Framework para generación y despliegue de monitores dinámicos de rendimiento en sistemas software autoadaptativos

Miguel Jiménez, MSc.
Gabriel Tamura, Ph.D

Citación

M. Jiménez y G. Tamura, "Framework para generación y despliegue de monitores dinámicos de rendimiento en sistemas software autoadaptativos," en *Bitácoras de la maestría*, vol. 3, *Monitores dinámicos de software - Despliegue de software - Monitoreo de espectro*, Cali, Colombia: Universidad Icesi, 2020, pp. 25-108.

RESUMEN

La prestación continua de servicios de software y el cumplimiento de niveles de rendimiento acordados respecto a dichos servicios son una preocupación de las empresas como respuesta a los requerimientos de un mercado cada vez más competitivo. Los avances en computación autonómica para fortalecer la capacidad de respuesta y recuperación en la prestación de servicios han promovido el diseño de sistemas reconfigurables, capaces de modificar su estructura en tiempo de ejecución. Garantizar niveles específicos de rendimiento en sistemas dinámicos implica disponer de mecanismos para evaluar métricas que deben ser actualizadas periódicamente, siguiendo la evolución de los requerimientos del sistema y su entorno. Para asegurar el rendimiento de los servicios, los administradores de sistemas requieren infraestructuras de monitoreo que midan continuamente la satisfacción de los diversos factores de rendimiento capaces de: actualizar de forma dinámica sus estrategias de monitoreo, de acuerdo con la evolución de los requerimientos del sistema o su entorno; desplegar e integrar los componentes de monitoreo en tiempo de ejecución; y proveer los medios para generar funcionalidades de monitoreo componibles, rastreables y controlables. Estos retos se abordaron en este proyecto mediante el diseño de una arquitectura de monitoreo dinámica y escalable, que implementa y resuelve preocupaciones de monitoreo dinámico en sistemas autonómicos sensibles al contexto, y del diseño e implementación –con base en ella–, de Pascani y Amelia, dos lenguajes de dominio específico que facilitan el desarrollo del monitoreo citado, adecuados para ser integrados en la arquitectura y para automatizar su despliegue en la infraestructura del sistema objetivo durante su operación.

INTRODUCCIÓN

Los sistemas software son una herramienta de apoyo fundamental para la realización de las actividades cotidianas en los contextos personal y empresarial: las personas, por una parte, dependen de una variedad de software para cumplir los deberes y responsabilidades de su diario vivir, en ello, las aplicaciones multimedia tienen un rol destacado pues permiten la interacción con las redes sociales, los servicios de mensajería y los *streaming online* de música, películas y programas televisivos; por su parte, las compañías son altamente dependientes de tecnologías de software para la correcta operación y el cumplimiento de sus objetivos de negocio, para ellas el software no es únicamente una herramienta para lidiar y manejar tediosas tareas administrativas, sino que es el puente para que la infraestructura de los negocios y el personal entreguen servicios de valor agregado a sus clientes. Estas dos perspectivas muestran la creciente dependencia de los usuarios en las aplicaciones de software y como resultado de ello sus crecientes expectativas de calidad de los servicios y las aplicaciones ofrecidos. Los usuarios, por ejemplo, desean un servicio de *streaming* de video con reproducciones continuas, sin pausas, por lo que, con el fin de fortalecer las líneas de negocio, los interesados (*stakeholders*) y los actores principales están pendientes del cumplimiento de los atributos de calidad (QA, *Quality Attributes*), especialmente de aquellos sensibles que impactan el comportamiento de los sistemas y la percepción del cliente final.

Con el fin de asegurar el cumplimiento de los QA del sistema (garantizar los acuerdos de nivel de servicio), los diseñadores de software consideran mecanismos para medir la satisfacción del cliente que les permitan predecir problemas y dificultades capaces de desviar al sistema de su adecuada operación. Dichas dificultades pueden identificarse, ya sea en etapas tempranas de desarrollo — permitiendo a los arquitectos su correcta resolución, desde una perspectiva de diseño— o en la etapa operacional, dejando a los administradores de sistemas como responsables de subsanarlas. Una efectiva aserción se puede alcanzar al identificar adecuadamente las causas raíz de los problemas detectados y generar acciones correctivas para solventarlos. Delegar tal responsabilidad en los administradores de sistemas implica que el personal de Tecnologías de la Información (TI) reaccione a cambios en el sistema durante horas de producción, lo que conlleva a un ineficiente aseguramiento de la calidad al permitir que los procesos de identificación y respuesta realizados por el personal retrasen las correspondientes acciones de mitigación; por el contrario,

el aseguramiento de la calidad en producción debería valerse de procesos sistemáticos; que deberían ser fundamentados en la realización de acciones de monitoreo y análisis de fallas como mecanismos de medición, monitoreo y control del comportamiento del sistema [1].

Como resultado, una tarea crítica para cumplir continuamente con los QA del sistema es medir y monitorear su comportamiento, no obstante, puesto que monitorear no se considera como una entidad de primera clase en el proceso tradicional de ingeniería de software, la medición de variables relevantes, relacionadas con los QA del sistema, se realiza generalmente al desarrollar y adicionar de manera manual porciones de código de medición en distintas ubicaciones del código fuente de la aplicación, lo que resulta en complicadas implementaciones de bajo nivel [1]. Pese a que este enfoque inicialmente parece simple, presenta múltiples problemas cuando los componentes de las aplicaciones son generados automáticamente: primero, después de la medición y generación de los componentes del sistema y su correspondiente modificación manual con el código de medición, los mecanismos de generación no pueden volver a utilizarse, pues hacerlo implicaría la pérdida del código insertado manualmente; segundo, como las variables de medición no están definidas como parte del mecanismo estándar de medición, son difíciles de ubicar y compartir entre desarrolladores [1]; y tercero, el código insertado manualmente para la medición de objetivos en el sistema lo hace no reutilizable para otros proyectos o versiones.

Más allá de procesar datos de monitoreo, debería existir una lógica de monitoreo a cargo de procesar y ensamblar información en forma de variables y eventos de contexto para reportarla a los interesados adecuadamente. Variables como el tiempo de respuesta y las ocurrencias de error son de especial interés para los administradores del sistema y los directores de proyectos, respectivamente. En la práctica se tienen en cuenta dos consideraciones de monitoreo: primero, cuando se descubre un problema, tal como un tiempo de respuesta del servicio excesivamente largo o un alto consumo de memoria en un corto intervalo, cómo identificar los componentes del sistema causantes del problema, por lo que los datos reportados deben contener la información necesaria para descomponer las mediciones monitoreadas que permita profundizar en las potenciales causas; segundo, los escenarios de calidad son sujetos a cambio ya sea porque son renegociados o porque otros han tomado más relevancia, por lo que el código de medición desarrollado debe ser analizado y actualizado cuidadosamente

por los desarrolladores de la aplicación, puesto que no ofrecen soporte para estrategias de monitoreo autoadaptativo buscando corregir cambios en los requerimientos de monitoreo [2].

El desarrollo de soluciones de monitoreo efectivas requiere, además de mecanismos funcionales, de elementos que provean una adecuada y estandarizada especificación de formatos con un nivel de abstracción y expresividad idóneo. Estas características configuran el escenario adecuado para considerar a los lenguajes específicos de dominio (DSL, *Domain Specific Languages*) como una alternativa para la automatización en la generación de componentes de monitoreo sistemáticamente. Los DSL pueden mejorar la flexibilidad y confiabilidad e incrementar así la productividad [3] y adicionalmente, reducir considerablemente los esfuerzos en componer medidas y especificaciones de monitoreo. Tales soluciones deben soportar operaciones en tiempo de ejecución –esto es, operaciones a ser aplicadas con el sistema en ejecución–, las que incluyen el manejo de parámetros y mediciones personalizadas, la modificación en la frecuencia de muestreo de medidas discretas y el control del mecanismo de monitoreo en sí. Con el objetivo de aplicar efectivamente estas operaciones, los administradores de sistemas deben ser capaces de desplegar automática y confiablemente la infraestructura de monitoreo necesaria para obtener información relevante del sistema. Para que estas operaciones en tiempo de ejecución sean confiables, los componentes generados deben ser capaces de reportar datos relevantes de su ejecución durante operaciones regulares, incluyendo los instantes previos y posteriores a la instalación de actualizaciones.

Aunque los administradores de sistemas están continuamente monitoreando el comportamiento del sistema, las condiciones de contexto pueden cambiar dramáticamente periodos cortos de tiempo, dejando pocas oportunidades de reaccionar oportuna y adecuadamente. Mientras esto sucede, las interrupciones de servicio emergen haciendo que las expectativas de calidad no se cumplan y reduciendo la confianza del cliente. También, puesto que los sistemas están en constante evolución y crecimiento hacia redes y componentes distribuidos, la efectiva administración y promesa de entrega del servicio se convierte en un desafío latente [4]. En el dominio de las redes sociales, un ejemplo destacado se da cuando Twitter presentó problemas con la infraestructura de red y su capacidad asociada: entre 2008 y 2012 reportaron diversas fallas generales del sistema [5] antes de un cambio mayor realizado en 2013 [6]. Algunos de estos

problemas generales –junto con incrementos en el tiempo de respuesta– fueron causados por peticiones de servicio inesperadas durante la Copa Mundial de la FIFA en 2010 [7] y los Juegos Olímpicos de 2012 [8]. Para superar estas situaciones, se automatizaron tareas administrativas dirigiendo el conocimiento del personal de TI hacia mecanismos automatizados capaces de responder adecuadamente a contextos cambiantes, enfoque que permitió mejorar las capacidades de respuesta y resiliencia del servicio en general [4]. Estas tareas se han catalogado como habilidades autogestionadas como paradigma de la computación autónoma, esto es sistemas que se gestionan a sí mismos de acuerdo con objetivos de alto nivel especificados por sus administradores [9].

A pesar de que las capacidades de autogestión reducen la intervención humana en la administración de sistemas, desarrollar mecanismos autoconscientes –mecanismos capaces de habilitar sistemas con cierto nivel de conciencia respecto a su propio comportamiento–, requiere de herramientas de monitoreo capaces de actualizar dinámicamente sus estrategias de medición, lo que conlleva a que los procesos de medición deban ser removidos de los componentes ya monitoreados y desplegados en nuevos componentes para empezar a monitorearlos. Asimismo, debe ser posible modificar el conjunto de variables monitoreadas para añadir nuevas o eliminar las existentes; como resultado, debe ser posible sustituir la lógica de monitoreo de forma dinámica, en tiempo de ejecución, como se analiza en los trabajos de Villegas et al. [10] y Tamura et al. [11]. Una renovación parcial o total de la infraestructura de monitoreo requiere desarrollar acciones de despliegue en los monitores, esto es: compilar nuevas implementaciones de monitoreo, transportarlas a sus respectivos nodos de computación, ejecutarlos y resolver las dependencias vinculadas [12]; también se deben remover versiones previas y recompilar algunas partes del sistema en la infraestructura en producción. Como en el caso de monitoreo, de acuerdo con el principio de separación de intereses [13], [14], dichas acciones de despliegue requieren formatos personalizados de especificación y un adecuado poder de expresión.

En resumen, la motivación detrás de esta investigación se encuentra en la necesidad de proveer sistemas software con infraestructuras de monitoreo generadas automáticamente y desplegadas en tiempo de ejecución; se requiere que dichas infraestructuras garanticen los QA en aplicaciones de software que enfrentan condiciones cambiantes en el contexto, las cuales puedan violar el requerimiento de dichos atributos en tiempo de ejecución. A pesar de las

propuestas existentes enfocadas en la ingeniería de software para sistemas software autoadaptativos –como el modelo de referencia Dynamico–, los cambios mencionados se encuentran abiertos [10].

El objetivo general de esta investigación fue definido como: "Desarrollar mecanismos estándar que generen infraestructuras de monitoreo para factores de desempeño (latencia, *throughput*, etc.) capaces de desplegar distintas estrategias de medición en tiempo de ejecución, con el fin de satisfacer continuamente indicadores de nivel de servicio en sistemas software basados en servicios–componentes". Para su logro, se establecieron los siguientes objetivos específicos: diseñar una arquitectura de referencia de software para la infraestructura de monitoreo requerida para supervisar la satisfacción del desempeño de los QA en sistemas software basados en componentes con protocolos de comunicación estándar bien definidos; desarrollar mecanismos estándar que generen componentes de monitoreo ajustables y trazables con los correspondientes procedimientos de medida, para la especificación de problemas de monitoreo relacionados con el desempeño de los QA; diseñar y desarrollar mecanismos estándar para desplegar componentes software, incluyendo la preparación de activos objetivo, su transporte a (posiblemente) nodos de computación distribuida, su ejecución y la limpieza de recursos del sistema; y diseñar y desarrollar una estrategia para integrar estrategias de medición al desplegar artefactos de monitoreo en tiempo de ejecución y al actualizar la infraestructura del sistema objetivo.

MARCO TEÓRICO Y ESTADO DEL ARTE

INGENIERÍA DE SOFTWARE BASADA EN COMPONENTES

La ingeniería de software basada en componentes (CBSE, *Component–Based Software Engineering*) es un enfoque del desarrollo de software basado en la reutilización de software enfocándose en un conjunto de principios de diseño y estándares para la implementación, documentación y despliegue encapsulados en un modelo de componente. En este paradigma, los componentes son unidades de software opaco con servicios bien definidos y dependencias explicitas sobre otros componentes; los servicios son visibles a través de interfaces, lo que hace posible a los componentes basarse en comportamientos previstos, lo que los hace desarrollables y desplegables independientemente de ellos [15]. La interacción

entre componentes se realiza al comunicar componentes uniendo servicios requeridos (dependencias) con los servicios prestados a través de protocolos de comunicación (SOAP, *Simple Object Access Protocol*; REST, *Representational State Transfer*; o RMI, *Remote Method Invocation*).

De acuerdo con la visión de la CBSE, los componentes son unidades de software independientes y flexibles que permiten la construcción de sistemas con predictibilidad mejorada basándose en las propiedades de los componentes, sus mercados y su reducido *time–to–market* (tiempo al mercado) [15]. Sin embargo, uno de los retos abiertos de la CBSE es la confianza de los componentes, es decir en la confiabilidad de los componentes de fuentes desconocidas y en quién certifica la calidad de dichos componentes. Estas y otras preocupaciones han sido abordadas por la comunidad de software con nuevos modelos de componentes (Arquitectura de Componentes de Servicio [16]) *middleware* como FraSCAti [17] y tecnologías relacionadas.

Extendiendo la visión de la CBSE, la especificación SCA (*Service Component Architecture*) define un enfoque general para ensamblar aplicaciones empresariales (EA, *Enterprise Applications*) basado en componentes y servicios y provee un mecanismo estándar para considerar servicios individuales en piezas de negocio de alto nivel [18], [19]. Por consiguiente, puesto que la SCA sigue la visión de la Arquitectura Orientada a Servicios (SOA, *Service Oriented Architecture*), soportar dichos componentes de alto nivel hace que SCA no sólo estandarice, sino que también simplifique la construcción, el desarrollo y la gestión de las EA.

Los componentes de alto nivel se ejecutan a través de compuestos, descriptores XML que contienen dependencias del servicio y componentes tanto de los servicios requeridos como de los provistos. Estos componentes internos en una aplicación SCA pueden implementarse utilizando diversos lenguajes de programación o tecnologías, tales como Java, C++, OSGi y BPEL, y vincularse mediante diferentes implementaciones de enlace, tales como: servicios Web (*Web Services*), JMS (*Java Message Service*), RMI y JCA (*Java EE Connector Architecture*).

Las actuales implementaciones de la especificación SCA incluyen proyectos de código abierto, como Apache Tuscany [20], Fabric3 [21] y FraSCAti [17]; y productos comerciales como el paquete de características de servicio de IBM (Websphere) [22] y Oracle Tuxedo [23]. No obstante, FraSCAti es la única implementación que se puede reconfigurar durante el tiempo de ejecución.

CBSE y SCA son relevantes para esta investigación porque definen las plataformas objetivo para la generación de código.

SISTEMAS SOFTWARE AUTOGESTIONABLES

Para abordar el incremento en la complejidad de los sistemas de computación actuales, los modelos de computación autónoma surgen como respuesta a la constante dificultad de gestionar sistemas más allá de la gestión de ambientes individuales de software [9]. Un sistema de computación autónomo puede manejarse a si mismo dadas ciertas directrices de alto nivel –políticas– de sus administradores. Estos sistemas monitorean continuamente su operación buscando detectar cambios en las condiciones internas o externas que afecten el cumplimiento de sus atributos de calidad y ajustan su operación para garantizar un alineamiento con los objetivos de alto nivel [9].

En el núcleo de la computación autónoma se encuentra la autogestión como un enfoque para reducir la intervención humana en las tareas de mantenimiento y administración detalladas. Dichas tareas se clasifican en cuatro propiedades de autogestión: autoconfiguración, esto es la habilidad de alcanzar la configuración autónoma de los componentes y sistemas desde políticas de alto nivel; autooptimización, que es la habilidad de los componentes y del sistema de continuamente buscar la eficiencia de sus parámetros y mejorar su rendimiento; autosanación, que corresponde a la habilidad del sistema para descubrir y diagnosticar fallas de software o hardware y recuperarse ante ellas; y autoprotección, que es la capacidad de anticiparse y defenderse de ataques maliciosos o fallos en cascada no planeados.

Kephart y Chess, investigadores de IBM, en su enfoque de arquitectura para sistemas autogestionables proponen la estructura de un elemento autónomo [9]. Los elementos de su enfoque utilizan el modelo de referencia Monitorear–Analizar–Planear–Ejecutar sobre una base de conocimiento compartida (MAPE–K, *Monitor–Analyze–Plan–Execute over a shared Knowledge base*), donde cada elemento tiene sus propias responsabilidades respecto de adaptaciones a nivel de sistema, sea por modificaciones estructurales o de comportamiento de este, así [24]:

- el monitor colecta información de contexto relevante del sistema objetivo (sistema gestionado) como latencia del servicio y *throughput*, y datos acerca del estado de la infraestructura de computación;

- el analizador analiza datos contextuales y determina si una adaptación debe ejecutarse o no;
- el organizador sintetiza un conjunto de acciones (plan de adaptación) para alterar el comportamiento del sistema de acuerdo con los síntomas de adaptación definidos por el analizador;
- los ejecutores realizan el plan de adaptación a través de los mecanismos de adaptación disponibles, tales como reconfiguración de la arquitectura y el ajuste de parámetros; y
- la base de conocimiento es el conjunto de fuentes de datos que habilitan el compartirlos –acción requerida para realizar decisiones de autogestión– entre monitores, analizadores, organizadores y ejecutores.

La relevancia de la concepción y visión del software autogestionado en esta investigación es que constituye la estrategia fundacional de la solución descrita en este documento, aun cuando se enfoque en el aspecto de monitoreo.

LENGUAJES ESPECÍFICOS DE DOMINIO

Como su nombre lo indica, un DSL es un lenguaje de propósito específico cuya sintaxis y semántica están adaptados para artefactos específicos de software (especificaciones del programa) en un dominio de aplicación particular (pruebas, monitoreo, seguridad); está diseñado para proveer una notación específica con el fin de expresar soluciones en diferentes niveles de abstracción, utilizando el vocabulario del dominio de la aplicación. Como resultado, un DSL bien diseñado es más flexible y efectivo que una librería tradicional, mejora la productividad del programador, mantiene los costos bajo control y permite una adecuada comunicación con expertos de dominio [25]. También, gracias a los DSL los principales interesados pueden valorar la importancia de validar y modificar dichas especificaciones del programa [26].

Los DSL incrementan, no sólo la productividad, sino también la flexibilidad y confiabilidad de los sistemas software [3]. Desarrollar un DSL puede conllevar a la generación y ensamblado automático de código, produciendo soluciones menos propensas a errores; sin embargo, a pesar de dichas ventajas, como mencionan Deursen y Klint [27], existen dos desventajas de mantenimiento: la primera, utilizar DSL implica un cambio desde el mantenimiento hecho a mano para aplicaciones a mantener programas DSL especificando cada aplicación

–aunque posiblemente con elementos reusados–, el compilador DSL y la librería DSL –que contiene el conjunto básico de objetos útiles–; la segunda, que aun cuando los lenguajes de programación son más difíciles de aprender y utilizar, existe suficiente material de aprendizaje disponible y profesionales experimentados, mientras que para los nuevos DSL, todo el material debe ser creado por sus desarrolladores. Adicionalmente, otros autores, como González [1] y Spinellis y Guruprasad [28], consideran como desventaja la falta de familiaridad acerca de cómo encajar un DSL en un proceso de desarrollo regular. Una parte importante de las contribuciones de esta investigación se basan en DSL.

MONITOREO DEL SISTEMA

En vista de la incesante competencia entre mercados y al continuo aumento de la demanda de servicios ubicuos, las compañías se preocupan por medir y mejorar la eficiencia operacional, a partir de la obtención de datos directamente desde la infraestructura, lo que ha incrementado la demanda hacia mecanismos avanzados que provean el monitoreo continuo de sistemas que soportan las actividades del negocio [1]. El monitoreo continuo difiere de otras técnicas de medición del desempeño (evaluación por perfil) en que su objetivo final no está relacionado con mediciones individuales, sino con su permanente aplicación. Desde un conjunto de medidas obtenidas se calcula un valor común y se compara con un valor de referencia como indicador de nivel de servicio. La información detallada acerca de las mediciones individuales es útil para analizar causas raíces de comportamientos inesperados del sistema, no obstante, capturar eventos se puede tornar difícil de implementar en aplicaciones en tiempo real [29], [30].

