso de la propuesta está en el uso de la técnica y sus resultados para reuso, sustitución de componente, testing, etcétera, y no en el uso para reacción frente a cambios en el entorno. Aun así, como ya hemos dicho, TFT puede ser una base para la definición de métodos que le permitan a los sistemas reaccionar frente a los cambios de entorno.

Acorde a los propósitos y a la forma en que hemos planteado la técnica TFT esperamos y confiamos en que el usuario de componentes obtenga una buena relación costo-beneficio con el uso de la misma. Si bien desde el punto de vista del usuario del componente puede presentarse cierta degradación en el tiempo de respuesta del componente debido a la sobrecarga del framework TFT, y por ende en el del sistema en general, y considerando que además él debe invertir en recursos como espacio de almacenamiento y procesadores de cálculo, debe tenerse en cuenta que 1) estos recursos son cada vez menos costos y están más al alcance de la mano; 2) a cambio se espera que sea mucho menor el tamaño del equipo IT requerido para mantener, controlar y analizar los sistemas en funcionamiento; y 3) dado que de este modo se logran componentes con información (y no faltos de ella), las promesas de reducción de costos con la reutilización de componentes parece ser alcanzable.

8.2 Trabajos relacionados

La técnica TFT toma las ideas subyacentes en el testing perpetuo de Pavlopoulou y Young acerca de extender el testing hasta la operación normal del software [PY/99]. Además comparte la idea de utilizar los datos recolectados in-field para testear y verificar sistemas software con otras investigaciones como el testing remoto ANTS [ROD/02], la recolección de datos de campo (field-data) del método Gamma de Orso et al. [OAH/03] y la equivalencia de comportamientos del método BCT de Mariani [MP/04]. En el diseño e implementación de la técnica se aprovechan propuestas de monitoreo genéricas como las de Garlan [GSC/01]. A diferencia de otros métodos, TFT permite enriquecer componentes en forma autónoma con información recolectada a lo largo de sus ejecuciones, formando una base de auto-conocimiento (self-knowledge) que evoluciona en cada nuevo entorno o escenario del componente, y a su vez proporciona un framework para administrar y consultar esa información a fin de dar soporte a las tareas de ingeniería de software principalmente a las vinculadas con la evolución del componente, como reutilización y actualizaciones.

La metodología design-for-testability apunta a manejar en general los mismos problemas que la técnica TFT pero el foco de la solución de cada uno está pues en lugares exactamente opuestos. Los métodos de design-for-testability, al igual que TFT, no manejan las causas de la falta de información, sino que brindan los medios para sobreponerse a las potenciales dificultades que puede encontrar el usuario del componente para testearlo [BG/03], pero en dichos métodos todo depende de lo que el proveedor del componente haya decidido articular en el componente, en tanto que muy por el contrario TFT enfrenta los problemas sin hacer suposiciones y sin depender de lo que el proveedor pudiera haberle brindado al usuario con el componente. Esta es una diferencia muy importante y una ventaja a favor de TFT ya que con ello, además de ayudar a superar los problemas de la falta de información, ayuda a superar el problema de la dependencia del proveedor del componente. Consideremos además que muchas de las técnicas basadas en la metodología design-for-testability requieren considerable trabajo manual, en tanto que TFT es esencialmente una técnica autónoma, y la recolección de la información no requiere intervención humana.

Raz et. al en [RKS/02] proponen un mecanismo para detectar desvíos del comportamiento esperado para piezas de software sin especificación o con especificaciones incompletas, como es el caso de los componentes desarrollados por terceras partes. Incluso proponen un mecanismo simple para asistir al usuario del componente en las tareas de reparación que pudieran ser necesarias. Al igual que TFT, este mecanismo pone el foco principalmente en las necesidades y capacidades del usuario del componente brindándole una solución que no depende del proveedor del componente. También comparte con TFT la idea de formar conocimiento (del comportamiento) a partir de las ejecuciones reales del software. A diferencia de TFT, este mecanismo sólo considera el descubrimiento de invariantes, su construcción y detección de violaciones, sin proveer un marco general para la recolección y uso de los diferentes tipos de información disponible en tiempo de ejecución del software. Tampoco considera el reuso de esa información, ni la evolución del software en cuestión.

TFT usa el framework definido en [OHR/00]. Ese framework permite proporcionar al usuario del componente distinto tipo de información, dependiendo del contexto y las necesidades específicas. El framework se basa en presentar la información en formato de metadato. El metadato describe aspectos estáticos y dinámicos del componente, puede ser accedido por el usuario, y puede ser utilizado para diferentes tareas. Tal como está allí definido, el proveedor del componente es el único proveedor de la información contenida en los metadatos, así la propuesta ataca las causas de los problemas en el desarrollo de CBS´s debidos a la falta de información. Por su parte, TFT extiende esa noción de metadato relajando la condición de que sólo el desarrollador del componente esté involucrado en su producción, considerando que cualquiera pueda producirla, siempre y cuando se respete el formato. Entonces, en particular, TFT trabaja con metadatos recolectados en forma autónoma por módulos de la técnica, referidos y asociados a propiedades del componente y su contexto en tiempo de ejecución. De este modo, a diferencia de aquel, TFT es una técnica que ataca los efectos de los problemas en el desarrollo de CBS´s debidos a la falta de información. Aun con esta generalización, los usos y ventajas del metadato de Orso, su formato y su manipulación son igualmente válidos para TFT.

