UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS.

Facultad de Ingeniería.

Maestría en ciencias de la información y las comunicaciones

TESIS DE MAESTRÍA

Representación visual de la ejecución de una arquitectura de software basada en componentes con especificación formal en cálculo ρ_{arq}.

Autor: **Jorge Alejandro Rico García.**

Director: **Henry Alberto Diosa PhD.**

Bogotá D.C, Colombia 2015

Resumen

La arquitectura de software ha sido un campo en constante crecimiento, en el cual han surgido diversas alternativas que solucionan la necesidad de descripción de arquitecturas de sistemas basados en software. Algunas de estas alternativas ofrecen herramientas formales para dicha labor de descripción; sin embargo, el esfuerzo y el conocimiento requerido para especificar y analizar arquitecturas usando tales lenguajes es alto. Esto ha conducido a la elaboración de una herramienta de software que facilite la interpretación de arquitecturas de software basadas en componentes descritas usando el cálculo ρ_{arq}, dicha herramienta brinda la facilidad para exponer de manera gráfica al arquitecto de software la estructura y el flujo de ejecución descritos en las expresiones de la arquitectura objeto de estudio.

Para la construcción de la herramienta de software fueron considerados varios módulos que, en conjunto, están en capacidad de interpretar expresiones conformes a la sintaxis y semántica operacional del cálculo ρ_{arq}, de acuerdo con el formato del lenguaje de edición científica LaTex, y de convertir dichas expresiones a elementos gráficos de la notación UML 2.x. De esta manera, dichos módulos despliegan la configuración de la arquitectura empleando los elementos gráficos de la notación UML y el flujo de ejecución resaltando las interfaces de provisión activadas al producirse reglas de ejecución propias del cálculo ρ_{arq}.

Con esta herramienta, el uso del cálculo ρ_{arq} se ve simplificado en la aplicación de la semántica operacional propia del cálculo, ya que el software se encargará de dicha aplicación y mostrará gráficamente el resultado de la aplicación de cada una de estas reglas, lo cual facilitará labores de análisis que el arquitecto deba realizar sobre su arquitectura.

Palabras Clave

Arquitectura de software, álgebra de procesos, lenguaje de descripción arquitectural, semántica operacional, cálculo ρ-arq, UML, flujo de ejecución arquitectónica.

Abstract

The software architecture has been a field in constant growing, it has proposed diverse alternatives that solve the description needs for software-based systems. Some of these alternatives offer formal tools for those description tasks; however, the effort and the knowledge required to specify and analyze architectures using such languages is high. This has driven to the elaboration of a software tool that ease the component-based software architecture interpretation described using ρ_{arq} calculus, this tool offers the facility to expose to the software architect on a graphic way the structure and the architectonic execution flow described in the architecture expressions under study.

For the construction of this software tool were considered some different modules that, in set, they have the capacity to interpret expressions conform to the ρ_{arq} calculus sintax and operational semantic, according to the scientific edition language (LaTex) format. Also to convert those expressions to UML 2.x notation graphic elements. This way, those modules expose the architecture configuration using UML notation graphic elements and the architectonic execution flow by highlighting the provision interfaces when a ρ_{arq} calculus execution rule is executed.

With this tool, the ρ_{arq} calculus use is simplified in the application of the calculus operational semantic, in asmuch as the software can take charge of that application and show the results of the rules application graphicly. With this, the analysis tasks that the architect must do over his architecture will be easy and he can focus on the architecture behaviour and not about the calculus manage.

Keywords

Software Architecture, process algebra, architecture description language, operational semantic, ρ-arq calculus, UML, architectural execution flow.

Agradecimientos

Dedico este trabajo a mi familia, en especial a mis padres quienes siempre me brindaron su apoyo en el desarrollo de esta investigación. También agradezco al profesor Henry Diosa quien aportó la idea de este proyecto y me permitió materializarla.

Tabla de Contenidos

Capítulo 1 Introducción 1
1.1 Planteamiento y justificación del trabajo...2
1.2 Hipótesis y objetivos 3
1.2.1 Hipótesis 3
1.2.2 Objetivo general 3
1.2.3 Objetivos específicos 3
1.3 Metodología seguida durante la investigación. 4
1.4 Organización de la tesis de maestría...5
Capítulo 2 Marco referencial sobre descripción arquitectónica...7
2.1 Lenguajes de descripción de arquitecturas (LDAs)..8
2.1.1 WRIGHT 9
2.1.2 UNICON...11
2.1.3 RAPIDE..11
2.1.4 ENFOQUE SINTÁCTICO..13
2.1.5 ACME...14
2.1.6 XADL 15
2.1.7 UML (UNIFIED MODELING LANGUAGE) 16
2.1.8 MÁQUINA QUÍMICA ABSTRACTA 17
2.1.9 CHIRON-2 18
2.1.10 DARWIN..19
2.1.11 LENGUAJE DE MARCAS PARA LA DESCRIPCIÓN DE ARQUITECTURAS (ADML) 20
2.1.12 KOALA 20
2.1.13 WEAVES 21
2.1.14 AO-ADL (ASPECT ORIENTED ARCHITECTURE DESCRIPTION LANGUAGE)..22
2.1.15 ASDL: Architectural Styles Description Language 23
2.1.16 ARCHIMATE 24
2.1.17 AADL (The Architecture Analysis and Design Language) 25
2.1.18 Pi-ADL 26
2.1.19 CÁLCULO ρ_{ARQ} 27
2.1.20 Utilidad pragmática de lenguajes de descripción arquitectónica. 34
Capítulo 3 Construcción de la solución...37

3.1 Evaluación de tecnologías para la Transformación e interpretación de modelos........37
3.1.1 MDA (Model-driven Architecture) 37
3.1.2 ANTLR (Another tool for language recognition)..39
3.1.3 DUALLy..40
3.2 Criterios de selección para la tecnología base 41
3.3 Incorporación de ANTLR...42
3.3.1 Definición de la sintaxis del cálculo ρ_{arq} con representación LATEX...........44
3.3.2 Construcción del API para interpretación del cálculo ρ_{arq} 46
3.4 Definición de la Arquitectura de la solución 51
3.4.1 Intérprete...53
3.4.2 Reescritor ρ_{arq} 58
3.4.3 Arquitecto 69
3.4.4 Transformador 74
3.4.5 Graficador...81
3.5 Pruebas de la herramienta...89
3.5.1 Aplicación de la semántica operacional. 90
3.5.2 Mapeo de la configuración arquitectural en UML 93
3.5.3 Exportación a XMI 102
Capítulo 4 Conclusiones y aportes...**111**
4.1 Interpretación de expresiones 111
4.2 Representación gráfica 112
4.3 Aportes originales...113
Capítulo 5 Trabajo Futuro...**115**
Anexo A. Guía de uso de la herramienta **123**

Tabla de Figuras

Figura 2.1. Separación Interfaces-Módulos en RAPIDE. Fuente: [15], [26] 12
Figura 2.2. Tipos de símbolos sintácticos Arquitecturales. Fuente [28]. 14
Figura 2.3. Especificación de componente con Darwin. Tomado de [9]. 20
Figura 2.4. Notación gráfica de Koala. Tomado de [40]. 21
Figura 2.5. Esquema de representación empleado por Weaves. Tomado de [41]............... 22
Figura 2.6. Conceptos arquitecturales de Pi-ADL. Tomado de [55] 26
Figura 3.1. Transformación de modelos. Tomado de [61] 38
Figura 3.2. Autómata finito determinístico para análisis LL(*). La notación Sn => i indica "El estado predice la alternativa i". Tomado de [64] 39
Figura 3.3. Vista conceptual de DUALLy. Tomado de [8] 40
Figura 3.4. Interfaz para el análisis de expresiones mediante el patrón de comportamiento observador. 48
Figura 3.5. Observadores de expresiones. Fuente autor. 49
Figura 3.6. Interfaz para el análisis de expresiones mediante el patrón de comportamiento visitador. 50
Figura 3.7. Visitadores implementados en el proyecto. Fuente autor. 51
Figura 3.8. Casos de uso PintArq. 52
Figura 3.9. Arquitectura de la solución. Fuente autor. 53
Figura 3.10. Caso de uso implementado por el intérprete. 54
Figura 3.11. Clases definidas para el módulo intérprete. 54
Figura 3.12. Ejemplo de árbol de expresiones construido por ANTLR. 55
Figura 3.13. Modelo dinámico caso de uso "Interpretar arquitectura". 57
Figura 3.14. Casos de uso implementados en el módulo Reescritor ρ_{arq}. 58
Figura 3.15. Diagrama de clases Reescritor $\rho_{arq.}$ 60
Figura 3.16. Modelo dinámico caso de uso "Identificar interacciones". 65
Figura 3.17. Modelo dinámico caso de uso "Reescribir arquitectura" para el caso de aplicaciones. 66
Figura 3.18. Modelo dinámico caso de uso "Reescribir arquitectura" para el caso de observaciones. 67
Figura 3.19.Modelo dinámico caso de uso "Modificar estados de la arquitectura". 68
Figura 3.20. Caso de uso implementado en el módulo Arquitecto. 69
Figura 3.21. Diagrama de clases para la configuración de arquitecturas. 70

Figura 3.22. Clase Arquitecto 71
Figura 3.23. Modelo dinámico caso de uso "Identificar patrones de definición de elementos". 73
Figura 3.24. Casos de uso implementados en el módulo Transformador. 74
Figura 3.25. Patrón puente para transformación de configuración a XMI. 75
Figura 3.26. Modelo de clases para Exportación XMI de una arquitectura. 78
Figura 3.27. Modelo dinámico caso de uso "Transformar arquitectura". 79
Figura 3.28. Modelo dinámico caso de uso "Generar archivo plano". 80
Figura 3.29. Modelo dinámico caso de uso "Exportar arquitectura". 81
Figura 3.30. Casos de uso implementados en el módulo Graficador. 82
Figura 3.31. Patrón puente para despliegue gráfico de una arquitectura. 82
Figura 3.32. Clase graficador. 83
Figura 3.33. Distribución de conectores para componentes. Fuente autor. 84
Figura 3.34. Diagrama de actividades para la ubicación de elementos de una arquitectura. 85
Figura 3.35. Modelo dinámico caso de uso "Ubicar componentes". 87
Figura 3.36. Modelo dinámico caso de uso "Ubicar conexiones". 88
Figura 3.37. Modelo dinámico caso de uso "Graficar arquitectura". 89
Figura 3.38. Resultado de interpretación para un componente con una interfaz de provisión y una interfaz de requerimiento. 94
Figura 3.39. Modelo UML para conexión de componentes 95
Figura 3.40. Visualización de interacción entre componentes. 95
Figura 3.41. Configuración para prueba de interacción sucesiva. 96
Figura 3.42. Primera interacción de la arquitectura. 96
Figura 3.43. Segunda interacción de la arquitectura 96
Figura 3.44. Representación gráfica de configuración compleja. 98
Figura 3.45. Ejecución de arquitectura compleja. 98
Figura 3.46. Representación gráfica para la ejecución No exitosa de F. 99
Figura 3.47. Configuración de arquitectura con combinador de selección condicionada. . 100
Figura 3.48. Configuración de sistema para prueba de elemento combinador. 101
Figura 3.49. Arquitectura con valor de la guarda verdadero. 101
Figura 3.50. Diagrama producto de importación XMI para arquitectura de un componente. 102
Figura 3.51. Arquitectura luego de importación XMI para arquitectura con conexión de componentes. 103
Figura 3.52. Arquitectura luego de importación XMI para ejemplo de ejecución sucesiva de la regla Aρarq. 104
Figura 3.53. Arquitectura importada usando XMI para ejemplo de ensamble complejo... 105
Figura 3.54. Arquitectura de PintArq 108
Figura 3.55. Componentes importados en Enterprise Architect. 110
Figura 3.56. Arquitectura PintArq al importar XMI en Enterprise Architect. 110
Figura A.5.1. Esquema visual de la herramienta. 124
Figura A.5.2. Controles para la ejecución de arquitecturas. 125
Figura A.5.3. Controles para carga de arquitecturas 125
Figura A.5.4. Controles para el cambio de arquitectura y la exportación a XMI. 125
Figura A.5.5. Cargue de arquitecturas en la aplicación. 126
Figura A.5.6. PintArq durante la ejecución de una arquitectura 127

