

Monitores dinámicos de software
Despliegue de software
Monitoreo de espectro

Miguel Jiménez

Gabriel Tamura Morimitsu

Luis Felipe Rivera

Norha Milena Villegas

Leonardo Vargas Bernal

Norha Milena Villegas

BITÁCORAS DE LA MAESTRÍA

**MONITORES DINÁMICOS DE SOFTWARE
DESPLIEGUE DE SOFTWARE
MONITOREO DE ESPECTRO**

BITÁCORAS DE LA MAESTRÍA

**MONITORES DINÁMICOS DE SOFTWARE
DESPLIEGUE DE SOFTWARE
MONITOREO DE ESPECTRO**

*Miguel Jiménez
Luis Felipe Rivera
Leonardo Vargas Bernal
Gabriel Tamura Morimitsu
Norha Milena Villegas
Andrés Navarro Cadavid*

Editorial Universidad Icesi, 2020

Monitores dinámicos de software - Despliegue de software - Monitoreo de espectro

© Miguel Jiménez, Luis F. Rivera, Leonardo Vargas, Gabriel Tamura, Norha M. Villegas, Andrés Navarro.

1 ed. Cali, Colombia. Universidad Icesi, 2020

210 p., 19x24 cm

Incluye referencias bibliográficas

ISBN: 978-958-5590-11-3 (PDF)

<https://doi.org/10.18046/EUI/bm.3.2020>

1. Software performance 2. Software engineering 3. Radio spectrum I.Tit
629 – dc22

© Universidad Icesi, 2020

Facultad de Ingeniería

Colección: Bitácoras de la maestría, vol. 3.

Rector: Francisco Piedrahita Plata

Decano Facultad de Ingeniería: Gonzalo Ulloa Villegas

Coordinador editorial: Adolfo A. Abadía



Producción y diseño: Claros Editores SAS.

Editor: José Ignacio Claros V.

Traducción: Oscar Arley Orozco / José Ignacio Claros V.

Impresión: Carvajal Soluciones de Comunicación.

Impreso en Colombia / *Printed in Colombia.*

La publicación de este libro se aprobó luego de superar un proceso de evaluación doble ciego por dos pares expertos. El contenido de esta obra no compromete el pensamiento institucional de la Universidad Icesi ni le genera responsabilidades legales, civiles, penales o de cualquier otra índole, frente a terceros.



Calle 18 #122-135 (Pance), Cali-Colombia
editorial@icesi.edu.co
www.icesi.edu.co/editorial
Teléfono: +57(2) 555 2334

La serie Bitácoras de la Maestría es una publicación de la Universidad Icesi que tiene como objetivo mejorar la difusión de los trabajos de grado meritorios de sus estudiantes, exponiéndolos a un público más amplio, no necesariamente académico, que pueda aprovecharlos en su cotidianidad. Se trata de “mover” las tesis, desde los anaqueles de las bibliotecas, hacia las manos de los actores de la vida diaria y establecer un vínculo entre autores y potenciales usuarios. En cada volumen se incluyen tres trabajos con temática diversa. Por lo heterogéneo de su contenido, el nombre de cada volumen está compuesto por el nombre de la serie y el de los tres temas que incluye.

Miguel Jiménez

Estudiante de doctorado (University of Victoria, Canadá) e Ingeniero de Sistemas y Máster en Informática y Telecomunicaciones (Universidad Icesi, Colombia), con experiencia en desarrollo para la Web y computación en la nube. Su principal área de interés en investigación es la ingeniería del software con énfasis en el diseño de arquitecturas e infraestructuras para sistemas autoadaptativos. Su actual proyecto de investigación se centra en la evolución automatizada del software en la nube a través de la experimentación continua basada en la calidad. miguel@uvic.ca

Luis Felipe Rivera

Estudiante de doctorado en Ciencias de la Computación en la University of Victoria, British Columbia, Canadá. Cuenta con una maestría en Informática y Telecomunicaciones y un título de Ingeniero de Sistemas, ambos otorgados por la Universidad Icesi. Sus áreas de interés incluyen: Digital Twins, modelos en tiempo de ejecución, Model-Driven Engineering (MDE), Model-Driven Architecture (MDA) y Software Engineering at Run-Time. rivera@uvic.ca

Leonardo Vargas Bernal

Ingeniero Telemático (2015) con Maestría en Informática y Telecomunicaciones (2009) de la Universidad Icesi (Cali, Colombia). Es docente de la Facultad de Ingeniería y miembro del Grupo de Investigación en Informática y Telecomunicaciones (i2t) de la Universidad Icesi. Sus áreas de interés profesional incluyen: comunicaciones móviles, gestión de espectro radioeléctrico, computación urbana y computación móvil. lvbernal@gmail.com

Gabriel Tamura Morimitsu

Doctor en Informática (Université Lille 1 Sciences et Technologies, Francia); Doctor en Ingeniería y Máster en Ingeniería de Sistemas y Computación (Universidad de Los Andes, Colombia); e Ingeniero de Sistemas y Computación (Universidad Javeriana, Colombia). Es profesor de tiempo completo de la Facultad de Ingeniería en la Universidad Icesi. Sus áreas de interés en investigación son: Context-Aware Self-Adaptive Software; gestión de tecnología e infraestructura informática; Model-Driven Software Development y Variability Modeling and Software Product Lines. Es IEEE Senior Member, Investigador Senior (Colciencias). gtamura@icesi.edu.co

Norha Milena Villegas

Doctora en Ciencias de la Computación (University of Victoria, Canadá) e Ingeniera de Sistemas y Especialista en Gerencia Informática Organizacional (Universidad Icesi, Cali-Colombia). Es profesora de tiempo completo y Directora del Programa de Ingeniería de Sistemas en Icesi. Sus principales áreas de interés en investigación son: ingeniería de software, sistemas de software autoadaptativos, gestión dinámica de contextos, sistemas físicos cibernéticos y analítica. Es IEEE Senior Member e Investigadora Senior (Colciencias). nvillega@icesi.edu.co

Andrés Navarro Cadavid

Doctor Ingeniero en Telecomunicaciones de la Universidad Politécnica de Valencia (España), Máster en Gestión Tecnológica e Ingeniero Electrónico de la Universidad Pontificia Bolivariana (Medellín, Colombia). Es profesor de tiempo completo y Director del Grupo de Investigación en Informática y Telecomunicaciones (i2t) de la Universidad Icesi (Cali, Colombia). Es: investigador senior (Colciencias), miembro senior del IEEE, presidente del capítulo Comunicaciones del IEEE Colombia, consultor internacional y miembro de Grupo de Estudio 1 de la Unión Internacional de Telecomunicaciones. anavarro@icesi.edu.co

Tabla de contenido

Presentación	23
Framework para generación y despliegue de monitores dinámicos de rendimiento en sistemas software autoadaptativos	25
Resumen	27
Introducción	28
Marco teórico y estado del arte	32
Ingeniería de software basada en componentes	32
Sistemas software autogestionables	34
Lenguajes específicos de dominio	35
Monitoreo del sistema	36
Despliegue de software	39
Metodología	41
Dominio del problema: monitoreo dinámico del rendimiento	42
Mape-k: monitoreando requerimientos	42
Requerimientos de la infraestructura dinámica de monitoreo	43
Dominio de la solución: un enfoque de DSL para el monitoreo y despliegue dinámicos	48
Diseño arquitectural global	48
Abordando restricciones de calidad	51

Diseño de la infraestructura de monitoreo dinámico	54
Pascani: un DSL para el monitoreo de desempeño dinámico	56
Amelia: un DSL para el despliegue dinámico de software	67
Implementación	76
Pascani	77
Amelia	84
Evaluación	90
Conclusiones y trabajo futuro	92
Limitaciones técnicas	93
Trabajo futuro	95
Referencias	96
Anexo 1. Definición gramatical de Pascani	101
Anexo 2. Definición gramatical de Amelia	105
Despliegue de software automatizado guiado por UML	109
Resumen	111
Introducción	112
Marco teórico	116
Diagramas de despliegue UML	117
Entrega continua	121
Ingeniería y arquitectura guiadas por modelos (MDE / MDA)	122
El DSL Amelia	124
Estado del arte	125
Automatización del despliegue de software	125
Despliegue de software en configuraciones de entrega continua y DevOps	128
Un enfoque guiado por el modelo para automatizar el despliegue de software	130
Usando especificaciones de despliegue basado en el modelo	130

Especificando despliegue de software en UML	131
El modelo de ejecución de Urano	133
Implementación	134
Definición y aplicación del perfil	136
Interpretación de diagramas de despliegue UML extendidos	137
Transformaciones M2M	142
Transformaciones M2T	145
Artefactos generados	148
Evaluación	149
Estudios de caso de despliegue	150
Aplicación y discusión de los criterios de evaluación	156
Conclusiones y trabajo futuro	160
Limitaciones técnicas	161
Trabajo futuro	162
Referencias	164
Sistema de benchmarking de redes móviles celulares y WiFi basado en dispositivos de usuario final y SDR	169
Resumen	171
Introducción	172
Marco teórico	175
OFDM	175
802.11	179
LTE	181
Indicadores de rendimiento	183
Drive test	184
Análisis de muestras	185

Minimización del drive test	186
WiFi offloading	187
Aplicaciones relacionadas	188
Aplicación móvil	189
Arquitectura	189
Implementación en android	191
Estricciones	193
Aplicación SDR	193
Herramientas	193
Aplicaciones	194
Backend	201
Conclusiones y trabajo futuro	204
Referencias	206

Índice de Tablas

FRAMEWORK PARA GENERACIÓN Y DESPLIEGUE DE MONITORES DE RENDIMIENTO DINÁMICO EN SISTEMAS SOFTWARE AUTOADAPTATIVOS

Tabla 1. Requerimientos funcionales	44
Tabla 2. Requerimientos funcionales del elemento monitor extendidos ...	45
Tabla 3. Consideraciones de calidad	47
Tabla 4. Proyectos Java que conforman la implementación de Pascani	77
Tabla 5. Pascani: paquetes de clases de la librería en tiempo de ejecución	79
Tabla 6. Pascani: paquetes de clases de la librería SCA	81
Tabla 7. Proyectos que conforman la implementación de Amelia	85
Tabla 8. Amelia: paquetes de clases de la librería en tiempo de ejecución	87

DESPLIEGUE DE SOFTWARE AUTOMATIZADO GUIADO POR UML

Tabla 1. Requerimientos de diseño e implementación previstos	129
Tabla 2. Descripción de los constructos UML usados en el proyecto	132
Tabla 3. Proyectos que componen la implementación de Urano	135
Tabla 4. Metaclasses definidas en el PSM usado por Urano	144
Tabla 5. Artefactos generados por Urano	148
Tabla 6. Criterios para la evaluación cualitativa	151

Tabla 7. Estilo de programación para Amelia	157
Tabla 8. Tiempos de ejecución / caso de estudio	159

**SISTEMA DE BENCHMARKING DE REDES MÓVILES CELULARES Y WIFI BASADO EN
DISPOSITIVOS DE USUARIO FINAL Y SDR**

Tabla 1. Historias de usuario	190
Tabla 2. Parámetros obtenidos y sus fuentes	191
Tabla 3. Variables del componente móvil Android	192
Tabla 4. Significado de los valores de encoding	198

Índice de Figuras

FRAMEWORK PARA GENERACIÓN Y DESPLIEGUE DE MONITORES DE RENDIMIENTO DINÁMICO EN SISTEMAS SOFTWARE AUTOADAPTATIVOS

Figura 1. Ciclo de vida del despliegue de software	39
Figura 2. Diseño arquitectural global (informal) de la propuesta	49
Figura 3. Patrón de diseño interceptor aplicado a los componentes sondas	52
Figura 4. Elementos principales que componen la arquitectura de monitoreo	55
Figura 5. Patrón de diseño Publish/Subscribe aplicado a elementos de la infraestructura de monitoreo	56
Figura 6. MonitorSpecificationModel	57
Figura 7. TypeDeclaration	58
Figura 8. MonitorDeclaration	58
Figura 9. ExtensionDeclaration	58
Figura 10. ImportEventDeclaration	59
Figura 11. ImportNamespaceDeclaration	59
Figura 12. NamespaceDeclaration	59
Figura 13. NamespaceBlockDeclaration	60
Figura 14. VariableDeclaration	61
Figura 15. MonitorBlockExpression	62

Figura 16. HandlerDeclaration	63
Figura 17. EventDeclaration	63
Figura 18. EventEmitter	64
Figura 19. EventEmitter	65
Figura 20. DeploymentSpecificationModel	68
Figura 21. TypeDeclaration	68
Figura 22. DeploymentDeclaration	68
Figura 23. SubsystemDeclaration	68
Figura 24. ExtensionDeclaration	69
Figura 25. IncludeDeclaration	69
Figura 26. DependDeclaration	69
Figura 27. SubsystemBlockExpression	70
Figura 28. InternalSubsystemDeclaration	70
Figura 29. VariableDeclaration	71
Figura 30. ConfigBlockExpression	71
Figura 31. RuleDeclaration	72
Figura 32 . Comando Cd	72
Figura 33. Comando Compile	73
Figura 34. Comando Run	73
Figura 35. Comando Transfer	73
Figura 36. Comando Eval	74
Figura 37. Comando personalizado	74
Figura 38. Pascani: diagrama de clases simplificado de la librería en tiempo de ejecución	80
Figura 39. Pascani: diagrama de clases simplificado de la librería SCA ..	81
Figura 40. Mapeo entre la definición de un espacio de nombres y su correspondiente elemento de clase de Java	83

Figura 41. Mapeo entre la definición Namespace y sus correspondientes elementos de clase Java Proxy	83
Figura 42. Mapeo entre la definición de un monitor y sus correspondientes elementos de clase de Java	84
Figura 43. Diagrama de despliegue de la infraestructura de monitoreo dinámico	85
Figura 44. Amelia: diagrama de clases simplificado de la librería en tiempo de ejecución	87
Figura 45. Mapeo entre la definición de subsistema y los elementos de clase Java generados	88
Figura 46. Mapeo entre despliegues y elementos de clases Java	90
Figura 47. Componentes del modelo de evaluación FQAD	91

DESPLIEGUE DE SOFTWARE AUTOMATIZADO GUIADO POR UML

Figura 1. Metodología	115
Figura 2. Actividades de un proceso usual de despliegue de software	117
Figura 3. Ejemplo de un diagrama de despliegue UML	118
Figura 4. Constructos UML para especificación de despliegue de software	132
Figura 5. Modelo de ejecución de Urano	133
Figura 6. Relación de los proyectos con la ejecución del modelo	135
Figura 7. Perfil propuesto para extender diagramas de despliegue UML	138
Figura 8. Aplicación de perfil en Eclipse Papyrus	137
Figura 9. Clases usadas para extraer y manipular elementos específicos ..	140
Figure 10. Interacción Urano-usuario: definición del diagrama de despliegue	141
Figure 11. Interacción Urano-usuario: selección del modelo a procesar ...	141
Figure 12. Interacción Urano-usuario: la ejecución de Urano	142

Figure 13. Interacción Urano-usuario: ejecución de las especificaciones de Amelia	142
Figura 14. Modelo PSM de la implementación actual de Urano	143
Figura 15. Configuración de despliegue BlockReduce para el caso de estudio MCM	153
Figure 16. Caso de estudio: clasificación - configuración de despliegue	154
Figura 17. Caso de estudio: procesamiento de grandes XML - configuración para el despliegue Reactor	156

SISTEMA DE BENCHMARKING DE REDES MÓVILES CELULARES Y WIFI BASADO EN DISPOSITIVOS DE USUARIO FINAL Y SDR

Figura 1. Estructura general de un sistema OFDM	177
Figura 2. Estructura de la PPDU	180
Figura 3. Transmisor y receptor OFDM 802.11	181
Figura 4. Estructura de tramas LTE	182
Figura 5. Arquitectura de la aplicación móvil	190
Figura 6. Implementación del componente móvil en Android	191
Figura 7. Aplicación móvil	193
Figura 8. Hardware de radiofrecuencia	194
Figura 9. Receptor OFDM en GNU Radio	195
Figura 10. Aplicación SDR usando gr-802.11	197
Figura 11. Aplicación SDR para monitoreo remoto	200
Figura 12. BladeRF conectada a un sistema embebido Raspberry Pi	201
Figura 13. Distribución del backend	202
Figura 14. Sistemas de monitoreo georeferenciados	203
Figura 15. Gestión remota del sistema de monitoreo	204

Acrónimos

FRAMEWORK PARA GENERACIÓN Y DESPLIEGUE DE MONITORES DE RENDIMIENTO DINÁMICO EN SISTEMAS SOFTWARE AUTOADAPTATIVOS

API	Application Programming Information
CBSE	Component-Based Software Engineering
DSL	Domain Specific Languages
EA	Enterprise Applications
EJB	Enterprise JavaBeans
FQAD	Framework for Qualitative Assessment of DSLs
JCA	Java EE Connector Architecture
JMS	Java Message Service
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge
MCM	Matrix-Chain Multiplication
ORM	Object-Relational Mapping
OSR	Online Store Retailer
QA	Quality Assessment
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	RemoteProcedureCalls
SCA	Service Component Architecture
SFTP	SSH File Transfer Protocol
SLOC	Single Lines Of Code
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol)
SSH	Secure SHell
TI	Tecnologías de la Información

DESPLIEGUE DE SOFTWARE AUTOMATIZADO GUIADO POR UML

CWM	Common Warehouse Metamodel
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GL	Guide Lines
IaC	Infrastructure as a Code
M2M	Model to Model
M2T	Model to Text
MCM	Matrix-Chain Multiplication
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
MTL	Model to Text Language
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
SCA	Service Component Architecture
SCM	Software Configuration Management
SCP	Secure Copy Protocol
SDLC	Software Development Life Cycle
SLOC	Source Lines of Code
SLR	Systemic Literature Review
SPLC	Software Product Life Cycle
SSLC	Software System Life Cycle
TAM	Technology Acceptance Model
UML	Unified Modeling Language
XMI	XML Metadata Interchange

SISTEMA DE BENCHMARKING DE REDES MÓVILES CELULARES Y WIFI BASADO EN DISPOSITIVOS DE USUARIO FINAL Y SDR

ADC	Convertidor Análogo-Digital
ANE	Agencia Nacional del Espectro
bladeRF-cli	BladeRF Command Line Interface
BLER	Block Error Rate

CDMA	Code Division Multiple Access
COST	European Cooperation in Science and Technology
CRS	Cell Specific Reference Signal
DAC	Convertidor Digital-Análogo
DFT	Discrete Fourier Transform
DMRS	Demodulation Reference Signal
DSSS	Direct Sequence Spread Spectrum
DVB-T	Digital Video Broadcasting – Terrestrial
eNB	Evolved Node B
EPC	Evolved Packet Core
ERE	Espectro Radioeléctrico
EV-DO	Evolution-Data Optimized
FDD	Frequency-Division Duplex
FFT	Fast Fourier Transform
GPS	Global Positioning System
GSM	Global System for Mobile communications
GUI	Graphic User Interface
HSDPA	High Speed Downlink Packet Access
HSPA	High-Speed Packet Access
i2t	Grupo de Investigación en Informática y Telecomunicaciones
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
IFOM	IP Flow Mobility
IMT	International Mobile Telecommunications
IoT	Internet of Things
iTEAM	Instituto de Telecomunicaciones y Aplicaciones Multimedia
JSON	JavaScript Object Notation
KPI	Key Performance Indicators
LIPA	Local IP Access
LTE	Long Term Evolution
MDT	Minimization of Drive Test
MIMO	Multiple- Input Multiple-Output
OFDM	Orthogonal Frequency Division Multiplexing
PLCP	Physical Layer Convergence Procedure
PLME	Physical Layer Management Entity
PMD	Physical Medium Dependent
PPDU	PLCP Protocol Data Unit
PPI	Power Preference Indication
PSC	Primary Synchronization Code

PSDU	Physical layer Service Data Unit
QoS	Quality of Service
RF	Radio Frecuencia
RSCP	Received Signal Code Power
RSRP	Reference Signals Received Power
RSRQ	Reference Signal Received Quality
RSSI	Received Signal Strength Indicator
RSTD	Reference Signal Time Difference
SC-FDMA	Single-Carrier Frequency Division Multiple Access
SDR	Software Defined Radio
SFTP	SSH File Transfer Protocol
SINR	Signal to Interference plus Noise Ratio
SIPTO	Selected IP Traffic Offload
SMS4DC	Spectrum Management System for Developing Countries
SNR	Signal-to-Noise Ratio
SSH	Secure Shell
TDD	Time-Division Duplex
UIT	Unión Internacional de Telecomunicaciones
UMTS	Universal Mobile Telecommunications System
USRP	Universal Software Radio Peripheral
UWB	Ultra-WideBand
VoIP	Voice over Internet Protocol
WCDMA	Wideband Code Division Multiple Access

Presentación

La computación autónoma surgió con el fin de obtener sistemas informáticos capaces de autogestionarse, esto es de al menos: descubrir y recuperar o prevenir fallas por sí mismos, para garantizar la continuidad en la prestación de los servicios; auto configurarse, para asegurar el ajuste de sus propiedades a los cambios del sistema y en su entorno; y optimizarse, para garantizar el mejor uso de los recursos disponibles.

Si bien los avances en esta materia han permitido desarrollar sistemas dinámicos, reconfigurables, capaces de modificar su estructura en tiempo de ejecución, garantizar niveles específicos de rendimiento en ellos requiere de mecanismos que permitan evaluar métricas que se actualizan periódicamente como respuesta a esos cambios en los requerimientos, esto es, indican los autores del primer capítulo, “de infraestructuras de monitoreo que midan continuamente la satisfacción de los diversos factores de rendimiento capaces de: actualizar de forma dinámica sus estrategias de monitoreo (...); desplegar e integrar los componentes de monitoreo en tiempo de ejecución; y proveer los medios para generar funcionalidades de monitoreo componibles, rastreables y controlables”.

El proyecto al que se refiere el primer capítulo abordó estos retos mediante el “diseño de una arquitectura de monitoreo dinámica y escalable, que implementa y resuelve preocupaciones de monitoreo dinámico en sistemas autónomos sensibles al contexto”. Con base en dicha arquitectura, diseñó e implementó dos lenguajes de dominio específico útiles para esas tareas de monitoreo.

El segundo capítulo de este libro también parte de reconocer la alta competitividad del mercado actual y resalta lo fundamental que es para las empresas asegurar la satisfacción de los requerimientos de sus clientes, lo que, a juicio de sus autores, “se materializa al añadir valor al cliente constantemente,

mediante actualizaciones, correcciones y funcionalidades”. Esta es una tarea compleja debido a la necesidad de instalar y mantener actualizados múltiples componentes de software que se deben configurar apropiadamente. Automatizar dicha tarea es importante, no solo por la necesidad de reducir el costo que representa el alto consumo de tiempo que implicaría hacerla manualmente, sino por la necesidad de reducir la propensión al error.

En el proyecto de investigación que se reporta en el segundo capítulo de este libro, se desarrolló un mecanismo dirigido por lenguaje de unificado de modelado (UML) para automatizar el despliegue de software, con un enfoque basado en los principios de la arquitectura dirigida por modelos (MDA), para generar automáticamente especificaciones de despliegue ejecutables a partir de diagramas de despliegue UML definidos por el usuario, “extendidos a través de un perfil de UML que captura la semántica y los requerimientos de las actividades de despliegue de instalación, configuración y actualización”.

El tercer capítulo de este libro se refiere al control del uso del espectro radioeléctrico –la base de los servicios de telecomunicaciones–. La relevancia del tema no tiene cuestionamiento, el avance en las comunicaciones ha llegado a niveles impensados y ha multiplicado por mucho, en pocos años, lo que la humanidad había logrado en algo más de un siglo. Y aunque las distancias, en términos de comunicación, se han roto por completo, el método para garantizar la calidad de los servicios sigue siendo el mismo: atribuir porciones específicas de espectro a servicios específicos; asignar porciones de esos rangos a unos operadores; y asegurar que los servicios que provee cada operador se prestan dentro del rango esperado.

Si bien la comparación entre la ocupación real y la prevista –en los términos establecidos por la Unión Internacional de Telecomunicaciones–, usualmente se apoya en costosos y sofisticados equipos, hoy es factible usando unidades de monitoreo simples, de operación desatendida y bajo costo, desarrolladas en un proyecto realizado en Icesi con financiamiento de Colciencias. En este tercer capítulo se reporta el desarrollo de una herramienta que permite evaluar las condiciones de los servicios móviles desde la perspectiva del usuario final, utilizando equipos sencillos (USRP y bladeRF), el primer paso firme en la construcción del SiMon, la citada herramienta de gestión del espectro.

José Ignacio Claros V.
Editor

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 autónoma 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 autónomos 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 explícitas 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 sí 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, empaquetado 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.

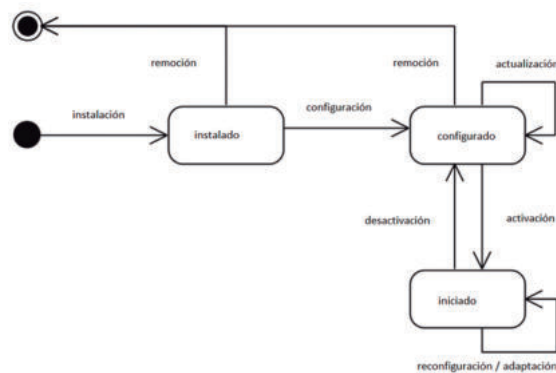


Figura 1. Ciclo de vida del despliegue de software [36]

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] (<i>e.g.</i> , 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 (<i>pull</i>) o por ocurrencia (<i>push</i>).
	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 (<i>knowledge manager</i>) 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>i.e.</i> , 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 (<i>Application Programming Interface</i>) 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>i.e.</i> , 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 redesplicue 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].

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

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.
Escalabilidad de la infraestructura de monitoreo	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.
	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>i.e.</i> , 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

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

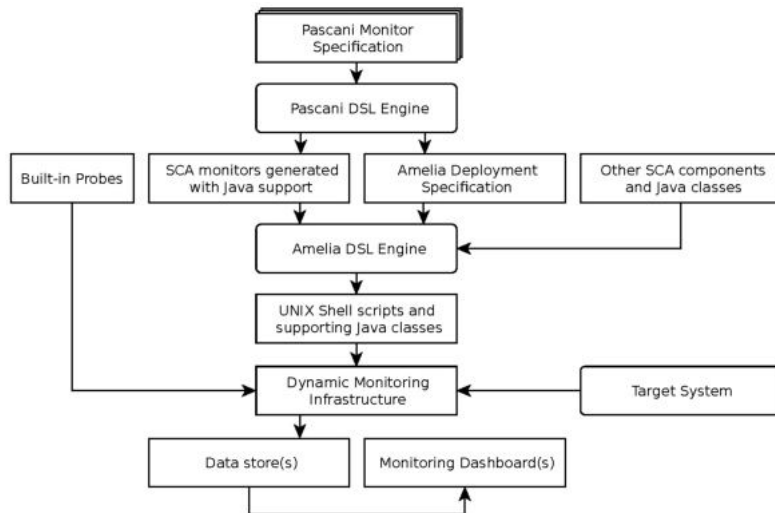


Figura 2. Diseño arquitectural global (informal) de la propuesta

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 cláusulas 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 redespiegue 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 *AspectJ*. 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.

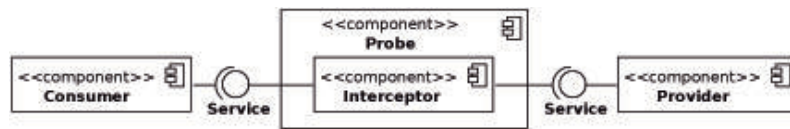


Figura 3. Patrón de diseño interceptor aplicado a los componentes sondas

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 sólida, 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 específicas 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.

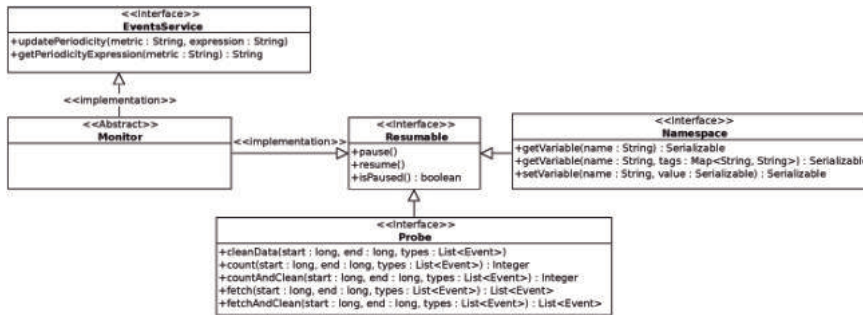


Figura 4. Elementos principales que componen la arquitectura de monitoreo

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.

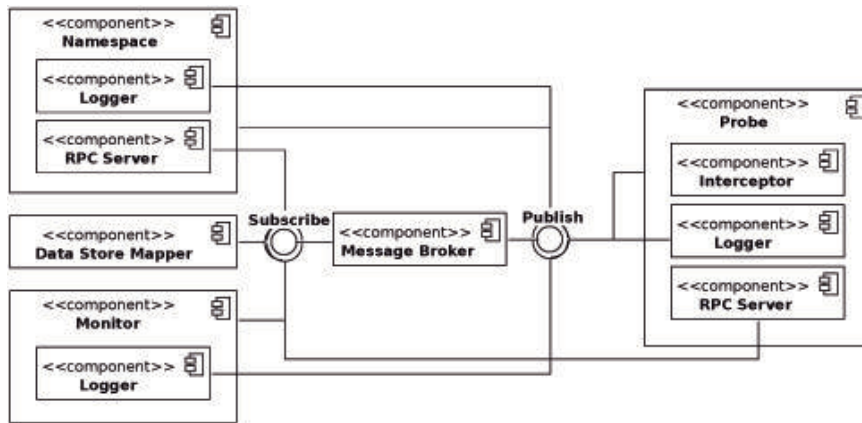


Figura 5. Patrón de diseño Publish/Subscribe aplicado a elementos de la infraestructura de monitoreo

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.

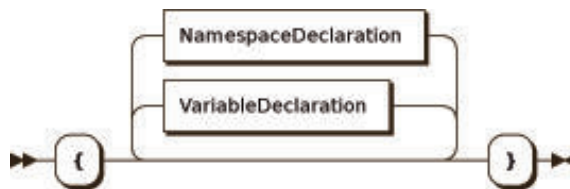


Figura 13. NamespaceBlockDeclaration

Una variable de contexto puede definirse como el valor inmutable de una variable. Si bien especificar su tipo es opcional, si no se especifica 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 válida en el lenguaje .

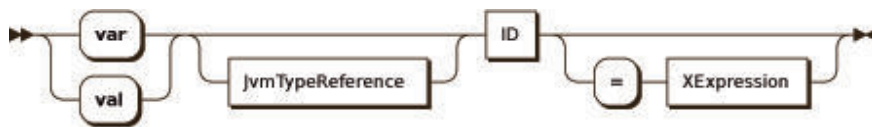


Figura 14. VariableDeclaration

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 inmutable, 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

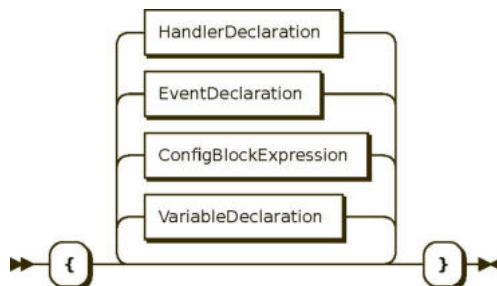


Figura 15. MonitorBlockExpression

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.

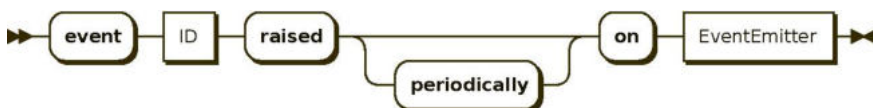


Figura 17. EventDeclaration

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.

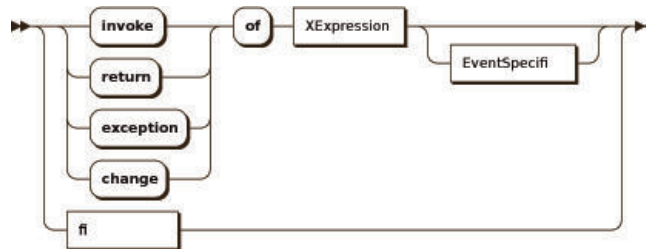


Figura 18. EventEmitter

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 específico. 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.

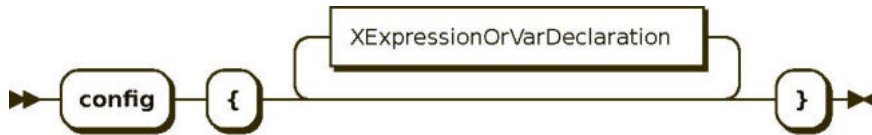


Figura 19. EventEmitter

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 updateThroughput(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.

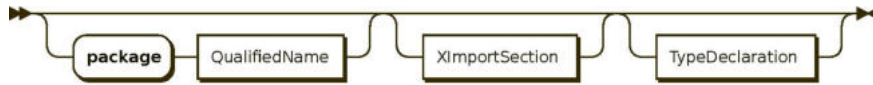


Figura 20. DeploymentSpecificationModel

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.

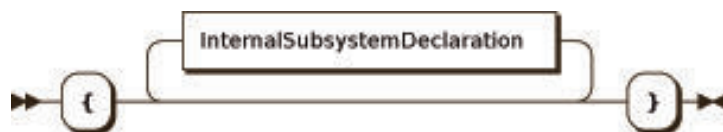


Figura 27. SubsystemBlockExpression



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 instanciados 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*.

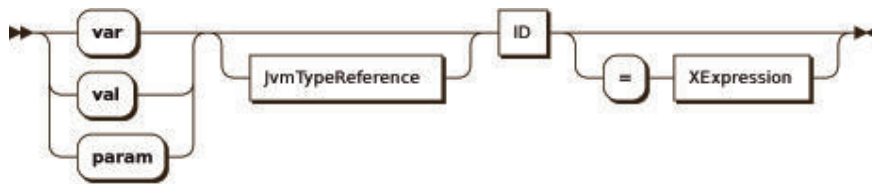


Figura 29. VariableDeclaration

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.

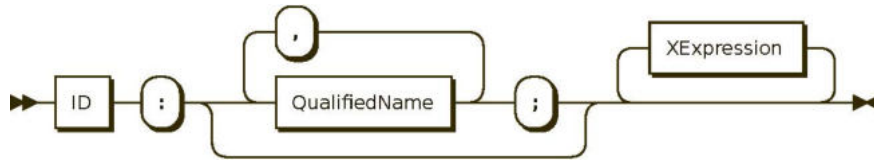


Figura 31. RuleDeclaration

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).