El principal objetivo en la implementación de sistemas de monitoreo es proveer los medios para reportar actividades de alto nivel que permitan el procesamiento para responder a preguntas relacionadas con un nivel de negocio, *e.g.*, ¿Qué está causando la falla en el sistema al cumplir el atributo de calidad X? ¿Por qué después de la última actualización del sistema las búsquedas de usuario se redujeron en Y%? Responder estas preguntas requiere poder analizar el sistema en términos de cómo se emplea, en lugar de hacerlo en términos de cómo se ha construido, común en el monitoreo de tecnología [29].

Para medir y controlar eficientemente las operaciones, se requiere de herramientas y mecanismos que permitan evaluar el estado del sistema, su

comportamiento y otras variables de desempeño y estado general. Actualmente, el enfoque principal para monitorear y evaluar dichas variables es a través del uso de métricas que se capturan en ubicaciones críticas de medición definidas por los componentes del sistema y permiten caracterizar la calidad en la prestación del servicio [31].

Los mecanismos efectivos de monitoreo deben considerar al menos dos tareas principales: el proceso de medición de los eventos y su correlación y evaluación. El proceso de medición se realiza con sensores in situ que generan eventos que contienen datos de bajo nivel relacionados con ejecuciones del servicio, tales como: tiempos de ejecución, consumo de memoria y comportamientos excepcionales; el proceso de correlación y evaluación, por su parte, reúne medidas de diferentes fuentes (sensores), las compone y las computa hacia valores puntuales. En caso de que se encuentre un comportamiento no deseado, los elementos de monitoreo reportan a otros elementos MAPE–K. La caracterización del comportamiento del sistema no es una tarea trivial, puesto que los sistemas producen millones de eventos por unidad de tiempo, lo que hace que entender el comportamiento del sistema sea un reto abierto [29]. La tecnología actual le permite a los sistemas únicamente ver dichos eventos, pero no razonar o tomar conciencia de ellos.

Un primer paso hacia el entendimiento del comportamiento de un sistema es monitorear la causalidad de los eventos, esto es, tener la capacidad de trazar la incidencia de un evento en la ocurrencia de eventos subsiguientes. La causalidad de eventos se denomina horizontal cuando los eventos causante y causado ocurren al mismo nivel conceptual en el sistema y vertical cuando dicha relación se determina por las diferentes capas que componen al sistema [29]. En esta última, los eventos son llamados comúnmente eventos de bajo o alto nivel, dependiendo de qué tan cerca se encuentren de la capa de negocio. Desde el monitoreo de la causalidad se pueden identificar complejos parámetros y correlacionarlos con información contextual, como la del estado actual del sistema o la época del año, lo que puede ayudar a evitar ciertos comportamientos gracias a la generación de mecanismos de defensa apropiados en el sistema.

Perfilar, monitorear y rastrear

Perfilar, monitorear y rastrear son técnicas empleadas para identificar comportamientos específicos en un sistema en operación, cada una de ellas tiene su valor agregado y un propósito principal que es específico para

una restricción dada. Monitorear es comúnmente una actividad constante donde se observan elementos o propiedades del sistema, con base en ello, los mecanismos de monitoreo disparan alertas y notifican a los interesados cuando los elementos en observación sobrepasan un valor específico. Perfilar software es una técnica de análisis dinámico de programas que mide el uso de recursos, tales como CPU, memoria, porcentajes de entrada y salida y tiempo de ejecución. y generalmente se usa para identificar código candidato para evaluaciones de desempeño –puesto que es una actividad que demanda una considerable cantidad de recursos, se emplea en ambientes de desarrollo, al contrario del monitoreo de software que se usa en producción–. el rastreo de software se refiere a seguir el rastro de la ejecución de un programa, lo que generalmente se usa en pruebas de software, y puede ser útil en diferentes escenarios, como: un rastreo de llamadas que ayuda a determinar por qué un programa falla o no responde como se esperaba; la cobertura de código, que registra qué partes de un programa se ejecutaron en un conjunto de pruebas; y la depuración en vivo, que permite ejecutar un programa instrucción por instrucción para identificar errores de programación.

Monitoreando el desempeño de un sistema

De acuerdo con Barbacci et al. [32], el desempeño de un sistema depende de la naturaleza de los recursos utilizados para cumplir peticiones y de cómo los recursos compartidos se gestionan cuando se presentan múltiples peticiones sobre dichos recursos. Según la taxonomía del desempeño presentada por estos autores, las restricciones de desempeño o requerimientos utilizados para especificar y corregir el desempeño de un sistema son: latencia, la ventana de tiempo durante la cual un evento debe ser procesado y se debe producir una respuesta; *throughput*, el número de respuestas que se han completado en un intervalo de observación dado; capacidad, la cantidad de trabajo que un sistema puede desarrollar, definida generalmente en términos de la cantidad máxima de *throughput* que es posible alcanzable sin violar los requerimientos de latencia [33]; y modos, la respuesta del desempeño de un sistema al encarar escenarios diferentes (o cambiantes) respecto de la latencia, el *throughput* y la capacidad.

Las anteriores restricciones de rendimiento ayudan a especificar el desempeño esperado de un sistema desde diferentes ángulos y pueden ser personalizadas a necesidades específicas para que las medidas de desempeño tengan un significado particular dependiendo del problema que el sistema resuelve.

DESPLIEGUE DE SOFTWARE

El despliegue de software es un proceso de posproducción que realiza el usuario durante el periodo que transcurre entre la adquisición del software y su ejecución [34], considerado como un conjunto de actividades interrelacionadas dentro del ciclo de vida del despliegue.

El ciclo de vida del despliegue de software

Hall, Heimbigner y Wolf presentan un ciclo de vida del despliegue de software compuesto por varias actividades interrelacionadas [35]; Dubus [36],por su parte, describe los diferentes estados de un sistema durante este ciclo de vida donde las transiciones son actividades de despliegue clasificadas en dos roles: productor, que consiste en crear una versión empaquetada del software para facilitar un producto entregable; y consumidor, que hace referencia a la configuración del paquete entregado para que sea utilizable (ver Figura 1).

En el lado del productor se identifican dos actividades: liberación (*release*) y retiro (*retire*). La liberación es la actividad puente entre el desarrollo y el despliegue, y se encarga de todas las tareas necesarias para la preparación, empacado y provisión de un sistema para despliegue en el sitio de un consumidor; el paquete liberado contiene tanto los recursos software como son las librerías, archivos de configuración y ejecutables, como los descriptores que especifican los recursos requeridos para el despliegue del software. El retiro es el proceso de eliminar el soporte para un sistema software o para una configuración dada de un sistema de software, lo que hace que no esté disponible para futuros despliegues.

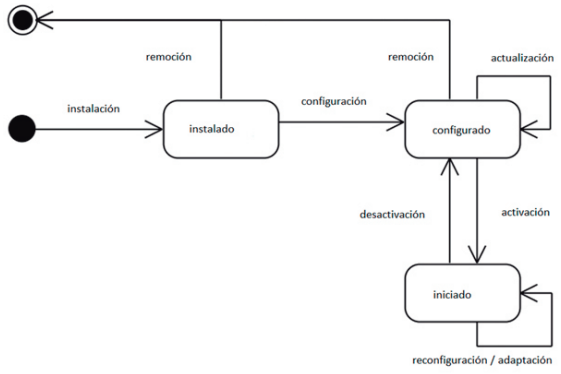

Figura 1. Ciclo de vida del despliegue de software [36]

En el lado consumidor se identifican cuatro actividades: instalación (*install*); activación (*activate*) y desactivación (*deactivate*); reconfiguración (*reconfigure*), actualización (*update*) y adaptación (*adapt*); y remoción (*remove*). La instalación es la primera actividad del consumidor y se encarga de la configuración y el ensamble de todos los recursos necesarios para usar el sistema software dado. Activación y desactivación son las actividades que permiten el uso del software; para tareas sencillas, estas actividades se realizan a través de la creación de ciertos comandos o iconos para ejecutar y detener un componente binario de la herramienta, mientras que los software complejos pueden estar formados por varios componentes que deben ser ejecutados en orden para su uso. Reconfiguración, actualización y adaptación son las actividades responsables de cambiar y mantener la configuración del sistema desplegado y pueden ocurrir más de una vez en cualquier orden; el objetivo de la actualización es desplegar una nueva configuración de software que no estuvo disponible previamente, mientras que la reconfiguración realiza cambios a un software previamente instalado, pero seleccionando una configuración diferente, y la adaptación se encarga de monitorear el sistema de software desplegado y responder a los cambios con el fin de mantener consistencia en el software. Finalmente, Remoción es la actividad que se realiza cuando el sistema de software desplegado ya no es requerido por el consumidor, e implica la remoción de todos los cambios causados por las todas las actividades de despliegue.

Construcción de software de automatización

Antes de que un producto software pueda ser utilizado por los usuarios finales, los desarrolladores deben compilar el código fuente para una plataforma destino particular o construir una versión del software (*software release*) para ella. El proceso de compilar código fuente toma como entradas los archivos de código fuente o un directorio raíz donde se encuentran todos los archivos que componen el programa. Después, se genera uno o varios binarios que son ejecutados o interpretados por otro programa. Cuando el código fuente hace uso de librerías externas, el compilador debe reconocerlas con el fin de encontrar posibles errores en el programa y optimizar la generación de los binarios.

A medida que el software crece, el proceso de compilación se vuelve más laborioso, requiere más tiempo y es más propenso a errores. Además, como los desarrolladores pueden trabajar en diferentes subsistemas, la compilación

manual de la totalidad del software, eventualmente, empeora el escenario. Para superar esta situación, la construcción de software generalmente se automatiza utilizando scripts o herramientas avanzadas como Make, Maven o Ant. Sin embargo, no es lo mismo automatizar la construcción que automatizar el despliegue, la integración continua y la entrega continua. Estos procesos se centran en desplegar o instalar una versión en un entorno particular, construir un producto de software a medida que los desarrolladores registran cambios en el código fuente y una combinación de ambos, respectivamente.

METODOLOGÍA

La metodología adoptada en esta investigación para el desarrollo y la validación de la solución planteada tiene un enfoque cualitativo [37]. En la fase de desarrollo se emplearon métodos cualitativos para explorar el estado del arte, se inició con una revisión de la literatura principalmente enfocada en la motivación y necesidad de sistemas software autogestionados, como un medio para avanzar en el diseño de sistemas autónomos [9].

Dado que el interés estaba centrado en automatizar el monitoreo (esto es, en desarrollar los medios para generar autoconciencia), se requirieron modelos de referencia y arquitecturas para la implementación efectiva de infraestructuras de monitoreo dinámico a través de mecanismos de autoadaptación. Adicionalmente, también fue necesario sondear mecanismos para especificar y desplegar componentes de software de monitoreo, con una aproximación desde los DSL

Por otra parte, en la fase de evaluación, se utilizaron métodos cualitativos para evaluar: la integridad de nuestra solución con respecto de los requisitos y los escenarios de calidad relacionados en este texto; y la expresividad y usabilidad de los dos DSL que componen la solución planteada en el documento, para lo que se realizó un taller para cada lenguaje, utilizando un estudio de caso no trivial y relevante. Con el fin de alcanzar los objetivos propuestos se establecieron los siguientes hitos:

- Extracción de requerimientos y diseño arquitectónico, lo que incluye: la identificación y especificación de los requerimientos de monitoreo para alcanzar autoconciencia en el desempeño de calidad; la especificación de los requerimientos de diseño para automatizar la distribución de componentes, la ejecución y el *binding* (unión) del servicio; el diseño de la

arquitectura de software para la infraestructura de monitoreo planeada; y el desarrollo de mecanismos estándar para proveer componentes software con capacidades de trazabilidad y registro.

- Diseño e implementación de dos DSL para especificar, generar, compilar y ejecutar pruebas de monitoreo de rendimiento y especificar y realizar actualizaciones a los despliegues en tiempo de ejecución, lo que comprende: el diseño de cada DSL con su semántica y sintaxis; el diseño de la gramática libre de contexto en correspondencia con la sintaxis propuesta; y el diseño e implementación de los modelos de traducción y ejecución de acuerdo con la semántica propuesta.
- Desarrollo de una implementación tipo "prueba de concepto", lo que incluye los artefactos de caso de estudio, los correspondientes monitores de QA y los descriptores de despliegue, utilizando los DSL desarrollados.
- Análisis y evaluación de resultados.

Como resultado de la aplicación de esta metodología, el proyecto logró crear: una arquitectura escalable para el monitoreo dinámico que implementa y resuelve problemas de monitoreo dinámico en el contexto de sistemas software autoadaptativos, con una arquitectura que fomenta la generación de componentes de monitoreo controlables, trazables y compuestos; un lenguaje de dominio específico (Pascani) para asegurar que los problemas de monitoreo puedan especificarse utilizando un formato estándar y adecuado con los niveles apropiados de abstracción; y un lenguaje de dominio específico (Amelia) para abstraer y facilitar la comprensión semántica del desarrollo del sistema, cuyos constructos ofrecen un alto poder de expresión para construir sistemas y ejecutar artefactos en infraestructuras de computación distribuida. El proceso de desarrollo y validación de estos productos, junto con las lecciones aprendidas durante él, se presenta en las secciones siguientes.

ANÁLISIS DEL DOMINIO DEL PROBLEMA: MONITOREO DINÁMICO DEL RENDIMIENTO

MAPE–K: MONITOREANDO REQUERIMIENTOS

Como se dijo, esta investigación busca proveer mecanismos estándar para monitorear el desempeño de aplicaciones de software en contextos dinámicos.

Alineado con la visión presentada por el modelo de referencia Dynamico [10], la propuesta de este documento debe ser capaz de soportar el cambio de valores y umbrales de las variables de contexto y la lógica de monitoreo para capturar dicha información. Puesto que los lazos de retroalimentación presentes en dicho modelo de referencia se basan en el MAPE–K [4], una primera aproximación hacia el diseño e implementación de la infraestructura de monitoreo de Dynamico considera los elementos sensor y monitor del modelo MAPE–K. Empero, se identificó una carencia de especificaciones de referencia detalladas y estandarizadas y de un diseño arquitectónico para ese modelo. Consecuentemente, Arboleda et al. [38],[39] presentan un diseño base para la construcción de sistemas autónomos utilizando MAPE–K, incluyendo los diseños arquitectónicos de estructura y comportamiento.

El mapa arquitectónico de IBM para la computación autónoma describe las funciones de alto nivel formando la estructura interna de un gestor autónomo [4]. De dicha propuesta, se puede identificar una serie de requerimientos funcionales para los elementos que componen el modelo de referencia MAPE–K (aunque los requerimientos en esta sección se enfocan únicamente en monitoreo, los demás se pueden consultar en [39]).

Los requerimientos se dividen en sensores y monitores: los sensores son elementos que constituyen un conjunto de propiedades que exponen el estado actual de cierto recurso gestionable y un grupo de eventos que ocurren cuando el estado de dicho recurso gestionable cambia: los monitores, en cambio, son elementos que recolectan detalles de los recursos gestionados utilizando una interfaz de gestión para las sondas y correlacionando la información con los síntomas que pueden ser analizados [4]. En adelante, en este documento se utilizará la denominación "sondas", para referirse a los sensores, puesto que dicho nombre es más adecuado para el enfoque que se presenta. Los requerimientos de las sondas de software y los monitores se presentan en la Tabla 1.

REQUERIMIENTOS DE LA INFRAESTRUCTURA DINÁMICA DE MONITOREO

Los requerimientos funcionales que se describen en la Tabla 1 son insuficientes para implementar la infraestructura de monitoreo para soportar las habilidades dinámicas especificadas para el lazo de retroalimentación de monitoreo en el modelo de referencia Dynamico, por lo que deben extenderse. A continuación se presenta el alcance funcional de los requerimientos de la infraestructura dinámica de monitoreo y sus consideraciones de calidad.

Tabla 1. Requerimientos funcionales

Elemento	Ítem	Requerimiento
Sondas	S1	Una sonda debe ser capaz de recolectar mediciones de las variables de interés (datos capturados), esto es de los atributos de calidad específicos de la serie de estándares ISO 25000 [40] (*e.g.*, desempeño del servicio, disponibilidad de recursos, información de topología, propiedades de configuración), en el contexto donde se ubica, esto es, en su contexto de ejecución o en el contexto del dominio al cual pertenece.
	S2	Una sonda debe guardar temporalmente la información recolectada. La capacidad de respuesta de los monitores se basa en la disponibilidad oportuna de la información recolectada, capacidad que se puede alcanzar al soportar almacenamiento temporal, lo cual permitiría a los monitores capturar información en cualquier momento. Sin embargo, dado que en este enfoque las sondas pueden utilizar espacio de memoria que debería estar disponible en el sistema objetivo, se deben considerar otras opciones de almacenamiento.
	S3	La sonda debe presentar un subconjunto de los datos obtenidos a los monitores, tanto cuando los monitores y sondas se hayan desplegado conjuntamente, como cuando hayan sido desplegados independientemente.
	S4	La sonda debe remover un subconjunto de la información capturada y almacenada temporalmente cuando así lo solicite un monitor.
	S5	Una sonda debe realizar operaciones primitivas (contar las repeticiones una medida en un intervalo dado) sobre un subconjunto de datos sensados. La transmisión de la información capturada desde las sondas hacia los monitores puede sobreutilizar los recursos de la red y dificultar así la operación regular del sistema objetivo. Trabajar con operaciones primitivas en sondas puede reducir considerablemente la cantidad de datos transmitidos por la red cuando los monitores no requieren la totalidad de la información capturada, sino los cálculos realizados sobre ella.
Monitores	M1	Un monitor debe obtener los datos sensados desde una o más sondas, de donde han sido capturados, a través de los modos de acceso requeridos, esto es, por solicitud (*pull*) o por ocurrencia (*push*).
	M2	Un monitor debe computar métricas (basado en la información capturada) relacionadas con las variables de interés para caracterizar el estado actual del sistema objetivo, dichos cálculos: pueden ser disparados periódicamente o por ocurrencia de medida, pueden producir cálculos instantáneos o promedio, y pueden involucrar la composición o correlación de métricas calculadas por otros monitores.
	M3	Un monitor debe facilitar los cálculos de métricas a través de un elemento denominado gestor del conocimiento (*knowledge manager*) a otros monitores, para que puedan realizar sus propios cálculos.
	M4	Un monitor debe filtrar las métricas calculadas antes de ser reportadas al elemento analizador, el filtro debe aplicarse a través de un conjunto de reglas de monitoreo dependientes del dominio sobre las métricas calculadas.
	M5	Un monitor debe reportar al elemento analizador síntomas de control como métricas simples o compuestas, que cumplan con las condiciones impuestas por las reglas de monitoreo.
	M6	Un monitor debe permitir cambiar la periodicidad en cómo calcula sus métricas.

Alcance funcional

La extensión de los requerimientos funcionales del elemento monitor se hace con el fin de orquestar una infraestructura dinámica de monitoreo compuesta por cuatro etapas (ver Tabla 2): adquisición de datos, agregación y filtrado de datos, persistencia de datos y visualización de datos. En las dos primeras, el enfoque consiste en proveer sintaxis y semántica "a la medida" para así separar el código de la aplicación de la lógica de monitoreo y facilitar la especificación de sondas y monitores; Para las otras dos variables, como su contexto puede diferir en su naturaleza, es necesario considerar diferentes almacenes de datos y tecnologías de visualización. Por esta razón, estas etapas son consideradas elementos conectables de infraestructura.

Tabla 2. Requerimientos funcionales del elemento monitor extendidos

Elemento	Ítem	Requerimiento
Adquisición de datos	M7	La definición de un monitor debe especificar las sondas que requiere para adquirir las medidas respecto de las variables de interés.
	M8	Las sondas deben ser desplegadas independientemente desde los componentes del sistema objetivo, lo que implica que el monitoreo de los componentes desplegados puede iniciar en cualquier momento.
	M9	Las sondas deben ser capaces de interceptar diversos tipos de eventos asociados con la ejecución del servicio, incluyendo: invocación, retorno, tiempo de ejecución, tiempo de comunicación y/o excepción.
	M10	Para eventos personalizados iniciados a un nivel más bajo que la ejecución del servicio debe existir una librería que permita programar sondas personalizadas, a las cuales se debe poder acceder de la misma manera que a las provistas por defecto.
	M11	Los desarrolladores deben especificar cómo crear, actualizar y obtener valores de medición para métricas personalizadas; en el caso de operaciones de obtención, se debe definir si el modo de acceso es por petición o por ocurrencia.
Agregado y filtrado de datos	M12	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para localizar, utilizar (*i.e.*, get y set) y compartir variables de contexto a través de un mecanismo estándar y dicho mecanismo debe soportar la adición o remoción de variables en tiempo de ejecución.
	M13	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para manipular colecciones de eventos y realizar cálculos sobre ellos.
	M14	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para definir valores de referencia (indicadores de nivel de servicio) y compararlos con los valores medidos, con el fin de notificarle a los servicios externos los comportamientos no esperados.

Tabla 2. Requerimientos funcionales del elemento monitor extendidos (cont.)