Tabla de Códigos

Código 3.1. Definición de lenguaje de ejemplo para análisis LL(*). Tomado de [64] 39
Código 3.2. Nombrado de expresiones alternativas. 43
Código 3.3. Definición de símbolos a descartar en el análisis léxico. 43
Código 3.4. Definición de aplicación reducida. 44
Código 3.5. Definición de la sintaxis del cálculo ρ_{arq} usando la representación del lenguaje LATEX 45
Código 3.6. Definición de inicio y fin de ecuaciones. 46
Código 3.7. Definición de etiquetas LaTex no consideradas en interpretación de arquitecturas. 46
Código 3.8. Preparación y utilización de ANTLR para construcción del árbol de expresiones. 55
Código 3.9. Identificación de la ecuación del sistema. 56
Código 3.10. Identificación de elementos en el árbol de expresiones. 56
Código 3.11. Sustitución de identificadores por definiciones. 58
Código 3.12. Reemplazo de identificadores en árbol de expresiones. 59
Código 3.13. Aplicación de regla Aρarq para abstracción y aplicación 61
Código 3.14. Identificación de interacción entre Abstracción y Aplicación 62
Código 3.15. Creación de expresión Reemplazo. 62
Código 3.16. Aplicación de regla Ejecτ para observación y ejecución exitosa. 63
Código 3.17. Definición del resultado de la observación de ejecución exitosa. 63
Código 3.18. Definición del resultado de una observación ejecución no exitosa. 63
Código 3.19. Aplicación de reemplazo de identificadores. 64
Código 3.20. Identificación de la configuración arquitectural. 71
Código 3.21. Identificación de elementos de la arquitectura 72
Código 3.22. Identificación de conexiones 72
Código 3.23. Operación mostrarComponente():void de ImplementadorXMI. 76
Código 3.24. Operación exportarArquitectura():StringBuffer de la clase Exportador 76
Código 3.25. Operación mostrarInterfaz():void de la clase ImplementadorXMI. 76
Código 3.26. Operación mostrarInterfazProvision():void de la clase ImplementadorXMI. 77
Código 3.27. Operación mostrarInterfazRequerimiento():void de la clase ImplementadorXMI. 77
Código 3.28. Operación mostrarInterfazConectada():void de la clase ImplementadorXMI. 77

Código 3.29. Operación exportarArchivo():StringBuffer de la clase Exportador................ 78
Código 3.30. Definición en latex de ecuación de prueba para regla Aρarq.......................... 90
Código 3.31. Resultado latex de la aplicación de la regla Aρarq. 90
Código 3.32. Definición Latex para prueba de ejecución secuencial de reglas. 91
Código 3.33. Resultado de ejecución sucesiva de regla Aρarq 91
Código 3.34. Código LaTex para prueba de regla Ejecτ con ejecución exitosa. 92
Código 3.35. Resultado de prueba para regla Ejecτ con ejecución exitosa. 92
Código 3.36. Código LaTex para prueba de regla Ejecτ con ejecución no exitosa. 92
Código 3.37. Resultado de prueba para regla Ejecτ con ejecución no exitosa 92
Código 3.38. Documento LaTex para prueba de mapeo de interfaz de requerimiento y provisión. 94
Código 3.39. Documento LaTex para prueba de conexión de componentes. 94
Código 3.40. Documento LaTex para prueba de ejecuciones sucesivas. 95
Código 3.41. Configuración ρ_{arq} para sistema complejo. 97
Código 3.42. Código LaTex con descripción del sistema complejo. 97
Código 3.43. Definición del sistema complejo con ejecución no exitosa de F. 99
Código 3.44. Definición LaTex del sistema con ejecución NO exitosa de F. 99
Código 3.45. Código LaTex con descripción de sistema usando combinador de selección condicionada. 100
Código 3.46. Documento XML generado para exportación a XMI de arquitectura con un componente. 102
Código 3.47. Documento XML para arquitectura con conexión de componentes............. 103
Código 3.48. Documento XML generado para la arquitectura de prueba usada en la ejecución sucesiva de la regla Aρarq. 104
Código 3.49. Documento XML generado para arquitectura de ensamble complejo. 105
Código 3.50. Código latex con definición de la arquitectura de PintArq. 107
Código 3.51. Código XML con definición de la arquitectura de PintArq en formato XMI. 109

Índice de Tablas

Tabla 2.1. xADL 2.0 Esquemas y características. Tomado de [30]. 16
Tabla 2.2. Sintaxis del cálculo ρ_{arq}. Tomado de [58]... 29
Tabla 2.3. Semántica operacional. Tomado de [4]. 30
Tabla 2.4. Reglas de reducción. Tomado de [4]. 30
Tabla 3.1. Correspondencia de elementos del cálculo y notación LaTeX. Fuente Autor. ... 46

Capítulo 1
Introducción

La arquitectura de software es un área de conocimiento de la ingeniería que ha tomado gran importancia dentro del ciclo de vida de un proyecto de esta naturaleza, permite identificar con claridad todas las consideraciones de alto nivel necesarias para satisfacer los requerimientos funcionales y no funcionales definidos para un producto de software.

Con el objetivo de permitir la descripción de arquitecturas de software surgen diversos lenguajes de descripción de arquitecturas (LDAs, en adelante) que contemplan características específicas para describir la estructura, configuración y el comportamiento de una arquitectura.

Uno de los estilos arquitecturales más natural a una arquitectura de software y que será considerado en detalle con este proyecto es la arquitectura de software basada en componentes. Este estilo como toda arquitectura de software permite identificar la interacción de los componentes y conectores; de tal forma, que debe posibilitar al arquitecto determinar si la arquitectura diseñada se comporta de la forma en que él estableció. Actualmente, algunos de los lenguajes de descripción arquitectural como Rapide[1], ArchWare [2], π-SPACE [3], ρ_{arq} [4], permiten la simulación de la ejecución de estas arquitecturas así que este proyecto se enfocará en uno de estos.

El cálculo ρ_{arq} permite modelar el flujo de la ejecución de una arquitectura de software mediante la reducción o reescritura de las expresiones definidas haciendo uso de reglas de reducción definidas en su semántica operacional; de esta forma, se puede observar la interacción de los elementos establecidos para la arquitectura.

El proyecto busca la representación gráfica del flujo de ejecución de una arquitectura de software basada en componentes y descrita con el cálculo ρarq; en la actualidad, para revisar el flujo de ejecución de una arquitectura descrita con este lenguaje se requiere la reducción de las expresiones que describen la configuración arquitectural de forma manual y una herramienta que realice dicha reducción de manera automática y lo refleje en un ambiente visual facilitaría la labor de análisis sobre las arquitecturas.

1.1 PLANTEAMIENTO Y JUSTIFICACIÓN DEL TRABAJO

Una de las herramientas más útiles para un arquitecto de software es el análisis de propiedades [5] de una arquitectura de software, tales como fiabilidad y vivacidad [6], existencia o ausencia de abrazos mortales [7], entre otras. Por esta razón, el proyecto se ha enfocado en brindar una herramienta gráfica que facilite para el arquitecto la tarea de observar el flujo de la ejecución de la arquitectura de software descrita.

En la actualidad solo algunos LDA tienen herramientas gráficas que facilitan la labor de análisis sobre arquitecturas de software. Es el caso de Rapide con su herramienta POV[1] (Partial Order Viewer) que despliega gráficamente los eventos producidos en la ejecución de una arquitectura. Algunas se apoyan en las características gráficas de otras herramientas como es el caso del proyecto DUALLY[8] que realiza una transformación de una especificación hecha con el LDA Darwin[9] a una notación basada en UML para mostrar una vista estática de la arquitectura. Sin embargo, la necesidad de herramientas que faciliten, a los investigadores y en general a los arquitectos de software, la labor de análisis es la que soporta la naturaleza de este proyecto.

Los cálculos asociados a LDAs brindan, en su mayoría, las herramientas más robustas para el análisis de arquitecturas, sus fundamentos matemáticos permiten la identificación de comportamientos de interés para el verificador; por esto, el proyecto se enfoca en la visualización gráfica de arquitecturas descritas con el cálculo ρ_{arq}. Así, el proyecto facilitará la labor de análisis de los arquitectos al brindar una herramienta que se encargará en forma automática de la reducción de expresiones para representar en forma gráfica, usando la notación UML de componentes, la estructura y flujo de ejecución de una arquitectura de software. Esta visualización hará que la tarea del arquitecto sea más fácil, pues podrá identificar en forma rápida las regiones de la arquitectura que eventualmente muestren comportamientos no deseados, y de esta forma, enfocarse directamente sobre los elementos involucrados. Esto permitiría reducir la labor de análisis ya que el arquitecto solo tendrá que revisar las expresiones del cálculo para los componentes y conectores en donde se hayan identificado los flujos no deseados, evitando labores innecesarias.

1.2 HIPÓTESIS Y OBJETIVOS

1.2.1 Hipótesis

A partir de una descripción arquitectural con cálculo ρ_{arq}, de un sistema de software basado en componentes, es posible realizar la traducción de la representación de los elementos a la notación gráfica utilizada por UML 2.x; así mismo, los estados (Configuraciones arquitecturales en un momento dado) por los que fluye la arquitectura, resultado de la semántica operacional sobre las expresiones definidas en el cálculo, podrán ser representados gráficamente usando notación UML 2.x.

1.2.2 Objetivo general

Construir un sistema basado en software que tenga la capacidad de interpretar las expresiones del cálculo ρ_{arq}, que describen una arquitectura de software basada en componentes, con el objeto de visualizar el flujo de ejecución de los módulos de dicha arquitectura de manera gráfica.