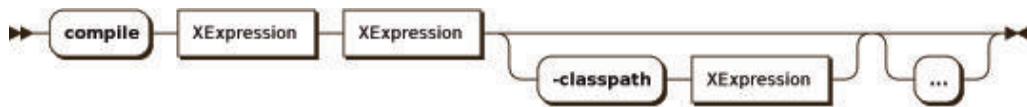


Figura 33. Comando Compile

- 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).

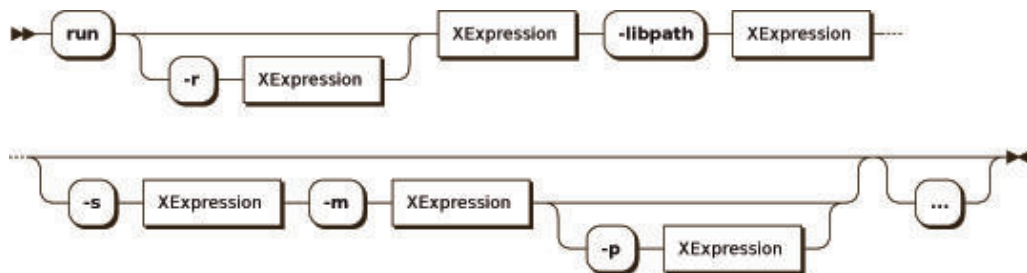


Figura 34. Comando Run

- 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).

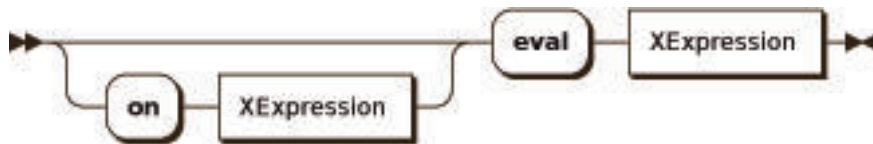


Figura 36. Comando Eval

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 "$FRASCATL_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
12  val remote = new Host("192.168.99.100", 25632, 41256, "username", "password")
13  set(newHelloworld(remote))
14
15  for (i : 1..10) {
16    helper.deploy([
17      start(true)
18    ], 3)
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",
5             PascaniPackage.Literals.HANDLER__PARAMS)
6     }
7     if
8         (handler.params.get(0).actualType.getSuperType(org.pascani.dsl.lib.Event) == null) {
9         error("The first handler.params.size > 1 first parameter must be subclass of Event",
10             PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
11     }
12     if (handler.params.size > 1) {
13         val actualType = handler.params.get(1).actualType.getSuperType(Map)
14         val showError = actualType == null
15         || actualType.typeArguments.size != 2
16         || !actualType.typeArguments.get(0).identifier.equals(String.canonicalName)
17         || !actualType.typeArguments.get(1).identifier.equals(Object.canonicalName)
18         if (showError)
19             error("The second parameter must be of type Map<String, Object>",
20                 PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
21     }
22 }

```


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.

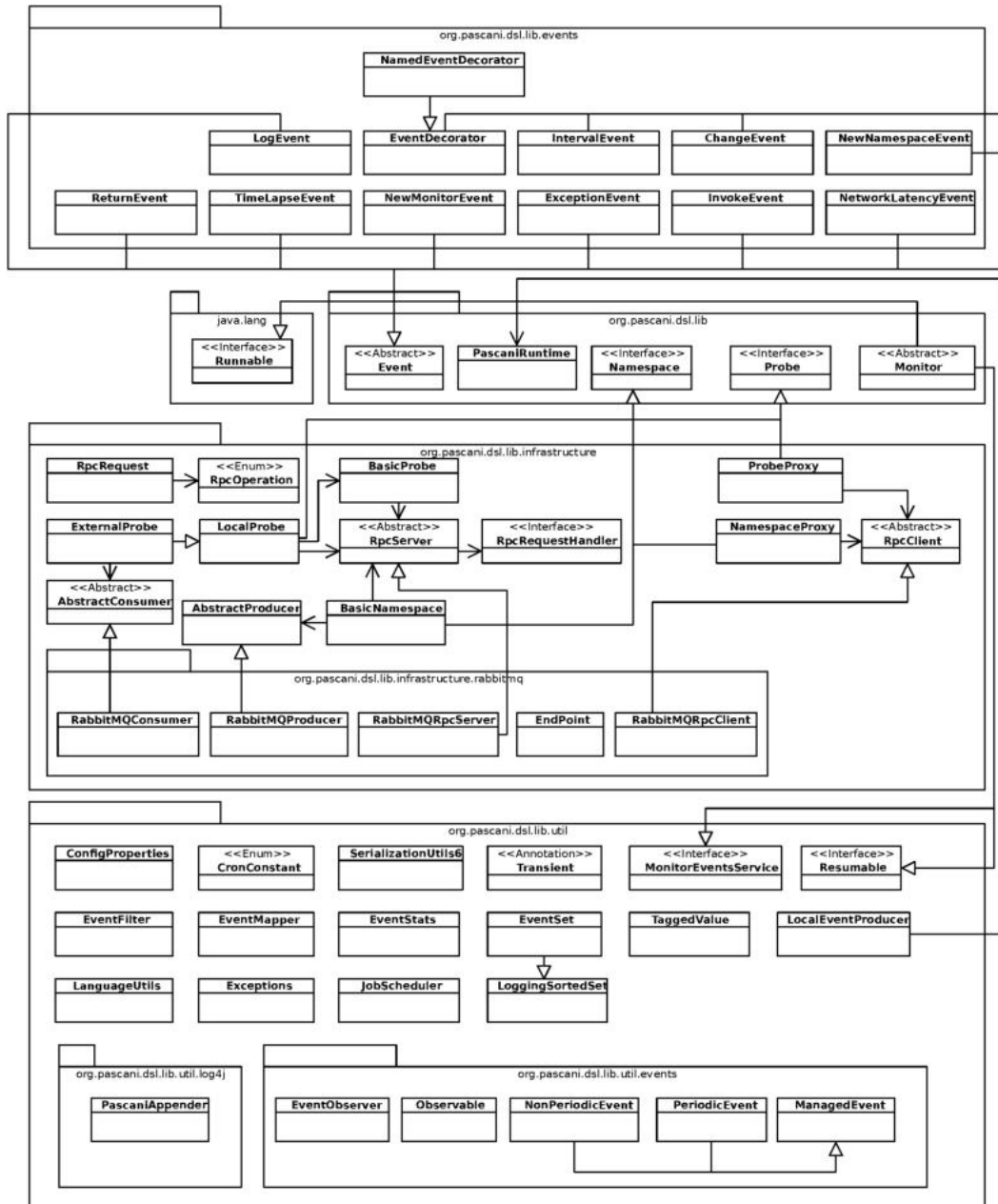


Figura 38. Pascani: diagrama de clases simplificado de la librería en tiempo de ejecución

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.

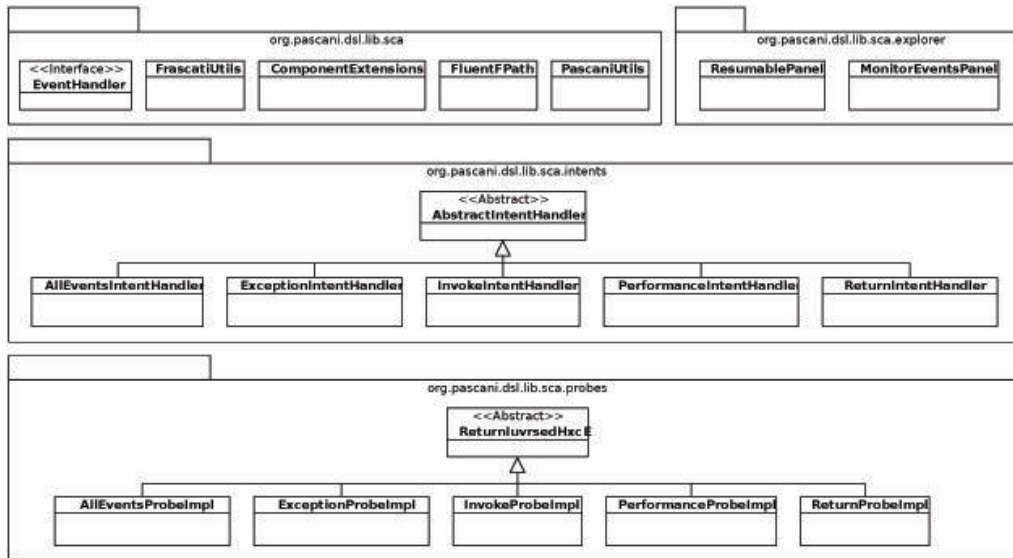


Figura 39. Pascani: diagrama de clases simplificado de la librería SCA

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

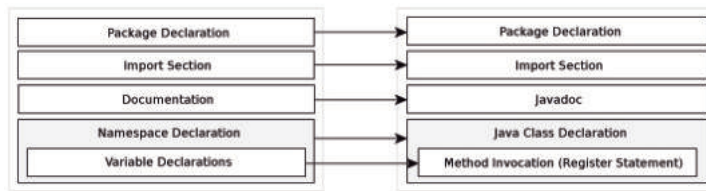


Figura 40. Mapeo entre la definición de un espacio de nombres (izquierda) y su correspondiente elemento de clase de Java (derecha)

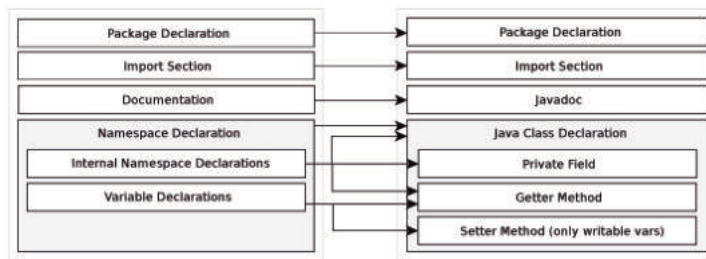


Figura 41. Mapeo entre la definición de un espacio de nombre (izquierda) y sus correspondientes elementos de clase Java Proxy (derecha)

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).

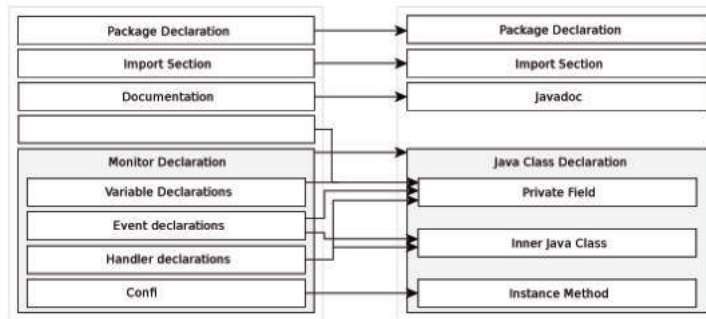


Figura 42. Mapeo entre la definición de un monitor (izquierda) y sus correspondientes elementos de clase de Java (derecha)

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 desplegarse 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.

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

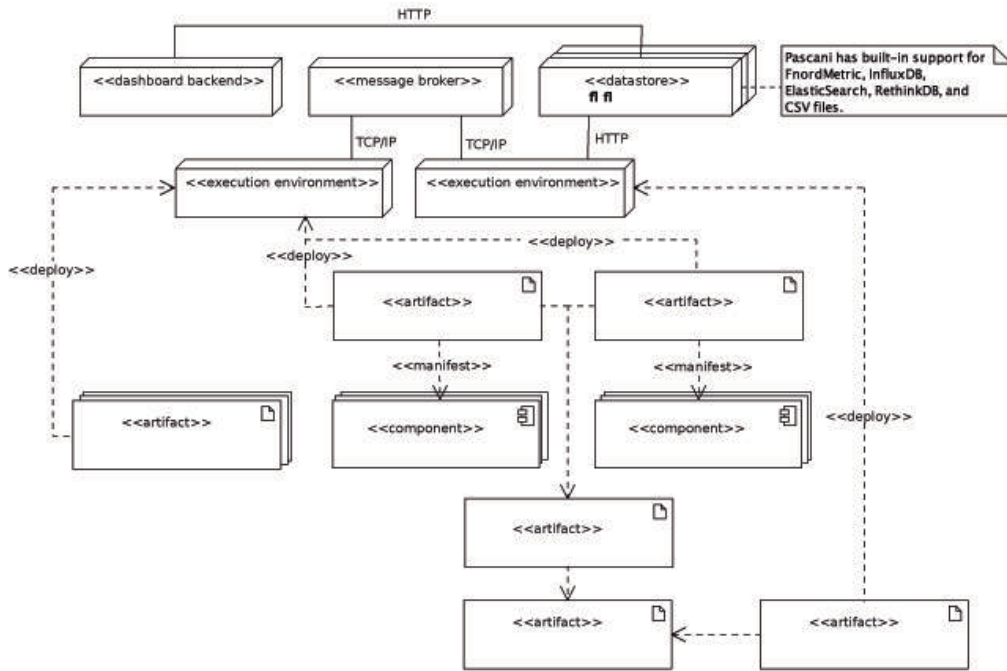


Figura 43. Diagrama de despliegue de la infraestructura de monitoreo dinámico

Tabla 7. Proyectos que conforman la implementación de Amelia

	Proyecto	Contenido	SLOC
Eclipse	org.amelia.dsl.feature	Definición del proyecto de <i>plug-ins</i> 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(OnHostBlockExpression blockExpression) {
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 ÃÃHost.simpleName or
6         Iterable<ÃÃHost.simpleName>, ÃÃtype.simpleName was found instead"
7     val showError = !isOk
8         || type.getSuperType(List).typeArguments.length == 0
9         || !type.getSuperType(Iterable).typeArguments.get(0).identifier.equals(Host.canonicalName)
10    if (showError) {
11        error(msg, AmeliaPackage.Literals.
12            ON_HOST_BLOCK_EXPRESSION__HOSTS, INVALID_PARAMETER_TYPE)
13    }
14 }

```

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

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

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

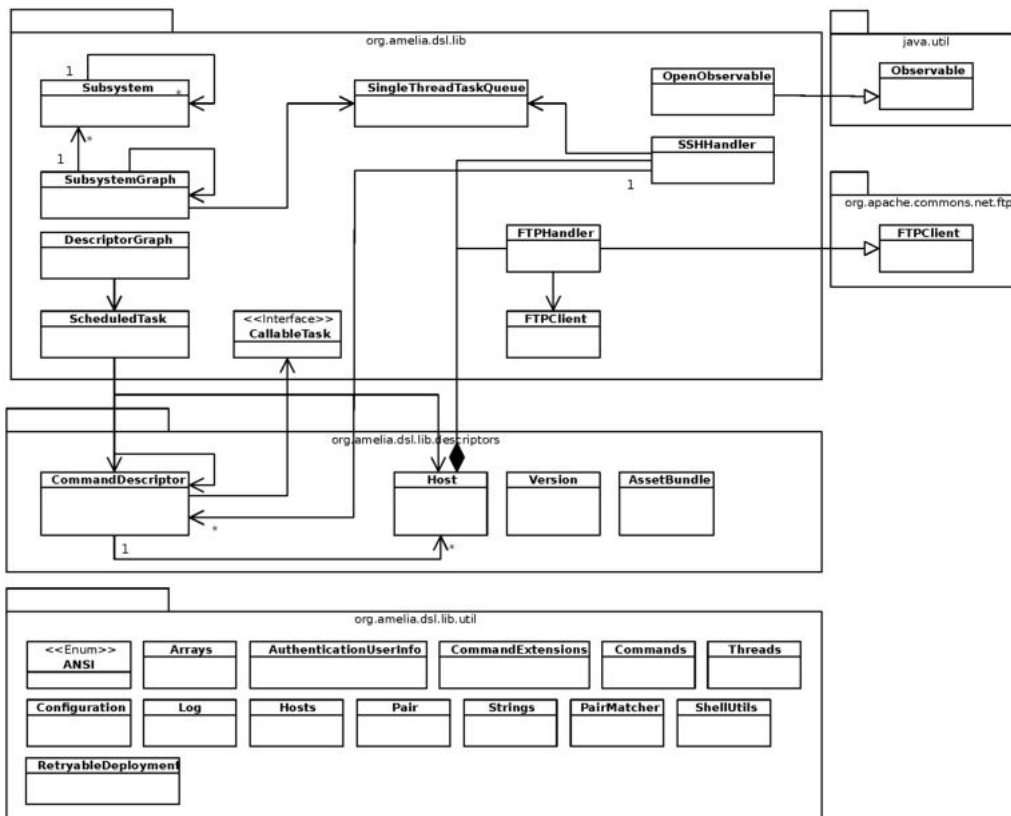


Figura 44. Amelia: diagrama de clases simplificado de la librería en tiempo de ejecución

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

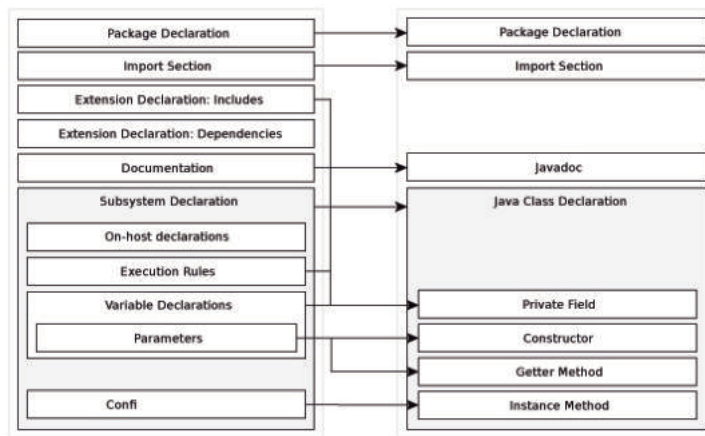


Figura 45. Mapeo entre la definición de subsistema (izquierda) y los elementos de clase Java generados (derecha)

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 último 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.

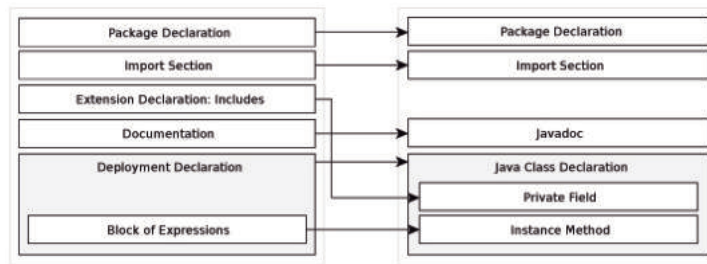


Figura 46. Mapeo entre despliegues y elementos de clases Java

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.

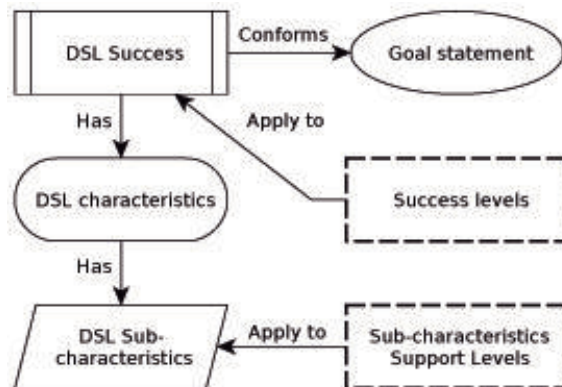


Figura 47. Componentes del modelo de evaluación FQAD [55]

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 rediseño 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 redesplesgarlos 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 redesplicue 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 redespigüe. 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 redesplicue 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], available: http://www.davidchappell.com/writing/Introducing_SCA.pdf
- [19] P. Shepherd (2009, August), *Oracle SCA: The power of the composite* [Online], Available: <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], Available: <http://www.fabric3.org>
- [22] IBM (2015), *IBM WebSphere application server feature pack for SCA* [Online], available: <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., *LNC3, vol. 8378, Models@run.time: Foundations, applications, and roadmaps* [Dagstuhl Seminar 11481, November 27 – December 2, 2011], Berlin–Heidelberg, 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 modèle pour le déploiement de systèmes 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,” *Ingeniería 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] *Influxdata* [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. Jiménez, “A framework for generating and deploying dynamic performance monitors for self-adaptive software systems,” MSc. thesis, Universidad Icesi, Cali, Colombia, 2016.

ANEXO I. 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 =
71     XExpression)?
72     ;
73
74 ConfigBlockExpression returns XBlockExpression
75     : {ConfigBlockExpression} 'config' '{' (expressions += XExpressionOrVarDeclaration '!')* '}'
76     ;
77
78 HandlerDeclaration returns Handler
79     : 'handler' name = ValidID
80     '(' params += FullJvmFormalParameter (',' params += FullJvmFormalParameter)* ')'
81     body = XBlockExpression
82     ;
83
84 EventDeclaration returns Event
85     : 'event' name = ValidID 'raised' (periodical ?= 'periodically')? 'on' emitter = EventEmitter -> '!'?
86     ;
87
88 EventEmitter
89     : eventType = EventType 'of' emitter = XExpression (=> specifier = AndEventSpecifier)?
90     | cronExpression = XExpression

```


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

```
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
154     : 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     (writable?= '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 +=
73         [RuleDeclaration | QualifiedName])*? ';')?
74     (commands += XExpression)*
75     ;
76
77 CdCommand
78     : 'cd' directory = XExpression (=> initializedLater ?= '...')?
79     ;
80
81 CompileCommand
82     : 'compile' source = XExpression output = XExpression
83     (=> '-classpath' classpath = XExpression)?
84     (=> initializedLater ?= '...')?
85     ;
86
87 RunCommand
88     :
89     'run' (hasPort ?= '-r' port = XExpression)?

```

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