Para la implementación de la clase de metadatos de contexto (recursos y elementos del entorno) de TFT, el área de la informática conciente-del-contexto (context-aware computing) tiene mucho en común con TFT, principalmente en lo que se refiere a definiciones y modelos del contexto, del entorno. Definitivamente, técnicas y métodos del área pueden proporcionar importantes aportes para el reconocimiento y la representación de los recursos y el entorno con metadatos.

8.3 Open issues y Trabajos Futuros

TFT monitorea, interpreta y registra, en forma autónoma, información del componente y su contexto en tiempo de ejecución de las corridas del componente a lo largo de todo su ciclo de vida. Confiamos en su utilidad, y creemos que la técnica es factible teniendo en cuenta técnicas similares que están siendo investigadas y desarrolladas por la comunidad científica informática. Pero sin lugar a dudas es necesaria una investigación más a fondo para comprobar la factibilidad de TFT. Es necesaria investigación y definición más detallada, principalmente en aspectos técnicos, resultados experimentales, análisis y evaluación de la técnica. Por ejemplo, la autonomía de la técnica es uno de sus pilares más importantes, y hasta la fecha no se tiene conocimiento de trabajos prácticos concretos, generales y aplicables a todas las tecnologías, del tipo que necesita utilizar TFT. Actualmente hay prototipos en desarrollo similares o útiles para la automatización de esta técnica, a partir de ellos se podrían reunir datos experimentales que permitieran verificar las hipótesis aquí planteadas y que a su vez permitieran identificar las debilidades de la técnica. El objetivo final debería ser refinar completamente la técnica, identificar los límites y las ventajas, y definir el campo de aplicabilidad.

Teniendo esto en mente, a continuación plantearemos algunos de los aspectos de TFT que requieren futuros trabajos de investigación:

- **Fases**
  En la *fase (1) (Selección de los registradores*) investigar la elaboración de conjuntos de sugerencias de metadatos, analizar sus utilidades, teóricas y prácticas, en profundidad. Investigar por ejemplo un conjunto mínimo de metadatos que pueda ser utilizado para ejecutar las tareas tradicionales de ingeniería de software, tales como testing, análisis, computación de métricas estáticas y dinámicas, y debugging. Más aun, hallar incluso relaciones entre esos conjuntos y taxonomías de componentes, o sistemas, por ejemplo podrían definirse conjuntos por estilos de arquitectura, conjuntos por tecnologías, conjuntos por tipos de errores o análisis, etcétera. También puede ser interesante investigar la posibilidad de brindar junto con la clase de metadato a recolectar algún tipo de información sobre el costo de la recolección del mismo, y por qué no algún valor del beneficio, de modo de asistir al usuario en la decisión sobre la elección de los elementos que conformarán el diario de viaje del componente.

En la *fase (2) (Generación e instalación de los registradores*) investigar técnicas no intrusivas para los componentes buscando el mayor grado posible de automatización para las diferentes tecnologías (ej., EJB, .NET, etcétera).
En la *fase (3) (Captura de las ejecuciones*) investigar "políticas de registro" para limitar el número de comportamientos simples a registrar, estudiando un tradeoff óptimo entre la cantidad de información registrada y los costos computacionales.
En la *fase (4) (Filtros de comportamientos*) investigar distintas "políticas de selección" para la elección de comportamientos simples relevantes basados en diferentes criterios de cubrimiento, impacto del comportamiento simple en los comportamientos inferidos e intereses del metadato.
En la *fase (5) (Destilación de información de viaje*) investigar criterios, políticas y métodos para sintetizar, aumentar y evolucionar los diferentes metadatos en cada experiencia.
En las *fases (6) y (7) (Verificación de desvíos y novedades y Consultas, análisis y verificación*) investigar todos los posibles usos de las alertas; algunos ejemplos son detección de fallas, detección de conflictos, mapeo y comparación de los diferentes metadatos entre experiencias, derivación de perfiles operacionales de usuario, feedback para testing, etcétera. También evaluar e investigar cuestiones de sincronización y overhead, es decir, analizar en qué orden y de qué forma es conveniente que se sucedan las alertas habiendo aplicado la técnica TFT a uno o varios componentes del mismo sistema; y en esos casos incluso valdría la pena investigar cuál de todos los diarios de viaje tiene la información que mejor describe un problema, o si es posible hallar o plantear alguna relación entre ellos. En particular, el tema del overhead tiene dos aspectos principales, por un lado afectar la semántica del metadato en sí mismo, y por el otro, en el caso de informes de los "últimos sucesos más llamativos" en tiempo de ejecución ("on-line"), si el reporte es lento hasta alcanzar el nivel más abstracto, entonces, puede ser que luego del tiempo requerido para la alerta, las condiciones en el entorno hayan cambiado nuevamente, o ya no fueran tales, o que no se justifique su atención.