Elemento	Ítem	Requerimiento
Agregado y filtrado de datos (cont.)	M15	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para adjuntar nuevas mediciones con información contextual. Etiquetar datos medidos permite categorizar valores de las variables, suministrando así información valiosa para buscar y filtrar mediciones para propósitos de visualización.
	M16	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para utilizar librerías de clase existentes para realizar cálculos o invocar API (*Application Programming Interface*] existentes con el fin de agrupar o filtrar medidas.
	M17	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para especificar la lógica de manejo para eventos de ejecución del servicio, eventos periódicos (basados en el tiempo) y cambios en variables de contexto.
	M18	La infraestructura de monitoreo debe proporcionar mecanismos estándar con protocolos establecidos para actualizar el conjunto de reglas de monitoreo que aplica al filtro de las métricas.
Persistencia de los datos	M19	La infraestructura de monitoreo debe soportar variables de contexto persistente en disco junto con su información contextual. Los mecanismos de persistencia deben ser aplicados independientemente de los monitores que actualizan las variables de contexto.
	M20	Diferentes tecnologías de persistencia se pueden aplicar a diferentes variables. El amplio uso de tecnologías NoSQL ha promovido la aparición de bases de datos simplificadas que buscan mejorar el desempeño y la escalabilidad en el almacenamiento de datos y consultas; los servicios y librerías de terceros para gestionar información de métricas y registro son una opción relevante.
Visualización de los datos	M21	Los desarrolladores deben ser capaces de crear visualizaciones (gráficas) resumiendo mediciones históricas asociadas con las variables de contexto.
	M22	La visualización de los datos debe permitir la representación gráfica de la información asociada con los eventos capturados por sondas.
	M23	Los desarrolladores deben ser capaces de preprocesar (*i.e.*, filtrar, transformar, correlacionar) medidas históricas con el fin de proveer visualizaciones de información relevante.

Consideraciones de calidad

Estas consideraciones incluyen aspectos como compatibilidad, coexistencia e interoperabilidad, que implican: el despliegue dinámico y el redespliegue de monitores y sondas; la escalabilidad de la infraestructura; y los mecanismos estándar de controlabilidad y componibilidad (capacidad de los artefactos para acoplarse y permitir su reuso para generar una adecuada relación costo/beneficio) [41].

Los escenarios de calidad asociados a estas consideraciones, presentes en la Tabla 3, se basan en las definiciones de atributos de calidad descritas por Barbacci et al. [32].

Tabla 3. Consideraciones de calidad [32]

Propósito	Ítem	Descripción
Coexistencia e interoperabilidad de monitores y sondas.	Escenario de calidad	Interoperabilidad en el despliegue dinámico y redespliegue de monitores y sondas.
	Atributos de calidad	Compatibilidad, coexistencia e Interoperabilidad.
	Justificación	Las tareas de despliegue y redespliegue pueden llevar a la infraestructura de monitoreo hacia un estado erróneo dado que los componentes (monitores y sondas) pueden ser reinsertados en el mismo middleware en ejecución o las variables de contexto pueden definirse más de una vez. Además, las actualizaciones en los componentes del sistema objetivo pueden detener la ejecución de las sondas.
	Estímulo	Un monitor ya desplegado es redesplegado.
	Fuente del estímulo	Un cambio en la lógica de monitoreo o en las variables de contexto.
	Artefacto	Los componentes de la infraestructura de monitoreo responsables de generar y desplegar los nuevos componentes.
	Respuesta	La infraestructura de monitoreo ejecuta las acciones necesarias para soportar la (posible) generación de nuevas sondas y monitores, y se asegura de que sean desplegadas correctamente. Si fueron ya desplegadas, pero requieren modificaciones en su lógica, deshace el despliegue y redespliega asegurando que las variables sean definidas una única vez.
Escalabilidad de la infraestructura de monitoreo	Escenario de calidad	Escalabilidad de la infraestructura de monitoreo.
	Atributo de calidad	Escalabilidad.
	Justificación	El monitoreo continuo puede generar grandes cantidades de eventos e información de inicio de sesión, la cual requiere mecanismos para utilizar recursos computacionales adecuadamente (si se cuenta con ellos). Los componentes monitoreados pueden llegar a un estado de no respuesta.
	Estímulo	Los monitores y sondas alcanzan niveles críticos de capacidad de recursos de computación (*i.e.*, disco, memoria, red).
	Fuente del estímulo	Los servicios del sistema objetivo están siendo altamente requeridos.

Tabla 3. Consideraciones de calidad [32] (cont.)

Propósito	Ítem	Descripción
Escalabilidad de la infraestructura de monitoreo (cont.)	Artefacto	Los componentes de la infraestructura de monitoreo responsables del reporte y despliegue de componentes nuevos o existentes en nuevos recursos computacionales
	Respuesta	Algunos componentes de la infraestructura de monitoreo (monitores) se despliegan en nuevos recursos computacionales.
Composabilidad de los monitores	Escenario de calidad	Composabilidad de los monitores.
	Atributo de calidad	Modificabilidad – modularidad
	Justificación	Los monitores deben ser capaces de reutilizar efectivamente elementos de monitoreo existentes, como sondas, con el fin de incrementar la productividad del desarrollo.
	Estímulo	Un monitor necesita recibir mediciones de sondas ya existentes y desplegadas.

DOMINIO DE LA SOLUCIÓN: UN ENFOQUE DE DSL PARA EL MONITOREO Y DESPLIEGUE DINÁMICOS

DISEÑO ARQUITECTÓNICO GLOBAL

En la presentación de los requerimientos funcionales para llevar a cabo la infraestructura de monitoreo dinámico propuesta y las restricciones de calidad relacionadas se identificaron dos grupos importantes: el primero, relacionado con las tareas de monitoreo de software; el segundo, con el despliegue de los artefactos de monitoreo. Estos requerimientos y restricciones se abordaron al diseñar dos DSL: Pascani y Amelia.

La Figura 2 ilustra el modelo basado en componentes del diseño arquitectónico global. Desde un conjunto de especificaciones de monitoreo de Pascani, el motor del lenguaje genera componentes de monitoreo compatibles con SCA implementados en Java y su correspondiente especificación de despliegue Amelia.

Desde su especificación de despliegue, el motor de lenguaje Amelia genera un programa ejecutable capaz de comunicarse con nodos de computación basados en UNIX con el fin de ejecutar las operaciones necesarias para desplegar los artefactos SCA específicos. Una vez que los componentes de monitoreo

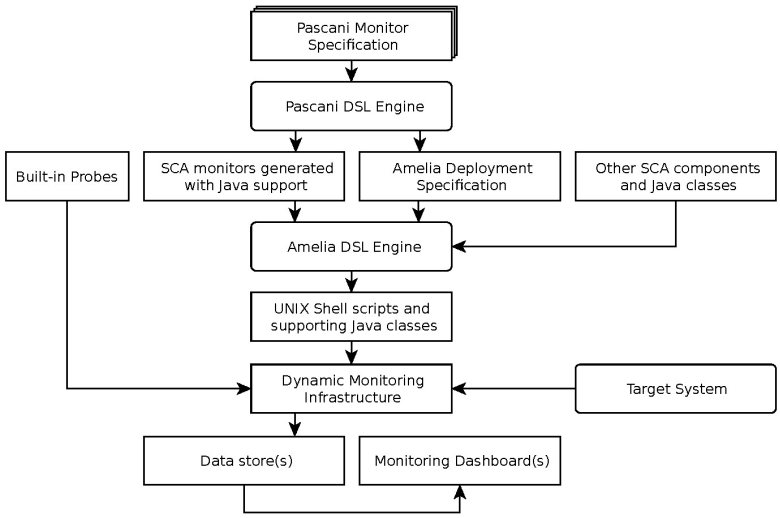

Figura 2. Diseño arquitectural global (informal) de la propuesta

se están ejecutando en la infraestructura, junto con un juego de sondas monitoras, la infraestructura recopila continuamente datos de rendimiento del sistema objetivo, lo que se hace al enviar mediciones desde las sondas hacia los monitores utilizando comunicación tipo *pull* o *push*. Los datos se utilizan para actualizar las variables de contexto –que son almacenadas en una o varias bases de datos–. Desde aquí, el personal de TI puede visualizar el rendimiento actual del sistema, modelado como variables de contexto, utilizando diferentes tecnologías para el panel de control.

Para entender mejor la solución propuesta, esta se explica utilizando un ejemplo que provee más detalle sobre cómo la arquitectura propuesta soporta el desarrollo de estrategias de monitoreo para un sistema de ventas en línea.

Monitoreo dinámico: la tienda de comercio en línea

En el comercio electrónico, una aplicación de tienda de este tipo (OSR, *Online Store Retailer*) permite a los clientes comprar productos y servicios a través de Internet. Los usuarios navegan por un catálogo de productos, seleccionan lo que desean y lo "añaden a un carrito" para realizar la compra; además, con la proliferación de contextos personales y recomendaciones centradas en el usuario, los usuarios esperan de las aplicaciones OSR recomendaciones de productos basadas en sus preferencias. Para completar la compra, los clientes inician el proceso de *checkout* ingresando la información de sus preferencias de pago y envío

(*e.g.*, dirección de envío, servicio postal a utilizar, número de tarjeta de crédito, código de seguridad, nombre). Las aplicaciones OSR, por lo general, utilizan varios servicios de terceros para recomendar productos relevantes, completar el proceso de *checkout* y confirmar la entrega del pedido. De la misma manera, para completar el proceso de *checkout*, las aplicaciones OSR utilizan un servicio para verificar la dirección postal para confirmar que las direcciones de envío y cobro existan. Un servicio adicional verifica toda la información acerca de las tarjetas de crédito y un servicio de entrega —con su correspondiente servicio de rastreo de paquetes— se utiliza para monitorear los envíos con el fin de despachar la orden y que el usuario pueda monitorear el proceso de entrega. Distintos vendedores de software proveen instancias de dichos servicios para ser utilizados manualmente o programados automáticamente a través de API públicas.

Asumiendo que un negocio local ha recibido quejas de sus consumidores durante las dos últimas semanas. De acuerdo con su sistema de tiquetes de soporte, existen retrasos largos y esporádicos cuando se realiza una compra en su aplicación OSR. Después de conversaciones con los consumidores afectados y analizar la infraestructura, el personal de TI asegura que este problema está relacionado con uno de los servicios de terceros involucrados en el proceso de *checkout*: el servicio de verificación de tarjeta de crédito. Sin embargo, la actual solución de monitoreo no provee información relevante acerca del problema, puesto que ella se limita a proveer información sobre la infraestructura de computación. Dado que los retrasos no pueden ser predichos o reproducidos, las pruebas iniciales que se ejecutaron en los servicios de terceros no demostraron que el proveedor de servicio estaba incumpliendo con la calidad acordada. Para descubrir qué está causando los problemas de rendimiento, se utilizó Pascani como sigue:

1. Los desarrolladores de la aplicación identifican las variables de contexto que permiten modelar el problema de desempeño en el proceso de *checkout*. En este caso, medir la latencia del servicio es suficiente para detectar cuál de estos está causando largos retrasos en el proceso de *checkout*.
2. Considerando los componentes del sistema objetivo –los elementos de alto nivel que componen la aplicación OSR–, los desarrolladores de la aplicación crean un monitor de desempeño Pascani con lógica para introducir una sonda en cada servicio objetivo en tiempo de ejecución y para recopilar la información necesaria para actualizar la variable de contexto de latencia.

3. Una vez que las especificaciones del monitor están completas, los desarrolladores ejecutan el motor Pascani para generar el monitor SCA y su correspondiente especificación de despliegue Amelia. Entonces, ejecutan el motor Amelia para generar un programa en Java para desplegar el monitor generado.
4. Con el fin de iniciar el monitoreo en la aplicación OSR, los desarrolladores ejecutan el programa Java generado por el motor Amelia, el cual compila el código fuente generado, transporta los artefactos resultantes hacia la infraestructura computacional y ejecuta los componentes de monitoreo.
5. Puesto que los monitores SCA generados se introducen en la infraestructura sin detener el sistema objetivo, la lógica de monitoreo es puesta en su lugar inmediatamente. Cada que un servicio se ejecuta (*i.e.*, cada que un cliente termina una compra), las variables de contexto se actualizan y sus nuevos valores son almacenados en los almacenes de datos.
6. Mientras que la infraestructura de monitoreo obtiene datos del sistema objetivo y llena los valores de las variables de contexto en los almacenes de datos, los desarrolladores crean visualizaciones útiles, como diagramas de barras o diagramas de Gantt, para descubrir qué servicio está causando el problema de desempeño.

Al visualizar la latencia asociada a cada servicio objetivo, los desarrolladores pueden descubrir la fuente de los retrasos inesperados. Después de descubrir los servicios que no se comportan de acuerdo con los acuerdos de nivel de servicio, el personal del negocio puede tomar decisiones acerca del proveedor del servicio, como por ejemplo, invocar alguna de las clausulas de los acuerdos de calidad del servicio. De manera similar, el proveedor del servicio puede analizar estos problemas y descubrir si se trata de un problema en su infraestructura o si viene directamente de la API de la compañía de tarjetas de crédito.

ABORDANDO RESTRICCIONES DE CALIDAD

Compatibilidad, coexistencia e interoperabilidad de monitores y sondas

La sección 2 de la Tabla 3 "Escalabilidad de la infraestructura de monitoreo" detalla los efectos de introducir componentes nuevos y existentes en la infraestructura. Este escenario puede dividirse en dos casos: la introducción de nuevos monitores y sondas, y el despliegue y redespliegue de monitores y

sondas. Respecto del primer caso, el introducir nuevos elementos en el sistema objetivo generalmente conlleva un chequeo de compatibilidad al momento de diseñar el sistema software, sin embargo, la infraestructura de monitoreo propuesta en este documento requiere la introducción de nuevos elementos en tiempo de ejecución, esto es, se necesita aumentar el procesamiento en la ejecución del servicio con código de monitoreo cuando ocurren las peticiones al servicio. Este es un escenario adecuado para utilizar el patrón de diseño interceptor, mientras que el middleware del sistema objetivo lo soporte, esto es, añadir elementos interceptores en tiempo de ejecución. Ejemplos de esto son los componentes *Intent* (intención) en el estándar SCA y el uso de programación orientada a aspectos en EJB (*Enterprise JavaBeans*) a través de anotaciones Java o archivos de configuración XML y OSGi a través de extensiones *Aspect*J. En la Figura 3 se muestra el uso de este patrón de diseño para producir datos de medida y para proveer acceso a los monitores a través del componente sonda.

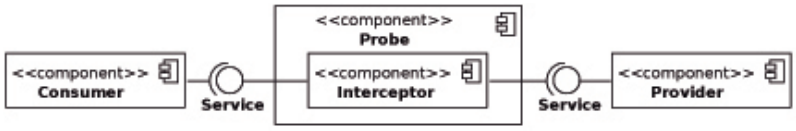

Figura 3. Patrón de diseño interceptor aplicado a los componentes sondas

Por su parte, el segundo caso requiere, no solo de la adición de nuevos componentes interceptores en tiempo de ejecución, sino también de su remoción. Como la comunicación entre monitores y sondas es bidireccional, esto puede ser fuente de errores debido a enlaces rotos en tiempo de ejecución. Esto conlleva a una nueva restricción de diseño respecto de los enlaces de dependencias estáticas, los cuales requieren de un proveedor y un consumidor del servicio para ser atados al despliegue del sistema y no pueden ser reemplazadas en tiempo de ejecución. En este caso, los monitores y sondas deben ser desacoplados con el fin de que: los datos de medición sean expuestos sin conocer los componentes consumiéndolos, y las mediciones se publican sin conocer los componentes que las utilizan. Estos casos de uso concuerdan con el contexto de aplicación del patrón de diseño publicar/subscribir (*publish/subscribe*).

Escalabilidad de la infraestructura de monitoreo

Alcanzar escalabilidad en una Infraestructura de monitoreo dinámico requiere de la habilidad de desplegar monitores en nuevos recursos computacionales, independientemente de los componentes del sistema objetivo. Asimismo, requiere manejar los siempre crecientes volúmenes de datos medidos y soportar el almacenamiento en memoria cuando el sistema objetivo esté bajo alta carga. Esto puede hacerse al introducir el estilo arquitectónico *Message Broker*, el cual desacopla los mensajes de transmisores y receptores (sondas y monitores) –como se discutió–, y permite distribuir los componentes de la infraestructura de monitoreo a través de múltiples recursos computacionales. Además, la acción de mover automáticamente componentes a través de diferentes recursos de computación requiere del diseño y la implementación de una infraestructura autónoma con relevancia especial para la función de planeación. El enfoque de este proyecto se limita a la automatización de los mecanismos de despliegue necesarios para realizar los planes de reconfiguración de dicha infraestructura.

Componibilidad de monitores

Los componentes componibles hacen referencia a los elementos autocontenidos (modulares) que pueden ser reutilizados efectivamente con el ánimo de incrementar la calidad de software, reducir los costos de mantenimiento y mejorar la productividad del equipo de desarrollo. Tal resultado no es fácil de alcanzar, la lógica de monitoreo depende de muchos factores, incluidas las variables de interés, los componentes desde los cuales dichas variables son calculadas y la arquitectura del sistema objetivo, entre otros. Estas dependencias hacen difícil crear componentes configurables que puedan ser seleccionados y ensamblados para construir nuevos componentes de monitoreo. Sin embargo, aunque esto hace difícil el reuso de monitores como un todo, se pueden proveer atributos deseables para promover la componibilidad desde diferentes frentes [42], así:

- arquitectura solida, una buena interfaz de diseño –especialmente para interfaces visuales– y una buena estructura de la arquitectura pueden facilitar considerablemente la composición;
- abstracción, proveer un nivel apropiado de abstracción a la especificación de la lógica de monitoreo puede simplificar la abstracción de capas técnicas relacionadas, esto puede ser benéfico para la composición de componentes de monitoreo cuando existe una adecuada independencia de los conceptos de monitoreo provistos; y

- modularidad, una infraestructura de monitoreo basada en abstracciones especificas encapsuladas en entidades bien definidas promueve componentes de monitoreo más modulares, la modularidad es benéfica para la componibilidad si las interfaces y especificaciones están bien definidas.

Controlabilidad de monitores

Controlar la ejecución de la infraestructura de monitoreo permite reaccionar adecuadamente a cambios en el contexto del sistema objetivo. Niveles críticos en el uso de memoria del sistema objetivo requieren de la reducción del uso de memoria en monitores y sondas desplegadas en los mismos recursos computacionales. Otros escenarios críticos requieren modificar la periodicidad del cálculo de métricas de medición e incluso la detención temporal de todas las actividades de monitoreo. Dichos requerimientos de controlabilidad se abordan al proveer todos los elementos de la infraestructura de monitoreo dinámico con interfaces de gestión o métodos para remover porciones o toda la información capturada (sondas) y modificar la periodicidad del cálculo de métricas (monitores). Estos y otros elementos de la infraestructura se proveen con una interfaz para pausar y resumir toda actividad de monitoreo, incluyendo la propagación de datos de medición y cambios en las variables de contexto.

Trazabilidad e información de registro de los monitores

Reportar detallados niveles de información acerca de cómo la infraestructura se desempeña es esencial para descubrir fuentes de error y comportamientos no deseados. Además, las trazas de la ejecución y los datos detallados de las transacciones son útiles para encontrar la causa de las anomalías. En la infraestructura de monitoreo se ha diseñado un sistema de registro basado en eventos que propaga eventos de *log* y despliegue a cada ubicación de componente. Dichos eventos son también almacenados y permiten reproducir una serie de eventos en un intervalo dado, lo que a su vez permite la visualización de *logs* como una línea de tiempo de eventos, con la posibilidad de filtrar por nivel de registro.

DISEÑO DE LA INFRAESTRUCTURA DE MONITOREO DINÁMICO (VISTA GENERAL)

Los elementos más relevantes que componen la arquitectura propuesta son: monitores, sondas (*probes*) y espacios de nombre (*namespaces*). Los primeros dos

elementos siguen las consideraciones que se han discutido en las secciones previas, los espacios de nombre son almacenamientos en memoria para valores asociados con nombres, elementos que permiten registrar variables de contexto en tiempo de ejecución y su correspondiente actualización de valores.

En la Figura 4 se describe el comportamiento básico para cada uno de estos elementos y las interfaces de gestión para controlar su ejecución. Estos tres elementos son concebidos como elementos SCA, los cuales estandarizan y simplifican la construcción, el desarrollo y la gestión de la infraestructura propuesta.

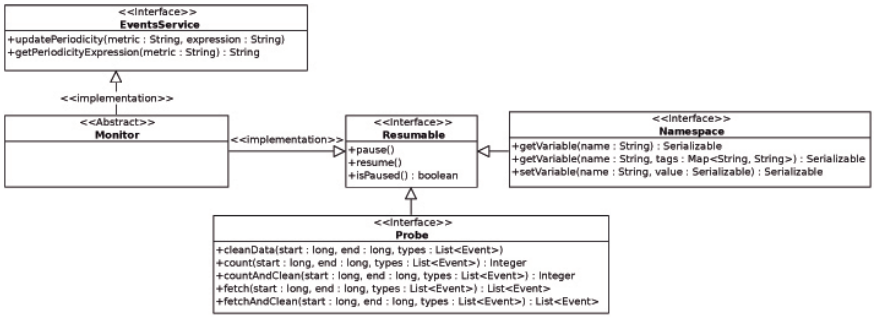

Figura 4. Elementos principales que componen la arquitectura de monitoreo

La interfaz probe en la Figura 4 expone un conjunto de operaciones básicas para buscar, contar y limpiar las mediciones realizadas en un intervalo dado, lo que permite a los monitores obtener las medidas (eventos) de las sondas y realizar cálculos para actualizar las variables de contexto. Como se analizó en la subsección Compatibilidad, coexistencia e interoperabilidad de monitores y sondas, los monitores y sondas necesitan estar desacoplados utilizando el patrón de diseño *Publish/Subscribe* (ver Figura 5). Con el fin de mediar entre publicaciones y suscripciones, se utilizó un message broker distribuido. En este esquema de comunicación, los monitores se suscriben a medidas de interés y posiblemente a cambios en variables de contexto. Esta información se publica por sondas y espacios de nombres, respectivamente. Puesto que se decidió remover los enlaces estáticos, la comunicación tipo pull se realiza a través de llamado a procedimientos remotos (RPC, *Remote Procedure Calls*), utilizando el mismo patrón de diseño.