```
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
125             expressions += XExpression)*)
126         expressions += RichStringLiteralEnd
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
154     | 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     ;

```

DESPLIEGUE DE SOFTWARE AUTOMATIZADO GUIADO POR UML

Luis Felipe Rivera Vera, MSc.

Norha M. Villegas, Ph.D

Gabriel Tamura, Ph.D

Citación

L.F. Rivera, N. M. Villegas, y G. Tamura, “Despliegue de software automatizado guiado por UML,” en *Bitácoras de la maestría*, vol. 3, *Monitores dinámicos de software - Despliegue de software - Monitoreo de espectro*, Cali, Colombia: Universidad Icesi, 2020, pp. 109-108.

RESUMEN

Los sistemas de software tienen un rol fundamental en las actividades cotidianas, tanto al nivel personal, como organizacional. Actividades como el transporte, la comunicación y la atención en salud son ahora fuertemente dependientes de ellos. Sobrevivir en una economía estrecha, con mercados altamente competitivos, como sucede hoy, hace que las empresas deban asegurar la satisfacción de los requerimientos de sus clientes, algo que, en sistemas software, se materializa al añadir valor al cliente constantemente, mediante actualizaciones, correcciones y funcionalidades. Garantizar los requerimientos de entregas frecuentes no es una tarea fácil, implica diseñar, desarrollar, desplegar continuamente y operar sistemas software complejos y robustos. El número de este tipo de sistemas crece constantemente, como también la complejidad de sus procesos de despliegue, especialmente en ambientes *cloud* y en ambientes relacionados con DevOps o *continuous delivery*. Esta complejidad se origina en la instalación y actualización de múltiples componentes de software que se deben configurar apropiadamente para lograr llevar a cabo sus interdependencias y sus entornos de ejecución distribuidos. Este proceso además consume mucho tiempo y suele ser propenso a errores, por lo cual, si es llevado a cabo de forma manual, puede conducir a reprocesos en sus actividades asociadas, con su correspondiente efecto en el costo. Este proyecto de investigación abordó este reto mediante un mecanismo dirigido por Lenguaje de Modelado Unificado (UML, *Unified Modeling Language*) para automatizar el proceso de despliegue de software, con un enfoque basado en los principios de la Arquitectura Dirigida por Modelos (MDA, *Model Driven Architecture*) para generar automáticamente especificaciones de despliegue ejecutables a partir de diagramas de despliegue UML definidos por el usuario, los cuales son extendidos a través de un perfil de UML que captura la semántica y los requerimientos de las actividades de despliegue de instalación, configuración y actualización.

INTRODUCCIÓN

En los últimos años las empresas buscan ofrecer productos basados en software y funcionalidades con la mayor frecuencia posible, de tal manera que sus clientes perciban continuamente una generación de valor. Un proceso automatizado de despliegue de software es clave para este objetivo, ya que hacerlo manualmente resultaría costoso y demorado y generaría otras consecuencias no deseadas [1]. En general, las prácticas actuales de despliegue de software, manuales y automatizadas, no cumplen con los requerimientos de las empresas y los clientes, por lo que los ingenieros de software deben invertir tiempo y esfuerzo en repetir las tareas de despliegue o mantener sus especificaciones sin contar con herramientas que ofrezcan el soporte completo durante ese proceso. De manera alternativa, un enfoque guiado por modelos podría, no solo alcanzar el nivel de abstracción de las actuales especificaciones de despliegue –y así reducir los esfuerzos necesarios para habilitar el proceso de automatización– sino también reducir la complejidad de las plataformas de despliegue disponibles y expresar efectivamente los conceptos del dominio del despliegue [2].

En un sentido amplio, el despliegue de software es un proceso primario de la ingeniería de software [3], que comprende un conjunto de actividades de posproducción que inician con la adquisición de un sistema –que ya ha sido desarrollado, probado, empaquetado y publicado por un proveedor de software– y se planea su despliegue [4]. Aunque esta definición del proceso de despliegue sigue vigente, la forma y frecuencia con la que se realiza ha cambiado con el tiempo. Informes recientes [5], [6] muestran cómo el número de implementaciones por unidad de tiempo ha aumentado en las empresas, y con ello ahora es mayor la frecuencia de entrega de productos y funcionalidades. Esta tendencia ha sido liderada por las grandes compañías de software (*e.g.*, Amazon, Facebook, Flickr, Google y Netflix), quienes implementan nuevas funcionalidades (atómicas) y correcciones de errores todos los días [7]. Dado que las empresas coexisten en una economía apretada, con mercados competitivos, no es sorprendente que la mayoría de ellas ahora busquen reducir los tiempos de despliegue para cumplir rápidamente con los requisitos cambiantes de los usuarios, algo que podría afectar decisivamente sus posibilidades de lograr una posición de mercado más sólida.

La necesaria capacidad de entregar “tan pronto como sea posible” nuevas funcionalidades y correcciones de errores ha hecho que surjan nuevos enfoques

de ingeniería de software inspirados en metodologías ágiles, tales como: DevOps, entrega continua (*continuous delivery*) y despliegue continuo (*continuous deployment*), en los cuales la automatización del despliegue se convierte en algo imprescindible para garantizar la posibilidad de entregar valor continuamente a los clientes [1].

Para proveer los medios requeridos para automatizar las tareas de despliegue han surgido diversas tecnologías: la organización en contenedores, propuesta en Docker [8] y Kubernetes [9], entre otros; y la infraestructura como código, presente en Chef [10], Ansible [11] y Puppet [12], entre otros, lideran este propósito. El factor común de estas tecnologías es la forma como se especifican las instrucciones de despliegue, esto es, en buena parte, con notas basadas en texto. Esto implica procesos de software más complejos porque la mayoría de estos marcos fueron diseñados originalmente para abarcar más que solo los procesos de despliegue. Por otra parte, la diversidad en alcances y objetivos implica que los ingenieros de software, no solo deben entender sus diferencias, sino también aprender y dominar los lenguajes de programación específicos usados en su desarrollo, los cuales, al ser, por lo general, específicos del proveedor, implican curvas de aprendizaje independientes para cada tecnología.

Buscando la estandarización –e inspirada por la comunidad de Ingeniería Dirigida por Modelos (MDE, *Model Driven Engineering*), el Grupo de Gestión de Objetos (OMG, *Object Management Group*) definió una iniciativa llamada Arquitectura Dirigida por Modelos (MDA, *Model Driven Architecture*), que es la realización OMG de MDE, basada en estándares OMG, tales como: MOF (*Meta Object Facility*), XMI (*XML Metadata Interchange*) y UML (*Unified Modeling Language*) [13]. Siguiendo los principios MDE/MDA, en lugar de las especificaciones basadas en texto, la automatización del proceso de despliegue puede ser impulsada por modelos estandarizados, como los diagramas de despliegue UML, los cuales no solo se utilizan para representar un conjunto de nodos de procesamiento (físicos) y sus interrelaciones [3], sino también para describir la arquitectura de ejecución de los sistemas y la asignación de artefactos de software a los elementos del sistema [14]. Si se modela correctamente, los diagramas de despliegue podrían proporcionar información valiosa para la automatización de algunas de las actividades principales en el proceso de despliegue de software.

Modelar diagramas de despliegue de manera adecuada para permitir su automatización podría requerir la extensión de los elementos sintácticos

UML actuales y la semántica correspondiente. Afortunadamente, el marco de metamodelado de OMG proporciona mecanismos de extensibilidad potentes, fundamentalmente los perfiles. El paquete perfiles en UML ofrece tres mecanismos principales: estereotipos, definiciones etiquetadas y restricciones, para permitir la extensión de las metaclasses de los metamodelos existentes [14]. El uso de perfiles para extender UML representa un medio esencial, tanto para permitir la automatización del proceso de despliegue, como para garantizar la expresividad suficiente en el contexto de un enfoque basado en modelos. Esto mitiga los problemas de consumo de tiempo y propensión al error del proceso de despliegue, particularmente cuando se realiza con mucha frecuencia, como en la estrategia de entrega continua o en entornos de nube y en procesos DevOps adoptados

Teniendo en cuenta el contexto mencionado, la investigación dirigió sus esfuerzos a desarrollar un mecanismo que permita automatizar el despliegue de software mediante la transformación de diagramas de despliegue UML en especificaciones de despliegue ejecutables. Para lograrlo, estableció como objetivos específicos:

- comparar implementaciones del metamodelo UML 2.x OMG que proporcionen herramientas para editar y procesar modelos de despliegue y seleccionar la que mejor satisfaga la especificación UML para diagramas de despliegue;
- seleccionar –y posiblemente completar–, un modelo independiente de plataforma (PIM, *Platform Independent Model*) que capture la semántica y los requisitos de las actividades del proceso de despliegue a automatizar: instalación, configuración y actualización;
- desarrollar un modelo específico de plataforma (PSM, *Platform Specific Model*) que permita generar una instancia del modelo independiente de la plataforma en un marco concreto;
- especificar un conjunto modelo de transformaciones requerido para derivar las operaciones de despliegue que permitan transformar una instancia de PSM en una especificación de despliegue ejecutable, donde la cadena de transformación de cuenta de la semántica de los diagramas de despliegue;
- diseñar y desarrollar un modelo de ejecución para realizar las transformaciones de modelo especificadas (por ejemplo, integrando los motores operativos de la cadena de transformación del modelo); y

- evaluar el mecanismo propuesto aplicándolo en al menos dos casos de estudio relevantes.

En la investigación se utilizó una metodología con enfoque cualitativo [15] que combina revisiones sistemáticas de literatura (SLR, *Systemic Literature Review*) [16], MDA, [17], [18] y estudio de casos [19], [20]. En la FIGURA 1 se esquematiza la aplicación de la metodología a lo largo de las fases del proyecto. El desarrollo de la investigación supuso ir alcanzando los siguientes hitos:

- la revisión sistemática de la literatura sobre mecanismos y especificaciones de despliegue automatizado para implementar diferentes tipos de sistemas de software, junto con una revisión sobre perfiles UML para despliegue de software;
- la selección de un PIM y la implementación de un despliegue del metamodelo UML 2.0, junto con el diseño de un perfil UML que captura la semántica y los requisitos de las actividades de instalación, configuración y actualización del despliegue;
- el diseño de un PSM que permita crear instancias del PIM en un marco concreto;
- la implementación de un conjunto de especificaciones de transformaciones de modelo requeridas para derivar la creación de instancias del PSM y las especificaciones de implementación ejecutables de una instancia de PSM;
- la implementación de un modelo de cadena de transformación compuesta por el PIM, el PSM, el modelo de transformaciones y el modelo de ejecución, que permita automatizar las actividades de despliegue específicas; y

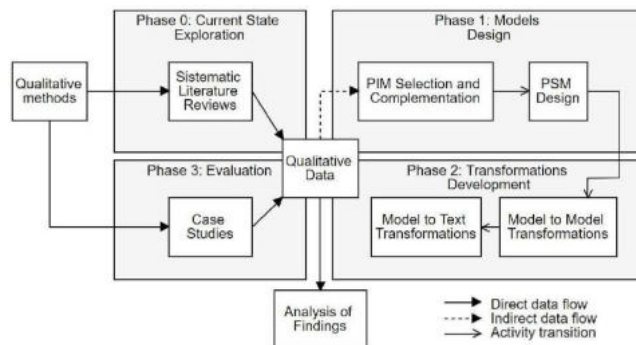


Figura 1. Metodología

- el desarrollo de un prototipo que implemente la solución propuesta y permita su aplicación en un escenario de despliegue de caso real.

MARCO TEÓRICO

El despliegue de software es un proceso primario de la ingeniería de software, no forma parte del ciclo de vida del desarrollo de software (SDLC, *Software Development Life Cycle*) pero si, combinado con otros procesos de inicio-a-retiro (*e.g.*, mantenimiento, soporte y evolución) y un SDLC integra un ciclo de vida del producto software completo (SPLC, *Software Product Life Cycle*). Este proceso de postproducción, como se dijo, inicia una vez que el producto ya ha sido desarrollado, empacado y publicado, y un implementador lo ha adquirido y planea implementarlo de acuerdo con sus necesidades particulares [4]. El OMG ha definido un conjunto de actividades generalmente presentes en un proceso de despliegue de software [4], el cual incluye: instalación, definida como la transferencia del paquete al repositorio del implementador; configuración, donde se habilita una configuración funcional del software para su uso futuro; planeación, donde las decisiones de cómo y dónde correrá el software en un ambiente objetivo se especifican en un plan de despliegue; preparación, definida como el conjunto de tareas que se llevan a cabo para permitir que el ambiente objetivo esté listo para ejecutar el software; y lanzamiento, donde el software se coloca en estado de ejecución.

Otros autores, como Carzaniga et al., [21] y Dearie proponen un nuevo conjunto de actividades interrelacionadas, más concretas: publicación, que engloba todas las actividades necesarias para preparar un software para ser empacado y transferido al sitio del consumidor; instalación, que cubre la inserción inicial y a configuración de un software en el sitio de un consumidor; activación, donde se ponen en marcha los elementos ejecutables de un software; desactivación, definida como el cese de la ejecución de los componentes de una software en uso; actualización, donde se instala una nueva versión del software; adaptación, que implica modificar, generalmente sobre la marcha, una versión del software previamente instalada; desinstalación, que corresponde a la remoción de un software instalado; y liberación, donde un software se marca como obsoleto y su proveedor discontinúa su soporte. En la FIGURA 2 se representa el conjunto de actividades interrelacionadas que forman parte del proceso de despliegue de un software.

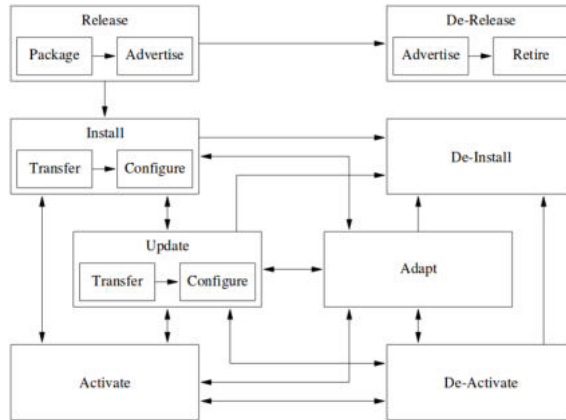


Figura 2. Actividades de un proceso usual de despliegue de software [21]

DIAGRAMAS DE DESPLIEGUE UML

Los diagramas de despliegue son parte del UML, un estándar técnico (especificación) que define un conjunto de elementos de notación legibles para humanos para el diseño, análisis e implementación de sistemas basados en software. Las especificaciones UML son gobernadas por OMG, entidad a cargo de su publicación como referencias normativas. La especificación UML v. 2.5 [14] es la última de un conjunto de más de diez versiones previas de este lenguaje gráfico, cada una, una versión ampliada de la anterior. Como quedó establecido en el objetivo general de esta investigación, el trabajo se enfocó en la transformación de diagramas de despliegue UML en especificaciones ejecutables de despliegue, esto es realizar el despliegue en una infraestructura dada, como se especifica en el diagrama. Además, los principales conceptos relacionados con UML que guiaron la presente investigación son: la unidad de lenguaje de las implementaciones y las extensiones de los diagramas de despliegue.

UNIDAD DE LENGUAJE DE LAS IMPLEMENTACIONES

Un diagrama de despliegue UML, descrito en la unidad de lenguaje de las implementaciones, define el conjunto de constructos (*e.g.*, artefactos, nodos de procesamiento, especificaciones de despliegue, dependencia, despliegue, manifestación) que no solo son usados para representar un conjunto de nodos de procesamiento (físicos) y sus interrelaciones [3], [22], sino también

para describir la arquitectura de ejecución de los sistemas y la asignación de artefactos software a los elementos del sistema [14].

En la FIGURA 3 se representa el uso de variadas constructos de diagramas de despliegue UML (nodo de procesamiento, componente, dependencia y asociación) para especificar la implementación de un conjunto de componentes software en una infraestructura objetivo. En este diagrama, algunos componentes software, como: *Order*, *ShoppingCart* y *Payment*, están

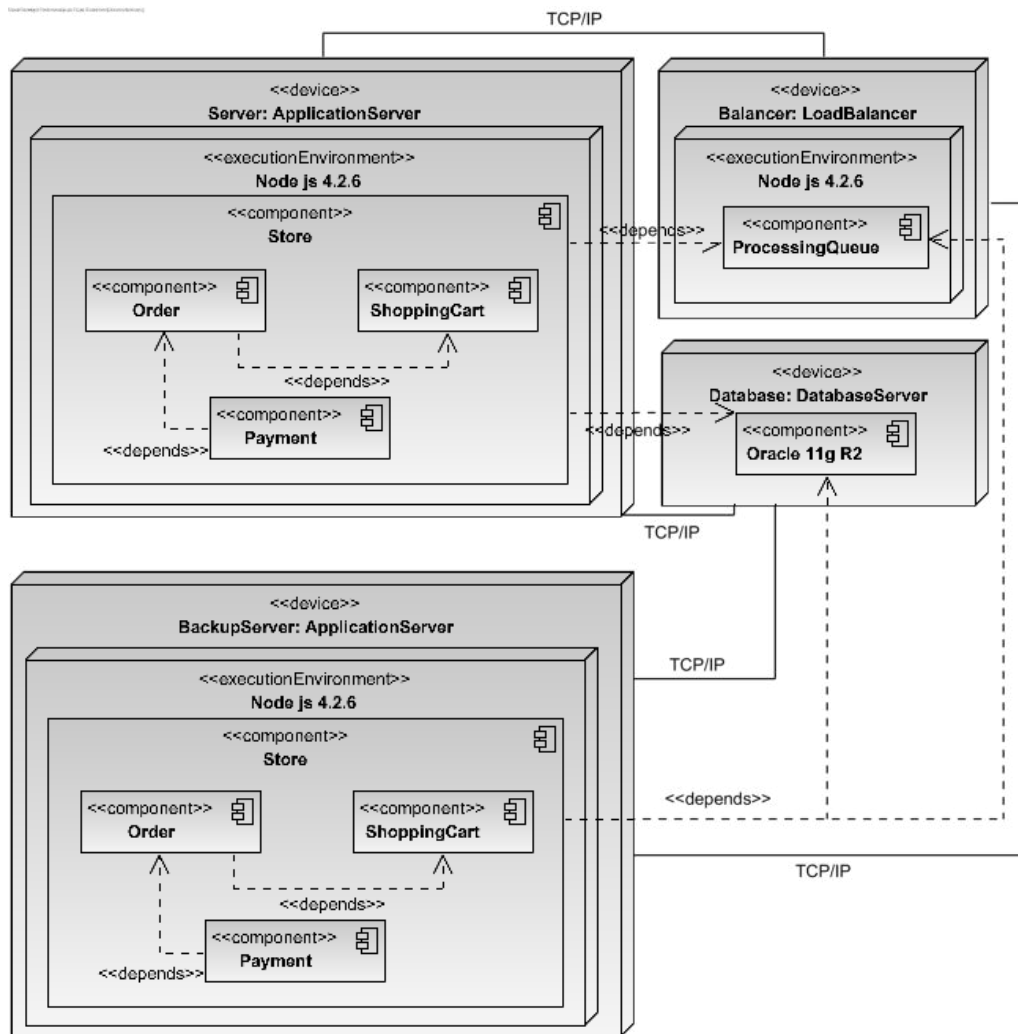


Figura 3. Ejemplo de un diagrama de despliegue UML

agrupados en otro componente denominado *Store*. Adicionalmente, algunas dependencias de ejecución se establecen entre los componentes al interior de la tienda y la ejecución de este componente compuesto depende de la ejecución de otro componente, como un *ProcessingQueue* y Oracle 11g R2. Además, los componentes están embebidos en nodos estereotipados y digitados (*e.g.*, los componentes *Store* y *ProcessingQueue* en el nodo js 4.2.6 y el componente Oracle 11g R2 en el nodo *Database*), lo cual significa que un componente se implementa en un nodo. En los nodos de este diagrama se utilizan dos estereotipos: *device*, que indica que un nodo diagramado representa un recurso computacional físico (*e.g.*, servidor, PC, smartphone); y *executionEnvironment*, que significa que un nodo ofrece una ejecución ambiental para un tipo específico de componentes desplegados ahí. Adicionalmente, en este diagrama se definen tres clases de nodos: *ApplicationServer*, *LoadBalancer* y *DatabaseServer*. Finalmente, en el diagrama se usan asociaciones para especificar los protocolos de comunicación que permiten las comunicaciones entre los nodos.

EXTENSIONES DE LOS DIAGRAMAS DE DESPLIEGUE UML

Aunque no de manera explícita, el OMG describe los dos principales mecanismos de extensión [14]: extensibilidad de primera clase, manejada a través de MOF, donde no hay restricciones en el conjunto posible de cambios al metamodelo del UML; y extensibilidad de no primera clase, usualmente manejada a través del mecanismo de perfiles, el cual permite la adaptación de metamodelos UML existentes con constructos que son específicos a un dominio, plataforma o método particulares. James Bruck, un antiguo contribuyente del Eclipse Modelling Project, y Kenn Hussey, un representante de la Eclipse Foundation en OMG, describe en [23] un conjunto más concreto de técnicas para diagramas UML extendidos, el cual se describe a continuación.

Extensiones peso pluma. Involucra el uso de palabras clave, las cuales usualmente se presentan como anotaciones de texto adjuntas a un elemento UML. Estas palabras clave se pueden utilizar como una manera de distinguir entre metaclasses que comparten formas gráficas similares (*i.e.*, clases e interfaces), especificando el valor del meta-atributo que está adjunto a un concepto UML (*i.e.*, la palabra clave *singleExecution*, que aparece adjunta al elemento *Activity*) o indicando estereotipos estándar. Este mecanismo de extensión, que no está estandarizado o formalmente definido por la especificación UML de superestructura, no permite adjuntar restricciones o adicionar propiedades a

metatipos existentes. Además, no hay no hay medios para compartir palabras clave comunes ni formas de validar su aplicación.

Extensiones “peso ligero”. Este tipo de extensión implica el uso de perfiles UML, los cuales definen extensiones limitadas a un metamodelo de referencia –como UML–, para su adaptación a una plataforma o dominio específico. Los perfiles se especifican en diagramas de estructura mediante la definición de estereotipos personalizados, definiciones etiquetadas con los valores correspondientes y restricciones, todo lo cual actúa como construcciones de extensión primaria. Los estereotipos se pueden usar para agregar palabras clave, restricciones, imágenes y propiedades (valores etiquetados) a los elementos del modelo. Este mecanismo de extensibilidad no promueve especificaciones de comportamiento ni la modificación de estructuras y restricciones actuales. Las extensiones ligeras están descritas en la especificación de superestructura de UML, pueden agregar estructura al metamodelo y su realización implica bajo costo de desarrollo y fácil despliegue.

Extensiones “Peso medio”. Este mecanismo permite la extensión de UML a través de la especialización de meta-tipos UML. Su realización requiere de ambos, extender por referenciación del metamodelo UML (*i.e.*, un conjunto de metatipos UML de diferentes unidades de lenguaje, fusionados, y agregando nuevo tipos específicos de dominio. Aunque este mecanismo de extensibilidad promueve la adición y la modificación del comportamiento, la estructura y las restricciones, puede ser difícil de mantener dada su dependencia de una versión específica de UML. Esta clase de extensiones por lo general involucran altos costos de desarrollo y requieren formatos de archivo para su implementación que no son estándar. Adicionalmente, se requiere extender la totalidad del metamodelo UML, así solo una pequeña porción de él lo necesite.

Extensiones "peso pesado". Este tipo de extensión implica reutilización vía copia o fusión, lo que significa que es necesario seleccionar únicamente las unidades de lenguaje que serán extendidas, fusionarlas e incluir los nuevos tipos específicos de dominio para ese conjunto. Aunque este mecanismo ofrece la mayor flexibilidad al extender el metamodelo UML, su desarrollo es costoso, supone un gran reto de mantenimiento, no puede modificar los comportamientos existentes y sus formatos de archivo no son estándar. Por otra parte, dado que se define un nuevo metamodelo, su interoperabilidad con otras herramientas basadas en UML puede verse comprometida.

En esta investigación se utilizó un mecanismo de extensión liviano, realizado a través de un perfil, para extender la semántica de los diagramas de despliegue UML y permitir la generación de especificaciones de despliegue ejecutables a partir de ellos.

ENTREGA CONTINUA

Este nuevo paradigma de la ingeniería de software demanda de los proveedores de software la capacidad de desplegar cualquier lanzamiento exitoso de un sistema software provisto, lo cual implica que una aplicación debe estar siempre en estado “actualizable” [1]. Dado que las empresas coexisten en un mercado estrecho y competitivo e inspirado por los principios del Agile Manifesto, el propósito final de las prácticas de entrega continua es proveer continuamente, tan frecuentemente como sea posible, valor agregado al usuario final, lo que desencadena en un proceso de retroalimentación constante. Los mayores beneficios de este enfoque son: el empoderamiento de los equipos de desarrollo y operaciones; la reducción de errores (*bugs*), con su consecuente efecto positivo en términos de menores costos y riesgos; la reducción del stress pre-lanzamiento de los equipos; y una mayor flexibilidad en los procesos de implementación [1]. Para lograr estos beneficios, un proveedor de software debe garantizar una cultura de colaboración entre todos los equipos involucrados en el proceso de entrega, el intercambio de conocimientos y herramientas entre los participantes, el establecimiento de métricas de medición y retrospectivas regulares para la mejora continua. Por lo tanto, los proveedores de software deben tomar ventaja de los principios de DevOps para aprovechar los beneficios de la entrega continua [24] y garantizar la definición de un proceso repetible y confiable para la liberación de software, la automatización de las actividades de desarrollo y operación, la definición de un proceso apropiado de Gestión de Configuración de Software (SCM, *Software Configuration Management*), la automatización de procesos de software “dolorosos” (*e.g.*, integración, prueba y lanzamiento), la definición de un proceso conveniente de aseguramiento de la calidad y una definición concreta de una funcionalidad de software “hecha” [1].

Como una respuesta a que no todas las organizaciones pueden desarrollar ese proceso de entrega continua “tal cual”, surgió un concepto similar, el de despliegue continuo, popularizado por Timothy Fitz [25], el cual es una alternativa para proveer constantemente valor agregado a los usuarios finales. La diferencia entre estos dos enfoques radica en que el proveedor de

software debe ser capaz de desplegar cada cambio que supera el respectivo test automatizado a producción para habilitar el despliegue continuo. Ambos métodos están ampliamente difundidos en las más grandes e influyentes compañías basadas en software del mundo (*e.g.*, Amazon, Facebook, Flickr, Google y Netflix) donde las nuevas funcionalidades (atómicas) o correcciones de errores se despliegan en periodos de tiempo muy cortos [7].

Nuestra investigación aunque provee un mecanismo automatizado de despliegue de software que contribuye a la entrega continua, no aborda directamente el proceso de implementación continua, pues ello requiere condiciones de desarrollo muy específicas para una aplicación exitosa e involucra otras metodologías, como la de integración continua.

INGENIERÍA Y ARQUITECTURA GUIADAS POR MODELOS (MDE / MDA)

Las practica de MDE surgieron en los años 80 con el fin de desarrollar una plataforma de alto nivel y abstracciones de lenguaje que permitieran mejorar la gestión de la complejidad del software [2]. Inspirados por las nuevas tendencias globales y buscando interoperabilidad, integración y estandarización, a inicios de 2000, el OMG propuso la MDA se compone de algunos estándares del modelado, tales como: UML, MOF, XMI y CWM (*Common Warehouse Metamodel*), los cuales conforman la base para establecer esquemas coherentes y unificados para autoría, publicación y gestión de modelos en una arquitectura guiada por el modelo [26].

MDA se dirige a derivar valor desde los modelos y a reducir la complejidad e interdependencia de los sistemas complejos, a través de la definición de la estructura, semántica y anotaciones de los modelos usando estándares de la industria. Usando estos modelos como vehículos de comunicación y habilitando derivaciones de ellos vía transformaciones automáticas, los modelos MDA se pueden utilizar para producir documentación, especificaciones del sistema, artefactos tecnológicos (*e.g.*, código fuente) y sistemas ejecutables. De esta manera, los modelos MDA promueven la ampliación de la agilidad de los procesos tradicionales en el ciclo de vida de los sistemas de software (SSLC, *Software System Life Cycle*) y beneficia la calidad y “mantenibilidad” de los productos resultantes [17]. Para aprovechar las capacidades de MDA, las primeras concepciones de las prácticas de MDA establecieron la necesidad de seguir al menos tres pasos: la definición de un PIM, que capture la funcionalidad y el comportamiento del negocio; la definición de un PSM, que particulariza

el PIM en una plataforma concreta; y la realización de transformaciones PSM, que permite producir los artefactos deseados [18].

Dado que en nuestra investigación se explotan los principios, fundamentos y pasos propuestos en MDE y MDA para lograr la automatización del proceso de implementación de software –es decir, la generación de especificaciones de implementación ejecutables que permitan la automatización–, a continuación se explican los conceptos de múltiples niveles de PIM y transformaciones modelo a modelo, ambos importantes para los enfoques MDE y MDA.

MÚLTIPLES NIVELES DE PIM

Los proyectos y contribuciones MDA son esperados para definir múltiples niveles de PIM, donde cada uno, excepto el nivel base, podría incluir aspectos de comportamiento tecnológico independientes de la plataforma [18]. Si bien el nivel base debe enfocarse en expresar únicamente la funcionalidad y el comportamiento del negocio, los siguientes si pueden incluir algunos aspectos (recurrentes) de la tecnología. Agregar estos conceptos a diferentes niveles de PIM, permite mapear un PSM de una manera más precisa [18]. La PIM definida en la presente investigación se puede considerar de dos niveles.

TRANSFORMACIONES MODELO A MODELO

Las transformaciones M2M (*Model to Model*) son parte esencial de los enfoques MDA, especialmente las transformaciones de modelos PIM a PSM. El OMG [27] definió un conjunto de características esenciales que constituyen la base de las transformaciones M2M para implementaciones MDA, así:

- Mapeo de modelos. Un mapeo provee especificaciones que habilitan la transformación de un PIM en un PSM. Hay tres clases de mapeos MDA: mapeo tipo, que especifica un mapeo desde modelos construidos usando tipos definidos in el lenguaje PIM a modelos expresados usando tipos de un lenguaje PSM; mapeo de instancias, donde ciertos elementos en el PIM se transforman de una forma particular, a través del uso de marcas –representaciones de un concepto en el PSM que puede ser aplicado a uno elemento del PIM–; y combinado (tipo e instancias). El mapeo se especifica a través de descripciones usando algún lenguaje –natural, de acción o de modelo de mapeo–, para describir la transformación de un modelo en otro.

- Grados y métodos de transformación de un modelo. Hay cuatro aproximaciones a la transformación de modelos: transformación manual, donde las decisiones de diseño se toman durante el proceso de desarrollo de un diseño que se ajuste a los requisitos de ingeniería en la implementación; transformación preparada de un PIM usando un perfil, donde se aprovechan los perfiles UML para preparar –y posiblemente marcar– un PIM para sus transformaciones; transformación usando patrones y marcas, donde las reglas especifican que todos los elementos del PIM que coinciden con un patrón particular se transformarán en instancias de otro patrón en el PSM; y transformación automática, donde no es necesario agregar marcas o usar datos de elementos adicionales (como perfiles), para poder generar el código.

En la presente investigación, para la transformación de PIM a PSM se utilizaron un tipo combinado y de mapeo de instancias, junto con un PIM preparado usando un perfil UML.

EL DSL AMELIA

Amelia es un lenguaje de dominio específico compacto (DSL, *Domain Specific Language*) cuya sintaxis y semántica están hechas a la medida de especificar y ejecutar flujos de trabajo de despliegue de sistemas software distribuidos. Amelia contiene, tanto expresiones declarativas e imperativas que facilitan el control del proceso de despliegue en general, como un control granular sobre las operaciones ejecutadas, en general. Amelia se basa en el lenguaje de expresión Xbase [28] y está completamente integrada con los sistemas tipo Java. Esta integración permite, no solo la reutilización de código java existente, sino también la extensión de la librería base de Java.

Una especificación Amelia es ambas cosas, la descripción de un subsistema o una estrategia de despliegue. La primera es una unidad modular que representa la estructura general del (sub)sistema a desplegar y sus correspondientes operaciones de despliegue; una descripción de un subsistema está compuesta por subsistemas y depende de ellos. Esta última es una especificación de flujo de ejecución que indica cómo realizar las operaciones de implementación que permite, por ejemplo, volver a intentarlo, al encontrar una falla, o repetir sistemáticamente la misma implementación, algo útil para "calentar" un sistema antes de ejecutar pruebas de rendimiento. La implementación actual de Amelia se puede utilizar como un compilador independiente o como un complemento

(*plug-in*) de Eclipse. A partir de una especificación, el compilador de lenguaje genera una aplicación Java ejecutable que resuelve automáticamente las dependencias e inclusiones del subsistema mientras se registra para las tareas de despliegue

Amelia es relevante en esta investigación porque proporciona los medios para generar especificaciones de implementación ejecutables compactas pero lo suficientemente potentes para sistemas de software distribuidos, que, en el contexto de nuestra investigación y contribuciones, se generan a partir de un diagrama de implementación UML definido por el usuario. Por lo tanto, los diagramas de implementación UML no solo se pueden usar para especificar una arquitectura de software sino también para generar las especificaciones ejecutables de Amelia que, en última instancia, permitirán desplegar los elementos descritos en la arquitectura.

Amelia es relevante para esta investigación, ya que provee los medio para generar compactas, pero suficientemente poderosas especificaciones de implementación ejecutables para sistemas distribuidos de software, lo que, en el contexto de nuestra investigación, se genera a partir de un diagrama de implementación UML definido por el usuario. Por lo tanto, los diagramas de despliegue UML pueden no solo ser usados para especificar una arquitectura de software, sino también para generar las especificaciones ejecutables Amelia que, en última instancia, permitirán desplegar los elementos descritos en la arquitectura.

ESTADO DEL ARTE

AUTOMATIZACIÓN DEL DESPLIEGUE DE SOFTWARE

Ketfi y Belkhatir [29] proponen DYVA, un marco de trabajo unificado para despliegue dinámico y reconfiguración de sistemas software basados en componentes. El marco se basa en su propio metamodelo jerárquico (PIM) el cual provee una visión abstracta de un modelo de componente. Este metamodelo puede personalizarse en un modelo de componente específico—como OSGi— [30]. Los cambios en este modelo personalizado (PSM) desencadenan el despliegue o el proceso de reconfiguración, el cual es realizado por un administrador de despliegue o reconfiguración. Aunque este marco se concibió utilizando principios MDA, no hay evidencia del uso de

estándares de modelado OMG, como UML, MOF y XMI, lo que compromete la interoperabilidad.

Sampaio y Mendonça [31] introducen Uni4Cloud, un enfoque que le permite a un modelo, desplegar y configurar aplicaciones complejas en múltiples infraestructuras en nube automáticamente. Uni4Cloud se compone de tres módulos principales: *Service Modeler*, que le permite al usuario final modelar y configurar la aplicación en la nube para ser desplegada usando plantillas de máquinas virtuales previamente construidas; *Service Manager*, responsable de desplegar y manejar las aplicaciones modeladas; y *Cloud Adapter*, cuyo objetivo es que el *Service Manager* se acople libremente sin depender de un proveedor de nube específico. Dado que este enfoque se basa en plantillas preelaboradas para modelar aplicaciones en la nube, carece de ciertos conceptos del dominio de despliegue (como componentes de software concretos, artefactos y dependencias entre componentes) y no suple las preocupaciones de estandarización e interoperabilidad que originaron los principios de la MDA.

Ribeiro et al. [32] proponen una solución guiada por el modelo para un despliegue automático de servicios en ambientes de nube. En su enfoque, un usuario final debe diseñar dos diagramas de despliegue UML: un modelo general, independiente de un proveedor de nube, que contiene elementos tales como servicios, dependencias, sistemas operativos, máquinas virtuales y bases de datos, entre otros; y un modelo específico, que contiene aspectos particulares relacionados con la infraestructura en la nube, tales como la clave de acceso, el repositorio del servicio de máquina virtual y la instancia en la nube de la máquina virtual. Definidos estos modelos, el desplegador de software los provee como una entrada para el sistema. Esto desencadena un proceso automático de despliegue que incluye: interpretación del modelo general, creación de la pila de software, asignación de los servicios, asociación de los servicios, interpretación del modelo específico y generación del código requerido para la automatización del despliegue. Considerando los principios MDA, aunque este enfoque está basado en especificaciones UML (uno de los estándares centrales en MDA), constructos de lenguaje, reglas de sintaxis and semánticas para diagramas de despliegue definidos en las especificaciones UML 2.x, no las reúne completamente. Además, las complejas interdependencias entre los componentes software pueden carecer de soporte.

La revisión permite concluir que: solo hay unas pocas soluciones para el despliegue automatizado de software basadas en el metamodelo UML v2.0

o superior, publicadas; hay poca evidencia de que las técnicas de modelado OMG estandarizadas, tales como MOF o XMI, se estén utilizando para impulsar la automatización de este proceso; hay poca evidencia de que los enfoques basados en modelos para el despliegue automático de software estén aprovechando una especificación de arquitectura de software para permitirla. Con respecto a las especificaciones para la implementación de software, de acuerdo con la revisión, la mayoría de los enfoques se basan en especificaciones propias (aunque bien desarrolladas) en lugar de usar estándares; solo unos pocos trabajos se basaron en estándares bien conocidos, como la topología OASIS y la especificación de orquestación para aplicaciones en la nube (TOSCA) y diagramas de implementación OMG UML.

REVISIÓN SOBRE PERFILES UML PARA DESPLIEGUE DE SOFTWARE

Jurjens [33] propone UMLSec, un perfil UML que permite la inclusión de información relevante de seguridad dentro de los diagramas en una especificación del sistema, facilitando así el desarrollo de sistemas críticos para la seguridad. Este perfil está destinado a proporcionar los medios para evaluar los aspectos de seguridad del diseño de un sistema. En este enfoque, los estereotipos, los valores etiquetados y las restricciones se utilizan para abordar: los escenarios de amenazas; y los requisitos, conceptos, mecanismos y primitivas de seguridad. Además, la seguridad física subyacente de los nodos informáticos se aborda mediante la aplicación de elementos de perfil a los diagramas de despliegue. Kallel et al. [34], con un enfoque similar, proponen un perfil UML que implementa su metamodelo MDS4MAS, el cual proporciona los conceptos necesarios para modelar sistemas seguros de agentes móviles. Estas dos extensiones livianas ofrecen valiosas definiciones y conceptos de seguridad que se pueden aprovechar en las especificaciones para el despliegue de sistemas de software distribuidos.

Apvrille et al., [35] presentan TURTLE-P, un perfil UML orientado a extender las semánticas de implementación UML para permitir la validación formal de sistemas críticos y distribuidos. El perfil permite la definición de descripciones detalladas de arquitecturas de comunicación, incluidos parámetros de calidad de servicio (QoS, *Quality of Service*), tales como *delay* y *jitter*, y un diseño unificado de componentes hardware y software con componentes UML extendidos y diagramas de despliegue. Si bien TURTLE-P no cuenta con información relevante para habilitar completamente la automatización del despliegue de

software, si incluye dos mejoras relevantes en los diagramas de despliegue que tienen el potencial de influir en los esfuerzos futuros: multiplicidad a nivel de nodo, es decir, el posible uso de un solo nodo gráfico para describir varios nodos físicos en los que se ejecutan varias instancias de los mismos componentes de software; y enlaces de comunicación dirigida, donde las relaciones de comunicación se definen en términos de componentes interfaces y no en términos de enlaces entre nodos.

Hughes y Løvstad [36] proponen capturar la estructura y las propiedades de rendimiento en las especificaciones de despliegue de software que permiten cuantificar la escalabilidad arquitectónica para sistemas distribuidos. Una de sus propuestas más significativas se basa en ajustar la resolución del modelo de despliegue de acuerdo con el contexto de diseño. Por lo tanto, la granularidad de los elementos del despliegue queda bajo el control del arquitecto o modelador. Aunque no se trata propiamente de un perfil UML, esta visión puede ser útil en la mayoría de los mecanismos de extensión para UML.

Esta revisión permitió comprender que los perfiles se han utilizado para adaptar la semántica UML a dominios como la atención médica, la optimización del consumo de energía, la seguridad, la validación formal y los sistemas de software basados en agentes. Si bien los diagramas de implementación han desempeñado un papel importante en estos logros, hay poca evidencia de perfiles UML que aborden directamente el despliegue de software, especialmente cuando estas tareas se automatiza, como en DevOps y entornos de entrega continua.

DESPLIEGUE DE SOFTWARE EN CONFIGURACIONES DE ENTREGA CONTINUA Y DEVOPS

Como se indicó, para poder cumplir con los requisitos cambiantes de los clientes, las prácticas de entrega continua requieren poner a disposición nuevas funciones –o nuevo software– tan pronto como se desarrollen, lo que implica garantizar que el código fuente esté siempre en un estado desplegable. La satisfacción de los clientes se consigue a través de la entrega oportuna y continua de software que tiene valor. Este objetivo está alineado con el movimiento DevOps, donde se necesita de la mayor colaboración de todos los involucrados en la entrega de software para liberar software con valor, de una manera rápida y confiable [1].

Las organizaciones que adoptan los principios y prácticas que dieron origen a las culturas de entrega continua y DevOps frecuentemente despliegan software –o nuevas funcionalidades– cientos, incluso miles, de veces por día [37]. Además, dado que la mayoría de las aplicaciones o funcionalidades del mundo de hoy, independiente de su tamaño, se componen de diferentes partes, (*i.e.*, componentes, módulos, archivos de configuración, etc.), cada despliegue tiende a ser complejo y propenso a fallas humanas [1]. En consecuencia, las organizaciones que implementan estas metodologías deben evitar realizar manualmente las actividades de despliegue, para así se puedan realizar de manera rutinaria y con bajo riesgo a lo largo del tiempo [37].

Los requisitos de alto nivel para el diseño y la implementación del mecanismo previstos en el proyecto para automatizar el despliegue de software en configuraciones de entrega continua y DevOps –especialmente en las relacionadas con el desarrollo de sistemas distribuidos basados en componentes–, se presentan, junto con su justificación, en la TABLA 1.

Tabla 1. Requerimientos de diseño e implementación previstos

Requisito	Lógica
Un mecanismo de despliegue debe automatizar la instalación de software en un entorno de destino. Esta actividad implica transferir artefactos de software producidos en la actividad de lanzamiento y configurarlos para que se ejecuten en una plataforma específica.	Por lo general, el software se implementa en entornos cuyos miembros difieren en tipo y configuración, lo que resulta en una actividad de instalación compleja y propensa a errores. La automatización de esta actividad garantiza un resultado predecible del proceso de implementación.
Un mecanismo de despliegue debe automatizar la activación de los componentes ejecutables instalados en un entorno de destino.	La activación de componentes requiere considerar múltiples factores, entre otros: dependencias, tiempo de espera de activación, parámetros de ejecución, puertos de comunicación. Automatizar la activación permitirá despliegues replicables y predecibles.
Un mecanismo de despliegue debe automatizar la actualización del estado del sistema desplegado.	Volver a implementar un sistema o una parte de él es un aspecto esencial para entregar valor a los clientes. Debe estar disponible tan pronto como sea necesario, ya que permite a las empresas verificar la utilidad de las características y las correcciones. El proceso de re-despliegue desencadena un flujo de retroalimentación que se utiliza para la toma de decisiones. La automatización es entonces crítica para garantizar la posibilidad de réplica y la consistencia en cada entrega.

Tabla 1. Requerimientos de diseño e implementación previstos (continuación)

Requisito	Lógica
Un mecanismo de despliegue debe ser barato, fácil de probar y no depender de la experiencia de ningún individuo.	Debe evitarse la producción y el seguimiento de una documentación extensa y detallada que describa los pasos que se deben seguir para garantizar el estado correcto de un sistema implementado; en cambio, las actividades de despliegue se deben realizar mediante un mecanismo confiable, rastreable y fácilmente repetible, que evite, tanto la pérdida de tiempo en la depuración de errores de despliegue, como la dependencia de las pruebas manuales para confirmar la ejecución correcta de la aplicación.
Las especificaciones de despliegue deben considerar los aspectos físicos y lógicos de los sistemas de software.	Los sistemas software son entidades complejas que deben entenderse desde diversas perspectivas para así caracterizarlos integralmente.
Las especificaciones de despliegue deben ser una representación exacta e integrada del estado actual de un sistema implementado.	En vista de que los cambios son frecuentes y el mantenimiento de la documentación es una tarea compleja, que requiere tiempo y, generalmente, la colaboración de varias personas, la información sobre el estado actual de un sistema desplegado tiende a ser incompleta o estar desactualizada. Lograr mantenerla completa y actualizada le permite a cada participante en el ciclo de vida del software, comprenderlo y prever los impactos de cambios futuros.

UN ENFOQUE GUIADO POR EL MODELO PARA AUTOMATIZAR EL DESPLIEGUE DE SOFTWARE

USANDO ESPECIFICACIONES DE DESPLIEGUE BASADO EN EL MODELO

Los modelos se utilizan en diferentes áreas de la ingeniería de software como un medio para aliviar la complejidad de las plataformas y expresar conceptos de dominio de manera efectiva, mediante la abstracción de elementos seleccionados de sistemas complejos [2]. Son útiles, tanto para entregar valor como para promover una mejor interacción y colaboración entre organizaciones, personas, hardware y software. Los modelos se pueden aprovechar para mejorar la agilidad de los (sub) procesos del ciclo de vida del software y su despliegue [17]. En el proyecto se utilizaron modelos UML para describir las especificaciones de despliegue, lo que genera valor de varias maneras [17], como se explica en los siguientes párrafos.

Modelos como vehículos de comunicación. Una vez que se define una arquitectura de software utilizando un diagrama de despliegue UML, este diagrama puede facilitar al equipo de desarrollo el consenso el sistema. El modelo que especifica la arquitectura se puede aprovechar para establecer el curso de otros procesos del ciclo de vida y mantener una visión unificada y práctica del estado actual del sistema, una vez que se ha desplegado o actualizado.

Derivación vía transformación automática. Siguiendo los principios y prácticas de la MDA, las especificaciones de despliegue ejecutables se pueden derivare automáticamente de un modelo de implementación UML que detalla una arquitectura de software. Esta transformación automática reduce el tiempo y el costo de realizar y mantener la implementación de un sistema, y promueve la coherencia entre los artefactos derivados. Esto es crucial para la automatización de las actividades de despliegue.

Análítica del modelo. Los arquitectos de software podrían aprovechar la analítica del modelo –validación, estadísticas y métricas– para descubrir información valiosa para la toma de decisiones, el monitoreo y la evaluación de la calidad. Este tipo de análisis es esencial en las primeras etapas del ciclo de vida del desarrollo de software (como ocurre en la metodología ATAM [38]), y cuando se requieren cambios frecuentes del sistema para satisfacer las necesidades del cliente, como en el caso de la entrega continua y los entornos DevOps.

ESPECIFICANDO DESPLIEGUE DE SOFTWARE EN UML

Aunque UML define una unidad específica para despliegues, el enfoque del proyecto combina múltiples unidades de lenguaje para definir una vista integrada que se adapte mejor a las especificaciones de los sistemas de software distribuidos basados en componentes. De esta manera, como se muestra en la FIGURA 4, los diagramas de implementación se componen de constructos de despliegue, componentes, estructuras compuestas, clases y unidades de lenguaje Kernel, lo que permite a los usuarios abordar, no solo las vistas físicas y estructurales de un sistema de software, sino también su lógica, lo que facilita el entendimiento pleno del sistema en cuestión.

Con base en la especificación UML 2.5 [14], el conjunto de constructos usado para las especificaciones de despliegue basadas en UML de la propuesta del proyecto se describe en la TABLA 2.

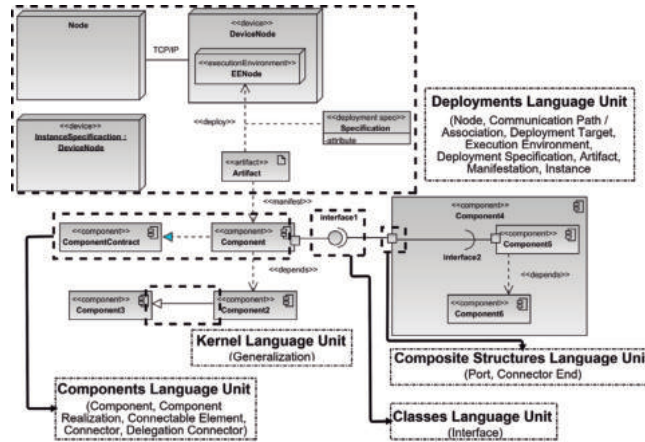


Figura 4. Constructos UML para especificación de despliegue de software

Tabla 2. Descripción de los constructos UML usados en el proyecto

Constructo	Unidad de lenguaje	Descripción
Artefacto	Despliegues	Pieza física de información que se usa o se produce en el proceso de despliegue.
Nodo	Despliegues	Recurso computacional donde se despliegan los artefactos (en el proyecto se utilizan dos especializaciones de este constructo: dispositivo, recurso físico computacional con capacidad de procesamiento; y entorno de ejecución, ambiente para la ejecución de componentes específicos).
Especificación de despliegue	Despliegues	Conjunto de propiedades que define el despliegue de un artefacto en un nodo
Asociación	Despliegues	Vínculo establecido entre dos nodos
Dependencia	Clases	Relación que indica que un elemento requiere de otro para su especificación o implementación
Despliegue	Despliegues	Relación que indica la asignación de un artefacto a un objetivo de despliegue (nodo).
Manifestación	Despliegues	La representación física concreta de uno o más elementos del modelo para un artefacto.
Componente	Componentes	Parte modular reemplazable de un sistema
Puerto	Estructura compuesta	Propiedad que especifica un punto de interacción entre un clasificador y su ambiente o sus partes internas.
Interface	Clases	Declaración de un conjunto de características públicas coherentes y obligaciones.

EL MODELO DE EJECUCIÓN DE URANO

En el proyecto se propone un enfoque basado en UML para la automatización del despliegue de software, basado en los principios y prácticas de MDA, para generar especificaciones de despliegue ejecutables, escritas en el DSL Amelia, a partir de las especificaciones de despliegue de UML. En la FIGURA 5 se presenta una visión general del diseño global del modelo de ejecución de Urano. El modelo de ejecución del proyecto permite automatizar las actividades de despliegue mediante la generación automática de especificaciones de despliegue ejecutables.

La FIGURA 5 también muestra la interacción de los usuarios con Urano: primero, un usuario define un diagrama de despliegue UML, donde se especifica el estado actual de una arquitectura de software y su despliegue asociado, para lo que se aplica un perfil UML que extiende la semántica de despliegue para proporcionar a los usuarios la expresividad necesaria para especificar las características principales del entorno donde se desplegará el sistema y los componentes y artefactos que lo conforman; una vez que el perfil se ha aplicado en el modelo de despliegue, los usuarios pueden enviarlo a Urano, cuando el sistema se vaya a implementar por primera vez o cuando sea necesario lanzar cambios en entornos de prueba, integración o producción; esto desencadenará un proceso de transformación de M2M, que implica transformar el diagrama

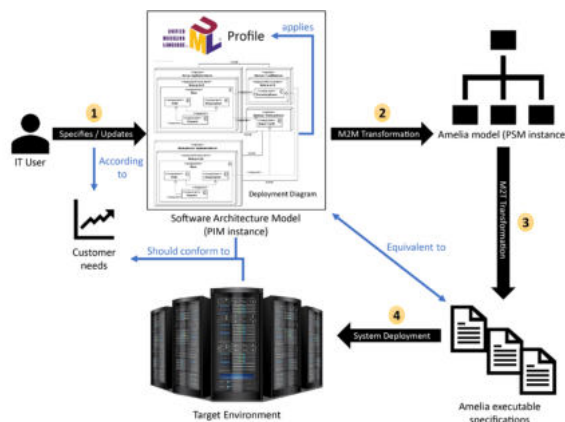


Figura 5. Modelo de ejecución de Urano: (1) especificación / actualización de la arquitectura de un sistema; (2) transformación M2M; (3) Transformación M2T; (4) Despliegue en un entorno de destino

de implementación extendida (instancia PIM) en un modelo (instancia PSM) que abstraiga los constructos definidos por Amelia; luego, el modelo pasa por un proceso de transformación de Modelo a Texto (M2T), el cual permite generar las especificaciones de despliegue ejecutables basadas en Amelia, que finalmente permitirán desplegar el sistema en un entorno objetivo.

IMPLEMENTACIÓN

En la construcción de Urano, la herramienta de despliegue automático de software basado en UML desarrollada en el proyecto, se utilizaron diversos componentes del proyecto de modelado Eclipse [39] en particular: Eclipse Modeling Framework (EMF) [40], que proporciona herramientas y soporte en tiempo de ejecución para permitir la implementación de las especificaciones modelo descritas en XMI; Eclipse Papyrus [41], un entorno de modelado altamente personalizable que permite crear y manipular diferentes tipos de modelos (incluidos aquellos basados en UML y EMF); Eclipse UML2 [42], que proporciona una implementación del metamodelo UML 2.5 basada en EMF; y Eclipse Acceleo [43], un generador de código que implementa el estándar *Model to Text Language* (MTL) [44] para generar cualquier tipo de código a partir de modelos basados en EMF. Dado que la plataforma Eclipse proporciona un entorno de código abierto, robusto, integrado y bien soportado para proyectos basados en MDA, se decidió usarlo, tanto para la implementación como para el despliegue del prototipo desarrollado en el proyecto.

Aunque podría usarse como una aplicación independiente, la implementación actual de Urano se ha personalizado para el IDE de Eclipse, a través de la definición de un conjunto de *plug-ins* que se pueden instalar como una característica de Eclipse, para así proveer un entorno integrado para el despliegue de software. La instalación de Urano solo requiere agregar el sitio de actualización correspondiente [45] y seguir las instrucciones de instalación [46].

Esta implementación de Urano comprende 2.866 SLOC (*Source Lines Of Code*) lógicos ejecutables, sin incluir el código fuente generado (la implementación, en términos de SLOC físico ejecutable, comprende 2,932 líneas de código) –lo que se determinó usando LocMetrics [47]–, distribuido en siete proyectos. Los detalles de la implementación y las funcionalidades más importantes de los principales proyectos se describen en las siguientes secciones y un resumen de ellas se presenta en la TABLA 3. En la FIGURA 6 se muestra cómo estos proyectos

se relacionan con la ejecución del modelo que se presentó en la descripción de Urano, así: (1) Extensión de diagramas de despliegue UML a través de la aplicación de nuestro perfil; (2) interpretación y procesamiento de diagramas de despliegue extendidos definidos por el usuario; (3) transformaciones M2M para instanciar el PSM basado en Amelia; y (4) transformaciones M2T para la generación de especificaciones de despliegue ejecutables, escritas en Amelia, desde la instancia de PSM.

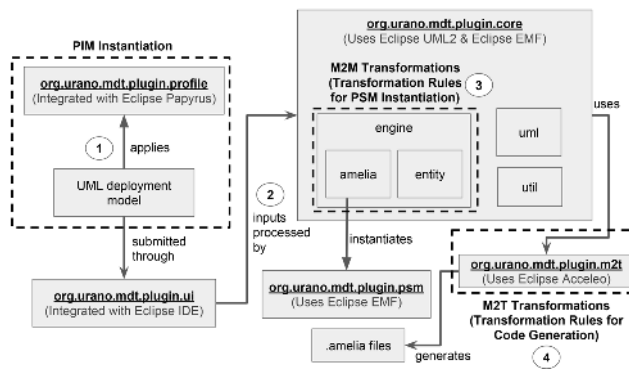


Figura 6. Relación de los proyectos con la ejecución del modelo

Tabla 3. Proyectos que componen la implementación de Urano

Eclipse ...	Proyecto	Descripción	SLOC
Sitios de actualización	org.urano.mdt.update.site.main	Permite organizar, construir y exportar la función Urano para poder instalarla en el IDE de Eclipse.	15
Características	org.amelia.dsl.feature	Permite agrupar los proyectos plug-in que componen Urano.	15
Plug-ins	org.urano.mdt.plugin.ui	Permite a los usuarios interactuar con Urano seleccionando el modelo UML que activará la generación de especificaciones de despliegue ejecutables basadas en Amelia.	81
	org.urano.mdt.plugin.core	Compuesto por las clases que permiten interpretar un modelo UML y la aplicación de perfil, permite generar una instancia del PSM, una transformación de M2M y coordinar la generación de textos basados en especificaciones ejecutables de Amelia — transformaciones M2T—. Es el proyecto principal, contiene el modelo de ejecución y organiza el proceso de generación.	2.416

Tabla 3. Proyectos que componen la implementación de Urano (continuación)

Eclipse ...	Proyecto	Descripción	SLOC
Plug-ins	org.urano.mdt.plugin. profile	Contiene el perfil UML propuesto para extender la semántica de los diagramas de despliegue y la definición del respectivo punto de extensión que habilita la integración del perfil con Eclipse Papyrus.	15
	org.urano.mdt.plugin. psm	Contiene el PSM utilizado en la cadena de transformación. Define un modelo de representación de las constructos de Amelia, que luego se instancian y transforman en especificaciones basadas en texto.	3.059
	org.urano.mdt.plugin. m2t	Contiene los módulos y plantillas de Acceleo que permiten generar especificaciones de implementación ejecutables basadas en Amelia desde el PSM.	324

DEFINICIÓN Y APLICACIÓN DEL PERFIL

DEFINICIÓN DEL PERFIL

Se utilizó el contexto de arquitectura de ingeniería de software de Eclipse Papyrus para desarrollar el perfil UML que extiende los elementos sintácticos y la semántica de los diagramas de implementación UML. El perfil se despliega como un complemento de Eclipse que se puede integrar en el entorno de modelado de Papyrus.

El perfil pretende mejorar las especificaciones de despliegue basadas en UML mediante la extensión de las propiedades (atributos) que pueden especificarse a partir de constructos en diagramas de despliegue. Para ello, se definió un conjunto de estereotipos UML agrupados en el perfil que pueden ser aplicados en elementos específicos del diagrama, tales como: componentes, artefactos, nodos, ambientes de ejecución, despliegue y declaración de relaciones y dependencias. Cada estereotipo define un conjunto de atributos (definiciones etiquetadas) que son heredados por los elementos que aplica el estereotipo (a este punto, las definiciones etiquetadas se conocen como valores etiquetados). Los estereotipos y sus respectivos valores etiquetados en el perfil habilitan, tanto el mejoramiento de las especificaciones de arquitectura de software basadas en UML –que comprenden conceptos físicos y lógicos detallados, no incluidos en

la especificación UML original—, como la automatización de las actividades de despliegue relacionadas con la arquitectura específica, ya que proporcionan suficiente información para lograrlo. En la FIGURA 7 (incluida en la siguiente página), se muestran los elementos (estereotipos y definiciones etiquetadas) definidos en el perfil propuesto para ampliar los diagramas de implementación de UML (para una descripción detallada del perfil, se recomienda consultar el Apéndice B en [48]).

APLICACIÓN DEL PERFIL EN ECLIPSE PAPYRUS

Como se describe en la FIGURA 6, la generación de especificaciones de despliegue ejecutables desde especificaciones basadas en UML inicia con la definición de un diagrama de despliegue (modelo) que aplica nuestro perfil. Este diagrama extendido de despliegue se introduce a Urano para iniciar el proceso de generación. En la FIGURA 8 se describe el proceso para aplicar y usar el perfil propuesto en Eclipse Papyrus, así: selección del perfil cuando se crea un diagrama de despliegue en un proyecto Papyrus (1); elemento raíz (modelo) con el perfil aplicado (2); aplicación de los estereotipos de perfil en elementos UML (3); y definición de valores etiquetados de estereotipo (nuevos elementos sintácticos y semántica respectiva) (4).

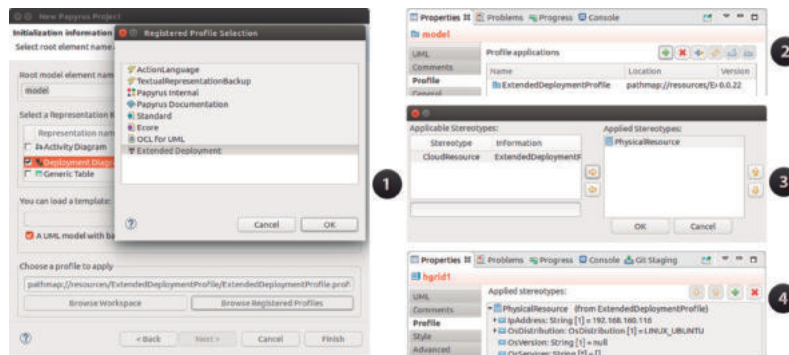


Figura 8. Aplicación de perfil en Eclipse Papyrus

INTERPRETACIÓN DE DIAGRAMAS DE DESPLIEGUE UML EXTENDIDOS

De acuerdo con el proceso de generación descrito en la FIGURA 6, en esta sección describe la manera cómo Urano lee (interpreta) los diagramas de

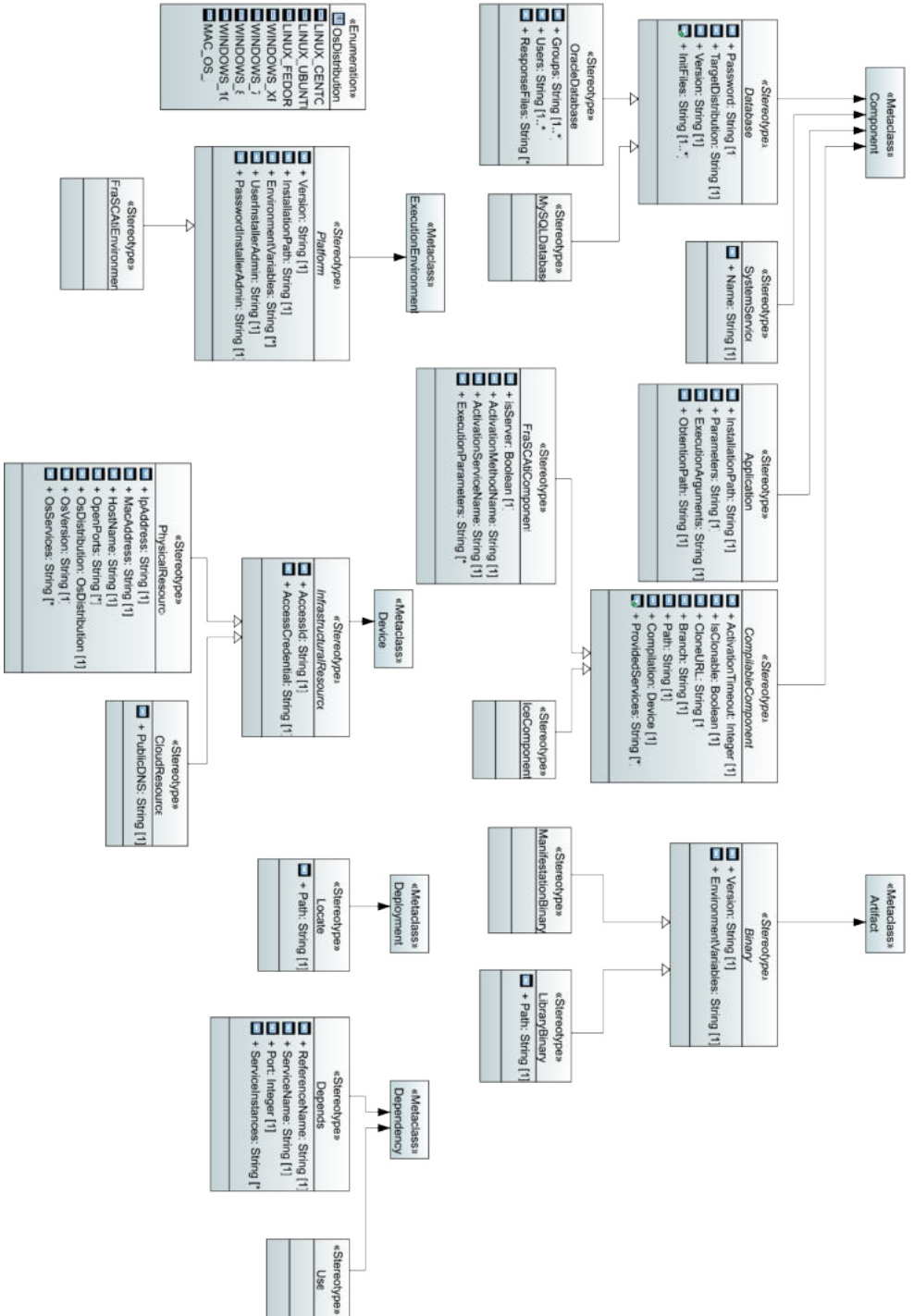


Figura 7. Perfil propuesto para extender diagramas de despliegue UML

despliegue extendidos: primero, se discuten los formatos y tecnologías usadas para almacenar y analizar los diagramas de despliegue UML; después, se explica cómo Urano procesa dichos diagramas.

ALMACENAMIENTO Y ANÁLISIS DE DIAGRAMAS DE DESPLIEGUE UML EXTENDIDOS

Por lo general, los diagramas de implementación basados en UML especificados por los usuarios se almacenan en archivos binarios propietarios, como por ejemplo en archivos .vpp de Visual Paradigm [49] y archivos .eap de Enterprise Architect [50]. Aunque el uso de este tipo de archivos obstaculiza la interoperabilidad y la estandarización, la mayoría de los entornos de modelado disponibles permiten a los usuarios exportar modelos UML utilizando el formato XMI de OMG, el mismo que está destinado a promover estas preocupaciones, permitiendo así las prácticas MDA. Los modelos descritos en Papyrus se almacenan automáticamente en un archivo basado en XMI con una extensión .uml .

Dado que XMI es un dialecto XML, es posible usar analizadores XML como un medio para recuperar información de los modelos UML. Por lo general, procesar esta información podría requerir su análisis en una implementación de metamodelo UML para procesar. En vista de que construir un metamodelo propio no es práctico (ver Apéndice A en [48]), se han propuesto varios enfoques para tratar directamente con el análisis XMI o para analizar la información de XMI en una implementación de metamodelo. Para recuperar y procesar información de diagramas de despliegue definidos por el usuario, se seleccionó la implementación del metamodelo UML proporcionado por el proyecto de modelado Eclipse (Eclipse UML2), porque: cumple con UML v.2.5 y ofrece soporte, tanto para versiones previas, como para la migración a futuras especificaciones; se actualiza constantemente, es ampliamente aceptado por la comunidad de modeladores –y ella es receptiva y colaboradora– [51]; es compatible con algunos de los ambientes comerciales y de código abierto más usados [52]; y provee un soporte robusto para la aplicación de perfiles en modelos UML.

En la FIGURA 9 se presentan las clases usadas por Urano, las cuales apalancan la implementación del metamodelo UML provista por Eclipse UML2 para extraer y manipular elementos específicos de un diagrama de despliegue UML. Para una visión completa de este diagrama, se recomienda revisar el Apéndice C de [48].

En las FIGURAS 10 a 13 se ilustra cómo un usuario interactúa con Urano para remitir el modelo de despliegue y desencadenar el proceso de generación. En la FIGURA 10, se define un diagrama de despliegue extendido, el cual especifica el despliegue de una aplicación cliente-servidor en la plataforma FraSCaTi [53], –un *middleware* para sistemas software basados en componentes–, donde el componente servidor expone un servicio de impresión a través de RMI y el componente cliente consume ese servicio para imprimir un mensaje en una salida estándar. Una vez el modelo se remite a Urano, se genera un conjunto de especificaciones de despliegue ejecutables que luego ejecuta el motor de Amelia, desplegando así, automáticamente, el sistema. En la secuencia de figuras de este proceso, la FIGURA 10 corresponde a la especificación de diagramas de despliegue UML extendidos en Eclipse Papyrus; la FIGURA 11, a la selección

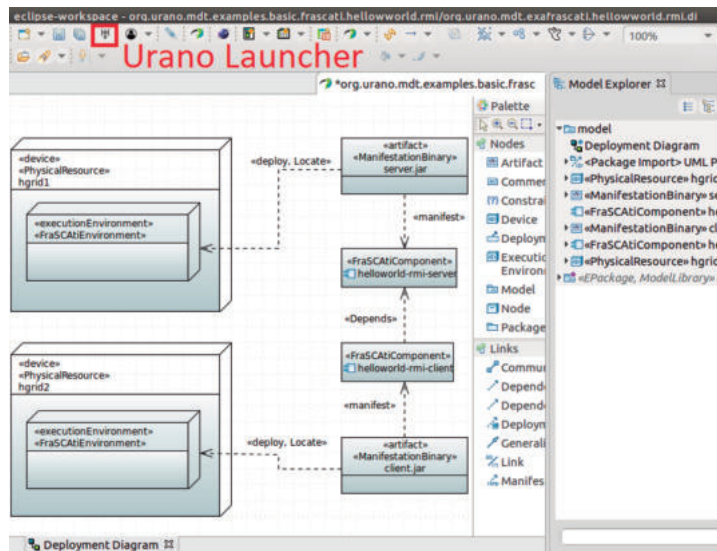


Figure 10. Interacción Urano-usuario: definición del diagrama de despliegue

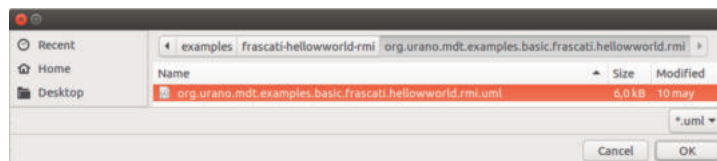
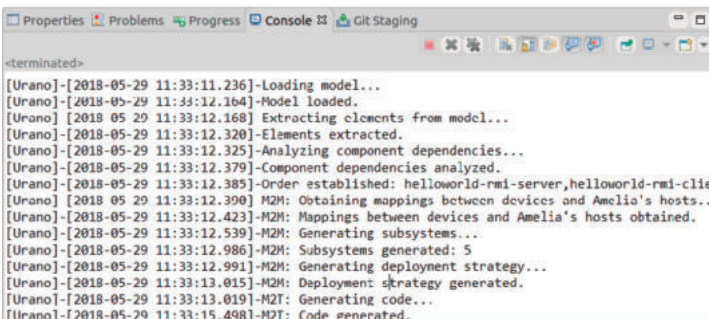


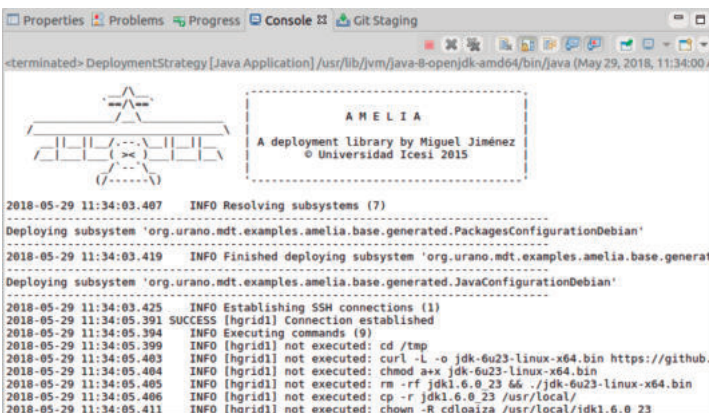
Figure 11. Interacción Urano-usuario: selección del modelo a procesar


del modelo a procesar; la FIGURA 12, a la ejecución de Urano en la consola Eclipse –esto es, la generación de las especificaciones de despliegue ejecutables del modelo remitido basadas en Amelia–; y la FIGURA 13, a la ejecución de las especificaciones de Amelia mostradas en la consola Eclipse.