- **Implementación y tecnologías de monitoreo**
  Trabajos como los presentados en [GSC/01] evalúan e investigan trabajos con probes y gauges, futuros planes incluyen aplicar diferentes clases de tecnologías de probes. La probe de Remos [LMSGSS/98] que han utilizado para observar información de bajo nivel es una probe que monitorea el entorno de red de una aplicación distribuida, y no necesita ser introducida dentro la aplicación, y por lo tanto es no-intrusiva. Las demás probes descriptas en ese trabajo, se codifican dentro de la aplicación, pero planean coordinar con otros investigadores desarrollar diferentes tipos de tecnologías de probes, para evaluar la generalidad de los gauges con respecto a las probes.

- **Escalabilidad**
  Investigar y analizar el impacto de aplicar la técnica TFT a diferentes escalas. Buscar los límites y estudiar propuestas de reducción de costos a fin de que la técnica alcance el mayor grado de escalabilidad posible. En este sentido, varias son las dimensiones a investigar: 1) escalar el número de metadatos a registrar para un componente, ya que mientras mayor sea el número de metadatos que se registren mayor será el impacto en la performance del componente y en consecuencia en la performance del sistema; 2) escalar el número de experiencias, ya que mientras mayor sea el número de experiencias de un componente mayor será la cantidad de información que tendremos del componente pero también es posible que los análisis sean más costos, o poco performantes, y esto puede impactar negativamente principalmente en las consultas y los análisis on-line, y a su vez al aumentar el número de experiencias sería deseable que la información, y la forma de utilizarla, indiquen pautas o tendencias y no que generen un dominio lo suficientemente amplio como para admitir cualquier comportamiento, incluso aquellos no deseados; 3) escalar el número de ejecuciones de un componente, algo así como estudiar el resultado de utilizar experiencias muy muy largas, analizar el impacto en el volumen de la información, y hasta que punto sería de utilidad continuar registrando información de un mismo contexto; 4) escalar el número de componentes, estudiar el resultado de aplicar TFT a varios componentes de un mismo sistema, incluso en sistemas de distintos tamaños.

- **Recolección de información durante una falla**
  Investigar, analizar y definir comportamiento, efectos y postura de TFT frente a fallas del componente y/o del sistema (comportamientos con fallas). Por ejemplo, si se registran datos de I/O como casos de test y oráculo, sería importante reconocer, o saber, si estos fueron tomados en momentos de falla o no. Es importante notar también que no todos los metadatos recolectados en tiempo de ejecución son impactados o tienen consecuencias a partir de una falla, incluso en algunos el impacto puede llegar a ser despreciable.

- **Puesta en práctica**
  Realizar un conjunto de estudios empíricos para evaluar la técnica propuesta. Por ejemplo utilizándose un sistema real y una configuración real, aplicar la técnica a un conjunto de componentes. Previa puesta en marcha seleccionar los metadatos a registrar para cada

uno. Ponerlo en producción para ser utilizado por usuarios reales. En sus ejecuciones ir recolectando la información. Finalmente aplicar verdaderas tareas típicas de la evolución del software, como por ejemplo reutilizar alguno de los componentes instrumentados en alguna otra aplicación, o bien instalar la aplicación sobre otra configuración, ambiente, contexto. Y en esos casos ver los resultados, avisos o análisis arrojados como evaluación de nuevas experiencias. El experimento debe ser armado cuidadosamente ya que mucho dependerá de las elecciones particulares que se hagan, como por ejemplo para qué y cómo sea reutilizado un componente, qué valores de configuración o del contexto cambien, qué y cómo se analicen los resultados obtenidos. Una posibilidad es hacer las mismas reutilizaciones y reinstalaciones con los componentes instrumentados y no instrumentados, y tratar de ver qué situaciones se van planteando en uno y otro caso, ver si la información recolectada contribuye a las soluciones necesarias o a las aplicadas, y si es posible, ver qué información ayuda más a esos análisis y soluciones.
El objetivo de los estudios sería ver, al menos para el sistema seleccionado, si los datos de campo recolectados pueden efectivamente mejorar la calidad de los análisis dinámicos considerados. También tratar de ver si para el mismo sistema, las técnicas tradicionales computarían resultados que reflejen o no el uso y las necesidades reales del sistema. Experimentos como los de [OAH/03] apoyan la idea de encontrar resultados de éxito para la evaluación de TFT.
También sería interesante analizar si la técnica propuesta puede ser un complemento o un sustituto de técnicas y métodos que tal vez por su formalismo o su precisión son demasiado costosas para el tipo de software no crítico cotidiano, como la mayoría de los sistemas de información. Y de ese modo ver si es posible lograr una mejor relación costo-beneficio entre el proceso y los recursos de validación y verificación de software y el esfuerzo requerido para ponerlos en práctica.