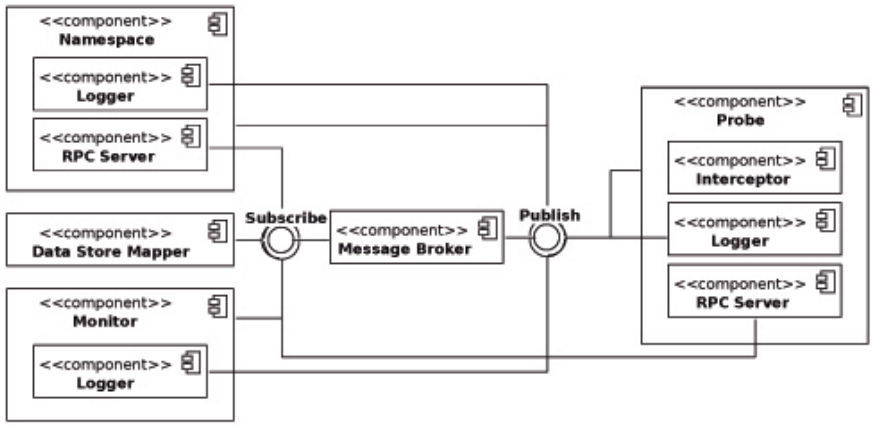

Figura 5. Patrón de diseño Publish/Subscribe aplicado a elementos de la infraestructura de monitoreo

Siguiendo el mismo esquema de comunicación, se añadió la persistencia y la habilidad de guardar *logs* en la infraestructura de monitoreo. Cada elemento en la infraestructura produce *logs* locales que se publican a través del message broker. La persistencia es realizada por el componente DataStoreMapper, el cual se suscribe a todos los cambios en las variables de contexto, nueva información de *log* y nuevos despliegues de monitores y espacios de nombre.

PASCANI: UN DSL PARA EL MONITOREO DE DESEMPEÑO DINÁMICO

Pascani es un DSL basado en componentes, de tipo estático, útil para especificar monitores de desempeño dinámico para sistemas software basados en componentes. Está altamente integrado con Java, lo que permite la integración de librerías existentes y proporciona algunas de las instrucciones de control de Java con una sintaxis más flexible basándose en el lenguaje de expresión Xbase [43]. De sus especificaciones, puede decirse que la implementación del lenguaje genera los artefactos para la infraestructura dinámica de monitoreo, incluyendo las especificaciones de despliegue. Los artefactos generados se componen de elementos de la librería de tiempo de ejecución Pascani y de la librería SCA (que se presentan en la sección siguiente). Para completar la automatización del monitoreo dinámico de desempeño, las especificaciones de despliegue son ejecutadas a través de Amelia, el segundo DSL, del cual se habla en la siguiente subsección.

Pascani comprende dos conceptos principales: monitores y espacios de nombre. Dichos conceptos representan una abstracción textual de las interfaces monitor y espacio de nombre introducidas en la Figura 4. Las semánticas asociadas con cada uno de estos conceptos se basan en el comportamiento definido por dichas interfaces. Los espacios de nombre son almacenamientos para valores asociados con nombres, identificados con un nombre de almacenamiento. Cada nombre dentro de un espacio de nombre corresponde a una variable de contexto. Los monitores son contenedores y gestores de eventos que especifican la lógica de monitoreo requerida para calcular métricas. Estos dos conceptos se han diseñado para ser utilizados conjuntamente a través de métricas, las cuales se utilizan para actualizar las variables de contexto definidas en los espacios de nombres.

Los diagramas de sintaxis que se presentan a continuación contienen una versión simplificada de la definición de la gramática de Pascani. Este tipo de diagramas, también denominados diagramas de ferrocarril, son una manera visual de representar diagramas libres de contexto, donde cada diagrama define un "no terminal". Un diagrama describe posibles caminos entre un punto de entrada y un punto de salida yendo a través de terminales (representados como elementos redondeados) y no terminales (representados por elementos cuadrados). La gramática completa se presenta en el Anexo 1. En todos los casos, las reglas de gramática cuyo nombre inicia con una X son heredadas de Xbase, a menos que sean explícitamente redefinidas en la gramática.

Un archivo válido en Pascani presenta una definición opcional de un nombre de paquete, una sección para importar desde Java y una sección de declaración de tipo (de unidad de compilación). Esto significa que los archivos vacíos son especificaciones validas que no generan ningún artefacto. Una unidad de compilación es una unidad traductora o archivo fuente que contiene la definición de un espacio de nombre o un monitor. Estas son las dos unidades de compilación de Pascani que corresponden a los elementos namespace y monitor.

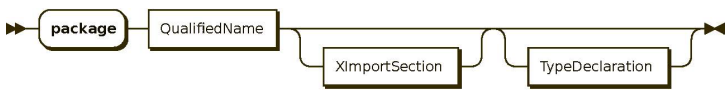

Figura 6. MonitorSpecificationModel

Cada unidad de compilación tiene una sintaxis y semántica particulares y ha sido diseñada para ser usada junto con otras. Utilizar espacios de nombre

dentro de los monitores hace transparente la acción de obtener y actualizar los valores de variables de contexto, lo cual facilita el trabajo de los desarrolladores al permitirles enfocarse en las variables en lugar de hacerlo en los detalles técnicos. En el ejemplo de comercio en línea, un espacio de nombre contendría variables de contexto como latencia del servicio o *throughput*. Pascani libera a los desarrolladores de aplicaciones de mantener y compartir el estado de dichas variables y es transparente a los protocolos de comunicación y a los detalles técnicos al obtener y actualizar los valores de las variables, puesto que la infraestructura puede estar distribuida entre diferentes nodos de computación.

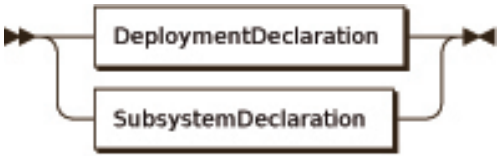

Figura 7. TypeDeclaration

Desde una perspectiva general, los monitores están compuestos de una sección de extensión, un nombre y un bloque de expresiones. El nombre calificado del monitor (la concatenación del paquete y el nombre separados por un punto) debe ser único en la ruta de la clase del proyecto.

Figura 8. MonitorDeclaration

Una declaración de extensión define el punto donde el monitor se extiende más allá de sus propias expresiones, desde donde es aumentado con declaraciones de otros tipos. Es un evento o un espacio de nombre importado.

Figura 9. ExtensionDeclaration

Una de las expresiones dentro de los monitores es la declaración de eventos. Cuando ella hace referencia a eventos de ejecución, ellos pueden ser reutilizados dentro de otros monitores para evitar la introducción innecesaria de sondas monitoras en el sistema objetivo. Esto se hace al importar explícitamente eventos, tratándolos como parte del monitor. La sentencia de importación del evento utiliza el nombre calificado del monitor declarado y el nombre de los eventos a importar.

Figura 10. ImportEventDeclaration

Importar espacios de nombre le permite a los monitores leer y actualizar variables desde diferentes contextos. Por ejemplo, si un desarrollador desea modelar indicadores de nivel de servicio en Pascani, puede crear un espacio de nombre con los valores de referencia y otro con los valores actuales del sistema. Esto promueve la separación y agrupamiento de variables de acuerdo con las diferentes restricciones de monitoreo.

Figura 11. ImportNamespaceDeclaration

La declaración de un espacio de nombre contiene la palabra reservada namespace seguida de un nombre y bloque de expresiones únicos.

Figura 12. NamespaceDeclaration

Los Códigos 1 y 2 muestran los contenidos de dos archivos válidos escritos en Pascani siguiendo las reglas de gramática descritas.

Código 1. Ejemplo mínimo de especificación de un espacio de nombre

```
1  package com.company
2
3  /*
4  * @date 2016/06/22
5  */
6  namespace SLI {
7     // Context variables
8  }
```

Código 2. Ejemplo mínimo de especificación de un monitor

```
1  package com.company
2
3  import java.util.List
4  using com.company.SLI
5
6  /*
7  * @date 2016/06/22
8  */
9  monitor Throughput {
10    // Monitor expressions
11 }
```

Reparto de variables de contexto

En Pascani, los espacios de nombre son almacenes jerárquicos de nombres que guardan variables de contexto. Un espacio de nombre se compone de una declaración de variables y espacios de nombre internos (opcionales) que crean la estructura jerárquica.

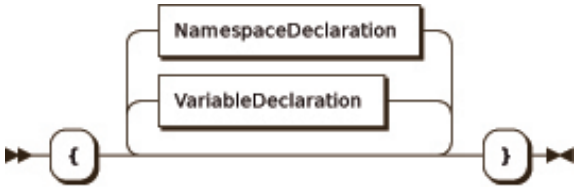

Figura 13. NamespaceBlockDeclaration

Una variable de contexto puede definirse como el valor inmutable de una variable. Si bien especificar su tipo es opcional, si no se especifique uno, se le

debe asignar un valor inicial a la variable. La parte derecha de la declaración es una expresión que indica que, no sólo se permiten tipos primitivos, sino también cualquier tipo de medios de cualquier expresión valida en el lenguaje .

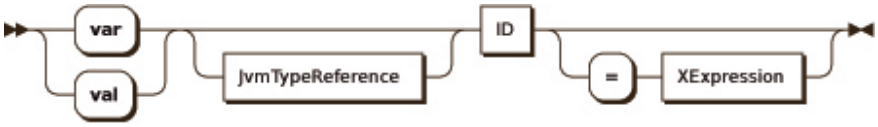

Figura 14. VariableDeclaration

Las variables soportan cualquiera de los tipos del sistema Java que sea serializado, incluyendo tipos personalizados; además, los espacios de nombre permiten importar clases Java. Los tipos deben ser serializados porque los valores de variables se envían a través de la red por diferentes componentes, como monitores que escuchan cambios en dichos valores. En un monitor, se accede a una variable utilizando su nombre calificado, el cual corresponde a la concatenación de la estructura jerárquica separada por un punto entre cada espacio de nombre y finalizando con el nombre de la variable. Por ejemplo, la variable reference en el Código 3 sería accesible con SLI.Performance.Throughput.reference. Es irrelevante que reference posea un valor inmutado, por consiguiente, puede únicamente leerse, no modificarse.

Código 3. Ejemplo de declaración de un espacio de nombre con espacios de nombres interiores y declaración de variables

```
1  namespace SLI {
2      namespace Performance {
3          namespace Throughput {
4              val reference = 100// transactions per minute
5              var Integer actual
6          }
7      }
8  }
```

El nombre calificado de una variable se utiliza como su nombre principal, su obtención y actualización se hace como se haría con una variable en cualquier lenguaje de propósito general como Java. Pascani también permite adjuntar información contextual al valor de una variable, lo que se hace al etiquetar la tarea utilizando la clase TaggedValue directamente o al utilizar el método de

ayuda *tag*. En el Código 4 se muestra un ejemplo de cómo obtener y actualizar el valor de una variable, incluyendo valores etiquetados.

Código 4. Obtención y actualización de variables

```
1   // Contextual information
2   val Server0 = #{ "node" -> "grid0", "component" -> "Server" }
3   val Server1 = #{ "node" -> "grid1", "component" -> "Server" }
4
5   // Update the value
6   SLI.Performance.Throughput.actual = 88
7   // Update the value and attach contextual information
8   SLI.Performance.Throughput.actual = tag(92, Server0)
9   SLI.Performance.Throughput.actual = tag(85, Server1)
10
11  // Get the current value
12  println("Reference value: " + SLI.Performance.Throughput.reference)
13  val Server0T = SLI.Performance.Throughput.actual(Server0)
14  val Server1T = SLI.Performance.Throughput.actual(Server1)
```

Especificación de componentes de monitoreo

Los componentes de monitoreo son reactivos por defecto, porque toda la lógica de monitoreo se dispara únicamente por un evento, es decir, no existe un método principal o un punto de entrada para invocar a un monitor directamente, sin importar que se trate de un evento: periódico, de ejecución o de cambio de variable. La especificación de un monitor puede contener declaraciones de variables y de eventos, manejadores (*handlers*) de eventos y bloques de configuración. La manera en que los eventos y sus manejadores trabajan juntos sigue el patrón de diseño Invocación Implícita [44], lo que significa que los

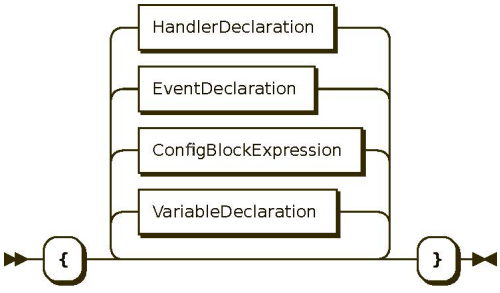

Figura 15. MonitorBlockExpression

eventos no conocen sus manejadores suscritos ni su lógica, y que los manejadores pueden ser añadidos o retirados en cualquier momento.

Los manejadores de eventos reciben dos parámetros: un objeto de evento y un diccionario de datos. Cada tipo de evento tiene una clase de evento con información acerca del evento a ser notificado. El primer parámetro permite acceder a datos del evento desde la lógica del manejador, mientras que el segundo es opcional y contiene información enviada en la suscripción del oyente del evento y es, por tanto, útil cuando un único manejador de evento maneja varios eventos. El Código 5 muestra un ejemplo de un manejador de evento simple con dos parámetros declarados.

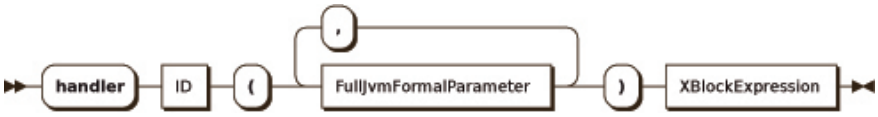

Figura 16. HandlerDeclaration

Código 5. Ejemplo de declaración de encargado de evento

```
1  handler ThroughputCalc(IntervalEvent e, Map<String, Object> data) {
2  // logic to calculate throughput
3  }
```

Las declaraciones de eventos se especifican con un nombre, la palabra clave opcional periodically –que indica si el evento está o no basado en periodos–, y el emisor del evento. Si el evento está basado en un lapso de tiempo, su emisor debe resolver a una expresión basada en el planificador de Unix (crontab), con una particularidad, Pascani permite agendar eventos en cuestión de segundos. En el caso contrario, el evento puede ser de ejecución de servicio o de cambio de variable.

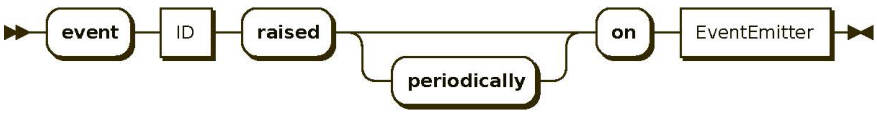

Figura 17. EventDeclaration

Los emisores de eventos son específicos para cada tipo de evento. Para eventos de ejecución, el emisor –también llamado objetivo– es una expresión FPath [45] apuntando al componente SCA, el servicio o la referencia a interceptar; cuando el objetivo es un componente, se interceptan todos sus servicios y referencias; para eventos de cambio, el emisor es una variable de un espacio de nombre. En este caso, el emisor se especifica utilizando el descriptor de acceso de la variable. Para eventos de cambio existe un parámetro adicional (opcional) llamado especificador de evento, elemento que permite ubicar condiciones lógicas sobre el nuevo valor para determinar si los manejadores suscritos deberían ser notificados o no. Esto se aplica únicamente a variables numéricas. En el Código 6 se muestran algunos ejemplos de declaraciones de evento.

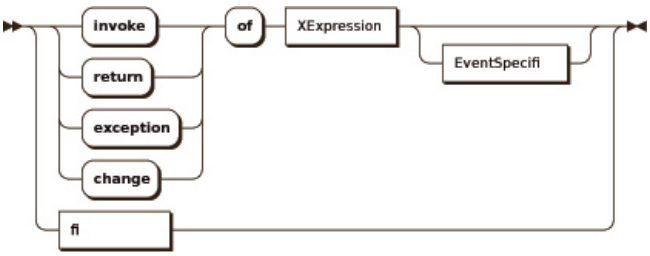

Figura 18. EventEmitter

Código 6. Ejemplos de declaración de evento

```
1  event e1 raised periodically on '*/5 * * * * ?'// every 5 seconds
2  event e2 raised on invoke of "$domain/scachild::Server/scaservice::print"
3  event e3 raised on invoke of SLI.Performance.Throughput.actual
4  event e4 raised on invoke of SLI.Performance.Throughput.actual below 80
5  event e5 raised on invoke of System.Performance.ResponseTime above 1.5 or below 0.5
6  event e6 raised on invoke of System.Performance.ResponseTime equal to 0.5
```

Las declaraciones de variables dentro de los monitores se especifican de la misma manera que las variables de contexto en los espacios de nombre. La última frase corresponde al bloque de configuración, el cual está destinado a: suscribir manejadores de eventos a eventos; especificar una URI cuando el emisor del evento no se despliega en el mismo nodo del monitor o especifica un puerto distinto al establecido por defecto; o indica que una sonda debería introducirse en el evento emisor especifico. Como los bloques de configuración se ejecutan después de la instanciación del monitor, es también útil para dar inicio a las variables del monitor que requieren mayor complejidad en su inicialización que la requerida por una expresión sencilla.

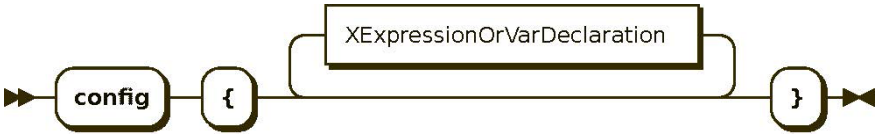

Figura 19. EventEmitter

Los Códigos 7 y 8 muestran dos especificaciones Pascani para monitorear *throughput* y tiempo de respuesta de un servicio SCA. Para mayor simplicidad, estos ejemplos incluyen únicamente la acción de actualizar variables de contexto y no la reacción a los cambios en ellas. Reaccionar a cambios contextuales requeriría la declaración de un evento change en la variable de interés y seguir el mismo patrón de suscripción descrito en los esquemas ya mencionados.

Código 7. Especificación de espacios de nombre para monitorear *throughput* y tiempo de respuesta

```
1  package com.company.monitoring
2
3  namespace SystemVars {
4      var throughput = 0d
5      var latency = 0d
6  }
```

Código 8. Especificación de monitor para seguimiento del *throughput* y el tiempo de respuesta

```
1  package com.company.monitoring
2
3  import java.util.Map
4  import org.pascani.dsl.lib.Probe
5  import org.pascani.dsl.lib.events.IntervalEvent
6  import org.pascani.dsl.lib.events.ReturnEvent
7
8  using com.company.monitoring.SystemVars
9
10 monitor Performance {
11
12     val searchTags = #{"service" -> "search", "host" -> "grid0"}
13     val paymentTags = #{"service" -> "payment", "host" -> "grid1"}
14     val server ="$domain/scachild::Server"
15
16     event minutely raised periodically on '0 * * * * ?'
17     event search raised on return of server + "/scaservice::search"
```

Código 8. Especificación de monitor para seguimiento del *throughput* y … (cont.)

```
18    event payment raised on return of server + "/scaservice::payment"
19
20    handler updateLatency(ReturnEvent e, Map<String, Object> data) {
21       SystemVars.latency = tag(e.value, data.mapValues[v|String.valueOf(v)])
22    }
23
24    handler updateTroughput(IntervalEvent e) {
25       val now = System.currentTimeMillis()
26       val searchCount = search.probe.countAndClean(-1, now)
27       val paymentCount = payment.probe.countAndClean(-1, now)
28       SystemVars.throughput = tag(searchCount, searchTags)
29       SystemVars.throughput = tag(paymentCount, paymentTags)
30    }
31
32    config {
33       #[ search, payment ].map[e | e.useProbe = true]
34       search.bindingUri = new URI("http://localhost:5000")
35       payment.bindingUri = new URI("http://localhost:5001")
36       search.subscribe(updateLatency, searchTags)
37       payment.subscribe(updateLatency, paymentTags)
38       minutely.subscribe(updateThroughput)
39    }
40 }
```

El Código 7 describe el espacio de nombre SystemVars declarando dos variables de contexto: Throughput y Latency, ambas de tipo double. Este espacio de nombre es utilizado por el monitor Performance que se presenta en el Código 8. Las líneas 3 a 6 de dicho código contienen importaciones de Java utilizadas en el resto de la especificación; la línea 8 declara que Latency utiliza el SystemVars; las líneas 16 a 18 declaran un evento basado en tiempo y dos eventos de ejecución –los servicios de búsqueda y pago–, mientras que la línea 33 indica que en estos dos eventos se debe introducir una sonda en el servicio correspondiente, pues sin ello no habría manera de recuperar los datos medidos para calcular el *throughput* del servicio; las líneas 34 y 35 le dicen a Pascani en dónde se está ejecutando el componente, con el fin de introducir la sonda de monitoreo; y las líneas 36 a 38 suscriben cada manejador al evento correspondiente.