```
<terminated>
[Urano]-[2018-05-29 11:33:11.236]-Loading model...
[Urano]-[2018-05-29 11:33:12.164]-Model loaded.
[Urano] [2018-05-29 11:33:12.168] Extracting elements from model...
[Urano]-[2018-05-29 11:33:12.320]-Elements extracted.
[Urano]-[2018-05-29 11:33:12.325]-Analyzing component dependencies...
[Urano]-[2018-05-29 11:33:12.379]-Component dependencies analyzed.
[Urano]-[2018-05-29 11:33:12.385]-Order established: helloworld-rmi-server,helloworld-rmi-clie
[Urano] [2018-05-29 11:33:12.390] M2M: Obtaining mappings between devices and Amelia's hosts...
[Urano]-[2018-05-29 11:33:12.423]-M2M: Mappings between devices and Amelia's hosts obtained.
[Urano]-[2018-05-29 11:33:12.539]-M2M: Generating subsystems...
[Urano]-[2018-05-29 11:33:12.986]-M2M: Subsystems generated: 5
[Urano]-[2018-05-29 11:33:12.991]-M2M: Generating deployment strategy...
[Urano]-[2018-05-29 11:33:13.015]-M2M: Deployment strategy generated.
[Urano]-[2018-05-29 11:33:13.019]-M2M: Generating code...
[Urano]-[2018-05-29 11:33:15.498]-M2M: Code generated.
```

Figure 12. Interacción Urano-usuario: la ejecución de Urano



```
<terminated> DeploymentStrategy [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (May 29, 2018, 11:34:00 A


A M E L I A  

    A deployment library by Miguel Jiménez  

    © Universidad Icesi 2015


2018-05-29 11:34:03.407 INFO Resolving subsystems (7)
-----
Deploying subsystem 'org.urano.mdt.examples.amelia.base.generated.PackagesConfigurationDebian'
2018-05-29 11:34:03.419 INFO Finished deploying subsystem 'org.urano.mdt.examples.amelia.base.generated.PackagesConfigurationDebian'
-----
Deploying subsystem 'org.urano.mdt.examples.amelia.base.generated.JavaConfigurationDebian'
2018-05-29 11:34:03.425 INFO Establishing SSH connections (1)
2018-05-29 11:34:05.391 SUCCESS [hgridl] Connection established
2018-05-29 11:34:05.394 INFO Executing commands (9)
2018-05-29 11:34:05.399 INFO [hgridl] not executed: cd /tmp
2018-05-29 11:34:05.403 INFO [hgridl] not executed: curl -L -o jdk-6u23-linux-x64.bin https://github.com
2018-05-29 11:34:05.404 INFO [hgridl] not executed: chmod a+x jdk-6u23-linux-x64.bin
2018-05-29 11:34:05.405 INFO [hgridl] not executed: rm -rf jdk1.6.0_23 && ./jdk-6u23-linux-x64.bin
2018-05-29 11:34:05.406 INFO [hgridl] not executed: cp -r jdk1.6.0_23 /usr/local/
2018-05-29 11:34:05.411 INFO [hgridl] not executed: chown -R cdloaiza /usr/local/jdk1.6.0_23
```

Figure 13. Interacción Urano-usuario: ejecución de las especificaciones de Amelia

TRANSFORMACIONES M2M

Como se presentó en la FIGURA 6, una vez el modelo definido por el usuario se carga en Urano, un conjunto de transformaciones M2M se desencadenan para instanciar una PSM, esto es, un modelo que abstrae los constructos Amelia. A continuación se presenta cómo se desarrolla este proceso en Urano, primero,

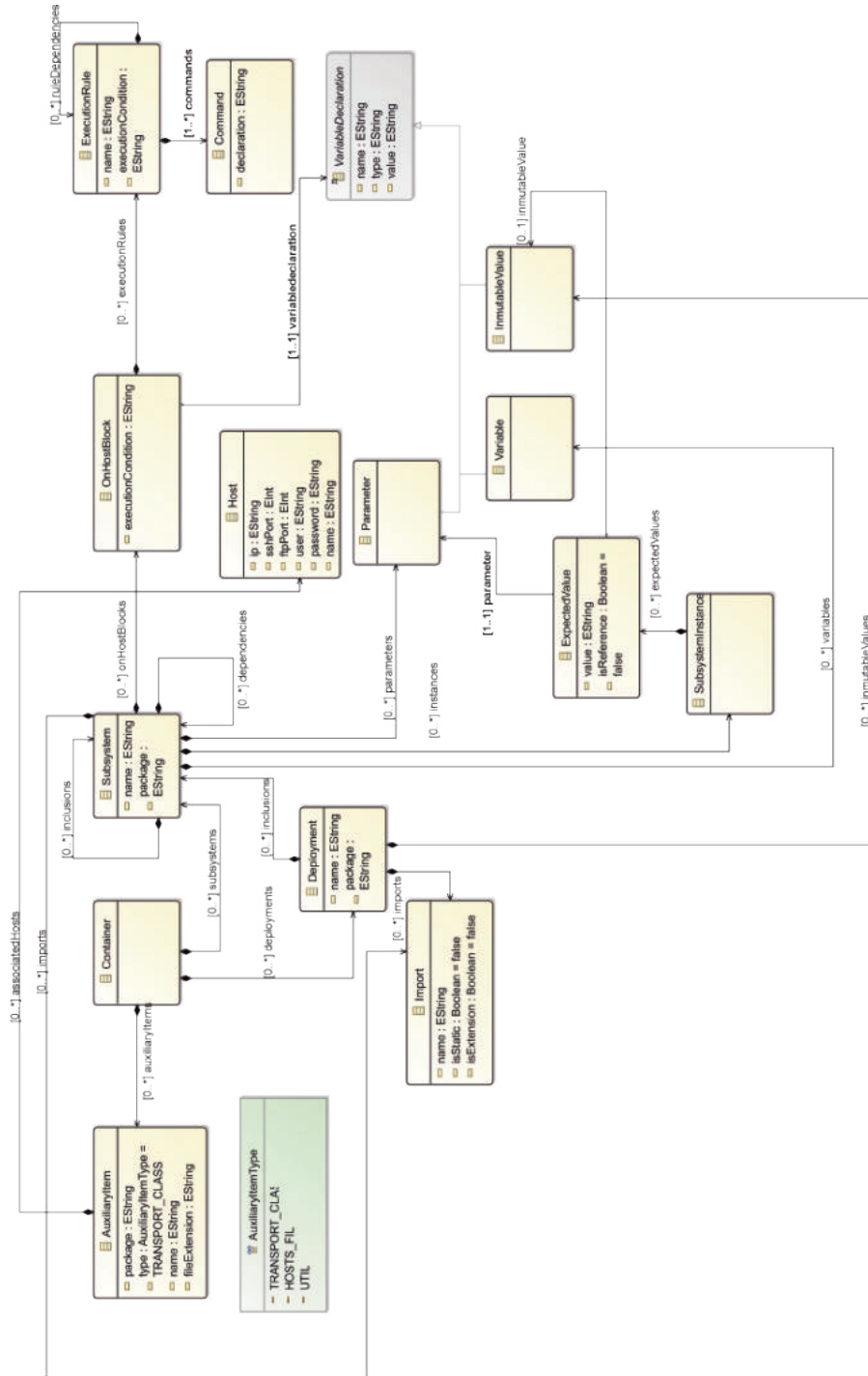


Figura 14. Modelo PSM de la implementación actual de Urano

una breve descripción del PSM basado en Amelia, después la descripción de las transformaciones.

EL PSM BASADO EN AMELIA

En la FIGURA 14 (ver página anterior) se presenta el modelo PSM usado para realizar la transformación M2M. Este modelo es una representación de los constructos de Amelia, el cual se almacena como un modelo ecore y se registra como un metamodelo en la plataforma Eclipse. En la TABLA 4 brevemente se describen las metaclases definidas en el modelo. Para producir una instancia de esta clase de modelos, el EMF provee dos mecanismos primarios: la instanciación estática, donde desde el modelo se generan unas clases Java

Tabla 4. Metaclases definidas en el PSM usado por Urano

Metaclase	Descripción
Container	Permite agrupar ítem auxiliares, subsistemas y despliegues, y así habilita la instanciación de los módulos Acceleo
AuxiliaryItem	Elemento que no forma parte de los constructos Amelia, usado para extender las operaciones del lenguaje
Subsystem	Unidad modular que describe la estructura general del (sub)sistema de despliegue y sus operaciones.
VariableDeclaration	Declaración de una variable que puede definirse en subsistemas o despliegues (<i>i.e.</i> , un parámetro, una variable o una variable inmutable)
Parameter	Variable requerida para la ejecución del subsistema.
Variable	Valor que podría cambiar durante el proceso de despliegue
ImmutableValue	Valor que no puede cambiar durante el proceso de despliegue (constante)
SubsystemInstance	Instancia de un subsistema definida en una estrategia de despliegue
ExpectedValue	Valor esperado en un parámetro por un subsistema en una ejecución exitosa
Host	Nodo de procesamiento de acuerdo con sus atributos de red (<i>e.g.</i> , IP, puertos SSH y FTP y nombre del host) que se debe utilizar para comunicación vía SSH y FTP
OnHostBlock	Elemento que agrupa reglas de ejecución que serán aplicadas a un host
ExecutionRule	Grupo de comandos (una actividad de despliegue) que se ejecutará en un host, dependiendo de ciertas condiciones
Command	Comando requerido para ejecutar una tarea de despliegue
Import	Librería requerida por un subsistema o una estrategia de despliegue
Deployment	Estrategia de despliegue que describe un flujo de ejecución que indica cómo desarrollar las operaciones de despliegue

que habilitan la instanciación; y la instanciación dinámica, donde lenguajes de transformación, como ATL [54], se usan para producir instancias de un modelo objetivo basadas en XMI, desde instancias de un modelo fuente, lo que implica que código no Java se genera desde el modelo objetivo para habilitar las transformaciones.

Para las transformaciones M2M de una instancia PIM a una instancia PSM, la implementación actual de Urano utiliza –aunque no se limita a–, el mecanismo de instanciación estática, el cual se seleccionó porque: permite registrar fácilmente modelos basados en EMF en la plataforma Eclipse; evita asumir el elevado nivel de la curva de aprendizaje del lenguaje ATL; y, como se utiliza Java, un lenguaje de propósito general, en el proceso de creación de instancias, da un mayor grado de control sobre las transformaciones y evita así, transformaciones intermedias y reglas de transformación muy complejas.

MAPEOS DEL MODELO

Fue definido un conjunto de mapeos y marcas del modelo para generar la instancia del PSM basado en Amelia (para un mayor detalle de ellas y su caracterización, se recomienda revisar el Apéndice D en [48]). Esta instancia se creó a partir de una especificación de despliegue basada en UML definida por el usuario, extendida mediante la aplicación de nuestro perfil. El motor de Urano implementado en el proyecto `org.urano.mdt.plugin.core` ejecuta los mapeos del modelo.

TRANSFORMACIONES M2T

Una vez Urano genera la instancia PSM, es decir, completa las transformaciones M2M de una instancia PIM (un diagrama de despliegue UML extendido instanciado de nuestro perfil) en un modelo que representa los constructos Amelia, se desencadena una transformación M2T y así se producen las especificaciones de despliegue ejecutables. A continuación se presentan el módulo *Acceleo* para la generación de los subsistemas de Amelia (CÓDIGO 1) y el módulo *Acceleo* para la generación de las estrategias de despliegue (CÓDIGO 2). Un archivo con plantillas para generar de código o consultas (*queries*) para extraer información de los modelos manipulados usadas para este propósito está disponible en [55] y las declaraciones *Acceleo* usadas para generar texto, en [56]. El conjunto completo de módulos se puede consultar en el repositorio del proyecto, en [57].

Código 1. Módulo Aceleo para la generación de los subsistemas de Amelia

```

1 [ comment encoding = UTF-8 / ]
2 [ module generateSubsystem('http://www.eclipse.org/emf/2018/Uranos')
3
4
5 [ template public generateElementSubsystem(aSubsystem :Subsystem) ]
6 [ file (aSubsystem.name.concat('.amelia'), false) ]
7 package [ aSubsystem._package / ]
8
9 // Subsystem imports
10 [ for (i:Import | aSubsystem.imports) ]
11 import [ if (i.isStatic = true) ] static [ / if ] [ if (i.isExtension = true) ]
    extension [ / if ] [ i.name / ]
12 [ / for ]
13
14 // Subsystem inclusions
15 [ for (s:Subsystem | aSubsystem.inclusions) ]
16 includes [ s._package / ]. [ s.name / ]
17 [ / for ]
18
19 // Subsystem depend encies
20 [ for (s:Subsystem | aSubsystem.depend encies) ]
21 depends on [ s._package / ]. [ s.name / ]
22 [ / for ]
23
24 subsystem [ aSubsystem.name / ] {
25
26 // Subsystem parameters
27 [ for (p:Parameter | aSubsystem.parameters) ]
28 param [ p.type / ] [ p.name / ]
29 [ / for ]
30
31 // Subsystem variables
32 [ for (v:Variable | aSubsystem.variables) separator ('\n') ]
33 var [ v.type / ] [ v.name / ] = [ v.value / ]
34 [ / for ]
35
36 [ for (o:OnHostBlock | aSubsystem.onHostBlock s) separator ('\n') ]
37 on [ o.variableDeclaration.name / ] [ if (o.executionCondition.size() > 0) ] ? [ o.
    executionCondition / ] [ / if ] {
38
39     [ for (er:ExecutionRule | o.executionRules) separator ('\n') ]
40 [ er.name / ] [ if (er.executionCondition.size() > 0) ] ? [ er.executionCondition
    / ] [ / if ] : [ for (rd:ExecutionRule | er.ruleDependencies) separator (',') ] [
    rd.name / ] [ / for ] [ if (er.ruleDependencies -> size() > 0) ] ; [ / if ]

```

Código 1. Módulo Acceleo para la generación de los subsistemas de Amelia (cont.)