9 CONCLUSIONES

En este trabajo hemos presentado la técnica TFT y sus principales elementos, una técnica que permite construir componentes self-knowledge, un intento de combinar la recolección de datos de campo con testing y verificación en tiempo de ejecución en un único framework para integrar y verificar CBS´s facilitando el reuso de componentes. TFT monitorea, interpreta y registra, en forma autónoma, información del componente y su contexto en tiempo de ejecución del componente a lo largo de todo su ciclo de vida. Con esa información, en formato de metadato, construye una base de conocimiento, su diario de viaje, buscando con ello proporcionarle al usuario del componente solucionar o reducir, sin depender del proveedor, los problemas a los que se enfrenta al utilizar y reutilizar el componente en el desarrollo de sus CBS´s, principalmente sobre aquellos componentes desarrollados por terceras partes.
Más específicamente, en este trabajo, hemos introducido un framework que proporciona un mecanismo general para monitorear externa y dinámicamente un componente, hemos presentado un diseño de la infraestructura de la técnica, y hemos visto qué información puede ser de utilidad monitorear y cómo puede ser utilizada. Si bien hemos ilustrado nuestra técnica en el contexto de un conjunto de atributos en particular, algunos tal vez muy cercanos sólo a algunas tecnologías, creemos que el framework es suficientemente general como para permitirnos utilizar varias tecnologías.
También hemos motivado la necesidad de varios metadatos de un componente que pueden ser explotados por los desarrolladores de una aplicación cuando ellos utilizan el componente en sus aplicaciones. Estos metadatos puede proporcionar información para ayudar en muchas de las tareas de ingeniería software en el contexto de sistemas basados en componentes. Hemos puesto el foco en el testing y en el análisis de los componentes, y con la ayuda de unos ejemplos discutimos el uso de metadatos para algunas tareas que un usuario de componente quizás quiera realizar en su aplicación. En algunos casos, la disponibilidad de los metadatos permitió disponer de información que típicamente no puede conseguir el usuario, mientras que en otros casos, mejoró el análisis, la precisión y, por lo tanto, la utilidad de la tarea que se estaba realizando. Estos son apenas algunos ejemplos de las clases de usos de los metadatos que imaginamos para sistemas basados en componentes. Hemos presentado un framework que se define genéricamente, permitiendo así manejar diferentes tipos de metadatos en diferentes dominios de aplicación, actuales y futuros.

Teniendo en cuenta que TFT permite crear auto-conocimiento en forma incremental y en forma continua a lo largo de toda la vida del componente, es decir, a través de los diferentes entornos de los que va formando parte y a través de sus evoluciones, podemos asegurar que TFT permite la auto-evaluación continua del componente a través de subsecuentes cambios ocurridos a lo largo de su ciclo de vida completo, y por lo tanto acerca la posibilidad de la auto-mejora del software, lo que encaja y sostiene las ideas del Análisis y Testing Perpetuo. A su vez vemos este trabajo como uno de los pasos preliminares para lograr el desarrollo de aplicaciones en el marco de la Informática Autónoma.

10 BIBLIOGRAFÍA

[AB/93] Arnold R. S. and Bohner S. A., "*Impact analysis - towards a framework for comparison*", in Proceedings of the International Conference on Software Maintenance, pages 292-301, September 1993.

[AH/02] Apiwattanapong T. and Harrold M. J., "*Selective path profiling*", in Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, pages 35-42, November 2002.

[ALR/01] Avizienis A., Laprie J.C. and Randell B., "*Fundamental Concepts of Dependability*", Research Report N01145, LAAS-CNRS, April 2001.

[BA/96] Bohner S. and Arnold R., *IEEE Software Change Impact Analysis*, Computer Society Press, Los Alamitos, CA, USA, 1996.

[BG/03] Beydeda S. and Gruhn V., "*State of the art in testing components"*, International Conference on Quality Software (QSIC, Nov. 06 - 07, Dallas, USA), IEEE Computer Society Press, 146-153, 2003.

[BGST/03] Beydeda S., Gruhn V., "*The Self-Testing COTS components (STECC) Strategy – a new form of improving component testability*", in Proceedings of the 29th Euromicro Conference (EUROMICRO'03), Belek-Antalya, Turkey, September 1-6, 2003.

[BIN/94] Binder R., "*Design for testability in object-oriented systems*", Communications of the ACM 37, pp. 87–101, 1994.

[BOH/02] Bowring J., Orso A. and Harrold M. J., "*Monitoring deployed software using software tomography*", in Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, pages 2-8, November 2002.

[CD/99] Cook J. and Dage J., "*Highly reliable ungrading components*", in Proceedings of the 21st International Conference on Software Engineering, pages 203-212, May 1999.

[CHGSS/04] Cheng S., Huang A., Garlan D., Schmerl B. and Steenkiste P., "*Rainbow: architecture-based self-adaptation with reusable infrastructure*", in Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC'04), New York, May 17 - 18, 2004.

[COR/98] Cherinka R., Overstreet C. M. and Ricci J., "*Maintaining a COTS integrated solution - Are traditional static analysis techniques sufficient for this new programming methodology?*", in Proceedings of the International Conference on Software Maintenance, pages 160-169, November 1998.

[COWL/02] Cobleigh J. M., Osterweil L. J., Wise A. and Lerner B. S., "*Containment units: a hierarchically composable architecture for adaptive systems*", in Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 159–165, ACM Press, 2002.

[DCOM/98] *Distributed Component Object Model protocol:DCOM/1.0*, January 1998.

[ECGN/01] Ernst M., Cockrell J., Griswold W. and Notkin D., "*Dynamically discovering likely program invariants to support program evolution*", IEEE Transactions on Software Engineering 27, pp. 99–123, 2001.