Desde las especificaciones del monitor, Pascani genera los elementos que componen la infraestructura de monitoreo dinámica y especificaciones de

despliegue correspondientes, las cuales están escritas en Amelia, cuya semántica y sintaxis se explican a continuación.

AMELIA: UN DSL PARA EL DESPLIEGUE DINÁMICO DE SOFTWARE

Amelia es un lenguaje específico de dominio declarativo, se basa en reglas, para automatizar el despliegue de sistemas software distribuidos basados en componentes; está inspirado en herramientas como Ant [46] y Maven [47], aunque su sintaxis se basa en la herramienta de automatización de Make [48]; provee comandos para facilitar la ejecución de tareas de despliegue a través de múltiples nodos de computación; sus expresiones y sentencias están basadas en Xbase[45].

Desde las especificaciones de Amelia, la implementación del lenguaje genera artefactos de despliegue ejecutables que realizan las tareas requeridas para transferir, instalar y configurar los componentes de software a desplegar en cada uno de los nodos de procesamiento especificados, utilizando los protocolos SSH (*Secure SHell*) y SFTP (*SSH File Transfer Protocol*) y ejecutando los comandos especificados en las reglas de Amelia.

Amelia consta de dos elementos principales: subsistemas y despliegues. Un subsistema contiene un conjunto de operaciones de despliegue para un sistema autocontenido que pertenece a un sistema más grande, mientras que un despliegue contiene sentencias de control de flujo que ejecutan el despliegue de un conjunto de subsistemas de una determinada manera. Los subsistemas están compuestos de reglas de ejecución que se ejecutan en nodos específicos de computación, estas reglas son contenedores dependientes de comandos que guían el despliegue de componentes software.

La presentación de la gramática de Amelia se hace utilizando la misma metodología que se mostró en la presentación de la gramática de Pascani. La gramática completa se presenta en el Anexo 2.

Un archivo de especificación válido en Amelia presenta una definición obligatoria del nombre del paquete y una definición –opcional– de una sección importada de Java y un tipo de declaración. Esto significa que los archivos sin declaración de tipo son una especificación válidas que no generan ningún artefacto. Una unidad de compilación es una unidad de translación, esto es, un archivo fuente que contiene la definición de un subsistema o un despliegue.

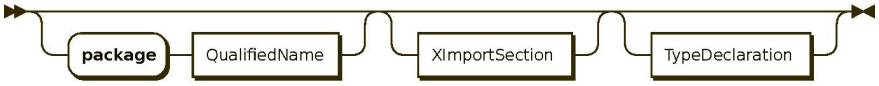

Figura 20. DeploymentSpecificationModel

Los subsistemas contienen la definición de los *hosts* (nodos de computación) y las reglas de ejecución, mientras que los despliegues permiten configurar estrategias de despliegue de declaraciones de subsistemas.

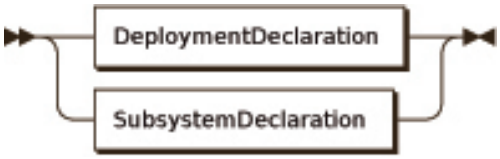

Figura 21. TypeDeclaration

La declaración de un despliegue provee los medios para realizar comportamientos personalizados en los despliegues, como sistemáticamente repetir el mismo despliegue un numero dado de veces o reintentarlo cuando falle. Este tipo de declaración se compone de una sección de extensión opcional, un nombre y un bloque de expresiones.

Figura 22. DeploymentDeclaration

Un subsistema se compone de una unidad dependiente del despliegue pensada para especificar cómo desplegar un conjunto de componentes en distintos hosts. Su declaración contiene una sección de extensión opcional, un nombre y un bloque de expresiones.

Figura 23. SubsystemDeclaration

Las declaraciones son extensiones útiles para incluir un subsistema o especificar una dependencia de ejecución. Por otra parte, incluir un subsistema dentro de otro implica que tanto los parámetros del sistema incluido como sus reglas de ejecución, se inserten y puedan ser tratadas como parte del subsistema. Los desacuerdos por nombres se manejan haciendo accesibles los parámetros de colusión, utilizando su nombre calificado. Asimismo, cuando un subsistema se incluye en una declaración de despliegue, él se toma en cuenta dentro de la estrategia de despliegue, permitiendo su instanciación utilizando diferentes valores para sus parámetros.

Figura 24. ExtensionDeclaration

Con el fin de incluir un subsistema, la declaración include debe especificar el nombre calificado del subsistema incluido, esto es, su nombre de paquete concatenado con su nombre, utilizando un punto para separar cada palabra.

Figura 25. IncludeDeclaration

Las dependencias del subsistema son una manera de establecer un orden de ejecución para asegurar la dependencia de los componentes. El especificar una dependencia requiere explícitamente declararla con el nombre calificado del despliegue o la declaración del subsistema.

Figura 26. DependDeclaration

En los Códigos 9 y 10 se muestran los contenidos de dos archivos validos escritos en Amelia utilizando las citadas reglas de gramática.

Código 9. Ejemplo mínimo de especificación de un subsistema

```
1  package com.company
2
3  includes Common
4  depends on Test1
5
6  subsystem Test2 {
7      // Variables and on-host declarations
8  }
```

Código 10. Ejemplo mínimo de especificación de un despliegue

```
1  package com.company
2
3  includes Test1
4  includes Test2
5
6  deployment CustomStrategy {
7      // Deployment expressions
8  }
```

El bloque de expresiones que componen un subsistema puede contener la declaración de variables, bloques de expresiones y bloques de configuración.

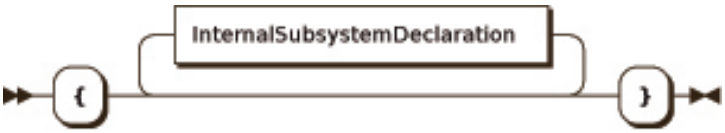

Figura 27. SubsystemBlockExpression

Figura 28. InternalSubsystemDeclaration

En Amelia, la declaración de una variable puede ser un valor inmutable, una variable o un parámetro. Los dos primeros casos se refieren a variables regulares privadas, mientras que el tercero tiene diferentes semánticas asociadas.

Los parámetros se incluyen en el constructor del subsistema, lo que indica que los subsistemas pueden ser instanciadas con diferentes argumentos. Cuando se incluyen los subsistemas, los parámetros incluidos se pasan al constructor del subsistema, por lo que en una cadena *include*, el constructor del último subsistema contiene todos los parámetros incluidos en todos los subsistemas en la cadena *include*. El orden se determina de acuerdo con el orden en las declaraciones *include*.

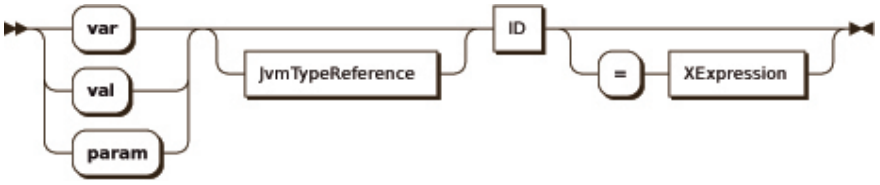

Figura 29. VariableDeclaration

El grupo de bloques de expresiones agrupa las reglas que serán ejecutadas en el *host* dado.

Figura 30. ConfigBlockExpression

Las acciones de despliegue se especifican utilizando reglas. Cada regla se compone de un objetivo, una enumeración opcional de dependencias (otros objetivos) y un conjunto de comandos. Cuando una regla depende de otras, sus comandos no pueden ser ejecutados hasta que todos los comandos declarados en las otras reglas se ejecuten.

En el Código 11 se muestra un ejemplo de declaración de reglas (Figura 31) y dependencias.

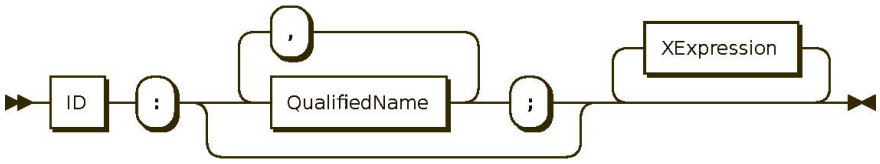

Figura 31. RuleDeclaration

Código 11. Ejemplo de declaración de ejecución de reglas

```
1  init:
2      // commands
3  server: init;
4      // commands
5  client: init, server;
6      // commands
```

Con el fin de facilitar la especificación de las tareas de despliegue, Amelia provee un conjunto de comandos con chequeo automático de error durante la compilación y ejecución. Además del reconocimiento de los estados exitoso y erróneo, los comandos permiten: cambiar el directorio de trabajo; compilar un componente FraSCAti; ejecutar un componente compilado; transferir archivos o directorios a un nodo de procesamiento remoto; evaluar expresiones FScript o una rutina en tiempo de ejecución FraSCAti; y ejecutar comandos Unix.

Adicionalmente, Amelia permite configurar cualquier valor predefinido para cada comando con el fin de modificar su comportamiento. Añadir "…" al final de la declaración de un comando hará accesible al constructor del comando, lo que permite modificar sus valores internos. Los comandos de Amelia son:

- Cd. Requiere únicamente el nuevo directorio de trabajo, el cual debe resolver a una expresión *string* (Figura 32).

Figura 32 . Comando Cd

- Compile. Se compone de expresiones *string* que representan, de izquierda a derecha, el directorio del código fuente, el archivo de salida y, opcionalmente, la ruta de la clase (Figura 33).

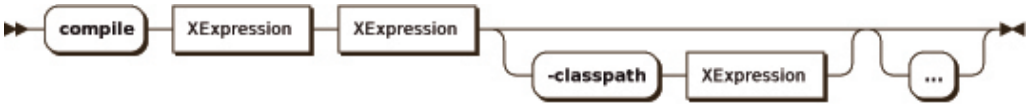

Figura 33. Comando Compile

- Run. Se compone de: una expresión integral opcional, que representa el puerto donde está expuesta la consola FraSCAti FScript; un *string*, que representa el componente (*i.e*, el archivo .composite); y la ruta de la clase (cuando se ejecuta el componente en modo cliente se deben ejecutar parámetros adicionales como los nombres del servicio y del método y, de manera opcional, una lista de argumentos, todos ellos expresiones *string* (Figura 34).

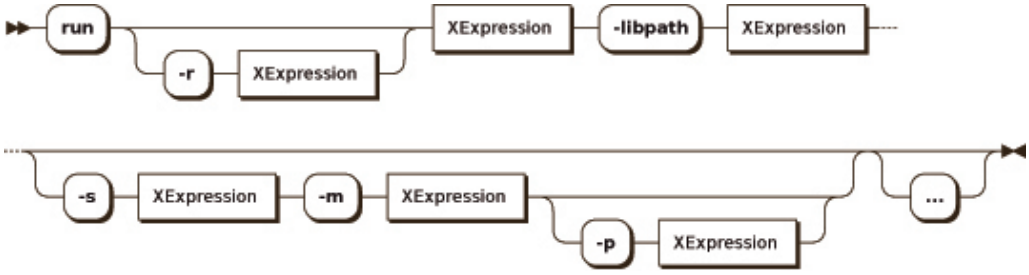

Figura 34. Comando Run

- Transfer. Requiere la ubicación local y remota de un archivo o un directorio (Figura 35). (si la ubicación remota no existe, será creada).

Figura 35. Comando Transfer

- Eval. De manera opcional, espera la URI donde el componente FraSCAti se ejecuta en tiempo de ejecución y el script FScript a evaluar (Figura 36).

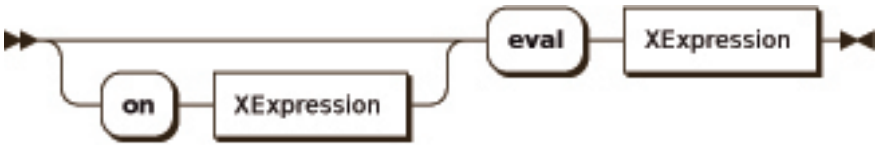

Figura 36. Comando Eval

Cualquier otro comando válido en Unix se presenta como un *string* (Figura 37). En el Código 12 se presentan varios ejemplos de especificación de comandos.

Figura 37. Comando personalizado

Código 12. Ejemplos de especificaciones de comandos

```
1  cd "/home/user/projects"
2  compile "src" "project" -classpath # [ "libs/lib1.jar" , "lib/lib2.jar" ]
3  run -r 5000 "server" -libpath # [ "server.jar" , "lib/common.jar" ]
4  run "client" -libpath # [ "client.jar" , "lib/common.jar" ] -s "r" -m "run"
5  scp "~/project/files" to "/tmp/project/files"
6  on new URI("http://localhost:5000") eval "some-procedure()"
7  cmd "date>> date.txt"
```

El ejemplo helloworld–rmi

Los Códigos 13 y 14 muestran la especificación de despliegue del ejemplo helloworld–rmi que está disponible con la distribución FraSCAti. Este ejemplo consta de dos componentes SCA: un servidor que expone un servicio de impresión a través de RMI y un cliente que consume dicho servicio para imprimir un mensaje en salida estándar. Nótese que $FRASCATI_HOME no se relaciona con Amelia de ninguna manera, sino que es únicamente una variable de entorno que se resuelve durante la sesión SSH.

Código 13. Especificación de subsistema para helloworld-rmi

```
1  package com.company
2
3  import org.amelia.dsl.lib.descriptors.Host
4
5  subsystemHelloworld {
6
7  param Host host = new Host("localhost", 21, 22, "username", "password")
8
9  on host {
10     compilation:
11           cd "$FRASCATI_HOME/examples/helloworld-rmi/"
12           compile "server/src" "s"
13           compile "client/src" "c"
14
15     execution: compilation;
16           run "helloworld-rmi-server" -libpath "s.jar"
17           run "helloworld-rmi-client" -libpath "c.jar" -s "r" -m "run"
18     }
19
20 }
```

Código 14. Especificación de despliegue para helloworld-rmi

```
1  package com.company
2
3  import org.amelia.dsl.lib.util.RetryableDeployment
4  import org.amelia.dsl.lib.descriptors.Host
5
6  includes com.company.Helloworld
7
8  deployment WarmingUp {
9
10 val helper = new RetryableDeployment()

11 val remote = new Host("192.168.99.100", 25632, 41256, "username", "password")
12 set(newHelloworld(remote))
13
14     for (i : 1..10) {
15         helper.deploy([
16             start(true)
17         ], 3)
18     }
19
20  }
```

La especificación de despliegue WarmingUp (línea 8, Código 14) combina dos estrategias comunes al despliegue de sistemas: ejecutar el sistema varias veces con el fin de preparar el ambiente para ejecutar pruebas de rendimiento y reintentar el despliegue si se identifica una falla.

La primera estrategia se especifica por medio del método start y la sentencia *for* (línea 14). El método *start* puede invocarse con dos parámetros: el primero indica si los componentes ejecutados deben detenerse después del despliegue o no, y la segunda si se debe apagar el despliegue, esto último útil si el usuario desea observar la salida de la sesión SSH (la salida estándar de los componentes ejecutados). La segunda estrategia se realiza a través de la utilidad RetryableDeployment, la cual ejecuta de nuevo la función lambda el número de veces indicado por el segundo parámetro. En este caso, como se nota en la línea 16, el despliegue podría ser ejecutado a tiempo y, si el resultado no es exitoso, se intentará de nuevo dos veces más.

Por defecto, los subsistemas se inicializan utilizando un constructor vacío; en caso de que haya parámetros sin inicializar, una instancia del subsistema debe ser proporcionada para evitar errores asociados con invocaciones de objetos tipo *null*. La línea 12 muestra cómo configurar la instancia de un subsistema utilizando un constructor diferente, lo que puede ser utilizado para proveer diferentes instancias de subsistemas por despliegue, en este caso, mover la invocación set dentro de la sentencia *for* permite desplegar el sistema en un *host* diferente en cada iteración, asumiendo que un objeto *host* se pasa al constructor del subsistema.

IMPLEMENTACIÓN

Pascani y Amelia se desarrollaron utilizando el lenguaje de programación Java y Xtext, un *framework* de ingeniería de lenguaje que asiste a los desarrolladores en la generación de implementaciones de lenguaje, incluido el *parser* (analizador sintáctico), el enlazador, el chequeador de escritura y el compilador, y soporta la edición desde IDE bien conocidos como Eclipse e IntelliJ desde una definición gramática. Las definiciones gramáticas completas para Pascani y Amelia se presentan en los Anexos 1 y 2, respectivamente.

Para soportar las funcionalidades de cada lenguaje, se desarrolló una librería en tiempo de ejecución abstrayendo elementos comunes utilizados por todas las aplicaciones generadas por el compilador, lo que le permitió

a cada compilador reducir la cantidad de líneas de código generadas y modularizar la implementación, de tal manera que se puedan crear nuevos elementos al combinar elementos existentes. En el caso de Pascani, existe una librería adicional que contiene recursos y elementos que soportan todos los conceptos relacionados con SCA.

La implementación de los dos lenguajes ha sido optimizada para el IDE Eclipse para ofrecer edición controlada por sintaxis, chequeo estático de errores, refactorización y generación de código. Las instrucciones para instalar Pascani o Amelia están disponibles en https://unicesi.github.io/pascani/releases/ y https://unicesi.github.io/amelia/releases/, respectivamente.

PASCANI

El código de implementación para el DSL Pascani comprende 11.028 líneas de código (SLOC, *Single Lines Of Code*), sin contar el código fuente generado, distribuidas entre los proyectos listados en la Tabla 4. Las semánticas del lenguaje están escritas utilizando las facilidades que proporciona Xtext, como chequear métodos que son invocados una vez que el modelo se ha analizado sintácticamente. A modo de ejemplo, el Código 15 describe una regla semántica para encontrar tipos de parámetros inválidos en manejadores de

Tabla 4. Proyectos Java que conforman la implementación de Pascani

	Proyecto	Contenido	SLOC
Eclipse	org.pascani.dsl.feature	Definición de los plug-in de los proyectos que componen el DSL	785
	org.pascani.dsl.build.feature	Sitio de actualización de PASCANI.	7
Plug-in de Eclipse	org.pascani.dsl	Clases que pertenecen a la librería núcleo del lenguaje, el compilador PASCANI, el generador de código, el modelo JVM, las semánticas de alcance, el tipo de sistema y la validación de semántica.	2257
	org.pascani.dsl.ide	Clases relacionadas con la IDE de Eclipse (ninguna, por el momento).	21
	org.pascani.dsl.ui	Definiciones de plug-in y sus correspondientes implementaciones de Java respecto de la interfaz de usuario de Eclipse; y clases que implementan o configuran el contenido asistido de la IDE, las reglas a resaltar, el etiquetado, el resumen y el arreglo rápido.	677
	org.pascani.dsl.lib.osgi	Librerías de tiempo de ejecución de PASCANI en un paquete OSGi.	346

Tabla 4. Proyectos Java que conforman la implementación de Pascani (cont.)

	Proyecto	Contenido	SLOC
Proyectos Maven	org.pascani.dsl.lib	Clases que soportan los componentes de la infraestructura dinámica de monitoreo generados por el compilador	2666
	org.pascani.dsl.lib.compiler	Clases de utilidad usadas por el compilador de Pascani	987
	org.pascani.dsl.lib.sca	Clases y recursos que soportan la actividad en tiempo de ejecución relacionada con el domino del SCA	1887
	org.pascani.dsl.dbmapper	Clases que mapean los eventos de monitoreo en datos planos que pueden ser enviados a una base de datos.	1128
	org.pascani.dsl.target	Definición del objetivo de Eclipse.	15
	org.pascani.tycho.parent	Configuración del ciclo de vida del constructor del plug-in y de proyectos Maven.	239
Proyectos Web	org.pascani.dsl.web	Clases y recursos para publicar un editor web utilizando el compilador PASCANI y los plug-in de UI de Eclipse como servicios.	346

Código 15. Regla de semántica para encontrar parámetros inválidos en manejadores de eventos (escrito en Xtend)

```
1  @Check
2  def checkHandlerParameters(Handler handler) {
3     if (handler.params.size> 2) {
4        error("Event handlers cannot have more than two parameters",
            PascaniPackage.Literals.HANDLER__PARAMS)
5     }
6
7     if
        (handler.params.get(0).actualType.getSuperType(org.pascani.dsl.lib.Event) == null) {
8        error("'The ÂńIF handler.params.size > 1 firstÂńENDIF parameter must be subclass of Event'",
            PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
9     }
10
11    if (handler.params.size> 1) {
12       val actualType = handler.params.get(1).actualType.getSuperType(Map)
13       val showError = actualType == null
14          || actualType.typeArguments.size != 2
15          || !actualType.typeArguments.get(0).identifier.equals(String.canonicalName)
16          || !actualType.typeArguments.get(1).identifier.equals(Object.canonicalName)
17       if (showError)
18          error("'The second parameter must be of type Map<String, Object>'",
               PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
19    }
20 }
```

eventos. La validación de semántica realizada en este método consiste en que los manejadores de eventos no pueden tener más de dos parámetros y que su tipo debe ser *Event* y *Map, con* tipos de parámetro *String* y *Object*, respectivamente.