```

41
42 [ for ( c : Command | er . commands ) ]
43 [ c . declaration / ]
44 [ / for ]
45 [ / for ]
46 }
47 [ / for ]
48 }
49 [ / file ]
50 [ / template ]

```

Código 2. Módulo Acceleo para la generación de las estrategias de despliegue

```

1 [ comment encoding = UTF -8 / ]
2 [ module generateDeployment ( ' http :// www . eclipse . org / emf /2018/ Urano ' ) ]
3
4
5 [ template public generateElementDeployment ( aDeployment : Deployment ) ]
6
7 [ file ( aDeployment . name . concat ( ' . amelia ' ) , false , ' UTF -8 ' ) ]
8 package [ aDeployment . _package / ]
9
10 // Deployment imports
11 [ for ( i : Import | aDeployment . imports ) ]
12 import [ i . name / ]
13 [ / for ]
14
15 // Deployment inclusions
16 [ for ( s : Subsystem | aDeployment . inclusions ) ]
17 includes [ s . _package / ] . [ s . name / ]
18 [ / for ]
19
20 deployment [ aDeployment . name / ] {
21
22 // Inmutable Variable definitions
23 [ for ( i : InmutableValue | aDeployment . immutableValues ) separator ( '\ n ' ) ]
24 val [ i . type / ] [ i . name / ] = [ i . value / ]
25 [ / for ]
26
27 // Shared Inmutable Variables
28 [ for ( s : Subsystem | aDeployment . inclusions ) ]
29 [ for ( i : SubsystemInstance | s . instances ) separator ( '\ n ' ) ]
30 [ for ( e : ExpectedValue | i . expectedValues ) ]

```

Código 2. Módulo Acceleo para la generación de las estrategias de despliegue (cont.)