1.2.3 Objetivos específicos

- Utilizar las herramientas del cálculo ρ_{arq} para la descripción de arquitecturas de software de tal forma que el sistema pueda interpretar la estructura del sistema definido.
- Emplear la notación gráfica provista por UML 2.x para el despliegue de los elementos definidos para el sistema de software diseñado.
- Emplear para la persistencia de los modelos en UML 2.x el lenguaje de marcado extensible XML.
- Determinar las tecnologías para la interpretación y transformación de expresiones que soporten la traducción de configuraciones arquitecturales expresadas en el cálculo ρ_{arq} a notaciones visuales de componentes de software conforme a UML 2.x.
- Construir una interfaz gráfica que permita visualizar el flujo de ejecución de una arquitectura, expresada como componentes de software alambrados con conectores de ensamble.

1.3 METODOLOGÍA SEGUIDA DURANTE LA INVESTIGACIÓN.

Dada la naturaleza del proyecto se acometieron las siguientes fases:

- Estudio de la sintaxis y semántica del lenguaje formal.
- Revisión de las tecnologías para transformación de modelos.
- Análisis de alternativas de implementación.
- Diseño de una solución.
- Implementación de prototipos.
- Pruebas de concepto.

Inicialmente, se realizó el estudio de la sintaxis y semántica operacional del cálculo ρ_{arq} para determinar los términos y reglas semánticas que debe interpretar la herramienta de software a construir. Seguido, se revisaron varias tecnologías que podrían apoyar la implementación de este proyecto, con el objetivo de conocer las posibles filosofías existentes para realizar la transformación y tener una base para la construcción o integración de las transformaciones necesarias entre el cálculo ρ_{arq} y UML. Finalmente, con esta información se procederá con la elección de una tecnología que permita la transformación de expresiones entre lenguajes y su manipulación.

Para la parte de diseño e implementación, se aplicó un híbrido entre las metodologías de desarrollo de Programación Extrema[10] y Proceso Unificado[11], con el objetivo de realizar las fases iterativas de análisis y diseño del prototipo.

Una vez finalizada la construcción de la herramienta de software se diseñaron un conjunto de pruebas de concepto en las cuales se definieron diversos escenarios de prueba en los que el software debía registrar una salida gráfica específica, a su vez, se realizaron las comparaciones haciendo el ejercicio manual de reducción de las expresiones que definen dicha arquitectura para determinar los estados y establecer su correspondencia con lo mostrado por el software.

1.4 ORGANIZACIÓN DE LA TESIS DE MAESTRÍA.

La primera parte de esta tesis se dedica a la revisión del estado del arte respecto a la descripción arquitectónica de software, luego se mostrarán las tecnologías revisadas para la transformación de modelos y se describirán los factores que fueron tenidos en cuenta para la selección de la tecnología empleada.

Dado el componente de desarrollo de software involucrado en el proyecto, se mostrarán los conceptos tenidos en cuenta durante el análisis y el diseño de la herramienta de software, detallando la arquitectura de la solución y explicando cada uno de los módulos propuestos en la aplicación y explicando los objetivos de cada uno de ellos.

Posteriormente se describirán las tareas acometidas en la construcción de la herramienta de software, se explicará en terminos generales la forma en la cual fue construido cada módulo indicando los problemas encontrados y las soluciones aplicadas a estos.

A continuación, se describirán los escenarios de prueba planteados para validar el funcionamiento de la aplicación y los resultados obtenidos.

Luego serán expuestas las conclusiones y aportaciones obtenidas en este trabajo de investigación y el trabajo futuro que se deriva a partir de este producto de software desarrollado.

Finalmente, se presenta como anexo una guía de configuración y manual de usuario de la aplicación.

Capítulo 2
Marco referencial sobre descripción arquitectónica

Las arquitecturas de software como lo mencionan Qianxiang y otros [12] requieren un control preciso sobre cada uno de sus componentes a lo largo de su ciclo de vida. Esta necesidad ha llevado al desarrollo de modelos, diagramas informales, lenguajes, estilos, etc., que permitan tener una vista de alto nivel (nivel de arquitectura) de los sistemas de software desarrollados. Así, se permite a los diseñadores la consideración de todos los elementos y propiedades que hacen parte fundamental de las herramientas a construir, adicionar ciertos beneficios y de alguna manera costos a los proyectos de desarrollo. De esta forma, este campo de conocimiento nace como una herramienta para la definición de la estructura, configuración y el comportamiento de sistemas de software que solucionan adecuadamente las necesidades de una organización.

El término arquitectura fue tomado del área de la construcción de edificaciones con el fin de caracterizar una estructura que será construida y representada abstractamente ya sea en diagramas o definiciones textuales. De manera general una arquitectura de software, como se expresa en el trabajo de D. E. Perry y otros [13], provee modelos y guías para describir la estructura, el comportamiento y propiedades clave de un sistema de software. Estas arquitecturas como lo menciona Rick Kazman [14], deben cumplir con tres principios que garantizan el entendimiento del sistema y su aplicación en un negocio específico:

- Principio 1: Una arquitectura de software debe estar definida en términos de elementos lo suficientemente amplios para el control del intelecto humano y lo suficientemente específicos para un correcto razonamiento.
- Principio 2: Los objetivos de negocio o misionales determinan los requerimientos para establecer atributos de calidad.
- Principio 3: Los requerimientos de atributos de calidad guían el diseño y el análisis de las arquitecturas de software.

Estos principios en gran medida determinan el propósito de una arquitectura. En el trabajo de Mary Shaw y otros [15] se menciona que la arquitectura de un sistema de software es definida en términos de componentes y de las interacciones entre dichos componentes; adicionalmente, esta muestra la correspondencia entre los requerimientos del sistema y los elementos del sistema construido. Para A. Jones [16], una arquitectura está definida como una estructura compuesta de componentes y reglas que caracterizan la interacción entre los componentes. Barry Boehm [17], la define como el conjunto de componentes, conexiones, restricciones y una base lógica. David Garlan y Mary Shaw [18], indican que una arquitectura de software es el conjunto de componentes, conectores y configuraciones.

Sin embargo, aunque parece no existir algún consenso sobre lo que debería definir una arquitectura de software, se pueden evidenciar algunos elementos comunes que pueden ser características para el establecimiento de lo que la define. Una arquitectura de software puede ser definida como la organización de un sistema de software en términos de sus componentes y la forma como estos tienen interacción. También se establecen algunas propiedades que complementan la definición de dichos elementos y restricciones que delimitan las posibilidades y el alcance. Con estos instrumentos, la arquitectura de software busca que el diseñador dirija su análisis del sistema a nivel de componentes y de sus interconexiones; esto permitirá el entendimiento de sistemas grandes, el reuso tanto a nivel de componentes como de arquitectura, la verificación y la validación del sistema y muchos otros beneficios tal como lo menciona el trabajo de Gang Huang y otros [19].

Ahora bien, de acuerdo a la configuración de estos elementos surge el concepto de estilos arquitecturales o patrones arquitecturales. Según Bass y otros [20] estos describen los elementos y tipos de relaciones, y establecen un conjunto de restricciones sobre cómo deben ser usados en la arquitectura; por si solos no son una arquitectura pero representan ciertas características que garantizan atributos de calidad. Algunos de los estilos manejados de acuerdo a las necesidades o la naturaleza de los sistemas a implementar son descritos por Garlan y Shaw [18].

Otros conceptos útiles en la definición de una arquitectura de software mencionados por Bass y otros [20] son los modelos y las arquitecturas de referencia. Los modelos corresponden a la descomposición estándar de un problema conocido en partes que en conjunto brindan una solución, y la arquitectura de referencia es la aplicación del modelo en elementos de software que cooperativamente implementan la funcionalidad descrita en el modelo. Estos elementos brindan opciones para las decisiones de diseño correspondientes a la arquitectura específica del sistema deseado.

2.1 LENGUAJES DE DESCRIPCIÓN DE ARQUITECTURAS (LDAS)

A lo largo del desarrollo de la ingeniería de software se ha visto la necesidad de herramientas que permitan la definición de los sistemas de software de manera organizada, bajo una semántica estándar y sin ambigüedades. Esta necesidad de caracterización para los sistemas de software concuerda con los análisis de David L. Parnas [21] respecto al

concepto de separación de funcionalidades (Separation of concerns), concepto estudiado por Dijkstra [22] y cuyo resultado indica que de esta forma se obtienen mejores resultados y ciertos beneficios en el diseño del software. Así, este modo de separación fue evolucionando a través de los modelos de programación y de igual forma en las herramientas de diseño, hasta llegar a las herramientas de descripción de arquitecturas conocidas en la actualidad.

Se han propuesto diversos lenguajes para la descripción de arquitecturas de software con el propósito de asegurar la calidad de los sistemas de software construidos y cuyo objetivo general ha sido la definición concreta de arquitecturas de software mediante una notación que permita la descripción de la configuración de sistemas de software de acuerdo a sus elementos y propiedades arquitecturales (componentes, conectores y restricciones).

Como común denominador Robert J Allen [23] menciona las siguientes características que debe cumplir un lenguaje de descripción arquitectural con el fin de permitir la especificación y el análisis una arquitectura:

- Descripción de configuraciones de arquitecturas. Debe permitir la definición de los componentes y las interacciones que ocurren dentro del sistema.
- Descripción de estilos arquitecturales. La herramienta debe permitir la identificación de familias de sistemas haciendo uso de características comunes que facilitan los esfuerzos de análisis e implementación.
- Capacidad de análisis sobre propiedades de interés. La descripción debe permitir la identificación y el análisis de algunas propiedades del sistema con el objetivo de establecer si el sistema cumple con los requerimientos definidos para él.
- Aplicación a problemas prácticos de la vida real. La utilidad del lenguaje no solo debe enfocarse en el análisis de propiedades valiosas para el sistema, también debe ofrecer una notación que pueda aplicarse en cualquiera de los escenarios de problemas reales y no solo en circunstancias con ciertas características.

Estas cualidades garantizarán las herramientas básicas para definir una arquitectura sobre sistemas requeridos en la vida real y permitirán el análisis de la arquitectura planteada con el objetivo de validar su conformidad con los requisitos iniciales. Una muestra de esto es lo que propone Diosa [4] para especificar arquitecturas de software basadas en componentes usando la notación de modelado propuesta por UML 2.x y extendiéndola con estereotipado de interfaces, para traducir a un cálculo formal que permite modelar ejecución arquitectural y chequeo de corrección.

2.1.1 WRIGHT

Esta propuesta de LDA se hace en el artículo A Formal Basis for Architectural Connection [24]; en este se busca aportar una notación y teoría subyacente que ofrezca una

semántica explícita a la conexión arquitectural; particularmentemente se describe un sistema formal para especificar tipos de conectores arquitecturales y lograr ejemplificaciones de los mismos (Facilitando el reuso). Se usa el mecanismo de protocolos para describir las interacciones entre partes de un sistema. Se usan ideas tradicionalmente aplicadas para caracterizar la comunicación de mensajes sobre una red, en este caso un subconjunto de la notación de proceso CSP (Communicating Sequential Processes) propuesta por Hoare [25], con la introducción de algunas convenciones notacionales adicionales.