[EDW/01] Edwards S., "*A framework for practical, automated black-box testing of component-based software*", Software Testing, Verification and Reliability (STVR), 11(2), 2001.

[GAO/95] Garlan D., Allen R. and Ockerbloom J., "*Architectural mismatch: why reuse is so hard"*, IEEE Software, November 1995.

[GC/03] Ganek A. G. and Corbi T. A., "*The dawning of the autonomic computing era"*, IBM Systems Journal, Special Issue on Autonomic Computing.Vol 42, No 1, 2003.

[GKW/02] Garlan D., Kramer J. and Wolf A, editors, *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems* (WOSS´02), ACM Press, November 2002.

[GS/02] Garlan D. and Schmerl B., "*Model-based adaptation for self-healing systems*", in Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, pp.27-32, November 2002.

[GSC/01] Garlan D., Schmerl B. and Chang J., "*Using gauges for architecture-based monitoring and adaptation*", in Proceeding of the Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[HAR/00] Harrold M. J., "*Testing: a roadmap*", in The Future of Software Engineering (special volume of the proceedings of the International Conference on Software Engineering (ICSE)), pages 63-72, ACM Press, June 2000.

[HOR/01] Horn P., "*Autonomic computing: Ibm's perspective on the state of information technology*", in IBM Research, October 2001.

[HUN/98] Hunt G. C., "*Automatic distributed partitioning of component-based applications*", Technical Report TR695, University of Rochester, Computer Science Department, August 1998.

[JAV/00] *Javabeans documentation*, http://java.sun.com/beans/docs/index.html, October 2000.

[KOO/03] Koopman P., "*Elements of the self-healing system problem space"*, Workshop on Architecting Dependable Systems (WADS), May 2003.

[LJ/98] Lindquist U. and Jonsson E., "*A map of security risks associated with using cots*", IEEE Computer, 31(6):pages 60-66, June 1998.

[LMSGSS/98] Lowekamp B., Miller N., Sutherland D., Gross D., Steenkiste P. and Subhlok J., "*A resource query interface for network-aware applications*", in Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing (Chicago, IL), July 1998.

[LR/03] Law J. and Rothermel G., "*Whole program path-based dynamic impact analysis*", in Proceedings of the 25th International Conference on Software Engineering, pages 308-318, May 2003.

[LR/98] Liu C. and Richardson D., "*Software components with retrospectors"*, in Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA), June 1998.

[MAR/03] Mariani L., "*A fault taxonomy for component-based software*", in Proceedings of International Workshop on Test and Analysis of Component-Based Systems (TACoS), in conjunction with European Joint Conferences on Theory and Practice of Software (ETAPS), volume 82 of ENTCS, Elsevier Science, April 2003.

[MP/04] Mariani L. and Pezzè M., "*A technique for verifying component-based software*", in Proceedings of the 2nd International Workshop on Test and Analysis of Component Based Systems (TACoS 2004), Barcelona (Spain) 27-28 March, 2004.

[MTY/01] Martins E., Toyota C. and Yanagawa R., "*Constructing self-testable software components*", in Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01), pp. 151–160, June-July 2001.

[NZ/99] Neumann G. and Zdun U., "*Filters as a language support for design patterns in object-oriented scripting languages*", in Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems, pages 1-14, The USENIX Association, 1999.

[OAH/03] Orso A., Apiwattanapong T. and Harrold M. J., "*Leveraging field data for impact analysis and regression testing"*, in Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 128–137, September 2003.

[OC/00] Osterweil L. J. and Clarke L. A., "*Continuous self-evaluation for the self-improvement of software"*, 1st International Workshop on Self-Adaptive Software (IWSAS2000), Oxford, UK, Springer-Verlag, April 2000.

[OGTJMQRW / 99] Oriezy P., Gorlick M.M., Taylor R.N., Johnson G., Medvidovic N., Quilici A., Rosenblum D. and Wolf A., "*An architecture-based approach to self-adaptive software"*, IEEE Intelligent Systems 14(3):54-62, May/June 1999.

[OHR/00] Orso A., Harrold M. J. and Rosenblum D., "*Component metadata for software engineering tasks*", in Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), Davis, CA, November 2-3, 2000.

[OJH/03] Orso A., Jones J. and Harrold M. J., "*Visualization of program-execution data for deployed software*", in Proceedings of the ACM symposium on Software Visualization, pages 67-76, June 2003.

[OLHL/02] Orso A., Liang D., Harrold M. J. and Lipton R., "*Gamma system: continuous evolution of software after deployment*", in Proceedings of the International Symposium on Software Testing and Analysis, pages 65-69, July 2002.

[OST/96] Osterweil L., "*Perpetually testing software*", The Ninth International Software Quality Week (QW'96), San Francisco, May 21-24, 1996.

[PBDLRR/04] Pezzè M., Baldini A., Denaro G., Lipari G., Rossi M., Rogai D., "*QUACK: A Platform for the Quality of New Generation Integrated Embedded Systems*", in Proceedings of the 2nd International Workshop on Test and Analysis of Component Based Systems (TAC0S 2004), Barcelona (Spain) 27-28 March, 2004.

[PEZ/03] Pezzè M., editor, *Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS)*, ENTCS 82(6), April 2003.