```

31 [if(e.inmutableValue <> null and e.isReference = false)]
32 val [e.inmutableValue.type / ][e.inmutableValue.name / ] = [e.value / ]
33 [/ if]
34 [/ for]
35 [/ for]
36 [/ for]
37
38 [for (s :Subsystem | aDeployment.inclusions) separator ('\n')]
39 // Instances of subsystem:[s.name / ]
40 [for (i:SubsystemInstance | s.instances)]
41 add (new [s.name / ]([for (e:ExpectedValue | i.expectedValues)separator ('
    ')][if(e.inmutableValue = null)][e.value / ][else][e.inmutableValue.name / ][
    / if][ / for]))
42 [/ for]
43 [/ for]
44
45 start (true)
46
47 }
48
49 [/ file]
50 [/ template]

```

ARTEFACTOS GENERADOS

Siguiendo el ejemplo de generación que se presentó en las Figuras 10 a 13, en la Tabla 5 presentan los artefactos producidos por Urano.

Tabla 5. Artefactos generados por Urano

Artefacto	Descripción
hosts.txt	Archivo de texto que lista los host (nodos de proceso) donde se desplegaran los componentes, usado por Amelia para establecer conexiones SSH.
Util.java	Define un conjunto de extensiones de comando para Amelia, que le permite al usuario mejor comunicación mientras el sistema se está desplegando.
TransferHelper.java	Define un conjunto de extensiones de comando para Amelia, que habilita el transporte de artefactos a través del protocolo de SCP (<i>Secure Copy Protocol</i>).

Tabla 5. Artefactos generados por Urano (continuación)

Artefacto	Descripción
PackagesConfiguration.amelia	Subsistema de Amelia que permite instalar paquetes requeridos en nodos de procesamiento específicos, antes del despliegue del sistema.
JavaConfiguration.amelia	Subsistema de Amelia que permite instalar y configurar las distribuciones Java requeridas por la plataforma FraSCATi en los nodos de proceso especificados.
FraSCATiConfiguration.amelia	Subsistema de Amelia que permite instalar y configurar la plataforma FraSCATi en los nodos de procesamiento especificados.
HelloworldRmiClient.amelia	Subsistema de Amelia que permite compilar, instalar (<i>i.e.</i> , transferir y configurar) y activar el componente cliente especificado en el diagrama.
HelloworldRmiServer.amelia	Subsistema de Amelia que permite compilar, instalar (<i>i.e.</i> , transferir y configurar) y activar el componente servidor especificado en el diagrama.
DeploymentStrategy.amelia	Estrategia de despliegue Amelia que incluye (crea instancias) los subsistemas generados y permite orquestar sus actividades de despliegue de acuerdo con lo que se especifica en ellas.

EVALUACIÓN

El propósito de esta actividad es evaluar que el producto sea lo suficientemente expresivo como para permitir la generación de especificaciones de despliegue ejecutables de alta calidad que faciliten la automatización de la instalación, configuración y actualización de las actividades de despliegue, especialmente cuando se realizan en el dominio de sistemas software distribuidos basados en componentes.

Se realizaron tres estudios de caso con la complejidad suficiente para evaluar las principales funcionalidades de Urano y su posible aplicación en contextos industriales. El proceso de evaluación se basó en algunos de los criterios propuestos por Mohagheghi [58] en su método para la evaluación empírica de los desarrollos de ingeniería basada en modelos. Estos criterios se basan en preguntas de investigación dirigidas a objetivos de las empresas industriales en la adopción y aplicación de ingeniería basada en modelos. Estos criterios se clasifican según el modelo de aceptación de la tecnología (TAM, *Technology Acceptance Model*) [59], el cual aboga por considerar diversos factores (*e.g.*, utilidad percibida, facilidad de uso percibida, compatibilidad de soluciones.) al analizar cómo los usuarios adoptan nueva tecnología.

Los criterios propuestos por Mohagheghi se dirigen a los siguientes principios de MDE: modelos en todas partes, que establece que los modelos se utilizan en la mayoría de los procesos del ciclo de vida del software para mejorar, tanto la comunicación entre las partes interesadas, como la calidad del software, mediante el uso de modelos en análisis y pruebas tempranas; metamodelado, donde los modelos se conciben y utilizan para promover la interoperabilidad y la estandarización; múltiples niveles de abstracción y separación de preocupaciones en los modelos, que está relacionado con la práctica de los modelos que se utilizan para aliviar la complejidad de los sistemas software; y generación de artefactos a partir de modelos, donde los modelos representan una tecnología crucial para lograr la automatización y reducir el trabajo manual.

La evaluación del proyecto se centró en la obtención de información cualitativa útil para identificar si cumple o no con los siguientes principios de MDE: múltiples niveles de abstracción, separación de preocupaciones en modelos y generación de artefactos a partir de modelos. Además se consideraron los dos aspectos principales de la propuesta de Mohagheghi y Agedal para evaluar la calidad de los desarrollos de MDE: transformabilidad, que determina si los modelos tienen la capacidad de transformarse en otros modelos de mayor detalle y en fragmentos de código ejecutables; y modificabilidad, que establece que los cambios realizados en el modelo se reflejen correctamente en los artefactos generados.

En la TABLA 6 se presentan los criterios de evaluación seleccionados del enfoque de Mohagheghi, donde cada criterio se clasificó de acuerdo con el principio MDE al que se dirige, y los aspectos para evaluar la calidad en enfoques MDE propuesto por Mohagheghi y Agedal [60].

ESTUDIOS DE CASO DE DESPLIEGUE

Se seleccionaron tres estudios de caso para evaluar las capacidades de Urano en términos de los criterios de evaluación definidos en la sección anterior. Cada estudio de caso, una solución a un problema académico o industrial, se realizó en un entorno de desarrollo controlado; cada uno consta de una o dos configuraciones arquitectónicas que requieren la especificación explícita de compilación, instalación, configuración y activación de actividades de despliegue. Las especificaciones de UML, modeladas utilizando Eclipse Papyrus y aplicando nuestro perfil, usadas en estos casos de estudio, están disponibles en el repositorio de Urano [61].

Tabla 6. Criterios para la evaluación cualitativa

Criterio	Aspecto de calidad	Principio MDE	Definición
Calidad de los artefactos generados	Transformabilidad	Generación de artefactos desde modelos	La calidad del código y de la documentación generada desde modelos está de acuerdo con la comprensión y el cumplimiento de los estándares de codificación, para lo que se definió un conjunto de pautas de programación para Amelia, que debe seguir una apropiada especificación de despliegue escrita en este lenguaje.
Idoneidad	Transformabilidad	Múltiples niveles de abstracción y separación de preocupaciones en modelos	La solución puede resolver el problema en cuestión, es decir, un enfoque basado en modelos puede habilitar la automatización del despliegue de la especificación de una arquitectura de software.
Eficiencia	Transformabilidad	Generación de artefactos desde modelos	El tiempo requerido por una herramienta para desarrollar una tarea, es decir, cuánto tiempo demora Urano para producir especificaciones Amelia ejecutables para especificaciones UML definidas por el usuario
Facilidad de cambio	Modificabilidad	Generación de artefactos desde modelos, niveles de abstracción múltiples y separación de preocupaciones en modelos.	El esfuerzo requerido para hacer un cambio y generar los activos requeridos.

Las siguientes secciones detallan cada estudio de caso, brindan una visión general del problema abordado, presentan las configuraciones de implementación utilizadas para resolverlo y explican la manera en que se aplicaron y analizaron los criterios de evaluación en ellos. Para el desarrollo de las configuraciones de despliegue, en los tres casos, siguió un proceso similar al que se muestra en las FIGURAS 10 a 13.

Resumiendo, se creó un diagrama de despliegue en Eclipse Papyrus para cada configuración de despliegue, se aplicó el perfil propuesto al diagrama, se completó la especificación de despliegue requerida usando elementos definidos en el perfil (estereotipos y valores etiquetados), se remitió el diagrama extendido de despliegue a Urano, y se usaron las especificaciones ejecutables de Amelia DSL generadas a partir de nuestra herramienta para el despliegue automático

en la configuración determinada. En este punto, se aplicaron y analizaron los criterios de evaluación.

CASO DE ESTUDIO I: EL PROBLEMA DE LA MULTIPLICACIÓN DE CADENAS DE MATRICES

La multiplicación de cadenas de matrices (MCM, *Matrix-Chain Multiplication*) es un problema de optimización donde se trata de encontrar la secuencia más eficiente para multiplicar un conjunto de matrices dado. La solución hallada con las herramientas del proyecto divide el problema en tres subproblemas: el problema de multiplicación de pares de matrices, el problema de colocación de paréntesis en cadenas de matrices, que encuentra la secuencia óptima de multiplicaciones de pares de matrices minimizando el número de sumas y multiplicaciones individuales, y el problema de programación de la multiplicación de subcadenas de matrices, que encuentra subconjuntos de multiplicaciones de matrices que pueden ser realizados simultáneamente para disminuir el tiempo total de multiplicación [62].

Esta implementación de la solución MCM aprovecha los recursos computacionales distribuidos para reducir el tiempo de ejecución al multiplicar una gran cantidad de grandes matrices. Con este fin, se desarrollaron dos estrategias de multiplicación: una basada en la arquitectura *map-reduce*, otra, en una variación de ella que reduce significativamente el uso de la red. Al final, las multiplicaciones locales se realizan utilizando el algoritmo de Strassen. La implementación sigue una arquitectura orientada a componentes (SCA, *Service Components Architecture*) y se ejecuta con el *middleware* FraSCAti [63].

Este caso de estudio dispone de dos configuraciones (estrategias) de despliegue: la configuración BlockReduce, que consiste en dividir cada matriz en bloques de tamaño fijo –submatrices– y multiplicarlos como si fueran una celda en lugar de un grupo de ellos; y la configuración híbrida, que introduce una mejora en términos de uso de red, pero es más exigente en uso de procesador y memoria. En la Figura 15 se muestran los elementos involucrados en el despliegue de la estrategia BlockReduce, en ella, determinar el tamaño del bloque es crucial para equilibrar la cantidad de datos transmitidos a través de la red y el tamaño de los bloques para multiplicar (para reducir el tiempo de multiplicación). Como se muestra en el diagrama, la implementación de esta configuración para el problema MCM requiere que los implementadores instalen y configuren la plataforma FraSCAti en múltiples nodos de procesamiento, una actividad que implica, no solo descargar y extraer la distribución de plataforma especificada,

sino también configurar las variables ambientales y del sistema necesarias para su correcta ejecución [64]. Además, los componentes del multiplicador de matriz deben compilarse y configurarse de acuerdo con los nodos donde se ejecutarán y los nodos donde se activarán sus dependencias, es decir, las configuraciones de enlace). Después de eso, los artefactos que manifiestan los componentes compilados deben transportarse a los nodos correspondientes y activarse teniendo en cuenta el conjunto de dependencias entre los componentes del sistema.

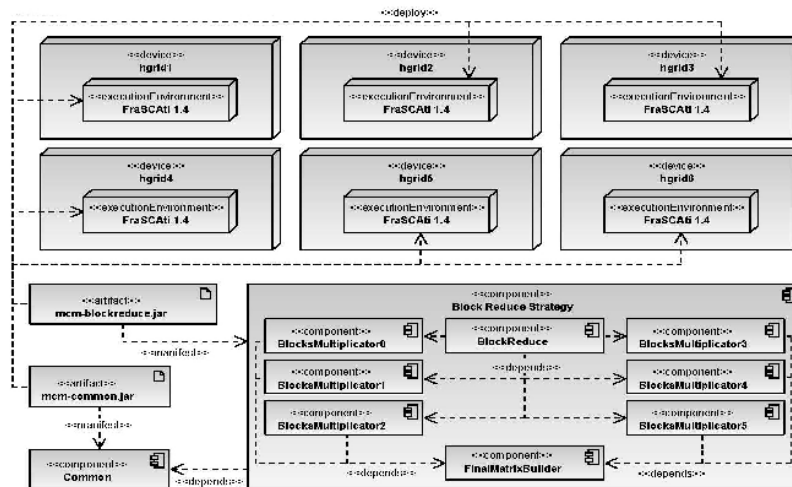


Figura 15. Configuración de despliegue BlockReduce para el caso de estudio MCM

CASO DE ESTUDIO 2: CLASIFICACIÓN DISTRIBUIDA

Los problemas de clasificación se usan con frecuencia en diversas áreas de las ciencias de la computación. Se definen como el arreglo de un conjunto de elementos en un orden específico. En este caso de estudio, se trata de clasificar una gran cantidad de datos a través de ordenamiento distribuido, esto es la distribución de las tareas de clasificación entre un conjunto de nodos de computación. Este caso se ha enfocado en una implementación distribuida de una algoritmo de combinación-ordenamiento para ordenar números expresados como potencia de dos de nodos organizados jerárquicamente (*e.g.*, $2^0=1$; $2^1=2$; $2^2=4$; $2^3=8$...). El arreglo de entrada que se debe organizar está particionado sucesivamente en mitades; cada nodo de una jerarquía hace un

ordenamiento parcial (de “su” parte) y se responsabiliza de combinar las partes ordenadas por los nodos inferiores inmediatos en la jerarquía. Este proceso se repite hasta que el nodo de mayor jerarquía combina las partes ordenadas producidas por sus nodos hijos inmediatos.

En la FIGURA 16 se aprecian los elementos involucrados en el despliegue de este caso de estudio. Tal como en las configuraciones de despliegue MCM, los componentes de ordenamiento y combinación basados en SCA (agrupados en componentes distribuidor) son desplegados en la plataforma FraSCAti corriendo en nodos de proceso distribuidos. Sin embargo, en este caso, componentes y dependencias adicionales, como Apache JMeter [65] y RabbitMQ [66], se deben desplegar para asegurar la correcta operación del sistema. Instalar y configurar esta clase de aplicaciones son actividades exigentes ya que se deben realizar múltiples operaciones propensas a error, para su correcta ejecución.

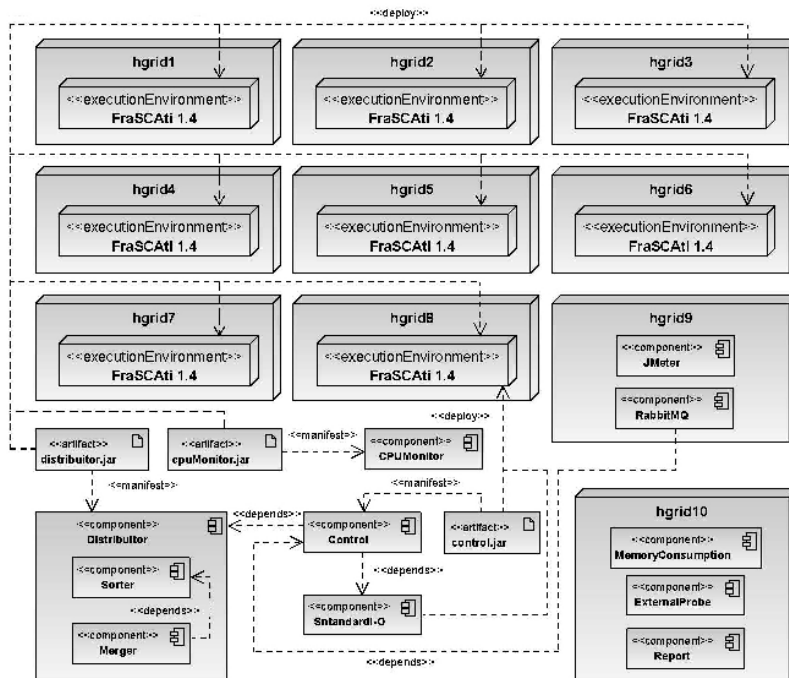


Figure 16. Caso de estudio: clasificación - configuración de despliegue

CASO DE ESTUDIO 3: PROCESAMIENTO DISTRIBUIDO DE GRANDES ARCHIVOS XML

Este estudio de caso industrial se refiere al problema de rediseñar la arquitectura de referencia para el motor central de una familia de productos software que procesa grandes archivos XML. La arquitectura de referencia rediseñada debe garantizar mejor rendimiento que los diseños arquitectónicos utilizados en productos anteriores de la familia. El sistema original, que fue diseñado para procesar documentos XML para diferentes dominios de aplicación, tiene una estructura monolítica. Sin embargo, es factible reestructurar este sistema en subsistemas distribuibles y escalables, para mejorar su rendimiento. Además, se pueden aplicar varios patrones de diseño –como productor-consumidor y reactor– para reestructurar y distribuir los subsistemas, generando así diferentes configuraciones arquitectónicas. Para cada una de estas configuraciones son posibles varias instancias, por ejemplo, variando el número de nodos de procesamiento esclavos, y cada una de ellas implica sus respectivos procesos de despliegue y ejecución, cuyas variaciones son complejas.

El procesamiento de archivos XML es una tarea común, intensiva en CPU, que soporta procesos centrales de negocios en un rango de dominios que va desde la transmisión y transformación de datos simples hasta la interoperabilidad de datos completos. En este caso de estudio, el procesamiento XML se utiliza para que los trabajadores del censo puedan recopilar datos demográficos en dispositivos móviles off-line en regiones donde no hay acceso a telecomunicaciones, y días o semanas después puedan sincronizar los datos censales recopilados con un servidor centralizado, a través de todo el país. Este proceso sufre retrasos graves y errores de tiempo de espera en la carga porque la gran cantidad de solicitudes sobrecarga el servidor central.

En este caso de estudio están disponibles dos configuraciones de despliegue: una que implementa el patrón de diseño Reactor, otra el patrón de diseño Productor-Consumidor. En la FIGURA 17 se presentan los elementos involucrados en el despliegue de la configuración basada en Reactor. De los casos de estudio, esta es la configuración con mayores demandas, toda vez que comprende dependencias complejas entre los componentes basados en SCA que constituyen el sistema, algunos despliegues de componentes en nodos de procesamiento que corren la plataforma FraSCAti y la instalación y configuración de componentes complejos, como la base de datos Oracle, la cual debe ser iniciada y restaurada previo a la ejecución del sistema; RabbitMQ, que debe activarse antes de lanzar los componentes del monitor; y Apache

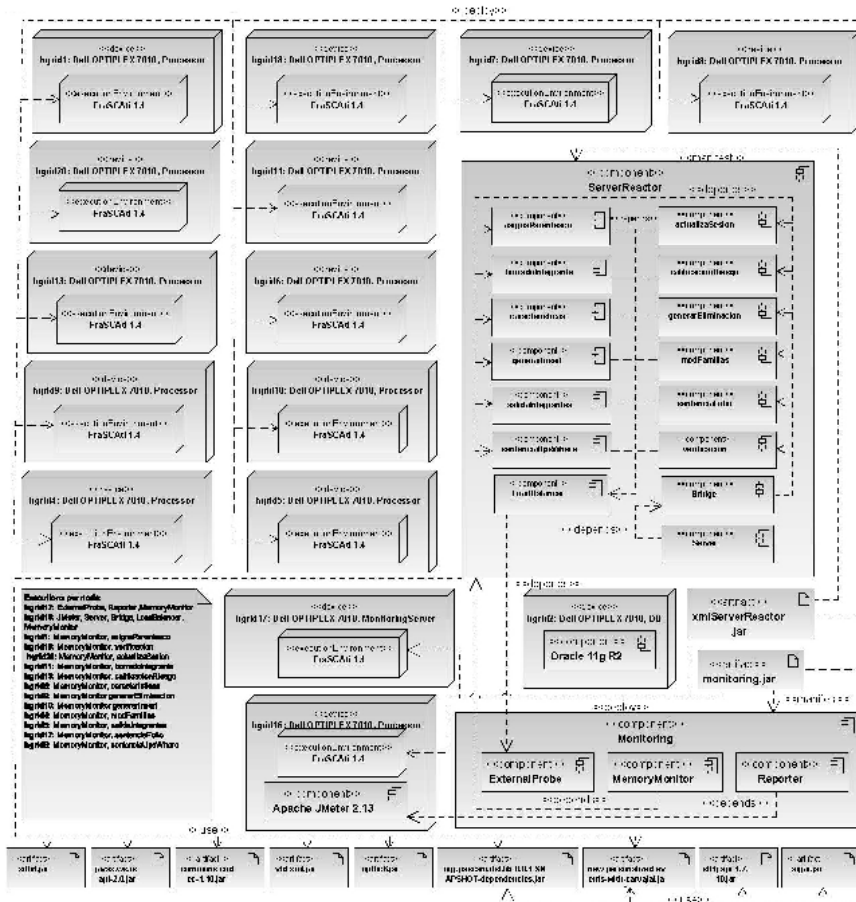


Figura 17. Caso de estudio: procesamiento de grandes XML - configuración para el despliegue Reactor

JMeter, que debe lanzar múltiples pruebas en el sistema de ejecución. Además, en este caso de estudio la mayoría de los componentes se basan en varias bibliotecas, lo que da lugar a una configuración y actividades de activación del despliegue complejas.

APLICACIÓN Y DISCUSIÓN DE LOS CRITERIOS DE EVALUACIÓN

CRITERIO DE CALIDAD PARA LOS ARTEFACTOS GENERADOS

Con el fin de evaluar este criterio se definió un conjunto de guías (GL, *Guide Line*) que buscan facilitar las especificaciones, tanto de despliegue, como el

mantenimiento del lenguaje Amelia. Estas pautas, incluidas en la Tabla 7, permiten evaluar la calidad de las especificaciones de despliegue ejecutables generadas por Urano.

Tabla 7. Estilo de programación para Amelia

ID	Descripción
GL 001	Los subsistemas son pequeños, cada uno encapsula el ciclo de vida de despliegue de uno de los elementos del sistema. Entre más pequeño es el subsistema, mayores son sus probabilidades de reutilización.
GL 002	Las cadenas usan interpolación variable en lugar de concatenación, lo que facilita la lectura del código, especialmente cuando la cadena es parte de una declaración de comando.
GL 003	Nombrar subsistemas y reglas de ejecución es relevante; los nombres son cortos pero descriptivos. Mezclar letras, números y símbolos no dice nada sobre el subsistema o sobre el significado de los comandos encapsulados por la regla.
GL 004	Los subsistemas no definen números o cadenas para el flujo de control, sino que hacen referencia a enumeraciones Java.
GL 005	Los subsistemas y variables se documentan utilizando comentarios similares a Java, las personas y herramientas ya están familiarizadas con ellas.
GL 006	Los subsistemas contienen la menor cantidad de variables posible
GL 007	Los subsistemas no contienen líneas largas, sin embargo, se prioriza la legibilidad sobre la longitud
GL 008	Los subsistemas usan métodos y extensiones Java para mejorar la legibilidad del código y, al mismo tiempo, promover su reutilización
GL 009	Los subsistemas usan parámetros e inclusiones para evitar duplicar código
GL 010	Los subsistemas usan instancias AtomicBoolean para controlar el flujo de ejecución y evitar la ejecución de comandos innecesarios.
GL 011	Los subsistemas no contienen rutas del sistema incrustadas directamente en el código fuente (hard-coded), pues ello dificulta el despliegue cuando cambia la ubicación del código fuente
GL012	Los subsistemas no declaran una cantidad excesiva de parámetros -hacerlo puede significar un nivel de abstracción inapropiado-, en cambio, se basan en los parámetros incluidos.

Para cada configuración de despliegue abordada en los casos de estudio, se evaluó si el código generado por Urano cumple con las pautas definidas en el estilo de programación. Si bien se encontró que la herramienta es totalmente compatible con la mayoría de las pautas (GL001-GL003, GL005, GL006 y GL008-GL012), el caso de estudio de procesamiento XML mostró que se podría mejorar el abordaje de las pautas GL004 y GL007. Primero, al generar el código para las configuraciones de despliegue de este caso de estudio, Urano

no aprovecha las enumeraciones para producir un flujo de control de ejecución claro, que permita tomar decisiones sobre los comandos que se ejecutarán, como por ejemplo, los comandos necesarios para inicializar la base de datos. Además, dado que algunos componentes de este caso de estudio requieren múltiples servicios de otros componentes, como por ejemplo, el componente del servidor, el código producido para compilar y configurar estos componentes contiene líneas de código relativamente largas. Mejorar estos problemas mejorará la legibilidad y con ello hará que el código sea más fácil de mantener.

LA IDONEIDAD DE LA SOLUCIÓN

Antes de abordar cada caso de estudio en particular, se realizó un conjunto de pruebas de transformaciones para evaluar el código generado por versiones anteriores de Urano y determinar si una herramienta basada en modelos podría ser completamente adecuada para automatizar la instalación, configuración, activación y actualización de actividades de despliegue. Estas pruebas se basaron en el caso de procesamiento XML, el más complejo de los tres. La expectativa con estas pruebas era depurar posibles errores y mejorar las especificaciones utilizando el perfil y los procesos de transformación y los artefactos generados.

La configuración de despliegue de cada caso de estudio fue especificada con éxito y desplegada automáticamente utilizando la versión actual de Urano. El perfil propuesto proporciona los medios para especificar y desplegar una arquitectura de software que combina vistas físicas (hardware), lógicas (software) y de comunicaciones. El perfil proporciona suficiente información al motor de Urano para producir artefactos (especificaciones de despliegue ejecutables) que permiten automatizar la instalación, configuración y actualización de las actividades de despliegue de sistemas software distribuidos basados en componentes heterogéneos, como los abordados en los casos de estudio. Además, el perfil propuesto es suficientemente completo como para permitir a los usuarios especificar, en detalle, componentes –del sistema, bases de datos, servicios de sistemas y aplicaciones–, dependencias, plataformas, nodos de procesamiento y artefactos, entre otros elementos utilizados en el despliegue de sistemas software.

EL CRITERIO DE EFICIENCIA

La Tabla 8 resume los tiempos tomados por Urano para generar las especificaciones de despliegue ejecutables a partir de los diagramas de

implementación UML definidos por el usuario de las diferentes configuraciones de implementación para cada caso de estudio. Como se puede observar, los tiempos son adecuados para los requisitos actuales del usuario para la implementación de software, especialmente en configuraciones de entrega continua y DevOps.

Tabla 8. Tiempos de ejecución / caso de estudio

Configuración de despliegue	Caso de estudio	Elementos del modelo procesados	Nodos procesados	Transformación M2M (segundos)	Transformación total (segundos)	Despliegue (segundos)
Estrategia BlockReduce	MCM	45	6	1	8.1	38
Estrategia híbrida	MCM	35	5	0.9	8	39
Estrategia original	Clasificación	56	11	2	47	164
Productora/ Consumidor	Procesamiento XML	64	8	2	49	174
Reactor	Procesamiento XML	86	15	2.8	71	210

EL CRITERIO DE FACILIDAD DE CAMBIO

Dado que en la configuración de entrega continua y DevOps los cambios son frecuentes, la actualización de software, una actividad que requiere volver a desplegar el sistema o la parte modificada de él, es una tarea crítica. La evaluación de este criterio implicó actualizar la parte de los diagramas de implementación de UML que representan cada configuración de despliegue para los casos de estudio y validar si los cambios se reflejaron correctamente en las especificaciones de despliegue ejecutables (re)generadas. La idea era evaluar si Urano pudo volver a implementar con éxito un sistema cuyos componentes deben reasignarse a diferentes nodos de procesamiento (cambiando así los enlaces).

Evaluar este criterio en el caso de estudio del problema de MCM implicó cambiar los nodos de procesamiento donde se despliegan los multiplicadores de matriz; en el caso de clasificación, se redespiegó el sistema una vez que se actualizaron los nodos de procesamiento donde se desplegaban los componentes del distribuidor; el caso de estudio de procesamiento de XML presentó el

escenario más exigente para evaluar este criterio, debido a las complejas dependencias entre los componentes que forman parte de las configuraciones de despliegue Productor-Consumidor y Reactor, al igual que con los estudios de caso anteriores, se modificó la forma en que algunos componentes, tales como *ServerReactor*, *Reporter* y *ExternalProbe*, se desplegaban en los nodos de procesamiento.

En todos los casos Urano permitió cambiar fácilmente las relaciones de despliegue, simplemente cambiando el objetivo de una relación de despliegue en el diagrama, por lo tanto, los componentes pueden redespigarse en un nodo informático diferente (*e.g.*, de acuerdo con las políticas o decisiones del usuario) modificando solo la especificación de arquitectura basada en UML del diagrama de despliegue. En cada caso, una vez que el modelo actualizado se envió a Urano, la herramienta produjo un nuevo conjunto de especificaciones de despliegue ejecutables que cumplían con los nuevos requisitos de despliegue especificados por el usuario.

CONCLUSIONES Y TRABAJO FUTURO

En esta investigación se abordó el desarrollo de un mecanismo para realizar la automatización del despliegue de software mediante la transformación de diagramas de despliegue UML –especificando una arquitectura de software–, en especificaciones de despliegue ejecutables. Para su resolución, se desarrolló Urano, un mecanismo de automatización de despliegue basado en modelos, que le permite a los desarrolladores de software producir especificaciones ejecutables a partir de una especificación de arquitectura basada en modelos, y automatizar así las actividades de despliegue.

Además de esta contribución, durante la investigación se caracterizaron los mecanismos y especificaciones de despliegue y se propuso un perfil que extiende los elementos sintácticos de despliegue UML –y la semántica correspondiente–, para proporcionar a los usuarios de Urano diagramas con la suficiente expresividad para especificar el despliegue de sistemas distribuidos, basados en componentes.

Se evaluó Urano realizando experimentos con tres casos de estudio desde las perspectivas académica e industrial. Los criterios de evaluación incluyeron la valoración de la calidad de los artefactos generados, la idoneidad de la solución, la eficiencia del enfoque y la facilidad de cambio en el modelo y

su replicación en el sistema implementado. El proceso de evaluación mostró que Urano cumple con estos criterios, por lo que es factible con él lograr la automatización de las actividades de despliegue a través de un enfoque basado en UML.

LIMITACIONES TÉCNICAS

- Dado que el despliegue de software es un proceso complejo que requiere, no solo considerar los aspectos físicos de los entornos de destino, sino también las características lógicas de los componentes desplegados, los servicios (interfaces) proporcionados y requeridos entre los componentes del sistema (las dependencias) son esenciales para comprender cómo se deben realizar las actividades de despliegue, sin embargo, las especificaciones UML no definen completamente la semántica para las interfaces proporcionadas y requeridas. Si un componente requiere (usa) una interfaz, y hay más de un componente que la proporciona, no hay una forma clara de saber quién atenderá la solicitud (enlace) y qué artefacto físico la manifestará; por lo tanto, existe una brecha semántica entre las condiciones requeridas de un sistema de software y su representación en un modelo UML. Una solución completa a este problema requiere extender UML con un mecanismo de peso medio o peso pesado, algo que está por fuera del alcance de la presente investigación.
- Eclipse Papyrus restringe los elementos de la interfaz de usuario que contienen el conjunto de constructos UML que se pueden arrastrar a un diagrama (las paletas), de acuerdo con el tipo de diagrama que se esté modificando, lo cual implica, de manera predeterminada, que los elementos que no pertenecen a la unidad de lenguaje de despliegue, no son visibles en la paleta mostrada para los diagramas de despliegue. Sin embargo, esta restricción no significa que los usuarios no puedan especificar componentes u otros elementos desde unidades de lenguaje diferentes en los diagramas de despliegue cuando usan Papyrus, de hecho, pueden usar el *Model Explorer* de Papyrus para agregarlos, pero se verán y se comportarán de manera diferente. Resolver esto requiere definir y registrar un contexto de arquitectura para diagramas de despliegue en Papyrus.
- Si bien lenguaje Amelia fue diseñado para automatizar tareas comunes de implementación de sistemas distribuidos basados en componentes,

como compilación de código fuente, ejecución y configuración de enlace. la implementación de ese tipo de sistemas también podría implicar tareas relacionadas con la configuración de la infraestructura -como la instalación de un sistema operativo, la configuración de hardware y propiedades de red, entre otros-. Aunque Amelia no cuenta con soporte para este tipo de tareas, tiene capacidades de extensión para lograrlo. Urano podría entonces orquestar la generación de especificaciones de despliegue ejecutables que integren a Amelia con lenguajes o herramientas basadas en secuencias de comandos que admitan capacidades IaC (*Infrastructure as a Code*), como Ansible [11], Chef [10] y Puppet [12].

TRABAJO FUTURO

El proceso de evaluación que se presentó no tuvo en cuenta atributos de calidad necesarios como son: usabilidad, rendimiento y escalabilidad. A futuro está prevista la evaluación de la usabilidad de Urano utilizando un marco propuesto por Condori-Fernández [67].

A continuación se presentan las expectativas de desarrollo de Urano en el corto plazo:

- Desarrollo de un contexto de arquitectura Eclipse Papyrus. Eclipse Papyrus es un entorno de modelado altamente personalizable que le permite a los profesionales de MDE y MDA adaptarlo según sus necesidades particulares. Se espera desarrollar un contexto de arquitectura que brinde a los usuarios una vista integrada que admita la administración de constructos UML -lógicos y físicos-, para diagramas de despliegue de software. También se espera incluir una paleta personalizada en el contexto de la arquitectura desde donde los usuarios puedan arrastrar los elementos estereotipados del perfil al diagrama, lo que facilitaría el proceso de aplicación del perfil. Esto mejoraría los tiempos de especificación.
- Soporte adicional de entornos de modelado. Actualmente Urano depende de Eclipse UML2, un proyecto compatible con múltiples entornos de modelado, tanto comerciales, como de código abierto, sin embargo, se espera admitir otras herramientas, como Visual Paradigm o StarUML, con lo que los usuarios podrían especificar diagramas de despliegue UML utilizando esas plataformas y generar especificaciones de despliegue ejecutables a partir de ellos al lanzar el modelo de ejecución de Urano.

- Abordar nuevas métricas de evaluación. La evaluación se centró principalmente en garantizar que los diagramas de despliegue basados en UML fueran suficientemente expresivos como para especificar sistemas distribuidos complejos, se espera evaluar la propuesta teniendo en cuenta otras métricas de evaluación, en particular aquellas relacionadas con la calidad.

De otro lado, las expectativas de desarrollo de Urano en el largo plazo son:

- Extensión del perfil UML. Aunque el perfil propuesto intenta ser tan general como sea necesario para generar especificaciones de despliegue de software en diversos dominios, se espera extenderlo con estereotipos o definiciones etiquetadas que capturen las semánticas de despliegue que se vayan requiriendo. Existe interés además en complementar el perfil con reglas de OCL (*Object Constraint Language*) que permitan que las arquitecturas de software y los desarrolladores validen el modelo de despliegue automáticamente.
- Desarrollar e integrar nuevos PSM. En este proyecto se abstraieron constructos de Amelia DSL y se diseñó un PSM que refleja las especificaciones del lenguaje, se espera integrar nuevos PSM basados en la abstracción de otros lenguajes de despliegue que promuevan el incremento de las capacidades de Urano. Principalmente hay interés en integrar lenguajes enfocados en especificaciones de infraestructura, tales como Chef [10], Ansible [11] y Puppet [12].
- Abordar la ingeniería inversa. Las prácticas de ingeniería inversa están bien difundidas para las clases, actividades y modelos de secuencia UML. Sin embargo, hay poca evidencia sobre su aplicación en diagramas de despliegue. Se espera que el estado actual de un sistema desplegado pueda traducirse automáticamente en un diagrama de despliegue que capture sus aspectos lógicos y sus aspectos físicos, y que, eventualmente, este tipo de abstracciones de modelos se puedan usar durante el tiempo de ejecución, de modo que proporcionen información en tiempo real sobre el estado de un sistema desplegado, algo que se podría aprovechar en su autoadaptación.

REFERENCIAS

- [1] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2010.
- [2] D.C. Schmidt, "Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.
- [3] P. Bourque, and R. E. Fairley, *Guide to the software engineering body of knowledge*, v. 3.0. Los Alamitos, CA: IEEE Computer Society, 2014.
- [4] Object Management Group (OMG), "Deployment and configuration of component-based distributed applications specification, v. 4.0," OMG, Needham, MA, OMG Document Number formal/2006-04-02, 2006.
- [5] O. Bossert, C. Ip, and I. Starikova, (2015, Sept.), *Beyond agile: Reorganizing IT for faster software delivery* [online], available: <https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/beyond-agile-reorganizing-it-for-faster-software-delivery>
- [6] *New Relic One is the first observability platform* [online], available: https://try.newrelic.com/rs/newrelic/images/DataCulture_SuretyReport_FINAL.pdf
- [7] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16*, pp. 21–30, New York, NY, ACM, 2016.
- [8] *Docker* [online], available: <https://www.docker.com/>
- [9] *Kubernetes* [online], available: <https://kubernetes.io/>
- [10] *Chef* [online], available: <https://www.chef.io/chef/>
- [11] *Ansible* [online], available: <https://www.ansible.com/>
- [12] *Puppet* [online], available: <https://puppet.com/>
- [13] J. Bezivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, May, 2005.
- [14] Object Management Group (OMG), "OMG Unified Modeling Language Specification, Version 2.5," OMG, Needham, MA, OMG Document Number formal/2015-03-01), 2015.
- [15] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, Thousand Oaks, CA: SAGE, March, 2013.
- [16] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," *Keele University and Durham University, UK, Technical Report EBSE 2007-001*, 2007.
- [17] Object Management Group (OMG), "MDA Guide, Revision 2.0," OMG, Needham, MA, OMG Document ormsc/2014-06-01, 2014.

- [18] J. Siegel, “Developing in OMGs model-driven architecture,” OMG, Needham, MA, Technical Report Revision 2.6], Nov. 2001. Retrieved from: <https://www.icmgworld.com/corp/developer/whitepapers/UsingMDA.pdf>.
- [19] S. Easterbrook, J. Singer, M.A. Storey, and D. Damian, “Selecting empirical methods for software engineering research,” in Guide to advanced empirical software engineering, pp. 285-311, London, UK: Springer.
- [20] R. K. Yin, Case study research: Design and methods. New Castle upon Tyne, UK: SAGE. 2013.
- [21] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L Wolf, “A characterization framework for software deployment technologies,” University of Colorado, Boulder, CO / Politecnico di Milano, Italia, Technical Report SU-CS-857-98, April 1998). Available: <https://www.ics.uci.edu/~andre/papers/T3.pdf>
- [22] D. Torre, Y. Labiche, M. Genero, M. Elaasar, T. K. Das, B. Hoisl, and M. Kowal, “1st international workshop on UML consistency rules (WUCOR 2015): Post workshop report SIGSOFT,” Software Engineering Notes, vol. 41 np. 2, pp. 34–37, May, 2016.
- [23] J. Bruck and K. Hussey (2009), Customizing UML: Which Technique is Right for You? [Online], Available: https://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html
- [24] J. Humble and J. Molesky, “Why enterprises must adopt devops to enable continuous delivery,” Cutter IT Journal, vol. 24, no. 8, pp. 6-12, 2011.
- [25] T. Fitz. (2009, Feb. 8). Continuous deployment [Online], available: <http://timothyfitz.com/2009/02/08/continuous-deployment/>
- [26] J. D. Poole, “Model-driven architecture: Vision, standards and emerging technologies,” in ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models, 2001.
- [27] Object Management Group [OMG]. MDA Guide Version 1.0.1. 2003, Needham, MA: OMG. Available: https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf
- [28] S. Etinge, M. Eysholdt, J. Kohnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, “Xbase: Implementing Domain-specific Languages for Java,” The ACM Special Interest Group on Programming Languages, vol. 48, no. 3, pp.112-121, 2012.
- [29] A. Ketfi and N. Belkhatir, “Model-driven framework for dynamic deployment and reconfiguration of component-based software systems,” in Proceedings of the 2005 Symposia on Metainformatics, MIS ’05, New York, NY: ACM, 2005.
- [30] OSGi [Online], Available: <https://www.osgi.org/developer/architecture/>
- [31] A. Sampaio and N. Mendonça, “Uni4cloud: An approach based on open standards for deployment and management of multi-cloud applications,” in Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing, SE-CLOUD ’11, pp. 15-21, New York, NY: ACM, 2011.

- [32] F. Magalhaes, T. da Rocha, J. Santos, and E. Moreno, “A model-driven solution for automatic software deployment in the cloud, Cham, Germany: Springer, 2016.
- [33] J. Jrjens. “UMLSec: Extending UML for secure systems development”, Lecture Notes in Computer Science, no. 2460, pp. 412-425, 2002.
- [34] S. Kallel, M. Loulou, M. Rekik, and A. H. Kacem, “MDA-based approach for implementing secure mobile agent systems,” in J. P. Muller and M. Cossentino [Eds.], Agent-Oriented SoftwareEngineering XIII, pp. 56–72, Berlin-Heidelberg, Springer-Berlin Heidelberg, 2013.
- [35] L. Apvrille, P. de Saqui-Sannes, and F. Khendek, “Turtle-p: A uml profile for the formal validation of critical and distributed systems,” Software and Systems Modeling, vol. 5, no. 4, pp. 449-466, 2006.
- [36] P.H. Hughes and J.S. Lvstad, “A generic model for quantifiable software deployment,” in International Conference on Software Engineering Advances (ICSEA 2007), Cap Esterel, France: IEEE, 2007.
- [37] G. Kim, P. Debois, J. Willis, J. Humble, and J. Allspaw, The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations, Portland, OR: IT Revolution Press, 2016.
- [38] R. Kazman, M. Klein, and P. Clements, “Atam: Method for architecture evaluation,” Software Engineering Institute, Pittsburgh, PA, CMU/SEI-2000-TR-004, Aug. 2000.
- [39] Eclipse Modeling Project [Online]. Available: <https://projects.eclipse.org/projects/modeling>
- [40] Eclipse Modeling Framework (EMF) [Online]. Available: <https://www.eclipse.org/modeling/emf/>
- [41] Eclipse Papyrus [Online]. Available: <https://projects.eclipse.org/projects/modeling.mdt.papyrus>
- [42] Eclipse MDT UML2 [Online]. Available: <https://projects.eclipse.org/projects/modeling.mdt.uml2>
- [43] Eclipse Acceleo [Online]. Available: <https://projects.eclipse.org/projects/modeling.m2t.acceleo>
- [44] MOF Model to Text Transformation Language [Online]. Available: <https://www.omg.org/spec/MOFM2T/1.0/>
- [45] Urano releases [Online]. Available: <https://lfrivera.github.io/urano/releases/>
- [46] Urano [Online]. Available: <https://github.com/lfrivera/urano>