La idea primordial es lograr describir relaciones de interacción antes que relaciones de implementación; distinguidas con base en tres razones:

1. Los dos tipos de relaciones tienen diferentes requerimientos de abstracción; las relaciones de implementación requieren adoptar normalmente las primitivas de un lenguaje de programación específico (Subyacente) para lograr que un componente sea computable. En contraste, las relaciones de interacción en un nivel arquitectural de diseño pueden involucrar relaciones que no sean soportadas directamente por lenguajes de programación.

2. Las relaciones de interacción involucran diferentes maneras de razonar sobre un sistema. En el caso de las relaciones de implementación, típicamente se hace un razonamiento jerárquico donde la corrección de un módulo depende de la corrección de los otros módulos que usa. En el otro caso, los componentes o módulos son lógicamente independientes. El comportamiento del sistema (agregado) depende del comportamiento de sus componentes constituyentes y la forma en que éstos interactúan.

3. Las relaciones de implementación chequean tipos para determinar si un uso de una facilidad es consistente con su definición. La completitud se chequea típicamente por medio de un cargador ("loader", en el inglés) que garantiza que toda facilidad requerida por un módulo es ofrecida por algún otro módulo. En el caso de las relaciones de interacción se está más interesado en si los protocolos de comunicación son consistentes, la completitud de un sistema está relacionada con si las suposiciones de cada componente sobre el resto del sistema se han satisfecho y si todos los participantes están presentes en las interacciones especificadas.

Este lenguaje de especificación de conexión arquitectural es capaz de expresar la noción de proveer/usar un conjunto de servicios, muy común en el desarrollo basado en componentes o la clase de conexión soportada por las cláusulas import/export, frecuente en lenguajes de interconexión de módulos. Además, WRIGHT provee otras facilidades que lo hacen versátil para describir secuenciación y escogencia del comportamiento dinámico. En [24] se dan ejemplos de estas facilidades.

Esta propuesta de Allen y Garlan [24] especifica la semántica asociada al concepto de conector, ilustra la manera de ejemplificar puertos y conectores y sus semánticas asociadas; provee mecanismos para hacer análisis de descripciones arquitecturales usando criterios de compatibilidad entre puertos y roles con una

generalización de chequeo de compatibilidad usando especificaciones traza como predicados aplicables a toda secuencia de eventos en el "glue"(Se puede asociar este concepto al protocolo subyacente en la conexión entre dos puertos). Así mismo, aporta elementos para lograr libertad de abrazos mortales ("deadlock freedom") en la especificación de componentes.

2.1.2 UNICON

En el artículo Abstractions for Software Architecture and Tools to Support Them [15] los autores hacen consideraciones sobre la manera de soportar abstracciones arquitecturales , localizando y codificando las formas en que interactúan componentes y distinguiendo entre los varios empaquetamientos de los mismos que requieren diferentes formas de interacción. El enfoque es bastante pragmático. Primero identifican, clasifican y soportan una variedad de componentes y sus posibles conexiones. Se refina la notación y luego se desarrolla un LDA llamado UNICON [15].

El modelo informal y la notación que proponen son caracterizados así:

1. Soporta idiomas de abstracción usados comúnmente por los diseñadores; por ejemplo, distingue explícitamente diferentes tipos de elementos y provee soporte de análisis de tipos específicos.
2. Especifica propiedades de empaquetamiento como aspectos funcionales de componentes; por ejemplo, distingue claramente entre funcionalidad liberada en la forma de un filtro de funcionalidad liberada en forma de un procedimiento.
3. Propone un sitio explícito llamado conector, donde se concentra la información sobre las reglas de interacción entre componentes; tales como protocolos, representaciones del intercambio y especificación de formatos de datos para lograr comunicación.
4. Se define una función abstracción para hacer corresponder código o constructos de bajo nivel a constructos de alto nivel. Similar a técnicas de Tipos Abstractos de Datos(TADs).
5. Es una propuesta abierta con respecto a herramientas de construcción y análisis desarrolladas externamente. Soporta colección y transporte de información relevante hacia una herramienta y retorno de resultados desde la misma.

2.1.3 RAPIDE

El artículo Specification and Analysis of System Architecture Using Rapide [26] describe este proyecto de la Universidad de Stanford, que ha venido evolucionando y madurando una herramienta prometedora en el campo de las arquitecturas de software. Entre los criterios de diseño que generan esta propuesta están:

1. Aportar entidades estructurales para definir prototipos ejecutables de arquitecturas.
2. Adoptar un modelo de ejecución en el que la concurrencia, sincronización, flujo de datos y temporización en un prototipo sean explícitamente representados.

3. Aportar restricciones formales y mapeos para soportar la definición de arquitecturas de referencia basadas en restricciones y probar sistemas frente a conformidad con estándares arquitecturales.
4. Direccionar algunos aspectos de escalabilidad involucrados en el modelado de arquitecturas de software a nivel industrial.
5. Desarrollar una nueva tecnología para construir sistemas multilenguaje distribuidos a gran escala.

En Rapide una arquitectura consiste de:

- **Un conjunto de especificaciones de módulos**. Una interfaz es la definición de funciones o acciones (características) ofrecidas a otras interfaces de módulos en una arquitectura y requeridas desde las mismas. Una interfaz contiene una definición abstracta del comportamiento de módulos subyacentes. Los comportamientos están definidos por reglas reactivas ejecutables. Típicamente, un comportamiento de interfaz especifica relaciones entre datos recibidos y datos generados por un módulo.
- **Un conjunto de reglas de conexión.** Definen la comunicación directa entre las interfaces. Las conexiones definen tanto la comunicación síncrona como asíncrona de datos entre las interfaces, también en una forma ejecutable reactiva.
- Un conjunto de restricciones formales que especifican patrones legales y/o ilegales de comunicación.

Estos constructos arquitecturales tienen un modelo de ejecución basado en eventos y que se denomina POSET (Partial Ordered Event Sets). Los comportamientos de las interfaces se ejecutan por recibir ciertos patrones de eventos y entonces reaccionar generando nuevos eventos. Una interfaz define la parte activa visible de los módulos y oculta lo que va al interior. Las conexiones definen cómo los eventos generados en algunas interfaces causan que otros eventos sean recibidos en otras interfaces. Las conexiones son también definidas por reglas reactivas. Las restricciones se aplican a actividades eventuales (De cadena de eventos) tanto en interfaces como sobre el conjunto de conexiones. Las restricciones son chequeables.

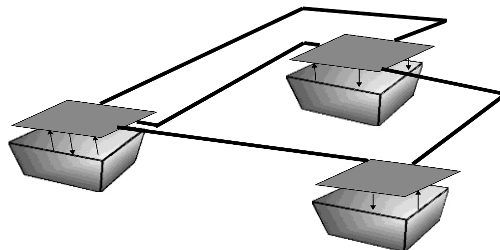

Figura 2.1. Separación Interfaces-Módulos en RAPIDE. Fuente: [15], [26]

La Figura muestra como se concibe una arquitectura desde la perspectiva de Rapide. Los módulos se comunican sólo a través de sus interfaces. Los eventos fluyen desde los módulos a sus interfaces, luego a lo largo de la arquitectura a través del

alambrado de conexiones entre las interfaces y desde las interfaces a los módulos. Cada módulo y conexión se puede ejecutar de manera independiente y concurrentemente.

Los eventos en cada interfaz deben satisfacer las restricciones en la misma y los eventos en todas las interfaces deben satisfacer las restricciones de la arquitectura.

De qué consta RAPIDE?

Rapide usa un Lenguaje de Definición de Arquitectura Ejecutable (EADL: Executable Architecture Definition Language, en el inglés). Esta herramienta se apoya en un conjunto de lenguajes:

1. Un lenguaje de tipos. Para describir interfaces de componentes.
2. Un lenguaje de arquitectura. Para describir el flujo de eventos entre componentes.
3. Un lenguaje de especificación. Para escribir restricciones abstractas sobre el comportamiento de componentes.
4. Un lenguaje ejecutable. Para escribir módulos ejecutables.
5. Un lenguaje de patrones. Para describir patrones de eventos. Rapide permite soportar el desarrollo basado en componentes para grandes sistemas basados en software y se integra a un conjunto de herramientas que favorecen un modelo de proceso evolutivo. La utilidad de Rapide como *Framework* para el desarrollo de sistemas, el detalle del modelo de ejecución POSET y la sintaxis de cada uno de los lenguajes antes puntualizados (que es extensa) se pueden conocer a fondo a través de los manuales de referencia publicados en el sitio web del proyecto Rapide [27].

2.1.4 ENFOQUE SINTÁCTICO

Esta propuesta es presentada por Thomas R. Dean y James R. Cordy [28]. Dicha teoría busca ser útil en tres sentidos:

1. Para exponer la estructura arquitectural de sistemas individuales.
2. El marco de trabajo (*Framework*) debe abstraer detalles de componentes particulares y proveer de un medio para categorizar, desde el punto de vista estructural, paradigmas arquitecturales.
3. El marco de trabajo debe aportar un medio para expresar paradigmas arquitecturales y describirlos en una forma que permita ejemplificar nuevos sistemas desde cada paradigma.

2.1.4.1 Qué requiere una sintaxis arquitectural

Uso de tipos sintácticos

Los tipos pueden imponer restricciones sintácticas sobre relaciones permisibles entre elementos de un sistema; en este caso interesa lo meramente estructural o sea la forma antes que la función o dinámica del sistema. Un lenguaje arquitectural simple podría estar conformado por los tipos resumidos en la Figura 2.2.

Figura 2.2. Tipos de símbolos sintácticos Arquitecturales. Fuente [28].

El concepto de sistemas parciales

Si se pueden representar grandes arquitecturas de sistemas, debe ser posible describir las partes de un sistema de manera separada; tales partes de un sistema tienen normalmente una conexión incompleta que es una conexión que pertenece al sistema pero con un nodo que no aparece o que está aparentemente perdido. Un conector MENSAJE puede conectar un número arbitrario de nodos: éste no puede ser una conexión incompleta. Este puede, no obstante, ser una conexión esperada, indicar que un conector de este tipo puede ser usado para conectar el sistema parcial a otros sistemas. Un SISTEMA PARCIAL es aquel que contiene conexiones incompletas o esperadas.

Algo interesante aquí es que las conexiones incompletas o esperadas pueden verse como variables conexión por las diversas posibilidades de ligado a conectores para incorporar el subsistema en otro sistema. Conectores incompletospueden pensarse como la conexión de uno o más nodos variable. Un conector incompleto puede usarse para conectar dos o más sistemas parciales al ligar conexiones esperadas de los sistemas al conector incompleto y ligar los nodos variable del conector incompleto a nodos de los subsistemas.

Los sistemas parciales se pueden categorizar en: subsistemas conector que cumplen el papel de conectores y subsistemas nodales que cumplen el papel de nodos.