Librería en tiempo de ejecución y librería SCA

La librería en tiempo de ejecución contiene las clases que soportan los componentes de la infraestructura de monitoreo dinámico generados por el compilador, las cuales se organizan en los siete paquetes descritos en la Tabla 5; la Figura 38 corresponde a un diagrama de clases simplificado que contiene las clases dentro de cada paquete.

La librería SCA, por su parte, contiene las clases y recursos que soportan la actividad en tiempo de ejecución relacionada con el dominio SCA. Los elementos que la componen son: un juego de interceptores configurado para producir varios tipos de eventos de ejecución soportados por el lenguaje; los procedimientos FScript, para añadir y remover sondas monitoras en tiempo de ejecución; y las instalaciones que facilitan la introspección de aplicaciones FraSCAti. En la Tabla 6 se resume la organización de las clases de esta librería, y en la Figura 39 se presenta un diagrama de clases simplificado de la librería.

Tabla 5. Pascani: paquetes de clases de la librería en tiempo de ejecución

Paquete	Contenido
org.pascani.dsl.lib	Clases derivadas de los elementos de la infraestructura de monitoreo dinámico en su forma básica (interfaces o clases abstractas).
org.pascani.dsl.lib.events	Tipos de evento que soporta la infraestructura (todos los subtipos de Event). El lenguaje únicamente soporta eventos basados en tiempo (InternalEvent), eventos de cambio de variable (ChangeEvent) y eventos de ejecución (InvokeEvent, ReturnEvent, TimeLapseEvent, y ExceptionEvent).
org.pascani.dsl.lib.infrastructure	Clases que realizan la estructura de monitoreo, como implementaciones de Namespace y Probe, y definiciones básicas de los elementos que participan en los mecanismos de comunicación (consumidor, productor, servidor RPC y cliente).
org.pascani.dsl.lib.infrastructure.rabbitmq	Implementaciones de los elementos que participan en los mecanismos de comunicación (para implementar el componente Message Broker y el esquema cliente servidor RPC, se utiliza RabbitMQ [A].
org.pascani.dsl.lib.util	Clases de utilidad empleadas por otras clases y las clases que trabajan con distintos sets de eventos.
org.pascani.dsl.lib.util.events	Clases para realizar manejadores, eventos y suscripciones a eventos.
org.pascani.dsl.lib.util.log4j	Clases de utilidad para adjuntar logs a la infraestructura.

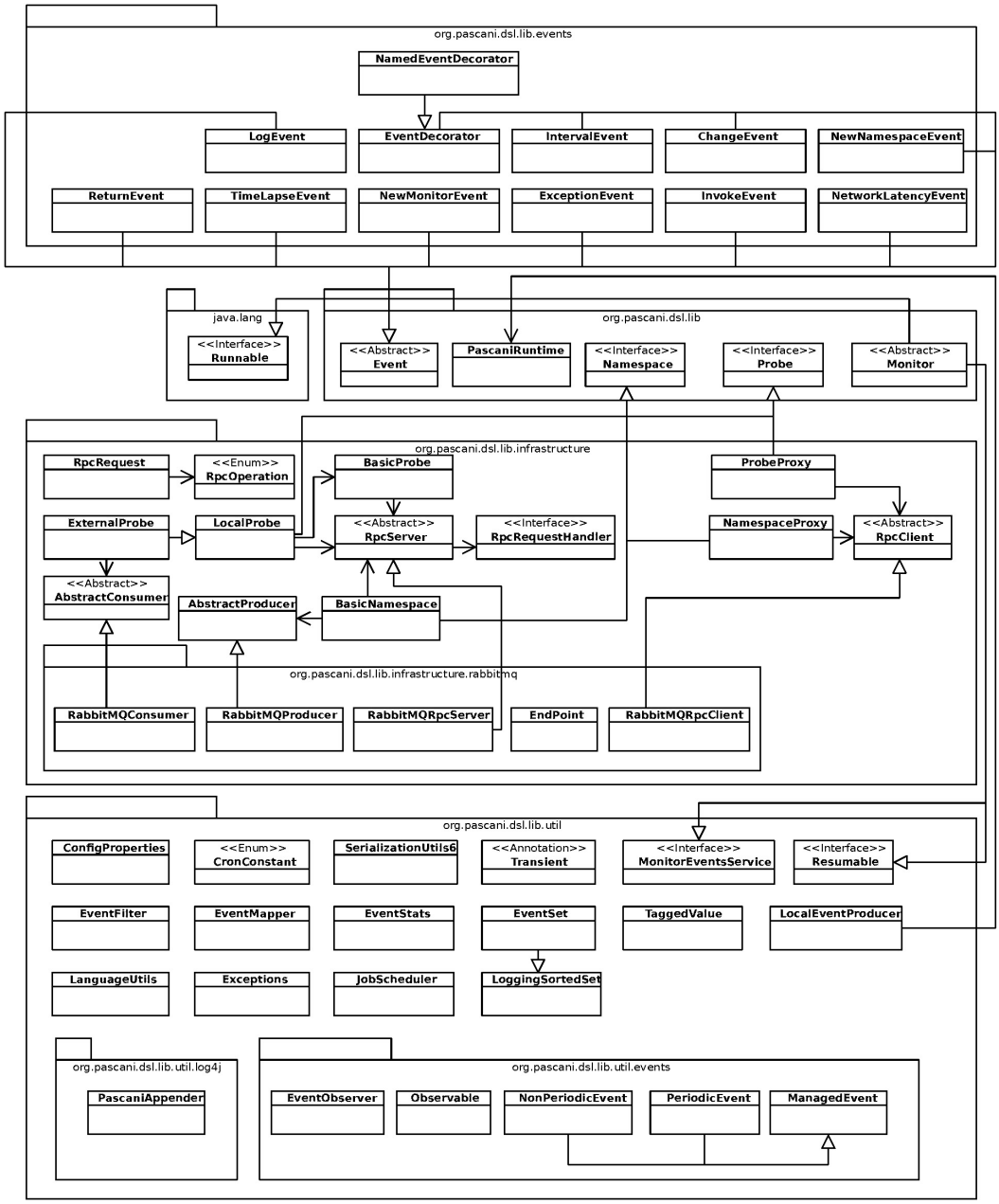

Figura 38. Pascani: diagrama de clases simplificado de la librería en tiempo de ejecución

Tabla 6. Pascani: paquetes de clases de la librería SCA

Paquete	Contenido
org.pascani.dsl.lib.sca	Clases de utilidad para realizar la introspección sobre las aplicaciones FraSCAti y manipular sondas monitoras en tiempo de ejecución.
org.pascani.dsl.lib.sca.explorer	Extensiones para la interfaz gráfica del explorador FraSCAti que permiten gestionar el estado de los monitores y sus eventos.
org.pascani.dls.lib.sca.intents	Implementación de los interceptores de servicio de acuerdo con los eventos de ejecución soportados por el lenguaje.
org.pascani.dsl.lib.sca.probes	Implementación de las sondas monitoras configuradas para manejar los eventos de ejecución soportados por el lenguaje.

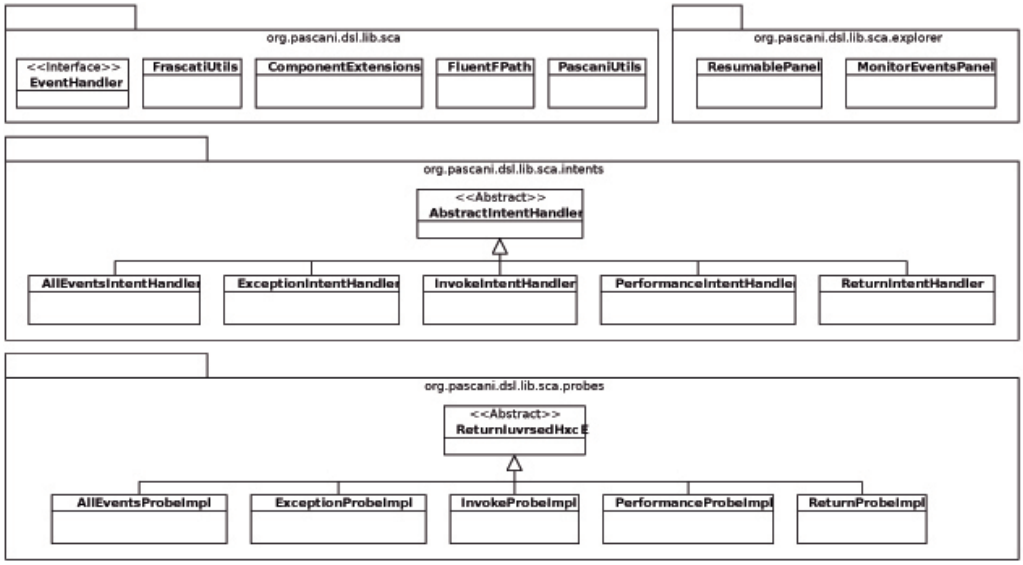

Figura 39. Pascani: diagrama de clases simplificado de la librería SCA

Método de translación de Pascani

El compilador Pascani traduce las especificaciones de los espacios de nombre y monitores en componentes SCA con una implementación de Java. Las clases Java generadas utilizan los elementos del tiempo de ejecución y de las librerías SCA, lo que reduce notablemente la cantidad de líneas de código generadas. A continuación se detalla el mapeo entre elementos desde las especificaciones de Pascani y sus correspondientes elementos de clase de Java.

Declaraciones de espacios de nombres

Desde las especificaciones de espacios de nombres, el compilador genera un archivo compuesto (un componente descriptor SCA) y dos clases Java: una implementación de espacio de nombres —la cual es también la implementación del componente SCA generado— y un proxy de espacio de nombres. La primera es una realización de la interfaz `Namespace` propuesta, con todas las variables de espacios de nombres registradas; el segundo es una clase proxy mediando entre elementos del cliente y la implementación de espacios de nombres (por ejemplo, cuando un monitor lee el valor de una variable de contexto, la petición pasa a través del proxy, crea una petición RPC y la envía a la implementación del espacio de nombre; éste responde a la petición de RPC y el proxy retorna el valor de la variable al monitor).

Las Figuras 40 y 41 corresponden a una vista abstracta del mapeo entre espacios de nombres y elementos de clases de Java. Desde la especificación de un espacio de nombre el compilador utiliza el paquete nombre, sección de importación y documentación para generar la implementación del espacio de nombres y el proxy. La implementación de un espacio de nombre es simple porque hereda de la clase `BasicNamespace` que ya implementa el comportamiento descrito en la vista general del diseño de la infraestructura de monitoreo dinámico. Cada una de las variables declaradas en el espacio de nombres –incluyendo las declaradas en espacios de nombres internos– se traduce en una sentencia de registro o invocación de método. Por su parte, un proxy para un espacio de nombres generado es más complejo que una simple clase.

La declaración de variables se traduce en métodos *getter* y *setter*, incluyendo variantes que permitan leer y actualizar valores que consideran la información contextual de la cuenta. Cada método reenvía la invocación a métodos equivalentes de la clase `NamespaceProxy`. Las declaraciones internas de espacios de nombre tienen el mismo tratamiento, solo que se convierten en campos privados del espacio de nombre contenedor. La clase generada es una utilidad que simplifica la notación en la especificación de lenguaje. Dado que Pascani está basado en el lenguaje de expresión Xbase, las invocaciones regulares, como `object.method(parameter)`, pueden ser reescritas como una asignación.

En este caso, `SLI.latency = 0.5` es equivalente a `SLI.latency(0.5)`. De la misma manera, las invocaciones sin parámetros pueden ser escritas sin paréntesis

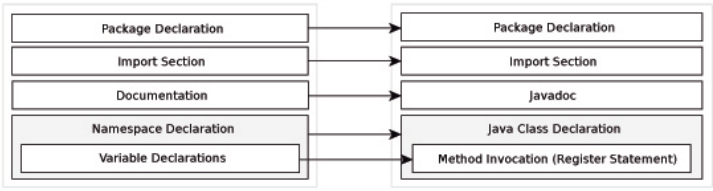

Figura 40. Mapeo entre la definición de un espacio de nombres (izquierda) y su correspondiente elemento de clase de Java (derecha)

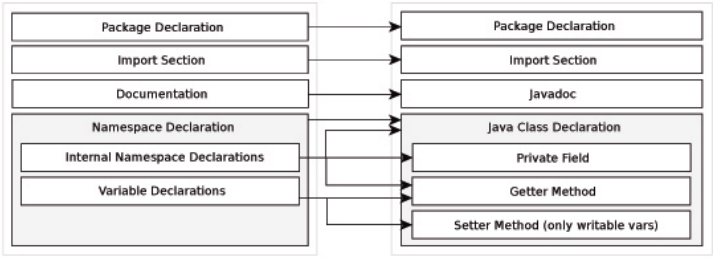

Figura 41. Mapeo entre la definición de un espacio de nombre (izquierda) y sus correspondientes elementos de clase Java Proxy (derecha)

como SLI.latency. En resumen, el *proxy* de espacios de nombres generado actúa como utilidad declarando la jerarquía del método representando la jerarquía en los espacios de nombres y variables.

Declaración de monitores

A partir de una declaración de monitores, el compilador de Pascani genera un archivo compuesto (*i.e.*, un descriptor de componentes SCA), junto con su correspondiente implementación en Java. Como para declaraciones de espacios de nombre, la clase Java generada utiliza el paquete, nombre, sección de importación y documentación, con si se declararan en el archivo de especificación. La utilización de los espacios de nombre y las declaraciones de variables se traducen en campos estáticos privados, haciéndolos accesibles desde la implementación del monitor, incluyendo clases internas.

La declaración de eventos también se traduce en campos y su tipo puede ser PeriodicEvent o NonPeriodicEvent. En el último caso, se crea una clase interna puesto que requiere una implementación. Desde los gestores de eventos, el compilador genera tanto un campo privado como una clase interna heredada de EventObserver.

Finalmente, los bloques de configuración se trasladan a métodos de instancia que se invocan secuencialmente, de acuerdo con su orden de aparición en la especificación del monitor (Figura 42).

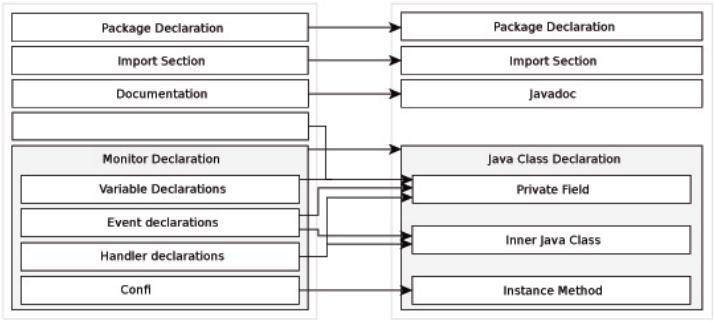

Figura 42. Mapeo entre la definición de un monitor (izquierda) y sus correspondientes elementos de clase de Java (derecha)

Almacenamiento y visualización de la información monitoreada

La infraestructura de monitoreo dinámico soporta diversas tecnologías para manejar y visualizar variables de contexto. Por defecto, Pascani soporta almacenar información en InfluxDB [49], ElasticSearch [50], RethinkDB [51], FnordMetric [52] y archivos CSV, desde donde se pueden usar productos de visualización de datos de código abierto, como Grafana [53], FnordMetric y Kibana [54], para ver representaciones gráficas de los datos monitoreados. En la Figura 43 se presenta el diagrama de despliegue de la infraestructura de monitoreo generada por el compilador Pascani. Las especificaciones de monitor y espacios de nombre, junto con los elementos del sistema objetivo, se compilan en archivos *.jar*, desplegados en un ambiente de ejecución FraSCAti. Ambos, monitores y espacios de nombre, dependen de las librerías en tiempo de ejecución de Pascani, por lo que deben se deben desplegar conjuntamente. El componente mapeador de datos no es un componente SCA, por lo que requiere únicamente un ambiente de ejecución Java. Este componente se suscribe a espacios de nombres utilizando el corredor de mensajes RabbitMQ [55].

AMELIA

El código de implementación para el DSL Amelia contiene 7.239 SLOC (sin contar el código fuente generado) distribuidos en los proyectos de la Tabla 7.

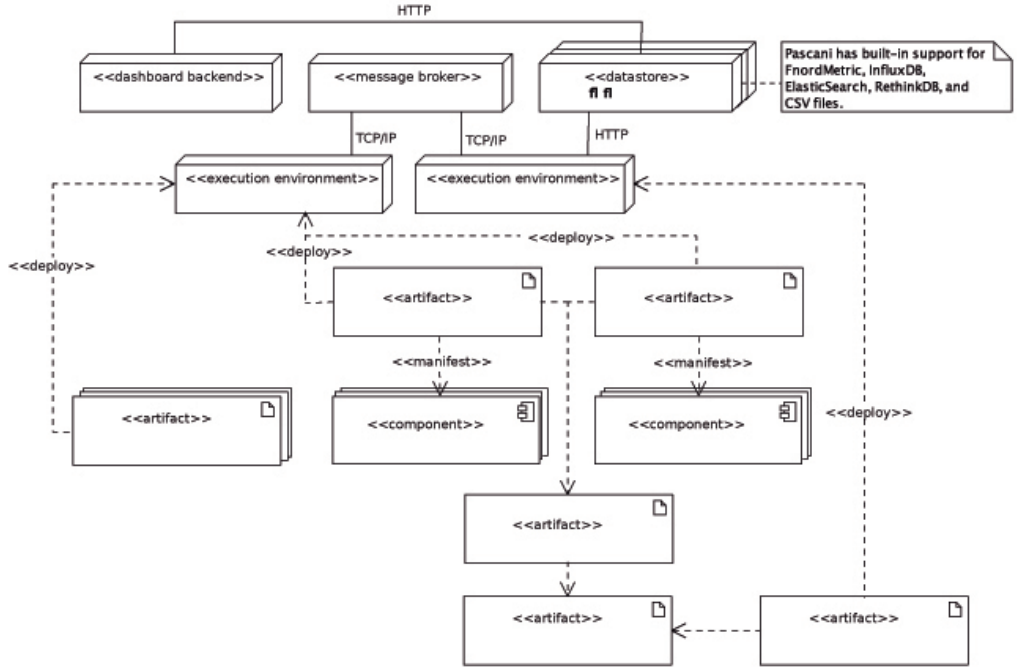

Figura 43. Diagrama de despliegue de la infraestructura de monitoreo dinámico

Tabla 7. Proyectos que conforman la implementación de Amelia

	Proyecto	Contenido	SLOC
Eclipse	org.amelia.dsl.feature	Definición del proyecto de *plug-ins* que componen el DSL Amelia.	782
	org.amelia.dsl.build.feature	Sitio de actualización de Amelia.	7
Plug-ins de Eclipse	org.amelia.dsl	Clases que pertenecen a la librería principal del lenguaje; compilador Amelia, generador de código, modelo JVM, semánticas de enfoque, tipo de sistema y validación de semántica.	1970
	org.amelia.dsl.ide	Clases relacionadas a la IDE de Eclipse (ninguna, por el momento).	21
	org.amelia.dsl.ui	Definiciones de plug-ins y correspondientes implementaciones Java relacionadas con la interfaz de usuario de Eclipse; clases que implementan o configuran el contenido asistido de la IDE, reglas para resaltar, editor, etiquetado, resumen y arreglo rápido.	603
	org.amelia.dsl.lib.osgi	Librería en tiempo de ejecución de Amelia en un paquete OSGi.	13

Tabla 7. Proyectos que conforman la implementación de Amelia (cont.)

	Proyecto	Contenido	SLOC
Proyectos Maven	org.amelia.dsl.lib	Clases que soportan los manejos de sesión SSH y FTP, programación de ejecución y gestión de dependencias; y funcionalidades de logueo y reporte.	3223
	org.amelia.dsl.target	Definición objetivo de Eclipse.	15
	org.amelia.tycho.parent	Elemento que configura el ciclo de vida del constructor del plug-in y de proyectos Maven.	262
Proyectos Web	org.amelia.dsl.web	Clases y recursos para publicar un editor web utilizando el compilador Pascani y los plug-in de UI de Eclipse como servicios.	343

Las semánticas del lenguaje fueron escritas utilizando las facilidades provistas por Xtext, esto es: chequear métodos que son invocados una vez el modelo ha sido analizado. Como ejemplo, el Código 16 describe una regla semántica para validar el parámetro host en expresiones *on hosts*. La validación de semántica realizada en este método es la siguiente: si el tipo actual (inferido) de expresión no es de tipo host o un objeto iterable de host, se muestra un error.

Código 16. Regla semántica para validar el tipo de parámetro *host* en expresiones con *host* (escrito en Xtend)

```
1  @Check
2  def void checkHost(OnHostBlockExpressionblockExpression) {
3     val type = blockExpression.hosts.actualType
4     val isOk = type.getSuperType(Host) != null || type.getSuperType(Iterable) != null
5     val msg ="'"The hosts parameter must be of type ÂńHost.simpleName or
          Iterable<ÂńHost.simpleName>, Âńtype.simpleName was found instead"'
6     val showError = !isOk
7        || type.getSuperType(List).typeArguments.length == 0
8        || !type.getSuperType(Iterable).typeArguments.get(0).identifier.equals(Host.canonicalName)
9     if (showError) {
10       error(msg, AmeliaPackage.Literals.
             ON_HOST_BLOCK_EXPRESSION__HOSTS, INVALID_PARAMETER_TYPE)
11    }
12 }
```

Librería en tiempo de ejecución de Amelia

Esta librería se compone de clases que implementan todos los conceptos en el lenguaje como subsistemas, comandos y dependencias. Los subcomponentes

más relevantes en esta librería se relacionan con: el manejo de sesiones SSH y FTP, la planeación en la ejecución y gestión de dependencias, los comandos, y las funcionalidades de logueo y reporte. Las clases en esta librería se organizan en los paquetes que se presentan en la Tabla 8. En la Figura 44 se presenta un diagrama de clases simplificado de la librería en tiempo de ejecución.