- [47] Locmetrics [Online]. Available: <http://www.locmetrics.com/>
- [48] L. F. Rivera, “UML-Driven Automated Software Deployment”, M.S tesis. Universidad Icesi, Cali, Colombia, 2018

- [49] Visual paradigm [Online]. Available: <https://www.visual-paradigm.com/>
- [50] Enterprise Architect in 30 minutes [Online]. Available: <https://sparxsystems.com.au/enterprise-architect/index.html>
- [51] Eclipse Community Forums [Online]. Available: <https://www.eclipse.org/forums/index.php/f/117/>
- [52] MDT-UML2-Tool-Compatibility [Online]. Available: <https://wiki.eclipse.org/MDT-UML2-Tool-Compatibility>
- [53] OW2 FraSCAti SCA Architecture [Online]. Available: <http://frascati.ow2.org/doc/1.4/ch12s04>
- [54] ATL-a model transformation technology [Online]. Available: <https://www.eclipse.org/atl/>
- [55] Acceleo/user guide / Modules [Online]. Available: https://wiki.eclipse.org/Accleo/User_Guide#Modules
- [56] Acceleo/user guide / Templates [Online]. Available: https://wiki.eclipse.org/Accleo/User_Guide#Templates
- [57] Urano M2T plugin [Online]. Available: <https://github.com/lfrivera/urano/tree/master/plug-ins/org.urano.mdt.plugin.m2t/src/org/urano/mdt/plugin/m2t>
- [58] P. Mohagheghi, “An approach for empirical evaluation of model-driven engineering in multiple dimensions”, Sintef, Oslo, Noeway, Technical report, 2010.
- [59] F. D. Davis, “Perceived usefulness, perceived ease of use, and user acceptance of information technology”, *MIS Q*, vol. 13, no. 3, pp. 319-340, September, 1989.
- [60] P. Mohagheghi and J. Aagedal, “Evaluating quality in model-driven engineering,” in *International Workshop on Modeling in Software Engineering, MISE’07: ICSE Workshop 2007*, p. 6, May, 2007.
- [61] Urano - Evaluación [Online]. Available: <https://github.com/lfrivera/urano/tree/master/examples/evaluation>
- [62] H. Lee, J. Kim, S. J. Hong, and S. Lee, “Processor allocation and task scheduling of matrix chain,” *Products on Parallel Systems*, vol. 14, no. 4, pp. 394-407.
- [63] 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, 2012.
- [64] About OW2 FraSCAti [Online]. Available: <http://frascati.ow2.org/doc/1.4/ch01s03.html>
- [65] Apache JMeter™ [Online]. Available: <https://jmeter.apache.org/>
- [66] Understanding RabbitMQ [Online]. Available: <https://www.rabbitmq.com/>
- [67] N. Condori-Fernandez, J.I. Panach, A. I. Baars, T. Vos, and S. Pastor, “An empirical approach for evaluating the usability of model-driven tools,” *Science of Computer Programming*, vol. 78, no. 11, pp. 2245-2258, Nov., 2013.

SISTEMA DE BENCHMARKING DE REDES MÓVILES CELULARES Y WiFi BASADO EN DISPOSITIVOS DE USUARIO FINAL Y SDR

Leonardo Vargas Bernal, MSc.

Andrés Navarro Cadavid, Ph.D

Citación

L. Vargas, y A. Navarro, “Sistema de benchmarking de redes móviles celulares y WiFi basado en dispositivos de usuario final y SDR,” en *Bitácoras de la maestría*, vol. 3, *Monitores dinámicos de software - Despliegue de software - Monitoreo de espectro*, Cali, Colombia: Universidad Icesi, 2020, pp. 169-209.

RESUMEN

En un mundo donde las telecomunicaciones tienen gran protagonismo es cada vez más importante garantizar el buen uso del espectro radioeléctrico, pues de ello depende que se puedan proveer más y mejores servicios. Monitorear la ocupación de las bandas de frecuencia es fundamental en ese propósito porque permite contrastar, de manera objetiva, su uso real frente a las asignaciones hechas por los reguladores. El proyecto Simones, del cual forma parte la presente investigación, abordó un reto importante: desarrollar unidades de monitoreo simples, de operación desatendida y bajo costo, pero capaces de cumplir con los requerimientos de la Unión Internacional de Telecomunicaciones, en particular con los de su *Spectrum Management Systems for Developing Countries*, usando tecnologías de radio software que permitieran ubicar funciones de procesamiento de señales en procesadores de propósito general. La contribución de este proyecto, como componente de Simones, fue el desarrollo de una herramienta que permitiera evaluar las condiciones de los servicios móviles desde la perspectiva del usuario final. Este trabajo logró su propósito utilizando equipos sencillos (USRP y el bladeRF), lo que representó el primer paso firme en la construcción de SiMon, una suite con herramientas de gestión del espectro radioeléctrico desarrollado en la Universidad Icesi.

INTRODUCCIÓN

La gestión del Espectro Radioeléctrico (ERE) es una tarea compleja y de gran responsabilidad, que involucra, además de la tradicional asignación de frecuencias –y la coordinación de estas asignaciones con los países vecinos–, retos derivados de: los nuevos usos del espectro y las nuevas tecnologías; el uso del espectro en el largo plazo; el monitoreo permanente de los servicios basados en él; y la adaptación de los parámetros de radio propagación a las condiciones locales. Los sistemas de monitoreo de espectro son herramientas fundamentales para esta labor. Se utilizan para detectar posibles violaciones de la regulación existente, ubicar emisiones no permitidas y ejecutar labores de limpieza de bandas de frecuencia. Los operadores móviles utilizan sistemas similares en sus operaciones de ajuste y despliegue de nuevos sitios para garantizar que estén libres de interferencias.

Una labor de las entidades reguladoras de las telecomunicaciones, como la Agencia Nacional del Espectro (ANE) en Colombia, es utilizar sistemas de comprobación técnica para monitorear el cumplimiento de los acuerdos de cobertura establecidos en las licencias para uso del espectro. En ese sentido, las medidas de los parámetros de señal tienen gran relevancia porque: muestran las áreas de servicio sobre las que influye un operador; le ayudan a identificar, de sus áreas de interés, cuáles están cubiertas –y cuáles no–; y le permiten tomar decisiones sobre potencia de transmisión, direccionamiento de antenas y ubicación de nuevas estaciones base. Estas actividades son pertinentes para las entidades reguladoras porque complementan sus registros de licenciamiento del espectro con datos reales sobre la disponibilidad del servicio y les ayudan a definir si existe correlación entre el uso real y la cantidad de recursos asignados.

Investigaciones en redes autoadaptativas, como el proyecto Sócrates [1], del Séptimo Programa Marco de la Unión Europea, estuvieron orientadas a proporcionarle capacidades de autoorganización, autoconfiguración, autooptimización y autosanación a la infraestructura de telecomunicaciones, las que deberían permitirle a las estaciones modificar su configuración y sus parámetros de forma automática, para asegurar la meta de cobertura establecida por el operador, de acuerdo con variables como: la densidad de usuarios y los requerimientos de ancho de banda. Los investigadores de Sócrates [1] proponen un modelo de distribución basado en el usuario, en el cual los terminales móviles son los que obtienen la información y permiten

alimentar el sistema. Este proyecto, encaminado a redes LTE (*Long Term Evolution*), demuestra el incremento en los indicadores de rendimiento de las redes autoadaptativas, respecto de las no adaptativas, y propone algoritmos de adaptación y modelos de obtención de datos, por ejemplo, para usuarios peatonales y móviles vehiculares.

El reporte técnico 3GPP TR 36.902 v.9.3.1 [2] define como una de sus prioridades la estandarización de las redes autoadaptativas en LTE y establece como una tarea operacional la optimización de las redes de acuerdo con su cobertura y capacidad. En ese sentido, resalta la importancia de las mediciones de parámetros de señal como información que permite identificar las características que deben modificarse en cada caso y los procedimientos de interacción entre las entidades involucradas. De acuerdo con Johansson, Hapsari, Kelley y Bodog [3], 3GPP propone además la minimización de *drive test* (MDT, *Minimization of Drive Test*), una característica que le permite a los operadores utilizar los dispositivos de los usuarios para recolectar información localizada sobre el rendimiento de la red.

El Instituto de Telecomunicaciones y Aplicaciones Multimedia iTEAM de la Universidad Politécnica de Valencia (España) presentó una propuesta en el COST (European Cooperation in Science and Technology) 2100 [4] en la que describe la medición de parámetros de redes HSDPA (*High Speed Downlink Packet Access*) usando *drive test* y algunos procesos para identificar la degradación del rendimiento en ese tipo de redes, y propone mecanismos para asegurar la disponibilidad de servicios; menciona: los parámetros que se deben medir, el proceso de medición y las diferencias respecto del monitoreo por *drive test* tradicional; y describe los problemas más comunes en las redes HSDPA y la forma de identificarlos a partir de los datos.

Existen varios fabricantes de sistemas de *drive test* convencionales, entre ellos: Anritsu [5], que tiene soluciones como ML8725A y ML8726A para medir y analizar de parámetros en redes LTE, WCDMA (*Wideband Code Division Multiple Access*), HSPA (*High-Speed Packet Access*), GSM (*Global System for Mobile communications*), CDMA (*Code Division Multiple Access*), cdma2000 y EV-DO (*Evolution-Data Optimized*), y hacer lecturas de RSSI (*Received Signal Strength Indicator*), SNR (*Signal-to-Noise Ratio*), RSRP (*Reference Signals Received Power*), potencia de transmisión e información de canal, entre otras; y JDSU [6] –cuya línea de productos para *drive test* pertenecía a Agilent–, que ofrece la plataforma de optimización de redes inalámbricas E6474A, que obtiene los parámetros

de red según la tecnología o el tipo de servicio (*e.g.*, video y *Voice over Internet Protocol*, VoIP). Los operadores han utilizado tradicionalmente este tipo de sistemas, pero ello implica disponer de múltiples unidades monitoreando la red continuamente en rutas determinadas por el operador de forma más o menos aleatoria. Otro método se basa en obtener las estadísticas del Nodo B por medio de los diferentes eventos que se definen en las redes móviles tipo UMTS (*Universal Mobile Telecommunications System*) o LTE, sin embargo, este mecanismo solamente brinda información general sobre una celda o sector, pero no incluye datos de localización.

Las medidas de parámetros de señal para redes WiFi son igual de relevantes que las de IMT (*International Mobile Telecommunications*) si se tienen en cuenta los avances en redes de pequeñas celdas y en femtoceldas, que consisten en pequeñas estaciones desplegadas por el usuario –de bajo consumo de energía y bajo costo–, que permiten ofrecer servicios de comunicaciones celulares en entornos residenciales o empresariales [7]. Esos conceptos le permiten a los operadores ofrecer servicios extendidos en sus licencias UMTS/LTE y en licencias libres, a través de dispositivos que se integran a las redes de macroceldas. Los algoritmos de búsqueda de femtoceldas basados en la potencia de transmisión o en la puntuación de las estaciones y en la localización del equipo de usuario, como el propuesto en la 3GPP TS 25.304, son áreas relativamente recientes de investigación [8]. Además, el estándar IEEE 802.11 [9] ha tomado mayor relevancia por las mejoras en rendimiento y el ahorro de energía, que permiten implementar las redes de sensores con conectividad IP necesarias para el Internet de las Cosas (IoT, *Internet of Things*) [10].

La presente investigación se realizó en el marco del proyecto “Sistema de monitoreo de espectro y benchmarking de sistemas móviles, usando radio software de dominio público –Simones” (Colciencias, Código 2117-502-27325), ejecutado por el Grupo de Investigación en Informática y Telecomunicaciones (i2t) de la Universidad Icesi (Cali, Colombia), cuyo propósito fue desarrollar unidades de monitoreo simples, de operación desatendida y bajo costo, que apoyen la labor de monitoreo y complementen los sistemas de gestión de espectro y las unidades de monitoreo existentes [11], usando tecnologías de radio software (SDR, *Software Defined Radio*), que permiten ubicar funciones de procesamiento de señales en procesadores de propósito general [12], incrementando así la flexibilidad y permitiendo reconfigurar el sistema de acuerdo con las tareas y requerimientos de monitoreo de cada región.

Se identificó la necesidad de contar con un sistema para monitorear los niveles de servicio en entornos metropolitanos, la contribución de este proyecto como componente de Simones es el desarrollo de una herramienta que permite evaluar las condiciones de los servicios móviles desde la perspectiva del usuario final, que le ayude a los operadores de telecomunicaciones y a las entidades reguladoras a: tomar decisiones acerca de la configuración de la red y la asignación eficiente del ERE; comparar los operadores de acuerdo con características clave, como son los indicadores de desempeño, los esquemas de modulación y el ancho de banda; y aportar a la investigación en mediciones de *drive test* en redes WiFi.

El proyecto definió como su objetivo general “Diseñar e implementar un sistema de benchmarking de redes móviles celulares y WiFi basado en SDR”, y como objetivos específicos: diseñar la arquitectura de software para un sistema de *drive test* que permita medir los parámetros de redes UMTS/LTE y WiFi; describir un modelo estadístico para el análisis de las muestras obtenidas durante el *drive test*; e implementar un analizador de modulación para *drive test* basado en SDR, un analizador de modulación para *drive test* basado en dispositivos de usuario final y un sistema de *backend* para almacenar y visualizar los resultados del monitoreo.

MARCO TEÓRICO

OFDM

OFDM (*Orthogonal Frequency Division Multiplexing*) es un método para multiplexar datos en diferentes subportadoras ortogonales. La serie de datos se pasa de serial a paralelo a través de un convertidor que la divide en unos canales – que posteriormente se modulan con portadoras de frecuencias distintas–, los cuales se combinan para generar un símbolo OFDM. Del lado del receptor, los símbolos se demultiplexan, se demodulan de forma independiente y se reconstruye la señal original con un convertidor de paralelo a serial sobre los componentes en banda base [13]. OFDM se utilizó inicialmente en ADSL y ha hecho parte de tecnologías como DVB-T (*Digital Video Broadcasting - Terrestrial*), redes ópticas [14], UWB (*Ultra-WideBand*) [15], IEEE 802.11 [9] y LTE.

N representa el número de canales paralelos para los que existen N moduladores con frecuencias de portadoras f_0, f_1, \dots, f_{N-1} . La diferencia entre

canales adyacentes se denota como Δf ; el ancho de banda del símbolo OFDM W es $N \Delta f$.

Un sistema OFDM se puede implementar utilizando la IDFT (*Inverse Discrete Fourier Transform*) y la DFT (*Discrete Fourier Transform*). El procedimiento consiste en multiplicar los valores del bloque de datos por un conjunto de subportadoras sinusoidales ortogonales y sumar esos componentes, es decir, obtener la IDFT y transmitir los coeficientes en forma serial. Se puede demostrar que al aplicar la IDFT sobre un bloque de datos $\{a_0, a_1, \dots, a_{N-1}\}$, se generan muestras con intervalos de tiempo T/N de una señal OFDM, donde T es el periodo en el que las sinusoidales son ortogonales [16].

Las implementaciones más eficientes de la IDFT y la DFT son la IFFT (*Inverse Fast Fourier Transform*) y la FFT (*Fast Fourier Transform*), que aparecen en su forma general en las ECUACIONES 1 y 2 [17].

$$w[k] = \frac{1}{N} \sum_{n=0}^{N-1} W[n] e^{j2\pi nk/N} \quad (1)$$

$$W[n] = \sum_{k=0}^{N-1} w[k] e^{-j2\pi nk/N} \quad (2)$$

Así, la muestra s_k del símbolo OFDM se puede expresar como se indica en la ECUACIÓN 3.

$$s_k = \sum_{n=0}^{N-1} a_n e^{j2\pi nk/N} \quad (3)$$

Para reducir la interferencia intersímbolo de las señales sobre canales dispersos, se agregó un intervalo de guarda o prefijo cíclico de tamaño N_{cp} entre los símbolos OFDM, iguales a las últimas N_{cp} muestras generadas por la IDFT. La FIGURA 1 muestra la estructura general de un sistema OFDM con la adición de prefijo cíclico.

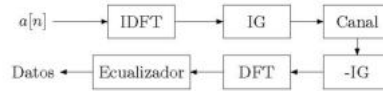


Figura 1. Estructura general de un sistema OFDM

En el dominio del tiempo, un símbolo OFDM de duración T tiene un prefijo cíclico de duración T_{GI} y un espacio disponible para transmisión de datos de tamaño T_{fft} , de tal forma que $T = T_{GI} + T_{fft}$. Si el intervalo de tiempo entre símbolos adyacentes es Δt , se pueden relacionar el dominio del tiempo y el de frecuencia con las ECUACIONES 4.

$$N = T_{fft}/\Delta t \quad (4a)$$

$$N_{cp} = T_{GI}/\Delta t \quad (4b)$$

$$\Delta f = 1/\Delta t \quad (4c)$$

La sincronización del lado del receptor se puede hacer utilizando el método de Schmidl & Cox [18], el cual demuestra que la diferencia en frecuencia o *frequency offset* Δf_o en una señal OFDM se puede estimar utilizando una secuencia de entrenamiento de dos símbolos, si cada uno tiene dos mitades idénticas en el dominio del tiempo, y aprovechando que el efecto del canal es el mismo en cada mitad. La diferencia en frecuencia causa una rotación de fase $2\pi t \Delta f_o$, así que, al multiplicar el conjugado de una muestra en la primera mitad del primer símbolo de entrenamiento por la muestra correspondiente en la segunda mitad, el efecto del canal se cancela y se obtiene un resultado con fase \emptyset , según la ECUACIÓN 5.

$$\emptyset = \pi T \Delta f_o \quad (5)$$

Si $P(d)$ es la suma de los pares de los productos en las muestras complejas r_m de la primera mitad del primer símbolo de longitud $2L$, como se muestra

en la ECUACIÓN 6, $R(d)$ es la energía recibida de la segunda mitad del primer símbolo, según la ECUACIÓN 7; y la métrica de tiempo $M(d)$ está dada por la ECUACIÓN 8 y $\hat{\phi}$ se puede estimar como el ángulo de $P(d)$ cerca al mejor punto de muestreo d .

$$P(d) = \sum_{r=0}^{L-1} (r^*_{d+m} r_{d+m+L}) \quad (6)$$

$$R(d) = \sum_{r=0}^{L-1} |r_{d+m+L}|^2 \quad (7)$$

$$M(d) = \frac{|P(d)|^2}{(R(d))^2} \quad (8)$$

A partir de $\hat{\phi}$ se puede estimar la diferencia en frecuencia, como se establece en la ECUACIÓN 9, y corregirla al multiplicar las muestras por $e^{-j2t\hat{\phi}T}$, una sinusoidal de frecuencia Δf_0 .

$$\Delta \hat{f}_0 = \frac{\hat{\phi}}{\pi T} \quad \text{si } |\hat{\phi}| < \pi \quad (9a)$$

$$\Delta f_0 = \frac{\hat{\phi}}{\pi T} + \frac{2z}{T} \text{ de otro modo} \quad (9b)$$

Si $2z/T$ es el valor restante después de corregir los dos símbolos de entrenamiento, es posible estimar el entero z al maximizar $B(\mathbf{g})$, y $\Delta \hat{f}_0$ se puede expresar según la ECUACIÓN 10. En la ECUACIÓN 11, \mathbf{x}_1, \mathbf{k} y \mathbf{x}_2, \mathbf{k} son las FFT de los dos símbolos de entrenamiento, \mathbf{v}_k son las componentes pares del segundo símbolo y X es el conjunto de índices de los componentes pares.

$$\Delta \hat{f}_0 = \frac{\hat{\phi}}{\pi T} + \frac{2\hat{g}}{T} \quad (10)$$

$$B(g) = \frac{|\sum_{k \in X} x_{1,k+2g}^* v_{k+2g}^* x_{2,k+2g}|^2}{2(\sum_{k \in X} |x_{2,k}|^2)^2} \quad (11)$$

802.11

El estándar IEEE 802.11 de 2007 [9] especifica los niveles físico y MAC para redes inalámbricas de área local e incluye varias mejoras. Sus versiones a y g usan OFDM en las bandas de 5GHz y 2.4GHz, respectivamente, con tasas de transmisión de 54Mb/s y canalización de 20MHz; la versión b utiliza DSSS (*Direct Sequence Spread Spectrum*) en la banda de 2.4GHz, con tasas de transmisión de 11Mb/s y la misma canalización; la versión n incorpora MIMO (*Multiple-Input Multiple-Output*), con lo que permite tasas de 600Mb/s, con canalizaciones de 20 y 40MHz; y la versión ac utiliza las bandas de frecuencia de 2.4 y 5GHz, con tasas de transmisión por encima de 1Gb/s [19].

La arquitectura de la capa física de 802.11 se compone de tres subcapas: PLCP (*Physical Layer Convergence Procedure*), PMD (*Physical Medium Dependent*) y PLME (*Physical Layer Management Entity*). PLCP actúa como una subcapa de adaptación que independiza la capa MAC de implementaciones específicas en la subcapa PMD, que establece las técnicas de modulación y codificación y accede directamente al medio inalámbrico, y PLME actúa como una subcapa de control de toda la capa física.

En 802.11g, la duración del símbolo OFDM $T = 4\mu s$, el prefijo cíclico $T_{GI} = 800ns$, el espacio disponible para transmisión de datos $T_{fft} = 3,2\mu s$ y el intervalo de tiempo entre símbolos adyacentes $\Delta t = 0,05\mu s$. De las ECUACIONES 4 se puede obtener: el número de subportadoras, $N = 64$; el tamaño del prefijo cíclico, $N_{cp} = 16$, y el espacio entre subportadoras, $\Delta f = 312,5KHz$.

De las 64 subportadoras sólo se utilizan $N_{ST} = 52$ y se agrega 0 en las otras doce para evitar la interferencia fuera de banda. El número de subportadoras piloto $N_p = 4$ corresponde a los índices $\{-21, -7, 7, 21\}$ y el número de subportadoras de datos $N_c = 48$ corresponde a los índices $\{-26, \dots, -22, -20, \dots, -8, -6, \dots, -1, 1, \dots, 6, 8, \dots, 20, 22, \dots, 26\}$, donde 0 es la portadora central o DC. Pueden estar moduladas con BPSK y QPSK o 16QAM y 64QAM.

La unidad de datos de PLCP (*PLCP Protocol Data Unit*, PPDU) se compone de un preámbulo de doce símbolos OFDM que se utilizan para sincronización en

el receptor; un símbolo OFDM SIGNAL modulado con BPSK y codificado a una tasa $R = 1/2$, que incluye el esquema de modulación y la longitud de los datos; y una serie de símbolos OFDM DATA con la carga útil: el campo SIGNAL y la PSDU (*Physical layer Service Data Unit*). La FIGURA 2 muestra la estructura de la cabecera PLCP y de la trama PPDU según el estándar.

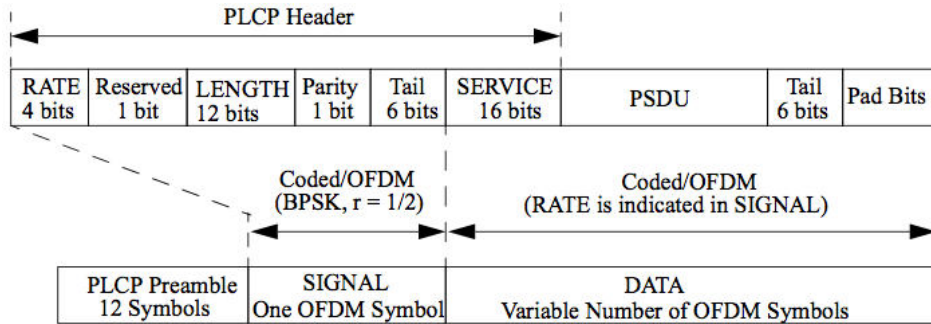


Figura 2. Estructura de la PPDU [18]

El preámbulo está compuesto por diez símbolos cortos y dos símbolos largos; cada símbolo corto consta de doce subportadoras moduladas por la secuencia S (ECUACIÓN 12), y cada símbolo largo, de 53 subportadoras moduladas por la secuencia L (ECUACIÓN 13); las dos secuencias en el dominio de la frecuencia se pueden consultar en [18, Tablas L-2 y L-5]. Los símbolos cortos tienen una duración de $0,8\mu s$, para una duración de secuencia de $8\mu s$; los largos tienen una duración de $3,2\mu s$ y se les adiciona un intervalo de guarda $T_{GI2} = 1,6\mu s$, para una duración de secuencia de $8\mu s$. La duración del preámbulo es de $16\mu s$, equivalentes a $N_p = 320$ muestras.

$$S_{-26,26} = \sqrt{13/16} \times \{0, 0, 1 + j, 0, 0, 0, -1-j, 0, 0, 0, 1 + j, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1 + j, 0, 0, 0, 0, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1 + j, 0, 0, 0, 1 + j, 0, 0, 0, 1 + j, 0, 0, 0, 1 + j, 0, 0\} \quad (12)$$

$$L_{-26,26} = \{1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 0, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, -1, 1, 1, -1, 1, 1, -1, 1, 1, 1, 1\} \quad (13)$$

SIGNAL está compuesto por 24 bits que abarcan los campos RATE, LENGTH y bits de paridad, de reserva y de cierre. RATE indica de forma implícita el esquema de modulación y la tasa de codificación de acuerdo con [18, Tablas 18-4 y 18-6], y LENGTH indica el número de octetos en la PSDU.

Para generar el símbolo SIGNAL y el resto de información en la PPDU, se utiliza codificación convolucional (*convolutional encoding*), intercalado (*interleaving*), mapeo de modulación, inserción subportadoras piloto y modulación OFDM, como se especifica en [18]. La estructura general del transmisor y el receptor OFDM en 802.11 se muestra en la FIGURA 3.

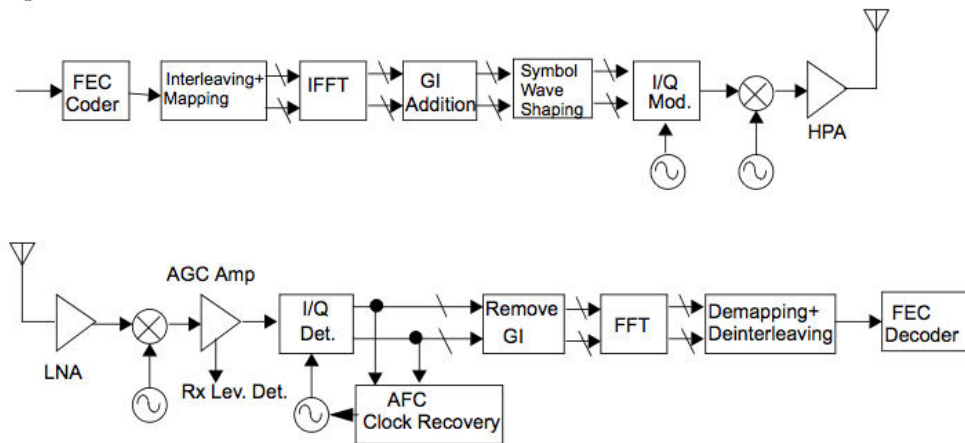


Figura 3. Transmisor y receptor OFDM 802.11 [18]

LTE

Es la tecnología de comunicaciones móviles de más rápido desarrollo. Su esquema de acceso múltiple está basado en OFDM en el canal descendente y en SC-FDMA (*Single-Carrier Frequency Division Multiple Access*) en el canal ascendente; soporta FDD (*Frequency-Division Duplex*) y TDD (*Time-Division Duplex*), es decir, que los canales de subida y bajada pueden estar separados por frecuencia o por tiempo [20]. El nivel físico se define a partir de bloques de recursos independientes del ancho de banda asignado, que pueden estar compuestos por doce subportadoras de 15KHz o veinticuatro de 7.5KHz, en un espacio de tiempo de 0,5ms. Tiene dos estructuras de trama, tipo 1 y tipo

2, que se aplican a FDD y TDD, respectivamente. La trama tipo 1 tiene una duración $T_1 = 10\text{ms}$ y consta de veinte franjas o *slots* de 0,5ms, equivalentes a diez subtramas, como se muestra en la FIGURA 4a. Los enlaces de subida y bajada están separados en frecuencia y tienen diez subtramas disponibles en cada intervalo de 10ms. La trama tipo 2 tiene una duración $T_2 = 10\text{ms}$ y consta de dos medias tramas de 5ms con cinco subtramas de 1ms cada una. Cada subtrama puede estar reservada para el enlace de subida, el de bajada o como subtrama especial, como se muestra en la FIGURA 4b. Una señal transmitida o recibida se puede describir en función de una malla de recursos RG , que posee múltiples elementos RE . Los bloques de recursos RB describen el mapeo de los RE a los canales físicos: cada RB es un conjunto de siete símbolos OFDM o SC-FDMA en el dominio del tiempo y doce subportadoras en el dominio de la frecuencia, es decir que cada RB posee 84 RE y tiene una duración 1 *slot* a 180 KHz para un canal de 20MHz [21].

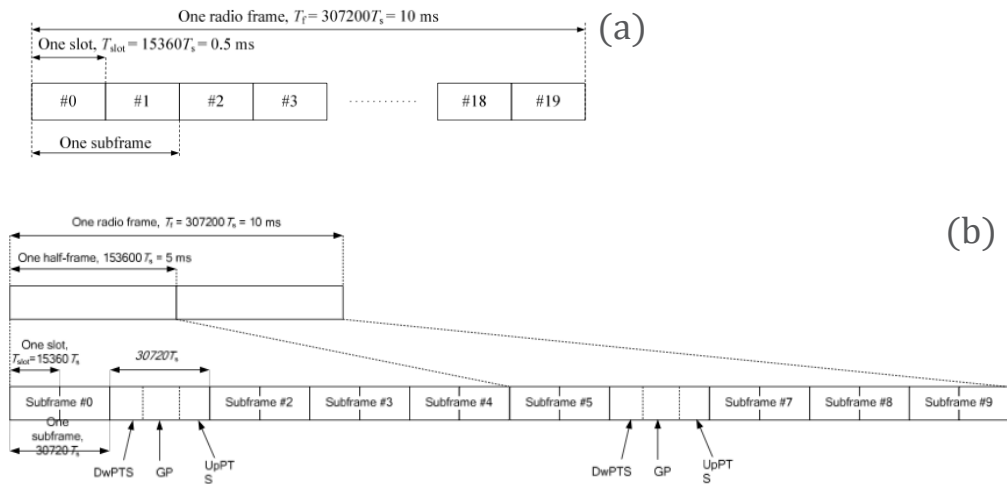


Figura 4. Estructura de tramas LTE (a) tipo 1; (b) tipo 2 [21]

LTE se puede desplegar en seis anchos de banda (1.4, 3, 5, 10, 15 y 20 MHz) y, con la agregación de portadoras de LTE *release* 10, se pueden agregar hasta cinco portadoras de bajada, para un máximo de 100MHz [22]. Los parámetros según el ancho de banda disponible se pueden consultar en [23, Tabla 2].

LTE utiliza señales de referencia específicas para la celda (CRS, *Cell Specific Reference Signal*) en el canal de datos, las que sirven para realizar mediciones de

movilidad, sincronización y estimación de canal para demodulación. Como alternativa, utiliza señales de referencia del equipo de usuario (*UE-specific*) o señales de referencia de la demodulación (DMRS, *Demodulation Reference Signal*). En el *release 11* se introdujo un canal de control mejorado, que se basa únicamente en señales de referencia del equipo de usuario y utiliza el canal de forma más eficiente, al reducir el uso de ancho de banda. El transmisor envía una señal de referencia por cada puerto de antena en el canal descendente, mapeada en el prefijo cíclico de los símbolos OFDM [21].

INDICADORES DE RENDIMIENTO

En tecnologías móviles, la estación base con señal más fuerte no es necesariamente la de mejor tasa de transmisión o la de mejor rendimiento, por eso es necesario identificar los indicadores clave de rendimiento para realizar comparaciones objetivas. La distribución de la relación señal a ruido más interferencia (*Signal to Interference plus Noise Ratio*, SINR) es la métrica más importante de las redes celulares [24] porque relaciona parámetros como el desvanecimiento de la señal, la potencia del ruido y la interferencia de otras estaciones base, y porque a partir de ella se puede obtener la tasa efectiva de transmisión. La ECUACIÓN 14 muestra la versión general, en donde P es la potencia de la señal, I es la potencia de la interferencia de otras señales en la red y N es la constante de ruido.

$$SINR(x) = \frac{P}{1+N} \quad (14)$$

El 3GPP especifica los parámetros que se deben medir en tecnologías de radio [25][26], entre ellos:

- la RSRP es el promedio de la energía de la CRS de una celda específica en el ancho de banda de interés;
- el RSSI incluye la potencia del ruido térmico y del receptor y la interferencia de otras celdas y canales adyacentes, en el ancho de banda de interés;
- la calidad percibida de la señal de referencia (*Reference Signal Received Quality*, RSRQ) se define como la relación entre la potencia de la señal de referencia y la potencia de la portadora (en la ECUACIÓN 15, N es el número de bloques de recursos en el ancho de banda de interés);

$$RSRQ = N X = \frac{RSRP}{RSSI} \quad (15)$$

- la potencia de código de la señal (*Received Signal Code Power*, RSCP) indica la potencia de un canal específico en la celda de interés;
- la energía recibida por *chip* dividida entre la densidad de potencia de la banda (E_c/N_0) es igual a la RSCP dividida entre la RSSI (ECUACIÓN 16);

$$E_c/N_0 = \frac{RSCP}{RSSI} \quad (16)$$

- la diferencia de tiempo entre las señales de referencia (*Reference Signal Time Difference*, RSTD) corresponde a la diferencia entre el momento en que se recibe la CRS de una celda j respecto de la CRS de la celda i , como se muestra en la ECUACIÓN 17;

$$RSTD = T_j - T_i \quad (17)$$

- la lista de celdas vecinas a la estación de referencia; y
- el esquema de modulación

DRIVE TEST

En el proceso de *drive test* los operadores de telecomunicaciones realizan mediciones de radio para identificar problemas y determinar cuándo un parámetro de la red debe ser ajustado [27] y detectar indicadores sobre la degradación del servicio y hacer correcciones para mantener las métricas de calidad de servicio (*Quality of Service*, QoS) en niveles aceptables. El método convencional se centra en la señalización de los niveles 2 y 3 de las interfaces y se utiliza para indagar sobre la calidad percibida por los usuarios finales, a partir de: parámetros de los enlaces de radio, como el RSCP y el E_c/I_0 ; de control, como la lista de vecinos y la configuración del *handover*; y de QoS, como el *throughput* y la tasa de errores de bloque (BLER, *Block Error Rate*) [4].

El análisis de los datos obtenidos ayuda a identificar problemas como: la falta de cobertura que se presenta cuando el RSCP está por debajo de -100dBm

o el E_c/I_o es menor a -12dB ; la cobertura de interferencia, que se presenta cuando el RSCP es mayor a -19dBm y el E_c/I_o está por debajo de -12dB ; el sobredimensionamiento de la celda, que se puede detectar cuando el terminal del usuario se conecta a celdas lejanas; y las celdas desasociadas que no están en la lista de vecinos y causan interferencia.

El *drive test* convencional implica el desplazamiento de personal y equipos especializados a las áreas de interés, y sus resultados están limitados a ellas. Algunos parámetros dependen de la carga de la celda, por lo que se deben medir en espacios de tiempo prolongados. Debido a los costos asociados a esa labor, el 3GPP estandarizó la MDT.

ANÁLISIS DE MUESTRAS

El método de Lee es recomendado por la Unión Internacional de Telecomunicaciones (UIT) para obtener valores promedio de los parámetros de una señal en un punto de una ruta. Las variables que se requieren para hacer los cálculos son: el rango en el que se toman las muestras $2L$, el número mínimo de muestras N y la distancia mínima entre las muestras para que estas no estén correlacionadas d . Los valores se determinan al promediar N muestras separadas a una distancia d dentro del rango $2L$ [28].

La envolvente de la señal $r(y)$, en la ECUACIÓN 18, está compuesta por una componente de variación rápida $r_0(y)$ superpuesta a una componente de variación a largo plazo $m(y)$, que representa la media de la señal, en donde y es la distancia recorrida por el receptor.

$$r(y) = m(y) \cdot r_0(y) \tag{18}$$

$2L$ y N se pueden estimar de acuerdo con las condiciones de las ECUACIONES 19, en donde σr es la desviación estándar de las muestras r_i de la señal $r(y)$, y d corresponde a la distancia en la que las muestras tienen una correlación menor a 0.2.

$$\frac{1}{2L} \int_{x-L}^{x+L} r_0(y) \cdot d(y) \rightarrow 1 \tag{19a}$$

$$1,65 \frac{\sigma_r}{\sqrt{N}} \leq 1dB \quad (19b)$$

Los resultados acogidos por la UIT para los parámetros del método de Lee son: $2L = 40\lambda$, $d = 0,8\lambda$, y $N = 50$, en donde λ es la longitud de onda de la señal, y la velocidad del receptor está dada por la ECUACIÓN 20, en donde $t_r(s)$ es el tiempo que toma el receptor en revisar una determinada frecuencia dos veces, es decir, el tiempo de barrido de todo el rango de interés [29].

$$V(km/h) \leq \frac{864}{f_{(MHz)} t_r(s)} \quad (20)$$

Aunque los valores se obtuvieron de estudios sobre las bandas de frecuencia VHF (30 MHz a 300 MHz) y UHF (300 MHz a 3000 MHz) suponiendo un canal con distribución *Rayleigh*, se recomienda utilizarlos en cualquier situación. En 2.4 GHz, $\lambda = 0,125m$, $2L = 5m$, $d = 10cm$ y $N = 50$.

MINIMIZACIÓN DEL DRIVE TEST

MDT es una característica estandarizada por el 3GPP en UMTS y LTE, habilita el uso de los dispositivos de usuario para medir parámetros de desempeño en la red asociados con información de localización, que pueden ser analizados en sistemas de operación, administración y mantenimiento [3]. Es una característica prioritaria para los operadores porque permite reducir los costos de optimización de la red. En su versión 10 se introdujo una funcionalidad básica enfocada en proveer información sobre las condiciones de cobertura utilizando mediciones en el canal descendente; y en la versión 11 se realizaron mejoras para obtener parámetros relacionados con la QoS en los canales ascendente y descendente, como el volumen de datos, las fallas en los enlaces de radio y la señalización de acceso y *handover*, y una mayor precisión en la localización [30]. Estas mejoras permiten analizar la QoS en cada equipo de usuario de forma independiente e identificar los segmentos de la red que generan problemas.

Existen dos modos para realizar mediciones de MDT: en el modo activo o inmediato los parámetros se obtienen y se envían cuando el terminal está

conectado y comunicándose con la estación base, lo que permite analizarlos al momento de la captura; en el modo inactivo o pasivo, los parámetros se obtienen en momentos de inactividad y se envían posteriormente. En ambos casos los terminales son capaces de realizar las mediciones independientemente de la configuración de la red, pero depende de ella que estén disponibles para su estudio. También es posible que la estación base seleccione los terminales para MDT. La frecuencia de las muestras, que influye en el consumo de energía del dispositivo y en la carga de señalización sobre la red, y la precisión de la localización son algunos ajustes de configuración asociados con la MDT [27].

En el modo inactivo se pueden obtener parámetros del canal descendente, como el RSCP y el RSRQ en LTE o el RSCP y el E_c/N_0 en UMTS de las celdas vecinas, mientras que en el modo activo se incluyen la frecuencia de la portadora y el indicador de la celda activa. Todos los registros deben tener el tiempo y la localización, que puede ser el identificador de la celda o la latitud y la longitud, si el terminal lo permite. Los parámetros de eficiencia (*throughput*) y volumen de datos se capturan en periodos activos, cuando se requieren niveles de QoS para conversación, interactividad, *streaming* o tareas en segundo plano, y se toman en los canales ascendente y descendente, de forma independiente.

WiFi OFFLOADING

Las aplicaciones que consumen datos desde Internet y la proliferación de dispositivos de usuario final han cambiado el perfil de tráfico de las redes móviles y las proyecciones que se tenían sobre el uso del canal de datos. El tráfico constante generado por aplicaciones de interacción social hace que los radios permanezcan encendidos todo el tiempo, pasando de estado activo a estado ocioso, sin llegar al inactivo, lo que incrementa el consumo de batería en los dispositivos y la sobrecarga de señalización de control en las redes móviles. Para LTE *release 11* se propuso un método para controlar las transmisiones en *background* de aplicaciones que no son utilizadas directamente por el usuario como servicios y sesiones activas; se denominó asistencia del equipo de usuario y permite pasar el dispositivo a un modo de red de bajo consumo utilizando el PPI (*Power Preference Indication*) [31]. Para administrar el tráfico controlado por el usuario, a partir de LTE *release 10* se han generado estrategias para densificar las redes utilizando pequeñas celdas, teniendo en cuenta aspectos como el ahorro de energía, los costos asociados, el soporte a diversos tipos de tráfico, la eficiencia espectral y las crecientes tasas de transmisión [32].

Otra estrategia es la descarga controlada de tráfico desde el operador hacia redes WiFi, denominada WiFi *Offloading*, cuyo objetivo es reducir la carga de datos en las redes celulares usando tecnologías de acceso público. Ese planteamiento es posible porque el *core* de las redes LTE (*Evolved Packet Core*, EPC) es extensible a otras tecnologías de acceso, incluso a las no relacionadas con el 3GPP. Se definieron tres mecanismos para el *offloading*: acceso local IP (*Local IP Access*, LIPA), descarga de tráfico IP seleccionado (*Selected IP Traffic Offload*, SIPTO) y movilidad del flujo IP (*IP Flow Mobility*, IFOM)[33].

En el método LIPA, el equipo de usuario se conecta a una estación base del hogar (*H(e)NB*) y transfiere los datos a través de la red local, sin pasar por una estación base principal (*Evolved Node B*, eNB); su objetivo es que las comunicaciones entre los equipos de la red privada se enruten internamente. En el método SIPTO parte del tráfico IP se transmite a través de la red local, para reducir la carga del sistema, si esta ofrece mejores prestaciones para el usuario. En el método IFOM, el equipo de usuario mantiene varias sesiones con la red externa o con redes locales y es el que decide, de acuerdo con los indicadores de servicio y la presencia de nuevas redes, la forma en que se distribuye el tráfico.

Aunque hay esfuerzos del 3GPP para estandarizar el *IP Traffic Offloading* utilizando soluciones específicas que les permitan a los operadores implementar políticas y ejercer un control similar al que tienen en sus redes privadas, algunas investigaciones, como la desarrollada por Korhonen, Savolainen, Ding y Kojo [34], están encaminados a utilizar herramientas como IPv6 y DHCPv6 para acelerar la adopción con soluciones livianas y compatibles con los dispositivos actuales en el lado del operador y del usuario.

APLICACIONES RELACIONADAS

Portolan [35] es una aplicación de monitoreo basada en usuarios finales, desarrollada en la Universidad de Pisa (Italia), permite tomar datos georeferenciados a gran escala sobre la potencia de la señal (RSS) para generar mapas de cobertura y obtiene la topología de la red celular utilizando el *traceroute* desde los dispositivos de usuario.

Network Signal Info [36] y 2G,3G,4G,CDMA Network Monitor [37] son aplicaciones móviles que permiten acceder a los parámetros de las redes GSM/UMTS y WiFi en terminales de usuario, ambas utilizan el API de

telefonía incluido en las herramientas de desarrollo para recolectar datos y generar informes locales. Sin embargo, no se encuentra evidencia de que esa información sea consolidada y analizada para generar informes sobre gestión de espectro a gran escala.

AZQ Android [38] es una aplicación comercial que permite realizar las pruebas de *drive test* con un terminal de usuario y analizarlas en un software propietario. Entre sus características está la generación de reportes en web y en hojas de cálculo, y la reproducción de las capturas para análisis en tiempo.