2.1.5 ACME

Es un lenguaje genérico para la descripción de arquitecturas de software propuesto por Garlan y su equipo de trabajo [29] que provee un marco de trabajo estructural para caracterizar arquitecturas. Incluye utilidades en la notación para información específica de otros ADL que le permiten su uso como un formato de intercambio común para herramientas de diseño de arquitecturas de software.

Los aspectos clave de este lenguaje son:

- El uso de una ontología arquitectural basada en siete elementos de diseño: componentes, conectores, sistemas, puertos, roles, representaciones y rep-maps (Mapas de representaciones).
- Uso de un mecanismo de notación flexible que soporta la asociación de información no estructural, usando sublenguajes externos.
- Uso de un mecanismo de plantillas para abstraer estilos e idiomas arquitecturales comunes y reusables.
- Uso de un marco de trabajo abierto de semánticas para analizar las descripciones arquitecturales.

2.1.6 XADL

Fue propuesto por Dashofy y otros [30] como una infraestructura para el desarrollo rápido de lenguajes de descripción de arquitecturas basados en XML. En general es un conjunto de esquemas XML reutilizables que pueden ser empleados como base para la creación de nuevos LDA.

xADL provee constructos que son útiles para la descripción de arquitecturas de software y garantizan tres características principales:

- Separación de los modelos de tiempo de ejecución y tiempo de diseño.
- Implementación de mapeos que relacionan la especificación ADL de una arquitectura con el código ejecutable.
- Habilidad para modelar aspectos de la evolución arquitectural y líneas de producto arquitecturales.

Los esquemas que provee xADL se pueden categorizar según las características del lenguaje como lo muestra la siguiente tabla.

<table>
<tr><td>Propósito</td><td>Esquema</td><td>Elementos</td></tr>
<tr><td rowspan="2">Modelos de tiempo de diseño y tiempo de ejecución</td><td>Ejemplificaciones</td><td>Componente de tiempo de ejecución, conector, interfaz, enlaces de ejempificaciones, subarquitecturas y grupos generales</td></tr>
<tr><td>Estructura y tipos</td><td>Componente de tiempo de diseño, conectores, interfaces y enlaces, subarquitecturas, grupos generales, componentes y tipos de interfaz.</td></tr>
</table>

<table>
<tr><td>Propósito</td><td>Esquema</td><td>Elementos</td></tr>
<tr><td rowspan="2">Mapeos de implementación</td><td>Implementación abstracta</td><td>Marcador de posición para datos de implementación de componentes, conectores e interfaces.</td></tr>
<tr><td>Implementación en java</td><td>Datos de implementación concreta para componentes del lenguaje java, conectores e interfaces.</td></tr>
<tr><td rowspan="3">Gestión de la evolución arquitectural / Arquitecturas de líneas de producto</td><td>Versiones</td><td>Grafos de versión para componentes, conectores y tipos de interfaz.</td></tr>
<tr><td>Opciones</td><td>Componentes de tiempo de diseño opcionales, conectores y enlaces.</td></tr>
<tr><td>Variantes</td><td>Componentes de tiempo de diseño variantes y tipos de conectores.</td></tr>
</table>

Tabla 2.1. xADL 2.0 Esquemas y características. Tomado de [30]**.**

2.1.7 UML (UNIFIED MODELING LANGUAGE)

UML es un lenguaje diseñado para el modelado de unidades de software a partir de las perspectivas funcional, estructural y de comportamiento. Las unidades lingüísticas que emplea UML no fueron pensadas inicialmente para definir una arquitectura de software, sin embargo, existen algunos elementos que permiten el modelado de componentes del sistema y sus interacciones.

De acuerdo a R.K. Pandey algunos académicos aceptan UML como un lenguaje de descripción de arquitecturas, sin embargo, se mencionan las siguientes debilidades [31]:

- No es adecuado para el análisis automático de verificación y validación de una arquitectura.
- Sus constructos carecen de semántica formal y por consiguiente pueden ser una fuente de ambigüedad e inconsistencia en algunos casos.

En la actualidad UML ha sido empleado como ADL y por esto Bass y otros [20] presentan algunas prácticas para la documentación de una arquitectura usando UML; no obstante, se reconocen sus debilidades y en otro sentido se consideran los mecanismos de extensión que permiten la inclusión de los elementos semánticos necesarios para definir una arquitectura. Estos mecanismos, descritos por Sandra Hurtado [32] son:

- Valores adicionados, que permiten incluir nuevas propiedades a los elementos del modelo UML.

- Restricciones, que permiten mediante OCL (Object Constraint Language) adicionar o modificar la semántica de los elementos.

- Estereotipos, que permiten crear nuevos elementos basados en los ya existentes.

Con estos mecanismos es posible establecer y definir arquitecturas de software adecuadas para representar en este nivel de abstracción un sistema, ganando en el aspecto de difusión de los modelos pues al emplear un lenguaje gráfico es posible compartirlo con varias personas del equipo de trabajo y no solo con aquellos que tengan el conocimiento sobre arquitecturas.

Si se quisiera extender UML con herramientas formales, en el trabajo de Jason E. Robbins [33] se consideran métodos de integración que hacen uso exhaustivo del mecanismo de extensión basado en restricciones para relacionar UML con LDA como C2 y WRIGHT, con el objetivo de extender la formalidad de los lenguajes con la facilidad de comprensión de un lenguaje gráfico.

El trabajo de Diosa [4] es una evidencia del uso de la extensión a través del estereotipado de interfaces de provisión y requerimiento de servicios de los modelos arquitecturales de software basados en la notación de componente, con el objetivo de hacerlos traducibles a un cálculo formal: el cálculo ρ-arq.

2.1.8 MÁQUINA QUÍMICA ABSTRACTA

Este modelo mas que considerarse como un LDA es un tipo de máquina abstracta adecuada para modelar computaciones concurrentes asincrónicas basada en la metáfora de las reacciones químicas, sin embargo como lo muestran Paola Inverardi y Alexander Wolf [34] puede ser empleado para describir arquitecturas. Este modelo propuesto por Berry y Boudol [35] surge a partir del paradigma del lenguaje γ de Banâtre y Le Metayer [36] donde se menciona que los estados de una máquina son soluciones químicas cuyas moléculas pueden interactuar de acuerdo a reglas de reacción.

La máquina química abstracta mantiene las nociones básicas del lenguaje γ así, un sistema es representado como una estructura de moléculas y reglas de reacción. Las soluciones son representadas por multiconjuntos de moléculas que garantizan las propiedades asociativa y conmutativa de la composición paralela. Las reglas de reacción corresponden a reglas de reescritura de los multiconjuntos, las cuales establecen la forma en que las soluciones evolucionan en la máquina modelada.

Las reglas son clasificadas en tres categorías: reglas de calentamiento, de enfriamiento y de reacción. Las reglas de calentamiento pueden descomponer una molécula en otras mas simples; las reglas de enfriamiento recomponen una molécula compuesta a partir de sus componentes; y las reglas de reacción son aquellas que determinan la manera de interacción entre las moléculas de una solución.

Este modelo contempla el concepto de membrana con el objetivo de incluir la abstracción y la programación jerárquica, permitiendo que una molécula pueda contener

una subsolución que estará contenida en la membrana. Esta membrana puede ser lo suficientemente porosa para permitir la comunicación entre la solución y su ambiente.

Todas las máquinas químicas abstractas obedecen un conjunto de leyes estructurales, definidas de la siguiente forma:

- **Ley de reacción:** Una instancia del lado derecho de una regla puede reemplazar la instancia correspondiente de su lado izquierdo. Así, dada la regla:

  $$m_1, m_2, \ldots, m_k \rightarrow m'_1, m'_2, \ldots, m'_l$$

  y $M_1, M_2, \ldots, M_k, M'_1, M'_2, \ldots, M'_l$ son las ejemplificaciones de los elementos m_i y m'_j de la regla, entonces:

  $$\{|M_1, M_2, \ldots, M_k|\} \rightarrow \{|M'_1, M'_2, \ldots, M'_l|\}$$

- **Ley química:** Las reacciones pueden ocurrir libremente dentro de cualquier solución.

  $$\frac{S \rightarrow S'}{S \uplus S'' \rightarrow S' \uplus S''}$$

- **Ley de membrana:** Una subsolución puede evolucionar libremente en cualquier contexto.

  $$\frac{S \rightarrow S'}{\{|C[S]|\} \rightarrow \{|C[S']|\}}$$

- **Ley de aislamiento (Airlock):** Una molécula puede ser extraída de una solución.

  $$\{|m|\} \uplus S \leftrightarrow \{|m \lhd S|\}$$

2.1.9 CHIRON-2

Chiron-2 o C2 como es conocido comúnmente, es un estilo arquitectural basado en componentes y mensajes propuesto por Richard Taylor y otros [37]. Fue diseñado para soportar las necesidades particulares de las aplicaciones con una estructura similar a las interfaces gráficas de usuario, permitiendo que los componentes de una interfaz de usuario como diálogos, modelos de gráficas estructuradas, administradores de restricciones, entre otros, puedan ser reutilizados con mayor facilidad.

El estilo puede ser resumido como una red de componentes concurrentes alambrados por dispositivos de enrutamiento de mensajes [37]. Así, los elementos clave de C2 son los componentes y los conectores; también se contempla el uso de ciertas reglas que gobiernan cómo estos elementos pueden ser correctamente compuestos.

El uso de este estilo está caracterizado por los siguientes principios:

- Soporte de independencia: Un componente no tiene conocimiento de los componentes debajo de él, así, el componente generará mensajes sin conocer si alguien los recibirá y responderá. Esto fomenta la sustitución y la reusabilidad de los componentes en la arquitectura.
- Comunicación basada en mensajes: Toda la comunicación entre componentes solo es lograda a través del intercambio de mensajes. Esta característica le permite a C2 ajustarse a ambientes distribuidos donde este modelo es ampliamente utilizado.
- Multi-hilos: Esta propiedad es sugerida por la naturaleza asincrónica de las tareas en el dominio de las interfaces gráficas de usuario. Simplifica el modelado y la programación de aplicaciones multiusuario y concurrentes.
- No suposición de espacios de direcciones compartidas: C2 no contempla ningún espacio de direcciones compartidas ya que esto estaría en contraposición con las premisas del lenguaje como los componentes heterogéneos, desarrollo en diferentes lenguajes, etc.
- Implementación separada desde la arquitectura: Algunos inconvenientes de desempeño pueden ser solucionados al separar el diseño de la arquitectura de las técnicas actuales de implementación.

2.1.10 DARWIN

Es un lenguaje de ligado declarativo propuesto por Jeff Magee y otros [9] que puede ser usado para definir composiciones jerárquicas de componentes interconectados. Soporta la especificación de estructuras dinámicas y estáticas para describir la arquitectura de sistemas distribuidos mediante sus abstracciones principales, los componentes y servicios.

Darwin emplea una semántica operacional que permite la elaboración de especificaciones tales que puedan ser empleadas en tiempo de ejecución para dirigir la construcción de sistemas de software. Su intención es ser una notación de propósito general para especificar la estructura de sistemas compuestos de diversos elementos que usan varios mecanismos de interacción. Estos elementos describen una jerarquía de componentes (especificación estructural) que en su nivel mas bajo tendrán componentes primitivos los cuales estarán descritos a nivel de comportamiento y podrán ser ubicados de manera distribuida.