[PY/99] Pavlopoulou C. and Young M., "*Residual test coverage monitoring*", in Proceedings of the 21th International Conference on Software Engineering (ICSE'99), pp. 277–284, May 1999.

[RH/97] Rothermel G. and Harrold M. J., "*A safe, efficient regression test selection technique*", ACM Transactions on Software Engineering and Methodology, 6(2):173-210, April 1997.

[RI/98] Richardson D. and Inverardi P., editors, *International Workshopon the Role of Software Architecture in Analysis and Testing (ROSATEA)*, June 1998.

[RKS/02] Raz O., Koopman P., and Shaw M., "*Enabling Automatic Adaptation in Systems with Under-Specified Elements*", 1st Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, November 2002.

[ROD/02] Rodriguez M. G., "*Automatic data-gathering agents for remote navigability testing*", IEEE Software 19, pp. 78–85, 2002.

[RT/01] Ryder B. G. and Tip F., "*Change impact analysis for object-oriented programs*", in Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, pages 46-53, June 2001.

[SHA/02] Shaw M., "*Self-healing: softening precision to avoid brittleness"*, in Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS´02), Charleston, South Carolina, pp.111-113, November 2002.

[TOS/04] Tosi D., "*Research perspectives in self-healing systems"*, Technical Report, Laboratorio di Test e Analisi del Software, Dipartamento di Informatico, Sistemistica e Comunicazione, Università degli Studi di Milano - Biocca, LTA:2004:06, June 2004.

[UY/00] Uchitel S. and Yankelevich D., "*Enhancing architectural mismatch detection with assumptions"*, in Proceedings of the Engineering of Computer Based Systems (ECBS 2000), pages 138–147, Edinburgh, Scotland, April 2000.

[VOA/98] Voas J., "*Maintaining component-based systems*", IEEE Software, 15(4):22-27, July-August 1998.

[WEY/98] Weyuker E., "*Testing component-based software: A cautionary tale*", IEEE Software, 15(5):54-59, September-October 1998.

[XML/00] *Extensible markup language (xml)*, http://www.w3.org/XML/, October 2000.

[XOTCL/00] *Xotcl - extended object tcl*, http://nestroy.wi-inf.uni-essen.de/xotcl/, November 2000.

11 APÉNDICE A. BIBLIOGRAFÍA DEL MAPA DE TECNOLOGÍAS E INVESTIGACIONES DE LOS SELF-HEALING SYSTEMS

[AAHLKP/97] Abowd G., Atkeson C., Hong J., Long S., Kooper R., and Pinkerton M., "*Cyberguide: a mobile context-aware tour guide*", Wireless Network, 3(5):421–433, 1997.

[AC/77] Avizienis A. and Chen L., "On the implementation of n-version programming for software fault tolerance during execution", in Proceedings of the IEEE COMPSAC 77, pages 149–155, November 1977.

[AHSWDKS / 02] Appavoo J., Hui K., Stumm M., Wisniewski R. W., Da Silva D., Krieger O., and Soules C. A. N., "*An infrastructure for multiprocessor run-time adaptation*", in Proceedings of the first workshop on Self-Healing Systems, pages 3–8. ACM Press, 2002.

[AK/02] Abercrombie P. and Karaorman M., "*jcontractor: Bytecode instrumentation techniques for implementing design by contract in java*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 70, Elsevier, 2002.

[AK/88] Ammann P. and Knight J., "*Data diversity: an approach to software fault tolerance*", in IEEE Transaction on Cmputers, volume 37, pages 418–425, 1988.

[BBGHO/98] Borg A., Blau W., Graetsch W., Herrmann F., and Oberle W., "*Fault tolerance under unix*", ACM Transaction Computer Systems, 7(1):1–24, February 1998.

[BDX/93] Bondavalli A., DiGiandomenico F., and Xu J., "Cost-effective and fexible scheme for software fault tolerance", Computer System Science and Engineering, 4:234–244, 1993.

[BEY/04] Beydeda S., "*The Self-Testing COTS Components (STECC) Method*", Martin Meidenbauer Verlag, Munchen, 2004.

[BFMW/01] Bartetzko D., Fischer C., Moller M., and Wehrheim H., "*Jass - java with assertions*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55, Elsevier, 2001.

[BIN/94] Binder R., "*Design for testability in object-oriented systems*", Communications of the ACM 37, pp. 87–101, 1994.

[BRO/96] Brown P. J., "*The stick-e document: a framework for creating context-aware applications*", in Proceedings of EP'96, Palo Alto, pages 259–272, also published in EP–odd, January 1996.

[CL/90] Cline M. P. and Lea D., "*Using annotated c++*", in Proceedings C++ at Work, September 1990.

[CNF/01] Cugola G., Nitto E. D., and Fuggetta A., "*The. JEDI event-based infrastructure and its application to the development of the OPSS WFMS*", Software Engineering, IEEE Transactions on, 27:827–850, September 2001.

[COWCL/01] Cobleigh J. M., Osterweil L. J., Wise A., Clarke L. A., and Lerner B. S., "*Architecting dynamic systems using containment units*", in Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[COWL/02] Cobleigh J. M., Osterweil L. J., Wise A. and Lerner B. S., "*Containment units: a hierarchically composable architecture for adaptive systems*", in Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 159–165, ACM Press, 2002.