Implementado con SDR, SDRDF [39], el de Balint Seeber es un sistema de *direction finding* basado en radio software diseñado para funcionar sobre un vehículo en movimiento con un arreglo de antenas en el techo; es un referente para la arquitectura de la aplicación y para la implementación de algoritmos que incluyan el mapeo de coordenadas de GPS (*Global Positioning System*) con información sobre parámetros de red.

AvMap [40] permite realizar el seguimiento de aviones a partir de la información difundida por las aeronaves y las torres de control aeroportuarias, es un referente para la representación de la información georeferenciada y los reportes que pueden generarse con esos datos.

gr-802.11 [41] es un receptor OFDM de 802.11a, g y p implementado en SDR, incluye la detección de tramas, la corrección del *frequency offset*, el alineamiento de símbolos, la corrección de fase, la estimación de canal, la decodificación de campo y la decodificación de las tramas.

APLICACIÓN MÓVIL

ARQUITECTURA

La FIGURA 5 muestra la arquitectura de software independiente de la plataforma que se definió para la aplicación móvil, a partir de las historias de usuario consignadas en la TABLA 1. Los componentes son: servicio de localización (*location service*), fuentes de radio (*radio sources*), perfilador (*profiler*), administrador de radios (*radio manager*), administrador de entrada y salida (*IO manager*), orquestador (*orchestrator*) e interfaz de usuario (GUI).

El servicio de localización, además de obtener la localización del usuario a través del GPS u otros métodos disponibles, administra la frecuencia con

Tabla 1. Historias de usuario

Nombre	Descripción
Iniciar medición	Iniciar el proceso de medición manualmente para capturar los parámetros de red
Terminar medición	Finalizar la medición en cualquier momento
Georreferenciación	Los parámetros deben estar localizados geográficamente
Exportar medición	La información de la captura debe ser exportada a un formato común
Visualizar medición en curso	Mostrar la información que está siendo recolectada
UIT	El sistema debe seguir las recomendaciones de la UIT para drive test

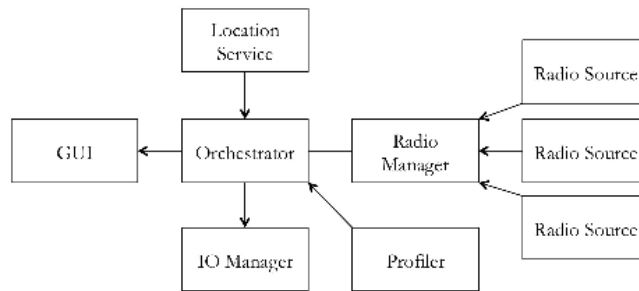


Figura 5. Arquitectura de la aplicación móvil

la que se obtiene la posición e identifica si un nuevo registro es mejor que el anterior o si debe descartarse. Las fuentes de radio obtienen los parámetros o los indicadores de rendimiento de una tecnología particular: UMTS, LTE o WiFi, y almacenan la lista de cambios en los parámetros hasta que sean solicitados por el orquestador. El perfilador genera el perfil del dispositivo, el cual indica las tecnologías de radio que están disponibles para realizar el *drive test* y sus limitaciones. El *radio manager* instancia y administra las fuentes de radio, de acuerdo con el perfil entregado por el orquestador y generado por el perfilador. El administrador de entrada y salida guarda las muestras del *drive test* en el sistema de archivos del dispositivo, asigna el nombre de los ficheros y controla su lectura. El orquestador es el componente principal del sistema, se encarga de administrar los ciclos de muestras, obtener la localización desde el servicio de localización, obtener los parámetros desde el administrador de radios, guardar las muestras a través del administrador de entrada y salida

y generar la información para la interfaz de usuario. La interfaz de usuario (*Graphic User Interface*, GUI) se encarga de la interacción con el usuario, para lo que recibe datos desde el orquestador y los despliega en la pantalla.

IMPLEMENTACIÓN EN ANDROID

Para el componente móvil en Android se utilizaron las herramientas de desarrollo más comunes y se implementó la arquitectura teniendo en cuenta actividades y servicios, los componentes básicos de las aplicaciones. Se desarrollaron cuatro módulos, como se muestra en la FIGURA 6. El primero, denominado módulo de monitoreo, obtiene los parámetros de red cada vez que se le solicite o cada vez que se presenta un cambio de celda (TABLA 2).



Figura 6. Implementación del componente móvil en Android

Tabla 2. Parámetros obtenidos y sus fuentes

Fuente	Parámetros
Dispositivo	Tipo de red que soporta (GSM, CDMA, SIP), identificador y versión de software.
Red	País, id y nombre del operador, id del suscriptor y tipo de red (UMTS, HSDPA, HSDPA+, LTE).
Celda	Identificador de la celda o de la estación base, código de localización de la celda o la estación base y lista de celdas vecinas.
Señal	BER, Ec/Io, SNR y RSSI.
Celdas vecinas	Tipo de red, identificador, código de localización, código de sincronización primario (Primary Synchronization Code, PSC) para UMTS y RSSI.

El segundo, denominado módulo de localización, obtiene la latitud, longitud, altitud y precisión del dispositivo frecuentemente, durante el tiempo de realización del *drive test*, y compara las muestras de ubicación con las anteriores para siempre retornar la más acertada. El tercero, denominado módulo de

almacenamiento, crea archivos de monitoreo en la memoria externa del dispositivo, guarda las muestras que se le indiquen y cierra el archivo; el cuarto, denominado módulo de control o de *drive test*, se encarga de la configuración inicial del monitoreo –lo que incluye la descripción y frecuencia con la que se van a tomar las muestras–, activa los otros módulos y controla el proceso de *drive test*, el cual, en cada iteración solicita la última localización al módulo de localización y el estado actual de la red al módulo de monitoreo, convierte la información de monitoreo en el formato de almacenamiento y envía la información de monitoreo al módulo de almacenamiento.

La FIGURA 7 muestra el componente móvil en Android. En la primera imagen aparece el formato de las muestras –que son objetos JSON (*JavaScript Object Notation*)– y contiene la variables detalladas en la TABLA 3; la segunda figura, con la interfaz adaptada, muestra los puntos en los que se realizaron las muestras, con un botón en la parte superior que permite iniciar o detener la captura; y la tercera muestra el momento en que se solicita la descripción antes de iniciar un nuevo *drive test*.

Tabla 3. Variables del componente móvil Android

Variable	Descripción
networkType	Valor entero, representa el tipo de red a la que está conectado el dispositivo, así: 1, GSM; 3, UMTS; 4, CDMA; 8, HSDPA; y 13, LTE.
networkOperator	Identificador numérico del operador de la red.
networkOperatorName	Texto con el nombre corto del operador de la red.
subscriberId	Identificador del suscriptor en la red.
deviceId	Identificador del dispositivo.
Location	Ubicación del dispositivo: latitud, longitud, altitud y precisión en metros.
cellId BSIId	Identificador de la celda a la que está conectado el dispositivo.
cellLac BSLoc	Ubicación de la estación base
Strength	RSSI según [?, 8.5], así: 0, –113 dBm o menos; 1, –111 dBm; 2 a 30, –109 a –53 dBm; 31, –15 dBm o más; y 99, desconocido o ausente
Ber	BER según [?, 8.2.4], así: 0, 0.14 %; 1, 0.28 %; 2, 0.57 %; 3, 1.13 %; 4, .26 %; 5, 4.53 %; 6, 9.05 %; 7, 18.10 %; y 99, desconocido / ausente.
dataState	Estado de la conexión, así: 0, desconectado; 1, conectándose; 2, conectado; y 3, suspendido.
dataActivity	Dirección del flujo de datos, así: 0, ninguno; 1, entrada; 2, salida; 3 entrada y salida; y 4, desconectado.
neighboringCellInfo	Colección de celdas vecinas y sus parámetros

Sistema de benchmarking de redes móviles celulares y WiFi basado en dispositivos de usuario final y SDR

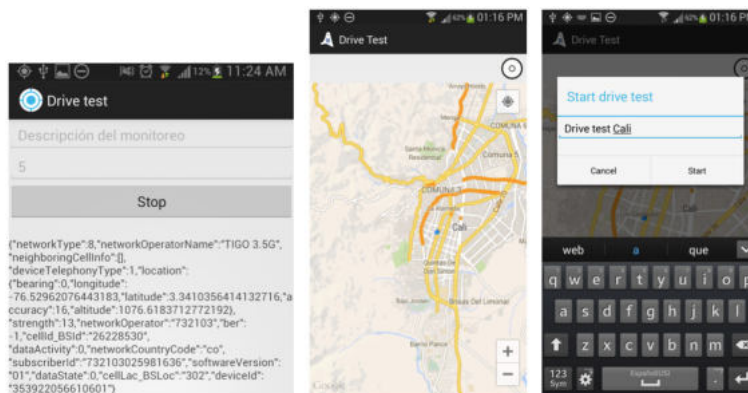


Figura 7. Aplicación móvil

RESTRICCIONES

Se identificaron restricciones para obtener datos de dispositivos Android debido a que no todos los fabricantes implementan correctamente las interfaces de desarrollo, lo que reduce la cantidad de parámetros medidos. Se obtuvieron los parámetros SNR, Ec/Io y BER y, en algunos dispositivos de prueba, la lista de celdas vecinas.

Se encontró que para obtener el resto de los KPI (*Key Performance Indicators*) es necesario modificar el sistema operativo y exponerlos desde los controladores de red. De esta forma, la implementación de características, como MDT, no depende de aplicaciones de terceros, como la que se propone aquí, sino de los desarrollos y ajustes que cada fabricante haga sobre el sistema operativo de acuerdo con el estándar y los parámetros que exponga en sus API.

APLICACIÓN SDR

HERRAMIENTAS

Para la aplicación SDR se decidió utilizar GNU Radio, el bladeRF y el USRP2. GNU Radio es una herramienta de código abierto que provee bloques de procesamiento digital de señales (DSP) para implementar radios en software basados en estructuras de flujos de trabajo, en lugar de utilizar *hardware* dedicado.

El bladeRF de Nuand y el USRP (*Universal Software Radio Peripheral*) de Ettus Research y National Instruments son equipos de radiofrecuencia (RF) capaces de procesar señales utilizando GNU Radio, sus componentes principales son un convertidor análogo-digital (ADC), un convertidor digital-análogo (DAC) y un FPGA. El bladeRF tiene frecuencias de operación de 300MHz a 3.8GHz, con un ancho de banda de hasta 28MHz, mientras que al USRP se le pueden adaptar *daughterboards* que cubren rangos específicos. La FIGURA 8 muestra los radios utilizados.



Figura 8. Hardware de radiofrecuencia

APLICACIONES

IMPLEMENTACIÓN USANDO LOS BLOQUES OFDM DE GNU RADIO

GNU Radio ofrece bloques genéricos para procesamiento OFDM que se pueden parametrizar para crear distintos receptores (ver FIGURA 9). El bloque *osmocom Source* actúa como puente con el bladeRF y se encarga de llevar las muestras desde el *hardware* hacia GNU Radio, en él se configura la frecuencia central (2.412GHz corresponde al canal 1 de 802.11g), el ancho de banda de 22MHz, la tasa de muestreo equivalente al ancho de banda (por tratarse de un sistema con muestras complejas) y la ganancia de la antena. La salida del bloque es un flujo de muestras complejas en el dominio del tiempo.

Schmidl & Cox OFDM synch se encarga de sincronizar la señal utilizando el método [17] que se describe en la sección de marco teórico. Se parametriza

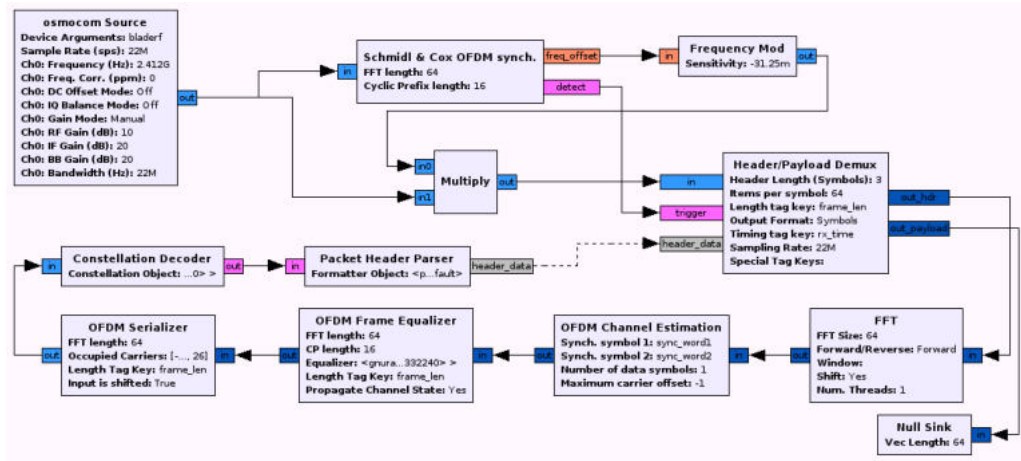


Figura 9. Receptor OFDM en GNU Radio

con el tamaño de la FFT N y el tamaño del prefijo cíclico N_{cp} , para obtener como salidas la diferencia en fase $\hat{\phi}$ y una marca de tiempo que indica la detección de una trama sobre el flujo de muestras. A diferencia de lo descrito en el marco teórico, $R(d)$ se calcula de acuerdo con la ECUACIÓN 21, es decir que representa la energía recibida de las dos mitades del primer símbolo, lo que se hace, según los desarrolladores, para prevenir detecciones falsas al final de las ráfagas, donde el nivel de energía cae repentinamente.

$$R(d) = \frac{1}{2} \sum_{m=0}^{N-1} |r_{d+m}|^2 \quad (21)$$

Schmidl & Cox OFDM synch indica la fase donde se debe corregir la señal, pero no la realiza, para hacerlo, *Frequency Mod* recibe el valor $\hat{\phi}$ y genera una sinusoidal $e^{-j2t\hat{\phi}/T}$, que se multiplica con la señal utilizando el bloque *Multiply*. Esto es posible porque las modulaciones en fase y frecuencia son ambas modulaciones angulares similares en sinusoidales simples.

El bloque *Header/Payload Demux* se encarga de separar el preámbulo y la cabecera del resto de la señal. Se parametriza con el número de ítems por símbolo N , el intervalo de guarda N_{cp} y la longitud de la cabecera N_p , para obtener los dos flujos independientes. Las entradas son el flujo de muestras, la marca de tiempo que indica el inicio de las tramas y la cabecera decodificada.

Primero descarta todas las muestras hasta que recibe la marca de tiempo, copia N_p muestras en la salida *out_hdr* y espera la entrada de la cabecera decodificada. Con la información de la cabecera, copia la cantidad de datos indicada en el campo *LENGTH* en la salida *out_payload* y descarta N_{cp} muestras.

La cabecera se pasa al dominio de la frecuencia con el bloque *FFT*, que recibe el número de componentes N y computa la transformada sobre cada una de las muestras. *OFDM Channel Estimation* separa los símbolos de sincronización del campo *SIGNAL*, por lo que debe parametrizarse con dos secuencias de sincronización correspondientes a los conjuntos de símbolos cortos y largos según [18, Tablas L-2 y L-5]; realiza la estimación de canal; y calcula una segunda corrección en frecuencia, esta vez en número de subportadoras, lo que corresponde a variaciones que se encuentren al identificar los símbolos de sincronización.

El bloque *OFDM Frame Equalizer* realiza la corrección de frecuencia y ecualiza el campo *SIGNAL* de acuerdo con el esquema de modulación BPSK con $R = 1/2$. Y *OFDM Serializer* remueve las portadoras piloto, por lo que es necesario parametrizarlo con la lista de portadoras de datos $\{-26, \dots, -22, -20, \dots, -8, -6, \dots, -1, 1, \dots, 6, 8, \dots, 20, 22, \dots, 26\}$.

La constelación se decodifica con *Constellation Decoder* y a partir de los primeros 4 bits del campo *SIGNAL*, en el subcampo *RATE* es posible identificar la tasa de transmisión y el esquema de modulación del resto de la señal, de acuerdo con las tablas 18-4 y 18-6 de [18], en donde se relaciona cada combinación de 4 bits con la tasa de transmisión y el esquema de modulación que se requiere para alcanzarla. Se realizó una prueba con un *access point* configurado con 802.11g en el canal 1 y se obtuvo el valor 0011, el cual corresponde a 54Mb/s, es decir, 64QAM con tasa de 3/4.

Los bloques que ofrece GNU Radio para OFDM son útiles para ambientes controlados en el que se conocen los parámetros del transmisor y el receptor, sin embargo, para implementar un estándar se requiere de bloques adicionales, como los que están disponibles en gr-802.11.

IMPLEMENTACIÓN USANDO GR-802.11

Se implementó una aplicación basada en gr-802.11, un módulo de GNU Radio que ofrece bloques de procesamiento para WiFi. Como se muestra en la FIGURA 10, la aplicación cuenta con tres secciones: detección de tramas, sincronización

y decodificación, de acuerdo con lo propuesto por Bloessl, Segata, Sommer y Dressler [41].

El bloque *UHD USRP Source* sirve de puente entre el *hardware* de RF, en este caso el USRP, y GNU Radio. Se parametriza con la frecuencia central de 2.412GHz y la tasa de muestreo de 20MHz. Para detectar las tramas se calcula la autocorrelación de las muestras en una ventana de tiempo aprovechando la secuencia de símbolos cortos en el preámbulo OFDM, la que consiste en un patrón que se extiende por dieciséis muestras y se repite diez veces, de tal forma que la mayor correlación se da al principio de cada trama.

El bloque *OFDM Sync Short* identifica los símbolos cortos y acumula una cantidad de muestras fijas cuando detecta una alta correlación en la entrada *cor*, se parametriza con el umbral de autocorrelación y el número de muestras que debe acumular. En esta implementación es necesario conocer de antemano el valor del campo *LENGTH* de la cabecera PLCP, lo que constituye una restricción importante al momento de tomar las muestras.

En *OFDM Long Sync* se aplica la corrección en frecuencia y se alinean los símbolos en el dominio del tiempo, es decir, se identifica en dónde empieza y en

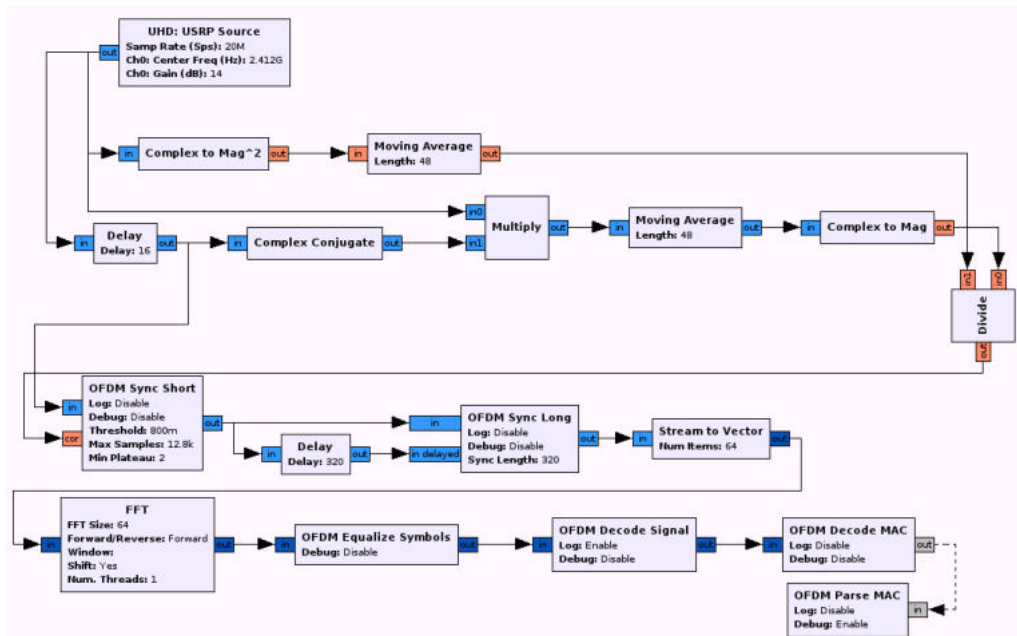


Figura 10. Aplicación SDR usando gr-802.11

dónde termina cada conjunto de 64 subportadoras, para lo que se aprovecha de que la secuencia de símbolos largos consiste en un patrón de 60 muestras que se repite 2.5 veces, lo que hace posible computar la correlación e identificar el primer símbolo de datos. Recibe como parámetro la longitud de la secuencia de sincronización $N_p = 320$, como se indicó en el marco teórico.

El bloque *FFT* computa la transformada sobre las muestras y *OFDM Equalize Symbols* se encarga de la corrección en fase y de la estimación de canal, lo que consiste en corregir la magnitud de las subportadoras para evitar problemas de decodificación. Retira las portadoras piloto y de guarda, por lo que la salida tiene una longitud $N_c = 48$. *OFDM Decode Signal* decodifica el símbolo *SIGNAL* del PLCP para obtener el esquema de modulación y la longitud de los datos. Su salida en pantalla es de tipo *gr::log:INFO: ofdm decode mac0 - encoding: 0 - length: 40 - symbols: 15*, en donde los valores de *encoding* están dados por la TABLA 4. Esa información pasa a *OFDM Decode MAC*, donde se hace: la demodulación, el *deinterleaving*, la decodificación convolucional y el descifrado de los datos de acuerdo con el estándar.

Tabla 4. Significado de los valores de encoding

Nombre	Descripción	Valor
0	BPSK	1/2
1	BPSK	3/4
2	QPSK	1/2
3	QPSK	3/4
4	16QAM	1/2
5	16QAM	3/4
6	64QAM	2/3
7	64QAM	3/4

Como restricción de esta implementación se encontró que la decodificación está limitada a modulaciones BPSK o QPSK y a conjuntos de datos de tamaño fijo; sin embargo, puede utilizarse para obtener periódicamente la información del campo *SIGNAL*, algo de gran interés para el *drive test*.

APLICACIÓN SDR PARA MONITOREO REMOTO

Con los objetivos de extender el sistema a otras frecuencias de operación y controlar remotamente el *hardware* de RF, se desarrolló una aplicación SDR

que ofrece las funcionalidades básicas de un analizador de espectro. Consta de dos partes, una en GNU Radio, que se describe en esta sección; otra Web, que se describe en la sección siguiente.

Las funcionalidades que se incluyeron en la aplicación fueron las de visualización de la señal, marcadores, búsqueda de picos (*peak search*), promedio (*average*), fijación de máximos (*max hold*), asignación de valores de frecuencia (inicial, final y central), *span*, nivel de referencia (*reference level*) y dB/div.

Para el *hardware* de RF se desarrolló la aplicación de la FIGURA 11. Ella: obtiene los valores de potencia de la señal, les agrega un patrón de sincronización y los envía como un flujo UDP. Primero convierte el flujo en vectores de tamaño $N = 1024$ utilizando el bloque *Stream to Vector*, equivalente al número de componentes de la transformada que se computa con el bloque FFT.

Complex to Mag obtiene la magnitud de los componentes de la transformada de acuerdo con la ECUACIÓN 22 y en *Log10* se computan los valores de potencia en dB. El bloque *Single Pole IIR Filter* es un filtro de respuesta infinita al impulso que previene una entrada de cero al logaritmo, porque ante entradas de impulso su salida tiene un número infinito de términos distintos de cero.

$$|z| = \sqrt{x^2 + y^2} \quad \text{para } z = x + yi \quad (22)$$

Vector to Stream convierte los vectores en un flujo de datos y *Vector Insert* inserta la palabra de sincronización `0x1271C351` cada N muestras, es decir, en la posición $N + 1$, para asegurar que quien reciba el flujo pueda reconstruir los vectores. El bloque *UDP Sink* transmite las muestras a la dirección IP y al puerto indicado.

Los parámetros se pueden modificar remotamente utilizando llamados XML-RPC, por lo que es posible especificar: las frecuencias central, inicial y final; el tamaño de la FFT; la tasa de muestreo (el *span* en los sistemas digitales); el ancho de banda y la dirección IP; y el puerto destino del flujo UDP. XML-RPC es un estándar, por lo que se pueden implementar clientes en diversas plataformas, teniendo en cuenta el nombre de los métodos en el dispositivo.

Es posible implementar una interfaz local con aceleración gráfica utilizando el módulo *gr-fosphor* de GNU Radio, el cual, aunque requiere de un gran

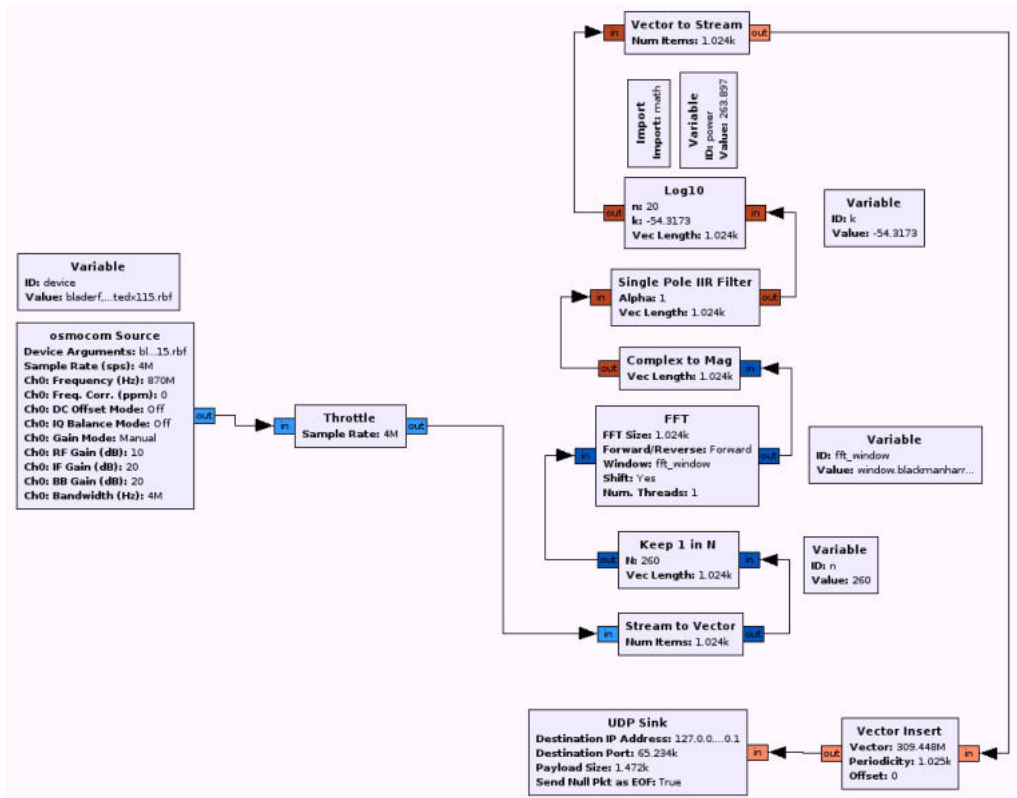


Figura 11. Aplicación SDR para monitoreo remoto

poder de cómputo, ofrece las funcionalidades básicas de un analizador de espectro. Ahí es posible observar el piso de ruido de la señal sobre -85dB y los impulsos de las subportadoras a -40dB. El pico que se observa en DC es causado por el *hardware* y se puede reducir al calibrar el dispositivo, sin embargo, ese comportamiento es común cuando el receptor es de conversión directa o de frecuencia intermedia cero, por la calidad de la electrónica.

También se pueden lograr despliegues autónomos utilizando *hardware* como el USRP E110, que incluye un microprocesador limitado en potencia, o para montajes con sistemas embebidos, como el que se muestra en la FIGURA 12. El primero tiene una versión de GNU Radio embebida en el dispositivo por lo que necesita ser conectarlo a un equipo que haga el procesamiento, pero su ancho de banda está limitado a 4MHz y a operaciones que no requieran de grandes volúmenes de datos. Para el segundo se utilizó un Raspberry Pi



Figura 12. bladeRF conectada a un sistema embebido Raspberry Pi

sin GNU Radio, sino con las herramientas propias del bladeRF, como es la interfaz por consola bladeRF-cli (*bladeRF Command Line Interface*), que permite realizar capturas en texto plano de los valores de potencia recibidos por el *hardware*, archivos que se pueden utilizar como fuente de datos para realizar análisis en modo desconectado en otro equipo con GNU Radio.

BladeRF-cli permite utilizar comandos como: *set frequency 24e8*, para asignar la frecuencia central en 2.4GHz; *set bandwidth 28e6*, para configurar el ancho de banda en 28MHz, el máximo permitido por el dispositivo; y *set samplerate 28000000*, para asignar la tasa de muestreo. Con *rx config file=filename.csv format=csv n=4096* se ordena una captura de 4096 muestras que se deben almacenar en el archivo *filename.csv*, utilizando un formato separado por comas; el comando *rx start* inicia la captura. Los datos registrados son de la forma x, y para muestras complejas $x + yi$.

Al tratarse de dispositivos embebidos con interfaces de red, tanto el USRP E110, como el Raspberry Pi, se pueden controlar a través de SSH (*Secure Shell*) y es posible acceder a su sistema de archivos utilizando SFTP (*SSH File Transfer Protocol*) para recuperar las capturas. Para analizar los datos, es indispensable contar con un *backend*.

BACKEND

Como *backend* se creó una aplicación que consta de dos servidores, uno para control, otro para administrar el flujo de información, como se muestra en

la FIGURA 13. El primero permite registrar nuevos dispositivos de monitoreo y controlarlos de modo remoto, haciendo la conexión entre el usuario y la aplicación SDR; el otro habilita el control de flujo de la información que llega de distintos dispositivos hacia distintos clientes.

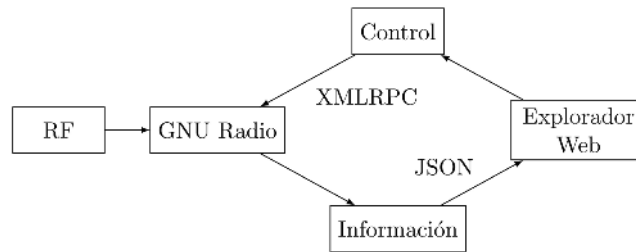


Figura 13. Distribución del backend

El servidor de control realiza llamados XML-RPC para cambiar los parámetros del dispositivo SDR –como se indicó en la descripción de la aplicación SDR para monitoreo remoto–, y expone servicios web RESTFUL que permiten al usuario realizar esos llamados desde la interfaz gráfica. Se desarrolló utilizando Django, un *framework* web para Python, e incluye una base de datos PostgreSQL con soporte geoespacial, que contiene información sobre las tablas de asignación de frecuencias, según el país, y registra la localización de los dispositivos de monitoreo. En la FIGURA 14 se aprecian los sistemas georeferenciados, los cuales pueden ser marcados como fijos (F), móviles (de *drive test*, M) y transportables (T). A cada uno se le asigna: nombre, descripción, dirección IP y localización inicial; para controlarlos basta con seleccionarlos en el mapa y abrir la interfaz de gestión remota.

El servidor de información utiliza UDP para recibir el flujo desde el dispositivo SDR y lo distribuye utilizando web sockets, lo que se hace para incrementar el rendimiento y reducir la latencia que implica utilizar otras tecnologías de transporte como TCP. Se desarrolló utilizando Node.js, un *framework* JavaScript para aplicaciones en tiempo real y la librería Socket.IO. Incluye una función para identificar la palabra de sincronización 0x1271C351 y para traducir el formato de punto flotante de C++, con el que llegan las muestras, a un formato de colecciones JSON que se puede utilizar en un explorador web. El proceso de visualización, que incluye el gráfico de la señal y comandos como

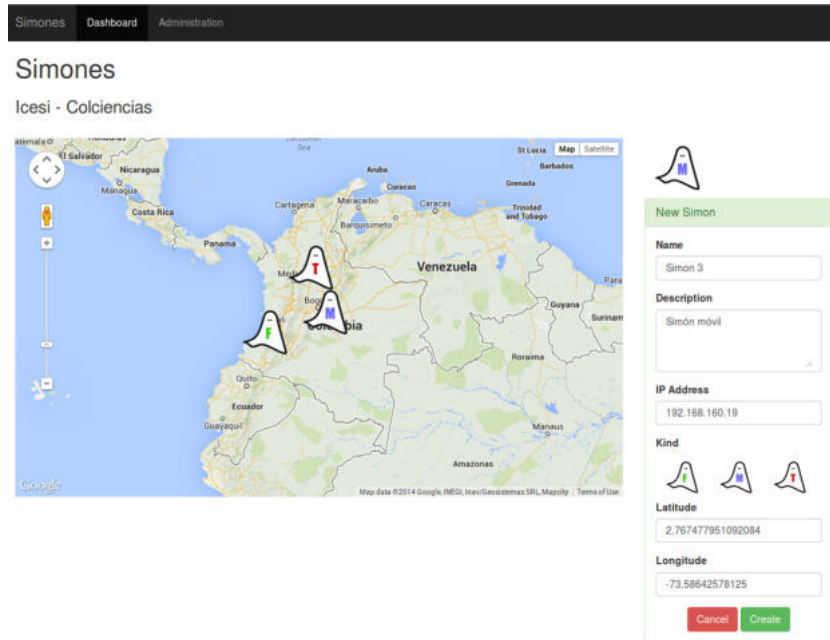


Figura 14. Sistemas de monitoreo georeferenciados

max hold, *average*, *peak search*, *dB/div* y *reference level*, se hace en un explorador web utilizando JavaScript y las librerías *Flot Charts* y *Angular.js*.

La FIGURA 15 muestra la interfaz de gestión remota: en el panel superior aparece información básica sobre el dispositivo (descripción, dirección IP y localización); en el panel izquierdo están los controles de un analizador de espectro, desde donde se pueden cambiar los parámetros y activar las funciones *average*, *peak search* y *max hold*; y en el panel central está la señal capturada desde el *hardware* de RF.

El panel central ofrece varias opciones de interacción: las líneas verticales indican el pico detectado (en rojo) y la ubicación del puntero del ratón (en verde); la gráfica amarilla representa la señal y la azul los valores máximos: En la parte inferior derecha están los detalles del pico: la frecuencia en la que se encontró y la potencia. Y en la parte inferior izquierda, los detalles de la frecuencia seleccionada por el usuario con el puntero, que son: la frecuencia, la potencia en la que se posicionó y el valor actual de potencia en ese punto.



Figura 15. Gestión remota del sistema de monitoreo

Como opciones de visualización se agregaron *reference level*, *dB/div* y *average*. En sistemas digitales no se tienen en cuenta la *resolution bandwidth* ni el *video bandwidth*, porque al tratarse de un sistema basado en FFT, siempre se cuenta con el mismo número de muestras N independiente del ancho de banda que se esté visualizando. En este desarrollo se utilizó $N = 1024$.

La implementación incluye las funciones de un analizador de espectro y permite realizar análisis básico de señales con *hardware* basado en FFT y aplicaciones SDR. Al tener monitores georeferenciados, es posible comparar los niveles de servicio en distintos sitios, lo que la constituye en una herramienta de gran interés para las agencias de control y los operadores de telecomunicaciones.

CONCLUSIONES Y TRABAJO FUTURO

La principal conclusión de esta investigación es que es posible construir aplicaciones de monitoreo remoto del espectro utilizando equipos sencillos,

como el USRP y el bladeRF, para desarrollar unidades de monitoreo simples, de operación desatendida y de bajo costo, que apoyen la labor de monitoreo y complementen los sistemas de gestión de espectro y las unidades de monitoreo existentes. Al contar con procesamiento local configurable en software, esas unidades pueden implementar las funciones mínimas requeridas en las recomendaciones de la UIT para países emergentes [42].

Es posible también implementar el nivel físico de un estándar utilizando SDR, sin embargo, información importante para optimización, como el BSSID en redes WiFi, se maneja en el nivel MAC, que aún está en etapa experimental en GNU Radio. Medidas como el nivel de señal, el ancho de banda y la radiación no ionizante se pueden obtener desde el nivel físico, para apoyar tareas de monitoreo de espectro, al tiempo que se trabaja en implementaciones sobre el nivel de enlace.

La implementación de características como MDT no depende de aplicaciones móviles de terceros, sino de los desarrollos y de los ajustes que cada fabricante haga sobre el sistema operativo, de acuerdo con el estándar, y de los parámetros que exponga en sus API. Sin los parámetros adecuados es difícil caracterizar las redes utilizando dispositivos de usuario final, se presenta entonces la oportunidad de desarrollar unidades de monitoreo basadas en los mismos dispositivos con un sistema operativo orientado a esa labor.

Los bloques que ofrece GNU Radio para OFDM son útiles para ambientes controlados donde los parámetros del transmisor y el receptor se conocen. Sin embargo, para implementar un estándar se requieren bloques adicionales, como los que están disponibles en gr-802.11. La implementación de estándares es un trabajo en progreso dentro de la comunidad de SDR, por lo que representa una oportunidad para hacer aportes.

Una tarea a futuro es la adaptación del desarrollo de *backend* para convertirlo en un sistema de monitoreo de espectro que cumpla con la recomendación ITU-R SM.1392-2 [42] y su integración con el SMS4DC (*Spectrum Management System for Developing Countries*) de la UIT y con TES Monitor, una herramienta de monitoreo de espectro desarrollada por TES America, la empresa aliada de Icesi en el proyecto Simones. Se espera que el sistema sea compatible con dispositivos SDR y con otros basados en FFT.

Con la llegada de 5G y las recientes propuestas sobre comunicación dispositivo a dispositivo, incluso en esquemas multisalto, toma más relevancia el *data*

offloading. Queda abierta una propuesta de investigación para la planificación y optimización de redes WiFi que permita soportar servicios de redes convergentes y procedimientos como el WiFi *offloading*. Las implementaciones en SDR de las otras versiones de 802.11 y de LTE/5G podrían apoyar la captura de datos para esa investigación y su utilización con fines académicos en la apropiación de conocimiento sobre OFDM.

Se propone la adaptación del sistema operativo Android para labores de monitoreo, de tal manera que sea posible superar las restricciones encontradas y exponer todos los KPI, y así poder construir un sistema de monitoreo basado en dispositivos móviles que permita validar si el análisis de los KPI del lado del usuario, en relación con las predicciones de cobertura, puede apoyar los procedimientos de monitoreo y optimización tradicionales y la caracterización de una red LTE/5G.

Con SDR y GNU Radio se puede realizar el procesamiento digital de señales y aprovechar todas las prestaciones del equipo, como la aceleración gráfica. Las características de los equipos requeridos para lograr esas aplicaciones muestran una oportunidad de mejora en la optimización de SDR, orientada a alcanzar rendimientos similares a los del *hardware* especializado.

REFERENCIAS

- [1] T. Kürner [Ed.], “Final report on self-organisation and its implications in wireless access networks,” Socrates-FP7, INFSO-ICT-216284 SOCRATES D5.9, 2010.
- [2] *LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Self-configuring and self-optimizing network (SON) use cases and solutions* *LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Self-configuring and self-optimizing network (SON) use cases and solutions*, 3GPP TR 36.902 v. 9.3.1 Release 9, 2011.
- [3] J. Johansson, W. A. Hapsari, S. Kelley, and G. Bodog. Minimization of drive tests in 3GPP release 11, *IEEE Communications Magazine*, vol. 50, no. 11, pp. 36-43, 2012.
- [4] J. Matamales, D. Martin-Sacristan, J. F. Monserrat, N. Cardona, “Performance Assessment of HSDPA Networks from outdoor drive-test measurements,” in VTC Spring, Barcelona, España, 2009, doi: 10.1109/VETECS.2009.5073782
- [5] *Anritsu Drivetest Solutions* [Online], available: <http://www.anritsu.com/>
- [6] *JDSU Drive Test Solutions* [Online], Available: <https://www.viavisolutions.com/en-us>
- [7] D. Calin, H. Claussen, and Uzunalioglu, “On Femto deployment architectures and macrocell offloading benefits in joint macro-Femto deployments,” *IEEE Communications Magazine*, vol. 48, no. 1, 26-32, Jan. 2010.

- [8] A. Golaup, M. Mustapha, and L. Patanapongpibul, "Femtocell access control strategy in UMTS and LTE," *IEEE Communications Magazine*, vol. 47, no. 9., pp. 117-123, Sept. 2009.
- [9] *Wireless local area networks*, Standard IEEE 802.11, 2018.
- [10] S. Tozlu, M. Senel, Wei Mao, and A. Keshavarzian, "Wi-fi enabled sensors for internet of things: A practical approach," *IEEE Communications Magazine*, vol. 50, no. 6, pp. 134-143, Jun. 2012.
- [11] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26-38, May. 1995.
- [12] L. Hanzo, W. Webb, and T. Keller, *Single and multi carrier quadrature amplitude modulation*, West Sussex, UK, John Wiley & Sons, 2000.
- [13] N. Cvijetic, D. Qian, and J. Hu, "100 gb/s optical access based on optical orthogonal frequency-division multiplexing," *IEEE Communications Magazine*, vol. 48, no. 7, pp. 70-77, July 2010.
- [14] C.M. Vithanage, M. Sandell, J.P. Coon, and Y. Wang, "Precoding in ofdm-based multi-antenna ultra-wideband systems," *IEEE Communications Magazine*, vol. 47, no. 1, pp. 41-47, Jan. 2009.
- [15] S.B. Weinstein. "The history of orthogonal frequency-division multiplexing [history of communications]," *IEEE Communications Magazine*, vol. 47, no. 11, pp. 26-35, Nov. 2009.
- [16] C.R. Johnson, W.A. Sethares, and A.G. Klein, *Software receiver design: Build your own digital communication system in five easy steps*, Cambridge, UK, Cambridge University Press, 2011.
- [17] T.M. Schmidl and D.C. Cox, "Robust frequency and timing synchronization for OFDM," *IEEE Transactions on Communications*, vol. 45, no. 12, pp. 1613-1621, Dec. 1997.
- [18] *Wireless LAN medium access control (MAC) and physical layer (Phy) specifications.*, IEEE Standard 802.11-2007.
- [19] G.R. Hiertz, D. Denteneer, L. Stibor, Y. Zang, X.P. Costa, and B. Walke, "The IEEE 802.11 universe," *IEEE Communications Magazine*, vol. 48, no. 1, pp. 62-70, Jan. 2010.
- [20] *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2*, 3GPP TS 36.300 version 11.6.0 Release 11, July 2013
- [21] *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Physical channels and modulation*, 3GPP TS 36.211 version 11.5.0 Release 11, Jan. 2014
- [22] C. Hoymann, D. Larsson, H. Koorapaty, and J-F, Cheng, "A lean carrier for LTE," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26-38, Feb. 2013.
- [23] *LTE PHY Layer Measurement Guide* (Online), available: <https://www.viavisolutions.com/en-us/literature/lte-phy-layer-measurement-guide-application-notes-en.pdf>

- [24] A. Ghosh, N. Mangalvedhe, R. Ratasuk, B. Mondal, M. Cudak, E. Visotsky, T. A. Thomas, J. G. Andrews, P. Xia, H-S. Jo, H. S. Dhillon, T. D. Novlan, “Heterogeneous cellular networks: From theory to practice”, *IEEE Communications Magazine*, vol. 50, no. 6, pp. 54-64, June 2012.
- [25] *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer; Measurements*. 3GPP TS 36.214 version 11.0.0 Release 11, Feb. 2013
- [26] *Universal Mobile Telecommunications System (UMTS); User Equipment (UE) procedures in idle mode and procedures for cell reselection in connected mode*, 3GPP TS 25.304 version 11.0.0 Release 11, Oct. 2012.
- [27] W. A. Hapsari, A. Umesh, M. Iwamura, M. Tomala, B. Gyula, and B. Sebire, “Minimization of Drive Tests Solution in 3GPP”, *IEEE Communications Magazine*, vol. 50, no. 6, pp. 28-36, Jun. 2012.
- [28] D. de la Vega, S. Lopez, J. M. Matías, U. Gil, I. Peña, M. Velez, J.L. Ordiales, and P. Angueira, “Generalization of the lee method for the analysis of the signal variability,” *IEEE Transactions on Vehicular Technology*, vol. 58, no. 2, pp. 506-516, Feb. 2009.
- [29] *Field-strength measurements along a route with geographical coordinate registrations*. ITU-R Rec. SM.1708-1, 2011.
- [30] *Universal Mobile Telecommunications System (UMTS); LTE; Universal Terrestrial Radio Access (UTRA) and Evolved Universal Terrestrial Radio Access (E-UTRA); Radio measurement collection for Minimization of Drive Tests (MDT); Overall description; Stage 2*, 3GPP TS 37.320 version 11.1.0 Release 11, Nov. 2012.
- [31] M. Gupta, S. C. Jha, A. T. Koc, R. Vannithamby, “Energy impact of emerging mobile Internet applications on LTE networks: Issues and solutions,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 90-97, Feb. 2013.
- [32] T. Nakamura, S. Nagata, A. Benjebbour, Y. Kishiyama, T. Hai, S. Xiaodong, Y. Ning, and L. Nan, “Autho trends in small cell enhancements in LTE advanced,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 98-105, Feb. 2013.
- [33] C.B. Sankaran, “Data offloading techniques in 3GPP rel-10 networks: A tutorial,” *IEEE Communications Magazine*, vol. 50, no. 6, pp. 46-53, June 2012.
- [34] J. Korhonen, T. Savolainen, A. Yi-Ding, M. Kojo, “Toward network controlled IP traffic offloading,” *IEEE Communications Magazine*, vol. 51, no. 3, pp. 96-102, March 2013.
- [35] A. Faggiani, E. Gregori, L. Lenzini, V. Luconi, and A. Vecchio, “Smartphone-based crowdsourcing for network monitoring: Opportunities, challenges, and a case study,” *IEEE Communications Magazine*, vol. 52, no. 1, pp. 106-113, Jan. 2014.
- [36] KAIBITS Software, *Network signal info application* [App.], Available: <https://play.google.com/store/apps/details?id=de.android.telnet&hl=es>
- [37] Sun Light, *2G,3G,4G,CDMA Network Monitor* [App], available: https://play.google.com/store/apps/details?id=msd.n2g.n3g&hl=es_CO

- [38] AZenQos, *AZenQos Android - DriveTest/benchmark Solution for LTE, WCDMA & gsm AZQ Android* [App.], Available: <http://www.azenqos.com>
- [39] B. Seeber, “Software defined radio direction finding,” In *GNU Radio Conference*, Atlanta, GA, 2012.
- [40] B. Seeber. “Aviation mapper,” In *GNU Radio Conference*, Atlanta, GA, 2012.
- [41] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, “An IEEE 802.11a/g/p OFDM receiver for GNU radio,” In *2nd ACM SIGCOMM Workshop of Software Radio Implementation Forum (SRIF 2013)*, ACM, Hong Kong, China, pp. 9–16, Aug. 2013.
- [42] *Essential requirements for a spectrum monitoring system for developing countries*, ITU-R SM.1392-2, Feb. 2011.

La preparación de este libro, que estuvo al cuidado de Claros Editores S.A.S., finalizó en enero de 2020. En su preparación, realizada desde la Editorial Universidad Icesi, se emplearon los tipos: Baskerville MT Std de 9, 10 y 12 puntos; Book antigua de 8 puntos; Cambria Math de 12 puntos; Gill Sans MT de 8, 9, 10, 14, 19, 26 puntos; y Times New Roman de 10 puntos.

En la colección “Bitácoras de la Maestría” se presentan los resultados de las investigaciones base del desarrollo de tesis meritorias, inicialmente provenientes de la Facultad de Ingeniería. Los dos primeros capítulos de “Monitores dinámicos de software - Despliegue de software - Monitoreo de espectro”, el tercer libro de esta colección, hablan de ingeniería de software. La primera se refiere al control del desempeño de sistemas dinámicos reconfigurables (aquellos capaces de modificar su estructura en tiempo de ejecución), en particular con su necesidad de evaluar métricas que se actualizan periódicamente como respuesta a los cambios, para lo que el proyecto diseñó una arquitectura de monitoreo dinámica y escalable que implementa y resuelve preocupaciones de monitoreo dinámico en sistemas autónomos sensibles al contexto y, con base en dicha arquitectura, diseñó e implementó dos lenguajes de dominio específico: Pascani y Amelia. La segunda investigación se enfoca en una permanente necesidad de las empresas desarrolladoras de software: instalar y mantener actualizados (y en su mejor versión) múltiples componentes de software. El proyecto, entendiendo la necesidad de automatizar dicho proceso –por costo, eficiencia y control de errores–, desarrolló un mecanismo para automatizar el despliegue de software con un enfoque basado en los principios de la arquitectura dirigida por modelos, para generar automáticamente especificaciones de despliegue ejecutables a partir de diagramas de despliegue UML definidos por el usuario. El libro cierra con un trabajo de investigación sobre monitoreo del espectro radioeléctrico, una tarea clave para garantizar la calidad en la prestación de los servicios de telecomunicaciones, que se constituyó en el primer “paso firme” en la construcción de sistema de monitoreo de bajo costo –adecuado para los presupuestos del mundo en desarrollo–, tan eficiente como los costosos y sofisticados equipos usados tradicionalmente para esta tarea.