En este lenguaje se considera a un componente en términos de los servicios que él provee a otros componentes para interactuar con ellos y los servicios que él requiere para interactuar con otros componentes. Para definir estas configuraciones Darwin ofrece un representación gráfica y una textual como se observa en la siguiente figura donde se especifica un componente Filtro:

```
Component  Filtro{
   provide   Salida<stream char>;
   require  Entrada<stream char>;
}
```

Figura 2.3. Especificación de componente con Darwin. Tomado de [9]**.**

2.1.11 LENGUAJE DE MARCAS PARA LA DESCRIPCIÓN DE ARQUITECTURAS (ADML)

Es un lenguaje de representación basado en XML originalmente desarrollado por Micro-electronics and Computer Technology Consortium (MCC) como parte del proyecto "Productividad de la ingeniería de software y de sistemas" (SSEP). ADML es un proyecto basado en el lenguaje ACME [29] y al cual adiciona cuatro nuevos elementos:

- Representación estandarizada que puede ser interpretada por cualquier lector XML.
- Habilidad de definir enlaces a objetos fuera de la arquitectura (Análisis, diseños, componentes, etc.)
- Habilidad para comunicarse directamente con repositorios comerciales.
- Capacidad de extensión transparente.

ADML es básicamente una traducción definida en un documento DTD del núcleo de ACME a XML. En ella se incluyen los conceptos de metapropiedades y tipos de propiedades que fortalecen la definición de ACME pero disminuyen la habilidad de XML para su extensión, como lo mencionan Dashofy y otros [38]. En otro sentido, Jhon Spencer [39] indica que al emplear XML como su lenguaje base, ADML permite el intercambio de descripciones arquitecturales entre herramientas de diseño de este tipo, y por el hecho de emplear una representación abierta, este lenguaje desacopla los modelos arquitecturales de una organización de las herramientas utilizadas y permite que estos sean útiles a pesar del cambio continuo en dichas herramientas.

2.1.12 KOALA

Es un lenguaje de descripción de arquitecturas creado por los arquitectos de software de Philips Electronics [40]. Está basado en el ADL Darwin [9] dada la necesidad de definir componentes de software en las familias de productos de televisión que establecieran explícitamente las interfaces que proveen y requieren para su funcionamiento. Su modelo busca establecer una separación entre el desarrollo de los componentes y la configuración en la cual serán incluidos. Así, los diseñadores de configuración no podrán

cambiar la estructura interna del componente ni los desarrolladores de componentes podrán asumir configuraciones específicas.

Los elementos utilizados por este lenguaje son los componentes, las interfaces y las configuraciones. Los componentes son unidades de diseño, desarrollo y reuso que se comunican con su ambiente a través de interfaces; estas a su vez permiten que el componente provea y requiera funcionalidades. El conjunto de componentes e interfaces una vez conectados para formar un producto es conocido como una configuración, y esta configuración puede estar controlada por módulos que definen diversas características de comportamiento para la arquitectura haciendo uso del lenguaje C. A través de dichos módulos es posible establecer los componentes a utilizar y las interfaces conectadas en tiempo de ejecución.

Este lenguaje al igual que Darwin, provee una notación gráfica y una notación textual. La notación gráfica fue diseñada de tal forma que los componentes lucen como un chip de un circuito integrado, las interfaces lucen como pines del chip y están acompañadas de un triángulo que indica la dirección del flujo de información.

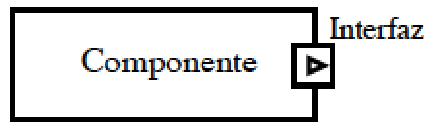

Figura 2.4. Notación gráfica de Koala. Tomado de [40]**.**

La notación textual usa para la definición de interfaces el lenguaje IDL (Interface Definition Language) en el cual se listan los prototipos de funciones con la sintaxis del lenguaje C y, los componentes son definidos con el lenguaje CDL (Component Description Language).

2.1.13 WEAVES

Esta herramienta fue diseñada por Gorlick y Razouk [41] con el objetivo de mejorar las prácticas de ingeniería de software en el dominio del procesamiento de telemetría satelital, específicamente sobre productos cuyos procesos de construcción y manutención son extremadamente difíciles debido a su dimensión y complejidad. La naturaleza de la herramienta permite la descripción de arquitecturas de software utilizando redes interconectadas de componentes, formalmente, multigrafos bipartitos dirigidos.

El lenguaje emplea para la descripción abstracciones denominadas fragmentos de herramienta que realizan una función definida en el sistema, produciendo y consumiendo objetos de software. Emplea puertos para realizar la tarea de transportar los objetos de software producidos por los fragmentos de herramienta para su comunicación, e incluye un tipo especial de componente denominado cola. Las colas son las encargadas de almacenar los objetos transmitidos por los fragmentos y sincronizar la comunicación entre ellos, así, los puertos conectan los fragmentos a través de colas. Los objetos que fluyen a través de un tejido (weave), los puertos y las colas a través de los cuales viajan, son pasivos. Solo los fragmentos de herramienta son activos, aceptando objetos desde los puertos, invocando

métodos implementados por ellos y ejecutando cálculos específicos de la herramienta.

Figura 2.5. Esquema de representación empleado por Weaves. Tomado de [41]**.**

Weaves está estructurado para permitir a lo largo del ciclo de vida del sistema los siguientes aspectos:

- **Cambio incremental continuo:** Weaves permite que los tejidos puedan ser reemplazados por otros subtejidos sin alterar el comportamiento de los fragmentos existentes, siempre y cuando se mantenga la misma estructura de comunicación.
- **Monitoreo y observación continua con baja carga de procesamiento:** Los tejidos pueden ser observados en cualquier momento y a cualquier nivel de detalle sin necesidad de hacer una provisión especial en el sistema, ya que todos los componentes se construyen con instrumentos de bajo costo para esta tarea.
- **Comunicación concurrente:** Los fragmentos de herramienta son incluídos en la arquitectura asumiendo que cada componente se ejecuta concurrentemente y las colas son las encargadas de sincronizar la comunicación. Así, el desarrollo de los componentes no se involucra con los aspectos de comunicación y los deja manos de las colas.
- **Oportunidades de paralelismo:** Weaves está diseñado para tomar ventaja de las arquitecturas de paralelismo disponibles como multiprocesadores de memoria compartida. De la misma manera, puede ejecutarse con eficiencia en máquinas de memoria no uniforme o arquitecturas distribuidas.
- **Escalamiento continuo:** Un fragmento de herramienta puede ser a su vez un tejido de componentes, así, permite una definición y un crecimiento jerárquico consistente con las prácticas modernas de ingeniería de software.

2.1.14 AO-ADL (ASPECT ORIENTED ARCHITECTURE DESCRIPTION LANGUAGE)

Este lenguaje tiene como objetivo incluir la filosofía de desarrollo propuesta en el paradigma de la programación orientada a aspectos por Kiczales y otros [42] en los lenguajes de descripción de arquitecturas. Fue propuesto por Mei Rong [43] a partir del trabajo de Zhu Xue-Yang y Tang Zhi-Song [44] XYZ/ADL, un lenguaje basado en lógica temporal.

Este lenguaje emplea los mismos constructos de XYZ/ADL, los componentes simples, los conectores y los componentes compuestos. Los componentes simples

contienen interfaces y computación, a su vez las interfaces contienen una descripción de puertos y una especificación funcional. Por otro lado, los conectores describen el carácter común de un comportamiento interactivo y están definidos por una interfaz y un protocolo interactivo. Los componentes compuestos son un tipo de componente de alto nivel cuyo comportamiento está representado por la conexión de diversos componentes.

Para lograr su objetivo, el lenguaje incluye dos conceptos adicionales denominados aspecto y conector de aspectos (Aspectual connector). El aspecto representa aquellas características que son transversales entre diversos componentes, este se compone de avisos y puntos de corte. Los avisos representan la descripción funcional completa de la característica transversal y los puntos de corte son un conjunto de puntos de unión que permiten la conexión con aquellos componentes que afecta. Los conectores de aspectos corresponden a los elementos que permiten establecer una relación transversal entre los componentes y un aspecto. Dada su naturaleza de conector, está compuesto por una interfaz y un protocolo interactivo, sin embargo, su interfaz está definida por un Rol Base que conecta los puertos de los componentes y un Rol de Aspecto que conecta el aspecto.

2.1.15 ASDL: Architectural Styles Description Language

Es un lenguaje creado para describir estilos arquitecturales basado en las herramientas formales brindadas por el lenguaje Z [45][46] y CSP (Communicating Sequential Processes) [25]. Fue propuesto por Michael D. Rice y Stephen B. Seidman [47][48] como un lenguaje que permite la descripción, el análisis y la comparación entre varios estilos arquitecturales y vistas lógicas.

Provee una colección de tipos de software que permiten la descripción de la ejecución, semántica de interfaces, vistas lógicas y relaciones entre estas. Estos son descritos usando esquemas en el lenguaje Z. La sintaxis y la semántica estructural de los módulos que componen una arquitectura de software está basada en tipos del lenguaje MIL(Module Interconnection Language) [49] que especifica los esquemas MIL_Library y MIL_Settings; no obstante, ASDL extiende estos tipos de esquemas adicionando la semántica de ejecución usando CSP y los denomina ASDL_Template, ASDL_Settings y ASDL_Unit [48].

Una especificación de plantillas (Template) representa interfaces de componentes que están disponibles para su inclusión en arquitecturas específicas de software; establece los tipos de partes y el comportamiento individual exteriorizado a través de sus interfaces. Una especificación de configuraciones (Settings) representa arquitecturas elaboradas a partir de la ejemplificación de plantillas y configuración de conexión de estas ejemplificaciones. Las unidades (Units) encapsulan configuraciones; de tal manera, que sus interfaces son externalizadas para constituirse en un componente encapsulado de una configuración arquitectural. En este sentido, ASDL soporta la jerarquización de las abstracciones de configuraciones arquitecturales.

2.1.16 ARCHIMATE

Es un lenguaje abierto e independiente definido por la organización de estándares The Open Group, creado en el periodo 2002 a 2004 en Holanda por un equipo de proyecto del Instituto de Telemática en conjunción con el gobierno, industria y la academia, incluyendo la Universidad Radboud de Nijmegen, el Instituto Leiden para ciencias de la computación avanzadas y el centro para matemáticas y ciencias de la computación "Centrum Wiskunde & Informatica".

Archimate ofrece un lenguaje de modelado para describir de manera integrada arquitecturas empresariales, su motivación, programas, proyectos y caminos de migración para implementarlas. Su utilidad para el modelado de arquitecturas de software se deriva de varios conceptos que lo hacen útil para este propósito y se presentan a continuación [50]. Archimate está compuesto por tres tipos de elementos:

1. Estructura activa: Está definida como una entidad que es capaz de desarrollar un comportamiento.
2. Elemento de comportamiento: Está definido como una unidad de actividad desarrollada por una o más estructuras activas.
3. Estructura pasiva: Es definida como un objeto sobre el cual se desarrolla un comportamiento.