Tabla 8. Amelia: paquetes de clases de la librería en tiempo de ejecución

Proyecto	Contenido
org.amelia.dsl.lib	Clases que implementan los manejadores de sesiones SSH y FTP y la ejecución y planeación de tareas.
org.amelia.dsl.lib.descriptors	Clases utilizadas principalmente para describir comandos, paquetes de activos y hosts.
org.amelia.dsl.lib.util	Clases con utilidades y para inicio de sesión

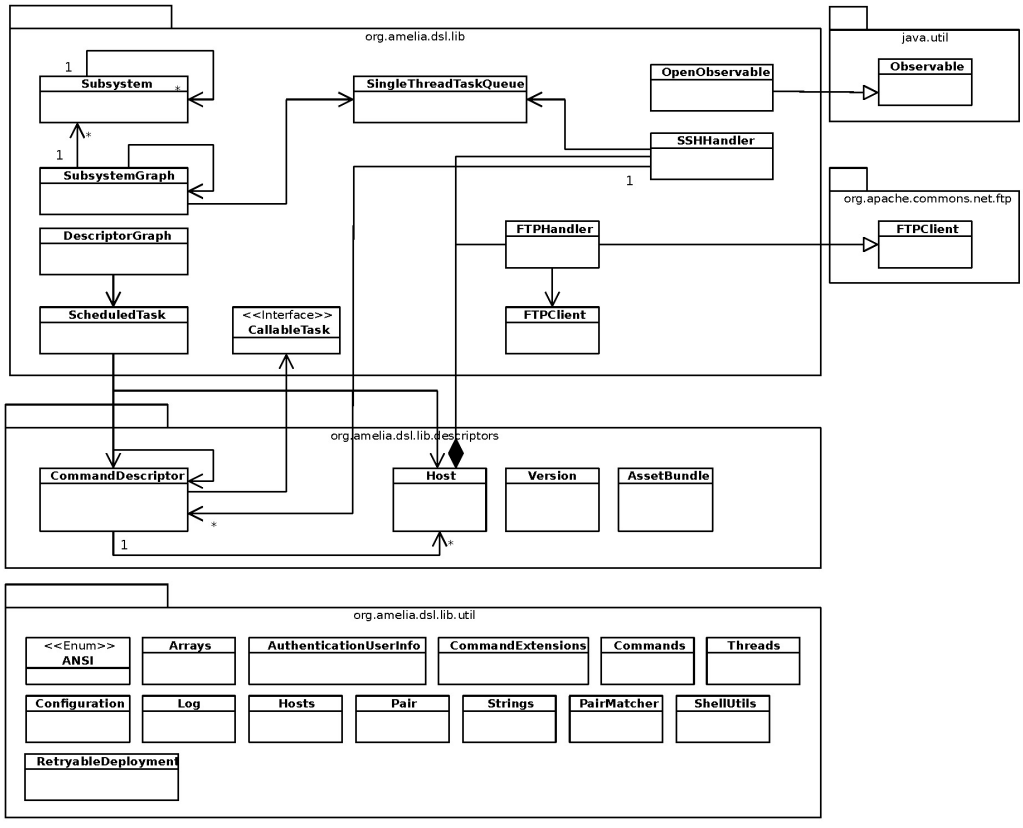

Figura 44. Amelia: diagrama de clases simplificado de la librería en tiempo de ejecución

El modelo de translación de Amelia

El compilador de Amelia traduce las especificaciones del subsistema y del despliegue en clases Java que son enteramente soportadas por la librería en tiempo de ejecución. No se generan otras clases, lo que implica que el código generado se ha reducido al mínimo, promoviendo la reusabilidad de los elementos de la librería. El detalle del mapeo entre elementos desde las especificaciones de Amelia y las clases Java se presenta a continuación.

Especificaciones del subsistema

En la Figura 45 se presenta una vista abstracta del mapeo entre elementos de la especificación de un subsistema y su correspondiente clase Java derivada. Desde un subsistema dado, el compilador genera una clase Java utilizando el paquete y nombre del subsistema. Esta clase importa las clases Java especificadas en la sección de importación del subsistema y las clases de la librería en tiempo de ejecución utilizadas en el resto del código generado. A pesar de que la aplicación Java resultante no haya sido ideada para ser analizada por un desarrollador humano, el compilador genera código legible e incluye la documentación disponible en la especificación del subsistema de Amelia.

La inclusión de un subsistema se traslada a un campo privado, cuyo tipo es de los incluidos en la clase derivada del subsistema. Este campo es utilizado para acceder a los parámetros incluidos y a las reglas de ejecución. La dependencia

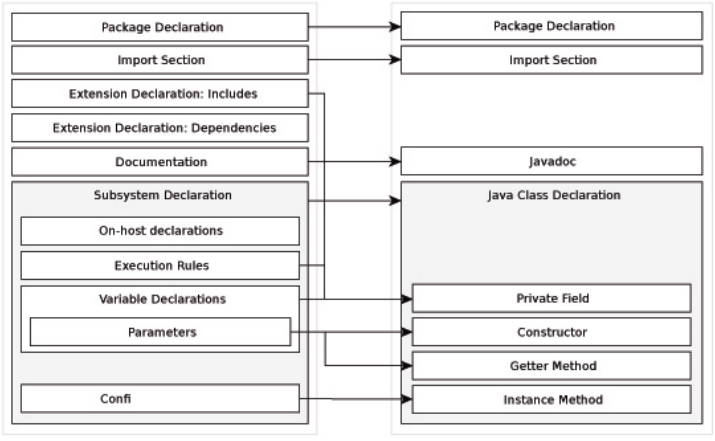

Figura 45. Mapeo entre la definición de subsistema (izquierda) y los elementos de clase Java generados (derecha)

de un subsistema —al contrario de la inclusión de un subsistema— no se utiliza en la clase derivada del subsistema; las dependencias se utilizan únicamente en clases derivadas de despliegues.

Una declaración *on host* no es traducida a un elemento Java específico, sino que se utiliza para configurar los comandos dentro de las reglas de ejecución. Esto es, por cada comando se tiene un *host* donde debería ejecutarse. Las reglas de ejecución se trasladan a campos de arreglos privados, cada campo se inicializa con un conjunto de instancias `CommandDescriptor`, representando cada comando con una regla.

Dentro de los subsistemas existen dos tipos de dependencias: comandos secuenciales y reglas de dependencias: el primer tipo de dependencias se representa en Java al configurar el comando n+1 como una dependencia del comando n, donde ambos comandos son elementos de la misma regla de ejecución; el segundo tipo de dependencias se soluciona al configurar el ultimo comando de cada regla de dependencia como una dependencia del primer comando de la regla dependiente, por ejemplo, en el código 13, el primer comando de la regla `execution` depende del último comando de la regla `compilation`.

La declaración de variables se traslada a campos privados. En caso de que existan parámetros del subsistema declarados o incluidos, se añade un constructor a la clase generada, incluyéndolos como parámetros (se crea un método *get* para cada parámetro). Los bloques de configuración se limitan a uno por subsistema, puesto que no hay razón de tener más. Si uno es declarado, se traslada a un método de instancia. Durante la ejecución, se ejecuta cuando todas las variables han sido inicializadas y todos los comandos han sido configurados.

Especificaciones de despliegue

La Figura 46 presenta una vista abstracta del mapeo entre elementos desde una especificación de despliegue y su correspondiente clase Java generada. Para subsistemas, el compilador genera una clase Java utilizando el paquete del despliegue, nombre y sección de importación. El subsistema es empleado para inicializar la instancia `Subsystem` utilizada por defecto en el constructor vacío del subsistema. También se utiliza para establecer dependencias de subsistemas cuando se configura la gráfica de ejecución (instancia `SubsystemGraph`). El cuerpo del subsistema se traslada a un método de instancia en su estado actual y se invoca después de la inicialización de todas las instancias del subsistema.

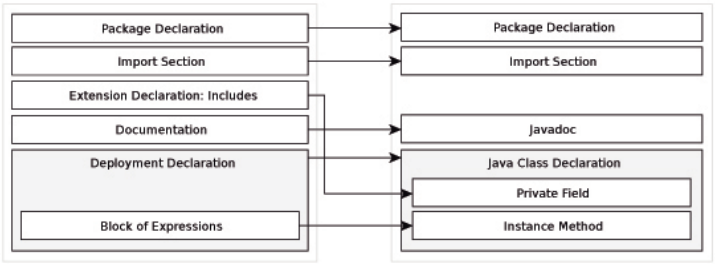

Figura 46. Mapeo entre despliegues y elementos de clases Java

EVALUACIÓN

La solución que se presentó en este documento consiste de dos elementos principales: el diseño de una arquitectura escalable para el monitoreo dinámico de desempeño y el diseño e implementación de Pascani y Amelia, dos DSL útiles para generar componentes de monitoreo acoplables, trazables y controlables, y desplegarlos en la infraestructura de un sistema objetivo. Con el fin de evaluar la efectividad de Pascani y Amelia se utilizó FQAD (*Framework for Qualitative Assessment of DSLs*) propuesto por Kahraman y Bilgen [56], el cual mejora un conjunto de características de calidad del estándar ISO/IEC 25010:2011, con el fin de utilizarlo en el análisis de DSL. Las características son:

- idoneidad funcional, es decir el grado en el que un DSL soporta el desarrollo de soluciones para satisfacer algunos requerimientos del dominio de la aplicación;
- usabilidad, que corresponde al grado en el que el DSL puede ser utilizado por ciertos usuarios para cumplir ciertas tareas;
- confiabilidad, la propiedad de ayudar a los usuarios a producir programas confiables;
- mantenibilidad, el grado en el que el lenguaje promueve la facilidad en el mantenimiento de programas;
- productividad, el grado en el que el lenguaje promueve la productividad en la programación;
- extensibilidad, el grado en el que el lenguaje provee mecanismos para que los usuarios puedan añadir características;
- compatibilidad, el grado en el que un DSL es compatible con el dominio y el proceso de despliegue;

- expresividad, el grado en que la solución de un problema (en el dominio) puede ser mapeada a un programa en el DSL naturalmente;
- reusabilidad, el grado en que las construcciones del lenguaje puedan utilizarse en otros lenguajes; e
- integralidad, la propiedad del lenguaje de ser integrado con otros lenguajes utilizados en el proceso de desarrollo.

La Figura 47 muestra los componentes en el modelo de evaluación FQAD, en donde un DSL exitoso corresponde a un conjunto de características interrelacionadas en él, que satisface colectivamente un requerimiento considerado relevante por él. En el esquema: sentencia meta describe el propósito de la evaluación; características del DSL a las características ya descritas, que corresponden a un conjunto de atributos únicos presentes en un DSL de alta calidad; y subcaracterísticas del DSL se utiliza para describir medidas de calidad relevantes para alcanzar la característica asociada.

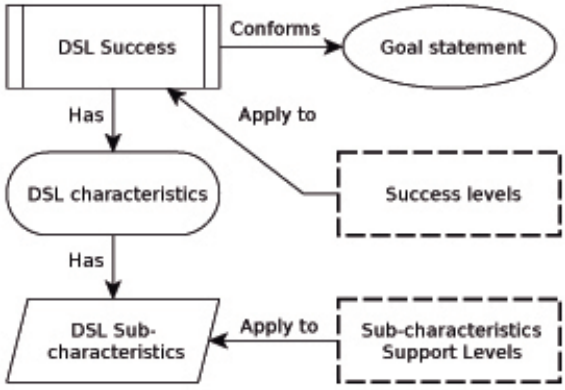

Figura 47. Componentes del modelo de evaluación FQAD [55]

El proceso de evaluación exitosa consta de tres etapas: en la primera, el evaluador asigna un ranking de importancia a cada característica de calidad seleccionada del DSL, de acuerdo con el alineamiento de las características con la meta de evaluación; en la segunda, con base en la retroalimentación del lenguaje provista por los usuarios que participan en el proceso de evaluación, el evaluador determina el nivel de soporte para cada característica; y en la tercera, los resultados de la evaluación se obtienen de acuerdo con las reglas definidas en el modelo FQAD.

Con el fin de establecer la efectividad de Pascani y Amelia, se diseñó un conjunto de ejercicios para utilizar cada lenguaje en un ambiente de desarrollo controlado y un cuestionario para evaluar la experiencia, ambos se aplicaron con un grupo de evaluadores en talleres de evaluación. La base de estos ejercicios fue el problema de multiplicación en cadena de matrices (MCM, *Matrix–Chain Multiplication*), un problema de optimización que consiste en encontrar la secuencia de multiplicación más eficiente para multiplicar un conjunto dado de matrices. Un detalle del proceso de evaluación se puede encontrar en [57], específicamente en la sección Evaluación y en los apéndices A y B. El resultado de la evaluación realizada permite afirmar que los dos DSL satisfacen su propósito.

CONCLUSIONES Y TRABAJO FUTURO

La entrega continua del servicio y el cumplimiento de los niveles acordados para el cumplimiento del desempeño del servicio requieren de información esclarecedora del estado actual del sistema, tanto de la infraestructura hardware como de los componentes software. Además, los avances en computación autónoma para fortalecer la respuesta y resiliencia del servicio han promovido el diseño de sistemas reconfigurables, capaces de modificar su estructura y comportamiento en tiempo de ejecución. Entonces, para asegurar la satisfacción continua de los factores de desempeño en estos sistemas, las infraestructuras de monitoreo deben ser capaces de: actualizar dinámicamente sus estrategias de monitoreo a medida que los requerimientos del sistema o el ambiente evoluciona; y de realizar el despliegue e integración de componentes de monitoreo en tiempo de ejecución. Conjuntamente, además de proveer al sistema de mecanismos de autoconciencia (mecanismos capaces de habilitar el sistema respecto a su propio comportamiento), dichas infraestructuras deben proveer los medios para generar capacidades de monitoreo acoplables, trazables y controlables.

Con el fin de proveer una solución para satisfacer estos retos se analizaron y convirtieron las necesidades citadas en requerimientos funcionales y consideraciones de calidad. Los requerimientos se clasificaron por componente, en monitores y sondas, y luego por etapa del proceso de monitoreo en Adquisición de Datos, Agregación de Datos y Filtrado, Persistencia de Datos y Visualización de Datos. Las restricciones de calidad identificadas se relacionan con el despliegue dinámico y el redespliegue de los elementos de la infraestructura, su acoplabilidad y controlabilidad y la escalabilidad de la infraestructura.

Este proyecto propuso una arquitectura de monitoreo dinámica basada en componentes dirigida a superar los retos identificados y cumplir con los requerimientos mencionados. Con el fin de abstraer detalles técnicos y de bajo nivel, se crearon Pascani y Amelia, dos DSL útiles para generar los componentes de monitoreo planeados y desplegarlos y redesplegarlos en la infraestructura en funcionamiento, respectivamente. Lo alcanzado en este proyecto aporta a los esfuerzos para avanzar en el desarrollo de mecanismos de autoconciencia (*self-awareness*) y a la retroalimentación de Dynamico.

LIMITACIONES TÉCNICAS

En el desarrollo de la solución presentada, se encontraron limitaciones técnicas que no permitieron proveer ciertas funcionalidades, como la medición de los tiempos de comunicación de red con Pascani y el enlace con servicios RMI en tiempo de ejecución con Amelia. Asimismo, se tomaron decisiones acerca del alcance de la implementación de la prueba de concepto que dejaron algunos requerimientos funcionales para desarrollos futuros, como es el caso de la recuperación del estado, después de un redespliegue en Pascani, y la configuración de ambientes de desarrollo, en Amelia. Estas limitaciones se describen brevemente a continuación.

Configuración de los ambientes de desarrollo

El lenguaje Amelia fue diseñado para desplegar sistemas distribuidos basados en componentes. El desplegar dichos sistemas incluye actividades como compilación de código fuente y configuración de ejecución y unión. Sin embargo, el despliegue también conlleva actividades relacionadas con la configuración de ambientes de ejecución –como la instalación de un sistema operativo, la configuración de propiedades hardware de las maquinas, la configuración de firewalls–. Los avances en la gestión de infraestructuras en la nube han promovido la virtualización de muchas de estas tareas, hoy existen muchas herramientas y lenguajes especializados en la creación y configuración de ambientes de desarrollo/producción.

Amelia no soporta directamente este tipo de tareas, sin embargo, como su tiempo de ejecución permite la comunicación con nodos remotos de computación utilizando sesiones SSH, puede con certeza ejecutar los comandos necesarios para disparar un conjunto de tareas de configuración del entorno.

Sondas de comunicación en red

Todas las operaciones de interceptación de tiempos de ejecución en la infraestructura de monitoreo dinámico se soportan en el middleware FraSCAti. Aunque éste fue diseñado para examinar aplicaciones SCA y medir o estimar el tiempo de comunicación de red requerido, no sólo para interceptar un componente, sino ambos lados de la comunicación en el contexto de una invocación sencilla del servicio. Para medir el tiempo que toma enviar datos por la red, se debe conocer el tiempo de invocación y el tiempo de finalización, el problema consiste en que dichos tiempos se miden en diferentes ubicaciones, en el consumidor del servicio y en el proveedor de componentes del servicio, pero aquí, las invocaciones no son identificables, no tienen un identificador o llave único que pueda ser adjunto. Este es un problema común cuando se miden tiempos de comunicación de red. Un enfoque utilizado por servidores RPC es la estrategia de colillas y esqueletos. Esta estrategia considera la generación estándar de dos componentes, una colilla y un esqueleto, para enviar un ID de transacción y el tiempo de inicio de la invocación junto con la petición original de la colilla al esqueleto; entonces, el esqueleto remueve la información adicional y reenvía la petición al proveedor de servicio. Dado el alcance de este proyecto, la versión actual de Pascani no cuenta con la implementación de una sonda de comunicación de red.

Amarre (binding) del servicio en tiempo de ejecución

Puesto que Amelia se implementó para desplegar componentes SCA generados por Pascani, FraSCAti es la plataforma middleware objetivo de Amelia. Esto limita a Amelia en el tipo de amarres que pueden realizarse en tiempo de ejecución para servicios web y REST, dejando por fuera amarres RMI.

Recuperación de estado en redespliegues

La actual implementación de Pascani no incluye un mecanismo para recuperar el estado después de realizar un redespliegue. Puesto que los elementos monitores están basados en eventos, esto no conlleva serias consecuencias en términos de perder el estado actual. Para los espacios de nombre esto es un problema relevante, puesto que los monitores continuarían trabajando con los valores por defecto de las variables de contexto en vez de los que se reflejan en el estado del sistema. Recuperar el estado de un espacio de nombre después de

un redespliegue requiere inicializar sus variables desde los valores almacenados en la base de datos, un proceso complejo puesto que debe existir un mecanismo de mapeo estándar para reunir y separar cualquier tipo de datos. Implementar tal mecanismo vendría a solucionar muchos de los problemas de las tecnologías de mapeo objeto–relacional (ORM, *Object–Relational Mapping*). Actualmente, Pascani soporta únicamente el almacenamiento de tipos de dato primitivos (el proceso de unir) pero con poco esfuerzo podría implementarse el separar los tipos de datos soportados.

TRABAJO FUTURO

Evolución de Pascani y Amelia

De acuerdo con la evaluación presentada, tanto Pascani como Amelia proveen un nivel adecuado de abstracción de dominio y mejoran la productividad del desarrollo. Sin embargo, se requiere realizar más pruebas respecto del rendimiento del monitoreo y despliegue del sistema en diferentes tipos de aplicaciones software. Aunque se diseñaron ambos lenguajes con la preocupación de incluir tantos conceptos como fuera sea posible de cada dominio del lenguaje, pueden surgir distintos tipos de requerimientos. Por consiguiente, se requiere adaptar y evolucionar la sintaxis y semántica de cada lenguaje.

Desarrollo de mecanismos con autoconciencia

La arquitectura propuesta en este proyecto permite la especificación manual de especificaciones de monitoreo y la generación automática y despliegue de componentes de monitoreo. Con el fin de progresar en el estado del arte y alcanzar mecanismos de autoconsciencia siempre relevantes, la solución que se ha presentado debe crecer, agregando capacidades autónomas para continuamente evolucionar la infraestructura de monitoreo. Una manera de lograrlo es implementar Dynamico con base en la arquitectura de monitoreo dinámico que se propone en el presente documento.