[DF/02] de Lemos R. and Fiadeiro J. L., "*An architectural support for selfadaptive software for treating faults*", in Proceeding of the first workshop on Self-Healing Systems, pages 39–42. ACM Press, November 2002.

[DM/02] Dabrowski C. and Mills K., "*Understanding self-healing in service-discovery systems*", in Proceedings of the first workshop on Self-Healing Systems, pages 15–20. ACM Press, 2002.

[DMP/03] Denaro G., Mariani L., and Pezzè M., "*Self-test components for highly reconfigurable systems*", in Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03), volume ENTCS 82(6), April 2003.

[DR/02] Demsky B. and Rinard M., "*Automatic detection and repair of errors in data structures*", Technical Report MIT-LCS-TR-875, MIT Massachusetts Institute of Technology, December 2002.

[DR/03] Demsky B. and Rinard M., "*Automatic data structure repair for self-healing systems*", in First Workshop on Algorithms and Architectures for Self-Managing Systems, June 2003.

[ECGN/01] Ernst M., Cockrell J., Griswold W. and Notkin D., "*Dynamically discovering likely program invariants to support program evolution*", IEEE Transactions on Software Engineering 27, pp. 99–123, 2001.

[EGN/99] Ernst Y. K. M. D., Griswold W.G. and Notkin D., "Dynamically discovering pointer-based program invariants", Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 1999.

[FLO/67] Floyd R. W., "*Assigning meanings to programs*", in Proceedings of the Symposium Application Math., volume XIX, pages 19–32. American Mathematical Society, April 1967.

[FPV/98] Fuggetta A., Picco G., and Vigna G., "*Understanding Code Mobility*", IEEE Transactions on Software Engineering, 24(5):342–361, 1998.

[FS/01] Finkbeiner B. and Sipma H., "*Checking finite traces using alternating automata*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55. Elsevier, 2001.

[FSS/02] Finkbeiner B., Sankaranarayanan S., and Sipma H., "*Collecting statistics over runtime executions*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 70, Elsevier, 2002.

[GMK/02] Georgiadis I., Magee J., and Kramer J., "*Self-organising software architectures for distributed systems*", in Proceedings of the first workshop on Self-Healing Systems, pages 33–38. ACM Press, 2002.

[GRMD/01] Gates A. Q., Roach S., Mondragon O., and Delgado N., "*Dynamics: Comprehensive support for run-time monitoring*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55, Elsevier, 2001.

[GS/02] Garlan D. and Schmerl B., "*Model-based adaptation for self-healing systems*", in Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, pp.27-32, November 2002.

[GSC/01] Garlan D., Schmerl B. and Chang J., "*Using gauges for architecture-based monitoring and adaptation*", in Proceeding of the Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[GVKMOPSL / 79] German S. M., von Henke F. W., Karp R. A., Milne P. W., Oppen D., Polak W., Scherlis W., and Luckham D. C., "*Stanford pascal verifier user manual*", Technical Report TR-79-731, Dep. of Computer Science, Stanford University, Program Analysis and Verification Group Rep. 11, March 1979.

[HH/94] Harter A. and Hopper A., "*A distributed location system for the active office*", Network, IEEE, 8(1):62–70, February 1994.

[HLMR/74] Horning J., Lauer H., Melliar-Smith P., and Randell B., "*A program strcture for error detection and recovery*", in Lecture Notes in Computer Science, volume 16, pages 177–193, 1974.

[HR/03] Havelund K. and Rosu G., "*An overview of the runtime verification tool java pathexplorer*", Netherlands, 2003, Kluwer Academic Publishers.

[ILL/75] Igarashi S., London R. L., and Luckham D. C., "*Automatic program verification i: A logical basis and its implementation*", in Acta Informatica, volume 4, pages 145–182, 1975.

[JR/02] Julien C. and Roman G., "*Egocentric context-aware programming in ad hoc mobile environments*", SIGSOFT Software Engineering, Notes, 27(6):21–30, 2002.

[KCL/02] Kumar S., Cohen P., and Levesque H., "*The adaptive agent architecture: achieving fault-tolerance using persistent broker teams*", in Proceedings of Fourth International Conference on MultiAgent Systems, pages 159–166. IEEE, 2002.

[KIM/84] Kim K., "*Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults*", in Proceedings of the Fourth International Conference on Distributed Computing Systems, pages 526–532, 1984.

[KKLSV/01] Kim M., Kannan S., Lee I., Sokolsky O., and Viswanathan M., "*Java-mac: a run-time assurance tool for java programs*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55, Elsevier, 2001.

[KKLSV/02] Kim M., Kannan S., Lee I., Sokolsky O., and Viswanathan M., "*Computational analysis of run-time monitoring - fundamentals of java-mac*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 70, Elsevier, 2002.

[KMSF/01] Kortenkamp D., Milam T., Simmons R., and Fernandez J. L., "*Collecting and analyzing data from distributed control programs*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55, Elsevier, 2001.

[KRA/98] Kramer R., "*icontract-the javatm design by contracttm tool*", in IEEE, editor, Technology of Object-Oriented Languages, Proceedings, pages 295–307, August 1998.