En el aspecto comportamental, Archimate distingue dichos elementos con vistas internas y externas, reflejando los principios de la orientación a objetos. De esta forma incluye los conceptos de servicio e interfaz:

- Servicio: Unidad de funcionalidad que un sistema expone a su ambiente, ocultando sus operaciones internas.
- Interfaz: Punto de acceso donde uno o más servicios son puestos a disposición en el ambiente.

Basado en la especialización de los tipos de elemento descritos, Archimate define tres capas:

- Capa de negocio: Ofrece productos y servicios desarrollados por la organización para clientes externos.
- Capa de aplicación: Apoya la capa de negocio con servicios de aplicación que son realizados por soluciones basadas en software. Normalmente aquí se hecha mano de UML y otros lenguajes de modelado de procesos de negocio como BPMN.
- Capa de tecnología: Ofrece servicios de infraestructura necesarios para ejecutar las aplicaciones.

Adicional al marco de trabajo descrito, es posible la integración de módulos de extensión que adicionan nuevos conceptos, relaciones, atributos que permiten la descripción de aspectos que a consideración del arquitecto sean de importancia como la motivación del proyecto, su implementación o planes de migración, entre otros. No se

podría aseverar que Archimate es un LDA en el sentido puro del término; sin embargo, su visión integrada de la arquitectura empresarial con las herramientas de modelado que incorpora para la capa de software lo hacen útil para algunos arquitectos que lo usan sólo en esta capa.

2.1.17 AADL (The Architecture Analysis and Design Language)

Se origina en el esfuerzo de contar con un lenguaje de descripción arquitectónica para sistemas de aviónica, su nombre inicial fue Avionics Architecture Description Language que germinó desde el lenguaje MetaH [51]. Sus abstracciones permitieron extenderse al modelado en el dominio de sistemas de tiempo real y empotrados para sistemas médicos y del ámbito automotriz. Como muchos otros LDAs puede describir la estructura de un sistema como un ensamble de componentes de software que pueden asignarse a elementos de hardware. Puede describir el flujo de control y datos entre las interfaces de los componentes alambrados. También puede capturar aspectos no-funcionales de los componentes de software como propiedades de los mismos [52]. AADL permite modelos multinivel de elementos de hardware y software interconectados. Este incluye abstracciones de software, hardware y componentes de un sistema para [13]:

- Especificar y analizar sistemas de tiempo real empotrados, sistemas de alta dependencia entre componentes, sistemas con subarquitecturas complejas y desempeño en sistemas especializados.
- Mapear componentes de software a elementos de hardware computacional.

AADL es efectivo para la elaboración de modelos arquitectónicos con elementos independientes de la plataforma y elementos específicos de la plataforma, la semántica bien definida del lenguaje permite el análisis cuantitativo a través del ciclo de vida de una solución basada en software. Sin información específica de la plataforma, el análisis de atributos de calidad operacionales no sería posible y no permitiría el descubrimiento temprano de problemas a nivel de sistema para una pronta mitigación de riesgos.

El propósito fundamental de AADL es representar la arquitectura de sistemas empotrados como modelos arquitectónicos que aportan una semántica analizable en tiempo de ejecución. La creación de estas descripciones cubre la identificación y detalle de componentes de hardware y software a través de especificaciones de interfaces (Tipos de componentes) y planos de su implementación (Implementaciones de componentes). Para los componentes de software (Hilos, procesos y datos) se definen sus características en tiempo de ejecución (Tiempos de latencia y de ejecución, scheduling y posibles umbrales o momentos de terminación de la ejecución). Además, se declara el tamaño de código en ejecución, lenguaje del código fuente, nombres de archivos de código fuente con su tamaño.

Para los elementos de hardware se definen los componentes de la plataforma de ejecución: Procesadores, memoria y buses. Se caracterizan estos componentes y su configuración (Velocidad del procesador, tamaño de palabra en memoria, entre otros).

La descripción arquitectónica completa se presenta en forma de una sola jerarquía que integra todos los componentes. Esta jerarquía establece todas las interacciones entre los componentes y la configuración arquitectónica para un sistema ejecutable. Esto incluye el intercambio de eventos y datos y las conexiones físicas entre componentes como también la asignación de software a elementos de hardware computacional [53].

2.1.18 Pi-ADL

Este lenguaje nace de la necesidad de describir arquitecturas estáticas y dinámicas de sistemas de software empleando herramientas formales, bajo el esfuerzo del proyecto europeo ArchWare. Permite la descripción formal de arquitecturas de software ejecutables haciendo uso de los conceptos del cálculo Pi [54]. Pi-ADL posee las siguientes características [55]:

- Formalidad: Es un lenguaje formal basado en el lenguaje de alto orden Cálculo Pi.
- Practicidad: Provee diversas sintaxis, gráfica y textual, empleando un perfil UML 2.0 para la representación de arquitecturas de software. Esto permite el ocultamiento al usuario de la complejidad asociada al uso del cálculo Pi.
  Adicionalmente ofrece una herramienta para la verificación y chequeo automático de algunas propiedades de la arquitectura.
- Ejecutable: Una descripción de arquitectura puede especificar como el sistema se comporta en términos de conceptos abstractos de arquitectura.

Para la representación de arquitecturas de software, este lenguaje emplea dos elementos principales: Componentes y conectores, los cuales a su vez son descritos empleando el elemento puerto y definiendo un comportamiento interno. En la Figura 2.6 se pueden observar los principales elementos utilizados en este LDA.

El rol del componente es definir unidades computacionales que proveen servicios del sistema y el rol de los conectores es proveer canales de comunicación entre dos elementos arquitecturales. Los puertos a su vez juegan el rol de unión de conexiones, permitiendo la definición de interfaces entre los componentes y su ambiente.

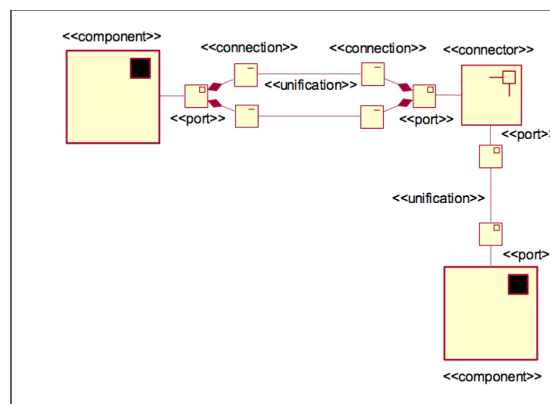

Figura 2.6. Conceptos arquitecturales de Pi-ADL. Tomado de [55]

Un componente puede enviar y recibir valores a través de conexiones, las cuales pueden ser definidas como de salida (output) permitiéndo sólo el envío de información, de entrada (input) permitiéndo sólo la recepción de información o mixtas (input-output) donde pueden ser recibidos o enviados valores.

El lenguaje, fue concebido bajo el enfoque de definición por capas, estableciendo tres sistemas formales, así [56]:

- El sistema formal de la capa base, denominada Pi-ADL_B, la cual provee sólo los elementos de conexión y comportamiento.
- El sistema formal de la capa de primer orden, Pi-ADL_{FO}, el cual extiende Pi-ADL_B con los elementos de datos y estructura: Tipos de dato base y tipos de constructores.
- El sistema formal de la capa de alto orden, Pi-ADL_{HO} la cual extiende Pi-ADL adicionando reglas para la definición de elementos otorgando derechos a dichos elementos.

Este LDA posee una notación gráfica para representar la estructura de las arquitecturas a describir, basado en un profile de UML 2.0 que establece cada uno de los elementos estructurales que propone Pi-ADL. Esta es una notación gráfica que extiende la sintaxis del lenguaje original pero solamente permite la representación de la configuración de una arquitectura. En la Figura 2.6 se puede observar el producto de la descripción gráfica haciendo uso del profile UML.

A diferencia de este proyecto, PintArq utilizará la misma notación gráfica de UML (No se implementará profile) pero resaltará las interfaces que producto de la aplicación de la semántica operacional del cálculo ρ_{arq} tengan una interacción, mostrando así el flujo de ejecución arquitectural. De esta forma, la diferencia principal radica en que Pi-ADL representa configuración con su notación gráfica, y el cálculo ρ_{arq} apoyado en PintArq representa configuración y ejecución de una arquitectura con la representación gráfica.

2.1.19 CÁLCULO ρ_{ARQ}

El cálculo ρ_{arq} es una extensión del cálculo ρ [57] para su aplicación en la especificación de arquitecturas de software basadas en componentes. La similaridad sintáctica y de la semántica operacional entre el cálculo ρ_{arq} y el cálculo π posibilita utilizar este con ligeras adaptaciones sintácticas a los propósitos de especificación de arquitecturas de software.

Una de las ventajas de este cálculo radica en la posibilidad de demostrar equivalencia comportamental entre sistemas constituidos por componentes de software usando la observabilidad de equivalencia propuesta por Milner para estos fines [54].

También, cuenta con una semántica operacional susceptible de ser traducida a Sistemas de Transición Rotulados, se pueden usar herramientas de chequeo de modelos para analizar propiedades dinámicas como fiabilidad, vivacidad y detección de abrazos

mortales y permite que el flujo de ejecución arquitectural pueda traducirse a redes de Petri.

Esta herramienta logra con el uso de restricciones que se permita condicionar la participación de componentes de software en posibles instancias arquitecturales ejemplificadas desde un modelo de referencia; esto a su vez permitirá activar o desactivar componentes de software en tiempo de ejecución desde un contexto controlado por restricciones almacenadas en un repositorio global [4].

Este cálculo debido a que es un método formal, tiene gran aplicación en la validación de comportamientos sobre sistemas de gran dimensión y que poseen una gran cantidad de estados, en donde la validación usando los métodos tradicionales tardaría demasiado tiempo debido al problema de la explosión de estados. Así mismo, en los escenarios donde la concurrencia o ciertas propiedades de la arquitectura definida puedan conducir a comportamientos no determinísticos, este método formal permite la identificación de estos comportamientos y la revisión de la estructura definida para su corrección. La aplicación de este método formal, desafortunadamente solo se ha dado en el contexto científico a nivel de investigación, debido al nivel de los conocimientos requeridos para el manejo de la sintaxis y los constructos semánticos que provee el cálculo.