Hacia el soporte de la generación automática de las especificaciones de Amelia y Pascani

Uno de los objetivos en desarrollar mecanismos de monitoreo dinámico consiste en proveer una línea base para gradualmente habilitar al sistema

para generar los componentes de monitoreo que permiten a la infraestructura permanecer pertinente. Sin embargo, la sintaxis y semántica de Pascani todavía está en un bajo nivel de abstracción, por lo que es difícil para el sistema ensamblar nuevas especificaciones con tal de cumplir requerimientos de monitoreo emergentes. Una solución adecuada a este problema es abstraer las especificaciones de Pascani a políticas de alto nivel, pues dichas políticas reducirían considerablemente las especificaciones de monitoreo, mientras esconden detalles técnicos irrelevantes para el sistema. Por ejemplo, para observar la latencia en un servicio dado, el sistema tendría que declarar un evento con su correspondiente servicio objetivo, crear un manejador de evento con la lógica para actualizar la variable de latencia y un bloque de configuración para subscribir el manejador al evento de ejecución; adicionalmente, el sistema debería declarar el paquete del monitor y nombre e importar las clases Java requeridas. Esto se vuelve complejo cuando existen diversas variables de contexto y servicios involucrados en el requerimiento de monitoreo, complejidad que aumenta cuando los requerimientos de monitoreo son dependientes entre ellos. Dichos casos requerirían diseñar una estrategia de derivación basada en plantillas acoplables y fragmentos. Sin embargo, el abstraer dichos requerimientos de monitoreo en políticas con enfoque claro haría más fácil derivar especificaciones de Pascani y representar estrategias/requerimientos de monitoreo en fuentes de conocimiento (elemento Knowledge del modelo de referencia MAPE–K).

Dichas consideraciones también aplican al lenguaje Amelia. El desplegar un componente requiere la declaración de expresiones *on–host*, reglas de ejecución y comandos. En este caso, las políticas podrían ayudar a reducir la cantidad de código requerido para expresar un requerimiento; sin embargo, esto podría no ser suficiente. Se espera que el despliegue de un componente puntual se de en el directorio del código fuente y provea el nombre del artefacto, las dependencias, las librerías y los comandos para compilar —y posiblemente configurar— y ejecutarlo. Se considera que el principio de configuración sobre convención es un buen complemento para impulsar el sistema para entender intrincadas estrategias de despliegue de manera simple.

REFERENCIAS

[1] O. González, "Monitoring and analysis of workflow applications: A domain–specific language approach," Ph.D. thesis, Universidad de los Andes, Bogotá, Colombia, 2010.

[2] N. M. Villegas, "Context management and self–adaptivity for situation–aware smart software systems," Ph.D. thesis, University of Victoria, BC, Canada, 2013.

[3] R. B Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering experiment in software component generation," In *IEEE 18th International Conference on Software Engineering, Proceedings of the*, Berlin, Germany, 1996, pp. 542-552.

[4] IBM, "An architectural blueprint for autonomic computing," [white paper], 2006.

[5] *Twitter / Outages* [Online], available: https://en.wikipedia.org/wiki/Twitter#Outages

[6] M. Honan (2013, Nov. 25), Killing the fail whale with twitter's Christopher Fry [Online], *Wired*, available: https://www.wired.com/2013/11/qa-with-chris-fry/

[7] T. Xu, Y. Chen, L. Jiao, B. Y. Zhao, P. Hui, and X. Fu, "Scaling microblogging services with divergent traffic demands," In *Middleware '11: Proceedings of the 12th International Middleware Conference*, Lisbon, Portugal, 2011, pp. 20-39

[8] C. Jones (2012, July, 26), "Twitter down but not out as Olympics test looms," *The Guardian* [Online], available: https://www.theguardian.com/technology/2012/jul/26/twitter-down-olympics

[9] J. O. Kephart, and D. M Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41-50, 2003.

[10] N. M. Villegas, G. Tamura, H. Müller, L. Duchien, and R. Casallas, "Dynamico: A reference model for governing control objectives and context relevance in self–adaptive software systems," In *LNCS*, vol. 7475, *Software Engineering for Self–Adaptive Systems 2*, Berlin–Heidelberg, Germany, Springer, 2013, pp. 265-293.

[11] G. Tamura, N. M. Villegas, H. Müller, J. Sousa, B. Becker, M. Pezzè, G. Karsai, S. Mankovskii, W. Schäfer, L. Tahvildari, and K. Wong, "Towards practical runtime verification and validation of self–adaptive software systems," In *LNCS*, vol. 7475, *Software Engineering for Self–Adaptive Systems 2*, Berlin–Heidelberg, Germany, Springer, 2013, pp. 108-132.

[12] P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, et al., "Ultra–large–scale systems: The software challenge of the future", , Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[13] E. W. Dijkstra, *A discipline of programming*, Englewood Cliffs, NJ: Prentice–Hall, 1976.

[14] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. Smith, J. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software engineering for self–adaptive systems: A second research roadmap," In *LNCS*, vol. 7475,

Software Engineering for Self–Adaptive Systems 2, Berlin–Heidelberg, Germany, Springer, 2013, pp. 1-32.

[15] F. Bachman, L. Bass, C. Buhman, S. Comella–Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Technical concepts of component–based software engineering" [DTIC ADA 379930], Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[16] M. Beisiegel, H. Blohm, D. Booz, J. J. Dubray, A. Colyer, Inter–face21, M. Edwards, D. Ferguson, J. Mischkinsky, M. Nally, and G. Pavlik, "Service component architecture: Building systems using a service oriented architecture," [white paper, v. 9], 2007.

[17] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J. B. Stefani, "A component–based middleware platform for reconfigurable service–oriented architectures. software," *Practice and Experience*, vol. 42, no. 5, pp. 559-583, May 2012.

[18] D. Chappell (2007, July), *Introducing SCA* [Online], avalaible: http://www.davidchappell.com/writing/Introducing_SCA.pdf

[19] P. Shepherd (2009, August), *Oracle SCA: The power of the composite* [Online], Avalaible: https://www.oracle.com/technetwork/topics/entarch/whatsnew/oracle-sca-the-power-of-the-composi-134500.pdf

[20] Apache Foundation (2015, Sep. 11), *Apache TuSCAny* [Online], available: http://tuscany.apache.org

[21] Metaform Systems (2015, Sep. 14), *Fabric3* [Online], Avaiblable: http://www.fabric3.org

[22] IBM (2015), *IBM WebSphere application server feature pack for SCA* [Online], avaiable: http://www-03.ibm.com/software/products/en/sca

[23] Oracle Corp. *Oracle Tuxedo* [Online], available: https://www.oracle.com/middleware/technologies/tuxedo.html

[24] N. Bencomo, R. B. France, B. Cheng, and U. Aßmann, Eds., *LNCS, vol. 8378, Models@ run.time: Foundations, applications, and roadmaps* [Dagstuhl Seminar 11481, November 27 – December 2, 2011], Berlin–Heildelberg, Germany, Springer, 2014.

[25] M. Fowler, *Domain–specific languages*, Upper Saddle River, NJ, Pearson Education, 2010

[26] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain–specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316-344, 2005.

[27] A. van Deursen and P. Klint, "Little languages: Little maintenance," *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75-92, March 1998.

[28] D. Spinellis and V. Guruprasad, "Lightweight languages as software engineering tools," In *Proceedings of the Conference on Domain–Specific Languages, DSL'97*, Berkeley, CA, 1997, p. 6.

[29] D. Luckham, *The power of events*, Boston, MA: Addison–Wesley, 2002.

[30] Q. Zhu, "Adaptive root cause analysis and diagnosis," Ph.D. thesis, University of Victoria, BC, Canada, 2010.

[31] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, et al., "Ultra–large–scale systems: The software challenge of the future" [DTIC ADA610356], Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[32] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock, "Quality attributes" [Technical Report CMU/SEI–95–TR–021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1995.

[33] P. Nikolaj, D. Bukh, and R. Jain, *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*, Hoboken, NJ: Wiley, 1992.

[34] A. Dearle, "Software deployment, past, present and future," In *2007 Future of Software Engineering*, Washington, DC, IEEE Computer Society, 2007, pp. 269–284.

[35] R. S Hall, D. Heimbigner, and A. L Wolf, "A cooperative approach to support software deployment using the software dock," In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, ACM, 1999, pp. 174-183.

[36] J. Dubus, "Une démarche orientée modele pour le déploiement de systemes en environnements ouverts distribués," Ph.D. thesis, Université des Sciences et Technologie de Lille, France, 2008.

[37] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, Los Angeles, CA, Sage, 2013.

[38] H. Arboleda, A. Paz, M. Jiménez, and G. Tamura. "A framework for the generation and management of self–adaptive enterprise applications," In *IEEE Colombian Computing Congress (10CCC)*, Bogotá, Colombia, 2015, pp. 1-10.

[39] H. Arboleda, A. Paz, M. Jiménez, and G. Tamura, "Development and instrumentation of a framework for the generation and management of self–adaptive enterprise applications," *Ingenieria y Universidad*, vol. 21, no. 1, pp. 303-316, 2016.

[40] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. Information technology standard*, ISO/IEC 25000:2014.

[41] M. Jimenez, Á. Villota, N. M. Villegas, G. Tamura, L. Duchien, et al., "A framework for automated and composable testing of component–based services," In *Maintenance and Evolution of Service–Oriented and Cloud–Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the*, Victoria, BC, Canada. 2014, pp. 1-10.

[42] P. G. Neumann, "Principled assuredly trustworthy composable architectures" [final report], SRI International, Menlo Park, CA, 2004.

[43] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, "Xbase: Implementing domain–specific languages for java," *ACM SIGPLAN Notices*, vol. 48, no. 3, pp.112-121, Sept., 2012.

[44] D. Garlan and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. 1, no. 3-4), 1993.

[45] P. C. David, T. Ledoux, M. Léger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures," *Annals of Telecommunications*, vol. 64, no. 1-2, pp. 45-63, 2009.

[46] *The Apache Ant project* [Online], available: http://ant.apache.org/

[47] *Apache Maven project* [Online], available: https://maven.apache.org/

[48] *GNU Make* [Online], available: https://www.gnu.org/software/make

[49] *Infuxdata* [Online], available: https://influxdata.com/

[50] *Elastic* [Online], available: https://www.elastic.co/

[51] *RethnkDB* [Online], available: https://www.rethinkdb.com/

[52] *Clip* [Online], available: http://fnordmetric.io/

[53] *Grafana* [Online], available: http://grafana.org

[54] *Kibana* [Online], available: https://www.elastic.co/kibana

[55] *Rabbit MQ* [Online], available: https://www.rabbitmq.com

[56] G. Kahraman and S. Bilgen, "A framework for qualitative assessment of domain–specific languages," *Software & Systems Modeling*, vol. 14, no. 4, pp. 1505-1526, 2015.

[57] M. Jimenez, "A framework for generating and deploying dynamic performance monitors for self–adaptive software systems," MSc. thesis, Universidad Icesi, Cali, Colombia, 2016.

ANEXO 1. DEFINICIÓN GRAMATICAL DE PASCANI

```
1  grammar org.pascani.dsl.Pascani with org.eclipse.xtext.xbase.Xbase
2
3  import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
4  import "http://www.eclipse.org/xtext/xbase/Xbase"
5
6  generate pascani "http://www.pascani.org/dsl/Pascani"
7
8  Model
9        : ('package' name = QualifiedName -> ';' ?)?
10         imports = XImportSection?
11         typeDeclaration = TypeDeclaration?
12       ;
13
14 TypeDeclaration
15       : MonitorDeclaration
16       | NamespaceDeclaration
17       ;
18
19 MonitorDeclaration returns Monitor
20       : extensions = ExtensionSection?
21         'monitor' name = ValidID
22         body = MonitorBlockExpression
23       ;
24
25 ExtensionSection
26       : declarations += ExtensionDeclaration+
27       ;
28
29 ExtensionDeclaration
30       : ImportEventDeclaration
31       | ImportNamespaceDeclaration
32       ;
33
34 ImportEventDeclaration
35       : 'from' monitor = [Monitor | QualifiedName]
36         'import' events += [Event | ID] (',' events += [Event | ID])* -> ';'?
37       ;
38
39 ImportNamespaceDeclaration
40       : 'using' namespace = [Namespace | QualifiedName] -> ';' ?
41       ;
42
43 MonitorBlockExpression returns XBlockExpression
44       : {MonitorBlockExpression} '{' (expressions += InternalMonitorDeclaration)* '}'
```

```
45        ;
46
47  InternalMonitorDeclaration returns XExpression
48        : VariableDeclaration -> ';'?
49        | ConfigBlockExpression
50        | EventDeclaration
51        | HandlerDeclaration
52        ;
53
54  NamespaceDeclaration returns Namespace
55        : 'namespace' name = ValidID body = NamespaceBlockExpression
56        ;
57
58  NamespaceBlockExpression returns XBlockExpression
59        : {NamespaceBlockExpression} '{' (expressions += InternalNamespaceDeclaration)* '}'
60        ;
61
62  InternalNamespaceDeclaration returns XExpression
63        : VariableDeclaration -> ';'?
64        | NamespaceDeclaration
65        ;
66
67  VariableDeclaration returns XExpression
68        : {VariableDeclaration}
69        (writeable ?= 'var' | 'val')
70        (=> (type = JvmTypeReference name = ValidID) | name = ValidID) ('=' right =
              XExpression)?
71        ;
72
73  ConfigBlockExpression returns XBlockExpression
74        : {ConfigBlockExpression} 'config' '{' (expressions += XExpressionOrVarDeclaration ';'?)* '}'
75        ;
76
77  HandlerDeclaration returns Handler
78        : 'handler' name = ValidID
79        '(' params += FullJvmFormalParameter (',' params += FullJvmFormalParameter)* ')'
80        body = XBlockExpression
81        ;
82
83  EventDeclaration returns Event
84        : 'event' name = ValidID 'raised' (periodical ?= 'periodically')? 'on' emitter = EventEmitter -> ';'?
85        ;
86
87  EventEmitter
88        : eventType = EventType 'of' emitter = XExpression (=> specifier = AndEventSpecifier)?
89        | cronExpression = XExpression
```

```
90          ;
91
92      enum EventType
93          : invoke
94          | return
95          | change
96          | exception
97          ;
98
99      AndEventSpecifier returns EventSpecifier
100         : OrEventSpecifier
101         (
102         {AndEventSpecifier.left = current}
103           operator='and' right = OrEventSpecifier
104         )*
105         ;
106
107     OrEventSpecifier returns EventSpecifier
108         : SimpleEventSpecifier
109               (
110                {OrEventSpecifier.left = current}
111                operator='or' right = SimpleEventSpecifier
112               )*
113         ;
114
115     SimpleEventSpecifier returns EventSpecifier
116         : (below ?= 'below' | above ?= 'above' | equal ?= 'equal' 'to')
117               value = XExpression (percentage ?= '%%')?
118         | '(' AndEventSpecifier ')'
119         ;
120
121     CronExpression
122         : lsymbol = ' '
123                   seconds = CronElement
124                   minutes = CronElement
125                   hours = CronElement
126                   dayOfMonth = CronElement
127                   month  = CronElement
128                   dayOfWeek = CronElement
129                   (year = CronElement)?
130             rsymbol = ' '
131         ;
132
133     CronElement
134         : CronElementList | IncrementCronElement | NthCronElement
135         ;
```

```
136
137  /*
138    * Options L and W of the Quartz scheduler are only supported
139    * in cases were they are found alone (by means of rule ValidID).
140    */
141  CronElementList
142        : elements += RangeCronElement (',' elements += RangeCronElement)*
143        ;
144
145  IncrementCronElement
146        : start = TerminalCronElement ('-' end = TerminalCronElement)? '/' increment =
              TerminalCronElement
147        ;
148
149  RangeCronElement
150        : TerminalCronElement ({RangeCronElement.start = current} '-' end = TerminalCronElement)?
151        ;
152
153  NthCronElement
164        : element = TerminalCronElement '#' nth = TerminalCronElement
155        ;
156
157  TerminalCronElement
158        : expression = (IntLiteral | ValidID | '*' | '?')
159        ;
160
161  IntLiteral
162        : INT
163        ;
164
165  XLiteral returns XExpression
166        : XCollectionLiteral
167        | XClosure
168        | XBooleanLiteral
169        | XNumberLiteral
170        | XNullLiteral
171        | XStringLiteral
172        | XTypeLiteral
173        | CronExpression
174        ;
```

ANEXO 2. DEFINICIÓN GRAMATICAL DE AMELIA

```
1  grammar org.amelia.dsl.Amelia with org.eclipse.xtext.xbase.Xbase
2
3  import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
4  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5
6  generate amelia "http://www.amelia.org/dsl/Amelia"
7
8  Model
9        : 'package' name = QualifiedName -> ';' ?
10         import Section = XImportSection?
11         typeDeclaration = TypeDeclaration?
12       ;
13
14 TypeDeclaration
15       : SubsystemDeclaration
16       | DeploymentDeclaration
17       ;
18
19 DeploymentDeclaration
20       : extensions = ExtensionSection?
21         'deployment' name = ID body = XBlockExpression
22       ;
23
24 SubsystemDeclaration returns Subsystem
25       : extensions = ExtensionSection?
26         'subsystem' name = ID body = SubsystemBlockExpression
27       ;
28
29 ExtensionSection
30       : declarations += ExtensionDeclaration+
31       ;
32
33 ExtensionDeclaration
34       : DependDeclaration
35       | IncludeDeclaration
36       ;
37
38 IncludeDeclaration
39       : 'includes' element = [TypeDeclaration | QualifiedName] -> ';' ?
40       ;
41
42 DependDeclaration
43       : 'depends' 'on' element = [TypeDeclaration | QualifiedName] -> ';' ?
```

```
44          ;
45  45
46  SubsystemBlockExpression
47          : {SubsystemBlockExpression} '{' (expressions += InternalSubsystemDeclaration)* '}'
48          ;
49
50  InternalSubsystemDeclaration returns xbase::XExpression
51          : VariableDeclaration -> ';' ?
52          | OnHostBlockExpression
53          | ConfigBlockExpression
54          ;
55
56  VariableDeclaration
57          : {VariableDeclaration}
58            (writeable?= 'var'|'val'|param?= 'param')
59            (=> (type = JvmTypeReference name = ValidID) | name = ValidID) ('=' right = XExpression)?
60          ;
61
62  ConfigBlockExpression returns xbase::XBlockExpression
63          : {ConfigBlockExpression} 'config'  '{' (expressions += XExpressionOrVarDeclaration ';' ?)* '}'
64          ;
65
66  OnHostBlockExpression
67          : 'on' hosts = XExpression '{' (rules += RuleDeclaration)* '}'
68          ;
69
70  RuleDeclaration
71          : name = ID ':'
72            (=> (dependencies += [RuleDeclaration | QualifiedName] (',' dependencies +=
                  [RuleDeclaration | QualifiedName])*)? ';')?
73            (commands += XExpression)*
74          ;
75
76  CdCommand
77          : 'cd' directory = XExpression (=> initializedLater ?= '...')?
78          ;
79
80  CompileCommand
81          : 'compile' source = XExpression output = XExpression
82            (=> '-classpath' classpath = XExpression)?
83            (=> initializedLater ?= '...')?
84          ;
85
86  RunCommand
87          :
88             'run' (hasPort ?= '-r' port = XExpression)?
```

```
89          composite = XExpression '-libpath' libpath = XExpression
90          (=>
91              hasService ?= ('-s' | '--service-name') service = XExpression
92              hasMethod ?= ('-m' | '--method-name') method = XExpression
93              (=> hasParams ?= '-p' params = XExpression)?
94          )?
95          (=> initializedLater ?= '...')?
96      ;
97
98  TransferCommand
99      : 'scp' source = XExpression 'to' destination = XExpression
100     ;
101
102 EvalCommand
103     : (=> 'on' uri = XExpression)? 'eval' script = XExpression
104     ;
105
106 CustomCommand
107     : 'cmd' value = XExpression (=> initializedLater ?= '...')?
108     ;
109
110 CommandLiteral
111     : CdCommand
112     | CompileCommand
113     | CustomCommand
114     | EvalCommand
115     | RunCommand
116     | TransferCommand
117     ;
118
119 RichString
120     :
121       {RichString} (expressions += RichStringLiteral)
122     | (
123           expressions += RichStringLiteralStart
124             (expressions += XExpression (expressions += RichStringLiteralMiddle
                    expressions += XExpression)*)
125           expressions += RichStringLiteralEnd
126       )
127     ;
128
129 RichStringLiteral
130     : {RichStringLiteral} value = RICH_TEXT
131     ;
132
133 RichStringLiteralStart
```

```
134         : {RichStringLiteral} value = RICH_TEXT_START
135         ;
136
137   RichStringLiteralMiddle
138         : {RichStringLiteral} value = RICH_TEXT_MIDDLE
139         ;
140
141   RichStringLiteralEnd
142         : {RichStringLiteral} value = RICH_TEXT_END
143         ;
144
145   XLiteral returns xbase::XExpression
146         : XCollectionLiteral
147         | XClosure
148         | XBooleanLiteral
149         | XNumberLiteral
150         | XNullLiteral
151         | XTypeLiteral
152         | XStringLiteral
153         | CommandLiteral
164         | RichString
155         ;
156
157   terminal RICH_TEXT
158         : """ ('\\' . | !('\\' | """ | '<' | '>') )* """
159         ;
160
161   terminal RICH_TEXT_START
162         : """ ('\\' . | !('\\' | """ | '<') )* '<'
163         ;
164
165   terminal RICH_TEXT_MIDDLE
166         : '>' ('\\' . | !('\\' | """ | '<') )* '<'
167         ;
168
169   terminal RICH_TEXT_END
170         : '>' ('\\' . | !('\\' | """ | '<') )* """
171         ;
172
173   terminal STRING
174         : '"' ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ | !('\\'|'"') )* '"'?
175         ;
```