[KT/87] Koo R. and Toueg S., "*Checkpoint and rollback-recovery for distributed systems*", IEEE Transaction on Software Engineering, 13(1):23–31, January 1987.

[LR/98] Liu C. and Richardson D., "*Software components with retrospectors"*, in Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA), June 1998.

[MAR/04] Mariani L., "Capturing and synthesizing the behavior of component-based systems", Technical Report LTA:2004:01, DISCO, University of Milano Bicocca. LTA lab., February 2004.

[MEY/97] Meyer B., "*Object-Oriented Software Construction*", ISE, 2nd edition, 1997.

[MIN/03] Minsky N., "*On conditions for self-healing in distributed software systems*", in Proceedings of the Autonomic Computing Workshop, pages 86–92. IEEE, 2003.

[MK/96] Magee J. and Kramer J., "*Self-organising software architectures*", in Proceedings of the 2nd International Software Architecture Workshop, 1996.

[MP/04] Mariani L. and Pezzè M., "*A technique for verifying component-based software*", in Proceedings of the 2nd International Workshop on Test and Analysis of Component Based Systems (TACoS 2004), Barcelona (Spain) 27-28 March, 2004.

[MTY/01] Martins E., Toyota C. and Yanagawa R., "*Constructing self-testable software components*", in Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01), pp. 151–160, June-July 2001.

[NE/01] Nimmer J. W. and Ernst M. D., "*Static verification of dynamically detected program invariants: Integrating daikon and esc/java*", in K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55, Elsevier, 2001.

[OGTJMQRW / 99] Oriezy P., Gorlick M.M., Taylor R.N., Johnson G., Medvidovic N., Quilici A., Rosenblum D. and Wolf A., "*An architecture-based approach to self-adaptive software"*, IEEE Intelligent Systems 14(3):54-62, May/June 1999.

[OHR/00] Orso A., Harrold M. J. and Rosenblum D., "*Component metadata for software engineering tasks*", in Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), Davis, CA, November 2-3, 2000.

[PIC/98] Picco G. P., "*μCode: A Lightweight and Flexible Mobile Code Toolkit*", in K. Rothermel and F. Hohl, editors, Proceedings of the 2nd International Workshop on Mobile Agents, volume 1477 of Lecture Notes in Computer Science, pages 160–171. Springer-Verlag: Heidelberg, Germany, 1998.

[PR/90] Peng and Reggia, "*Abductive Inference Models for Diagnostic Problem Solving*", Springer-Verlag, 1990.

[PY/99] Pavlopoulou C. and Young M., "*Residual test coverage monitoring*", in Proceedings of the 21th International Conference on Software Engineering (ICSE'99), pp. 277–284, May 1999.

[RJP/04] Roman G., Julien C., and Payton J., "*A formal treatment of context-awareness*", in Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004), pages 12–36, Springer, April 2004.

[RTB/02] Reilly D., Taleb-Bendiab A., and Badr N., "*A conflict resolution control architecture for self-adaptive software*", in Proceedings of the International workshop on Architecting Dependable systems (WADS), 2002.

[RTLB/02] Reilly D., Taleb-Bendiab A., Laws A., and Badr N., "*An instrumentation and control-based approach for distributed application management and adaptation*", in Proceedings of the first workshop on Self-Healing Systems, pages 61–66. ACM Press, 2002.

[SAW/94] Schilit B., Adams N., and Want R., "*Context-aware computing applications*", in Proceedings of Workshop on Mobile Computing Systems and Applications, pages 85–90, December 1994.

[SDA/99] Salber D., Dey A. K., and Abowd G. D., "*The context toolkit: aiding the development of context-enabled applications*", in Proceedings of the SIGCHI conference on Human factors in computing systems, pages 434–441, ACM Press, 1999.

[SG/02] Schmerl B. and Garlan D., "*Exploiting architectural design knowledge to support self-repairing systems*", in Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering, pages 241–248. ACM Press, 2002.

[SGM/85] Scott R., Gault J., and McAllister D., "*The consensus recovery block*", in Proceedings of th Total System Reliability Symposium, pages 74–85, 1985.

[SY/85] Strom R. and Yemini S., "*Optimisitc recovery in distributed systems*", ACM Transaction Computer Systems, 3(3):204–226, August 1985.

[TDJ/99] Traon Y. L., Deveaux D., and Jezequel J. M., "*Self-testable components: from pragmatic tests to design-to-testability methodology*", in Technology of Object-Oriented Languages and Systems (TOOLS), pages 96–107, IEEE Computer Society Press, 1999.

[VK/02] Valetto G. and Kaiser G., "A case study in software adaptation", in Proceedings of the first workshop on Self-Healing Systems, pages 73–78. ACM Press, 2002.

[VK/03] Valetto G. and Kaiser G., "*Using process technology to control and coordinate software adaptation*", in Proceedings of the 25th international conference on Software Engineering, pages 262–272. IEEE Computer Society, 2003.

[YCSSSM/02] Yang Z., Cheng B. H. C., Stirewalt R. E. K., Sowell J., Sadjadi S. M., and McKinley P. K., "*An aspect-oriented approach to dynamic adaptation*", in Proceedings of the first workshop on Self-Healing Systems, pages 85–92. ACM Press, 2002.