La sintaxis de este cálculo está compuesta por un conjunto de símbolos y expresiones que permiten la descripción de los elementos de una arquitectura de software. Así, los símbolos están compuestos de un alfabeto infinito de nombres y un alfabeto infinito de variables. Para la representación de configuraciones arquitecturales, el cálculo ofrece el siguiente conjunto de expresiones:

- T(Null): Representa un componente nulo, es decir aquel que no tiene ningún comportamiento.
- Composición: Representa ejecución concurrente de componentes de una arquitectura.
- E(int): Representa la parte interna o encapsulada de un componente denominado E. Esta expresión permite separar la representación de la parte externa de un componente de la parte interna.
- Combinador de selección condicionada: Tiene la sintaxis if (C1) … (C2) else G. Es una generalización del condicional SI que permite la activación de componentes cuando alguna de las guardas es satisfecha, si ninguna de las guardas de la expresión es satisfecha, entonces se activa el componente definido en la palabra else.
- Abstracción: Es utilizada para representar el recibimiento de una entidad simbólica a lo largo de un canal para ser utilizado por un componente.
- Aplicación: Representa el envío de una entidad simbólica de un componente a través de un canal.
- Reacción Interna (τ/E): Esta expresión es utilizada en el cálculo para reducir el número de estados de una arquitectura y evitar de alguna forma la explosión de estados. Representa aquellos estados que no son de interés en el chequeo de propiedades de una arquitectura.
- Declaración ($\exists w E$): Introduce una referencia en el ámbito de otra expresión o un componente. Dicha referencia solo puede ser conocida en el ámbito de E.

- Replicación de abstracción: Esta expresión es utilizada para la ejemplificación de componentes tantas veces como la ejecución de la arquitectura lo permita. Permite que un tipo de componente pueda ejemplificarse cada vez que se requiera los servicios que este provee.
- Los términos $E_{\top}$ y $E^{\perp}$ indican el éxito o fracaso de la ejecución de un componente.
- Expresión On Success Of: Permite la observación del resultado de ejecución de un componente y dependiendo del resultado activar para su ejecución ciertos elementos de la arquitectura.

SÍMBOLOS			
x, y, z		variables	
a, b, c		nombres	
$u, v, w ::= x \mid a$		referencias	
EXPRESIONES			
$E, F, G ::=$		$\top$	**Null**
	$\mid$	$E \wedge F$	**Composición**
	$\mid$	$if(C_1 \cdots C_n)\ else\ G$	**Combinador de selección condicionada**
	$\mid$	$x :: \overline{y}/E$	**Abstracción**
	$\mid$	$x\overline{y}/E$	**Aplicación**
	$\mid$	τ/E	**Reacción interna**
	$\mid$	$\exists wE$	**Declaración**
	$\mid$	$x : \overline{y}/E$	**Replicación**
	$\mid$	$E^{\top}$	**Ejecución exitosa del componente E**
	$\mid$	$E^{\perp}$	**Ejecución no exitosa del componente E**
	$\mid$	$OSO(E)\ do\ F\ else\ G$	**On Success Of**
$\phi, \psi ::=$		$\dot{\top}$	**Verdad Lógica**
	$\mid$	$\perp$	**Falsedad Lógica**
	$\mid$	$x = y$	**Restricción ecuacional**
	$\mid$	$\phi \dot{\wedge} \psi$	**Conjunción de restricciones**
	$\mid$	$\dot{\exists}\phi$	**Cuantificador existencial**

Tabla 2.2. Sintaxis del cálculo ρ_{arq}. Tomado de [58].

Adicionalmente, el cálculo provee una semántica operacional que permite modelar la ejecución de una arquitectura. Esta incluye un conjunto de reglas que determinan el cambio de estados de la arquitectura modelada.

El cálculo ρ_{arq} mantiene la propiedad del cálculo ρ respecto a que restricciones, aplicaciones, abstracciones y condicionales pueden combinarse libremente por medio de composición y declaración [58]. La siguiente tabla muestra los axiomas de congruencia estructural que permiten la composición y declaración.

$(\alpha - conversión)$	Cambio de referencias ligadas por referencias libres
(ACI)	$\wedge$ es asociativa, conmutativa y satisface $E \wedge \top \equiv E$
$(Intercambio)$	$\exists x \exists y E \equiv \exists y \exists x E$
$(Alcance)$	$\exists x\ E \wedge F \equiv \exists x(E \wedge F)\quad si\ x \notin \mathcal{FV}(F)$
$(Equiv.Restricciones)$	$\phi \equiv \psi \quad si\ \phi \models\!\mid_{\Delta} \psi$ y $\mathcal{FV}(\phi) = \mathcal{FV}(\psi)$
$(Replicación\ de\ observación)$	$!OSO(E)\ do\ F\ else\ G\ \equiv\ OSO(E)\ do\ F\ else\ G \wedge !OSO(E)\ do\ F\ else\ G$
$(Éxito/Fracaso\ \ Observacional)$	$[v/w]E\ \equiv\ \dot{\top}\ \wedge\ if\ [\ (\dot{\top}\ then\ E^{\top})\ ,\ (\dot{\top}\ then\ E^{\perp})\ ]\ else\ (\ \top\ )$

Tabla 2.3. Semántica operacional. Tomado de [4].

Como parte de la semántica operacional del cálculo se contemplan algunas reglas de reducción que se describen en la siguiente tabla:

$(A_{\rho_{arq}})\quad \phi\ \wedge\ x:\overline{y}/E\ \wedge\ x'\overline{z}/F \longrightarrow \phi\ \wedge x:\overline{y}/E\ \wedge\ [\overline{z}/\overline{y}]E\ \wedge\ F\quad si\ \phi \models_{\Delta}\ x = x', \mathcal{V}(\overline{z}) \cap \mathcal{BV}(E) = \emptyset$
$(C_{\rho_{arq}})\quad \phi_1\ \wedge\ \phi_2 \longrightarrow \psi \qquad si\ \phi_1 \dot{\wedge}\ \phi_2 \models\!\mid_{\Delta} \psi$
$(Comb_{\rho_{arq}})\quad \phi\ \wedge\ if\ (C_1)\ldots(C_n)\ else\ F\ fi \longrightarrow \begin{cases} E_k, & si\ \phi \models_{\Delta}\ \psi_k \\ F, & si\ \phi \models_{\Delta}\ \neg\psi_k\ ; \forall k = 1,2,\ldots,n \end{cases}$ $Con\ \ C_k ::= \exists\overline{x}(\psi_k\ Then\ E_k)\ ;\ k = 1,2,\ldots,n$
$(Ejec_{\tau})$ (a) $[OSO(E)\ do\ F\ else\ G]\ \wedge\ E^{\top} \longrightarrow F, debido\ a\ que\ hay\ ejecución\ exitosa\ del\ componente$ (b) $[OSO(E)\ do\ F\ else\ G]\ \wedge\ E^{\perp} \longrightarrow G, debido\ a\ que\ no\ hay\ ejecución\ exitosa\ del\ componente$

Tabla 2.4. Reglas de reducción. Tomado de [4].

En la regla $A_{\rho arq}$ existe una replicación de abstracción que ejemplifica un componente E para que reciba un servicio a lo largo de x y una aplicación donde F envía Z a lo largo de x'. Aquí ambos componentes reaccionan, Z reemplaza a y en E y F continua su ejecución.

La regla $C_{\rho arq}$ es empleada para realizar la reducción de restricciones cuando sea posible hallar una restricción que sea equivalente a las originales.

La regla $Comb_{\rho arq}$ dispara la ejecución de un E_k si la restricción de contexto es suficientemente fuerte y permite deducir desde ϕ la guarda ψ_k del condicional respectivo.

La regla $Eject_{\tau}$ permite la ejecución de un componente u otro dependiendo del resultado de ejecución de otro componente. Así para el ejemplo, cuando E es ejecutado correctamente E^{T} se libera el componente F. Cuando su ejecución no es exitosa $E\perp$ se libera el componente G.

2.1.19.1 Representación arquitectural

Para representar una arquitectura de software utilizando el cálculo ρ_{arq} es necesario definir la manera en la cual es posible especificar cada elemento estructural. Así, para representar un componente, este debe ser considerado como un proceso encapsulado que solo expone al exterior sus puertos de entrada y sus puertos de salida [58]. De esta

forma, la comunicación entre componentes está definida por la conexión de los puertos compatibles que se exponen entre ellos. Para permitir dicha conexión cada puerto tiene asociada una interfaz con la cual es posible la exposición de servicios o el requerimiento de estos.

Una interfaz de provisión o interfaz de salida se encarga de externalizar un servicio prestado por un componente, para representarla en el cálculo se emplea la entidad sintáctica replicación, queda definida en términos del puerto de salida del componente y el servicio prestado, así, para definir una interfaz de salida de un componente nombrado E, cuyo puerto de salida es p y el servicio es s, se tiene la siguiente expresión:

$$PROV_E(p,s) = p_e{:}\,x/xs_e$$

Ecuación 2.1. Interfaz de provisión.

Esta expresión puede ser leída de la siguiente forma: “Se espera a lo largo de p_E un valor de una locación a la cual se enviará el servicio s_E”[58]. Así, la interfaz estará pendiente de recibir dicha locación y de enviar servicios a todas las locaciones recibidas, lo cual es posible con las expresiones del cálculo debido a las propiedades de la entidad replicación que permiten a la interfaz concurrentemente atender una solicitud de servicio y estar a la espera de una ubicación.

Una interfaz de requerimiento o interfaz de entrada es aquella que se encarga de externalizar un puerto de entrada a través del cual entrará un servicio requerido como insumo para los procesos de un componente. La expresión para definir una interfaz de requerimiento está compuesta de dos partes, la primera encargada de enviar al puerto de la interfaz de provisión la ubicación a donde se debe prestar el servicio y la segunda encargada de recibir en dicha ubicación el servicio prestado. Por lo tanto, en el cálculo ρ_{arq} la expresión que representa una interfaz de requerimiento del componente E con puerto *r*, ubicación *l* y parametro de entrada *i* esta constituida por la ejecución concurrente de las siguientes expresiones:

$$REQ_E(r,l,i) = \exists l_E[(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})]$$

Ecuación 2.2. Interfaz de requerimiento.

De esta forma, la ubicación l_E será enviada al puerto recibido en r_E y de regreso en dicha ubicación será obtenido el servicio para ser incorporado en el interior del componente E. Así, la expresión puede ser interpretada como “Se espera a lo largo de r_E un nombre de acceso a un servicio que al ser aplicado a la ubicación l_E que es local a la expresión (al componente que pertenece), externalice la ubicación para recibir a lo largo de ésta el servicio solicitado, que al ser recibido a lo largo de l_E podrá reemplazar el parámetro de entrada i_E en $E^{(int)}$, que representa la parte interna del componente”[58].

Haciendo referencia a la forma de representar un componente, mediante la exposición de sus puertos de entrada y salida, un componente con una interfaz de entrada y una interfaz de salida podría representarse como se ve a continuación:

$$E = PROV_E(p,s) \wedge REQ_E(r,l,i)$$

Ecuación 2.3. Definición del componente E

Donde se puede observar la ejecución concurrente de las interfaces de provisión y requerimiento descritas previamente.

Ahora es necesario, definir la manera en la cual una interfaz de salida y una de entrada cuyos puertos son compatibles pueden ser alambradas mediante las expresiones del cálculo ρ_{arq}. Para esto se define un conector empleando la entidad sintáctica aplicación reducida utilizando como canal de envío el puerto de la interfaz de requerimiento y enviando como ubicación el puerto de la interfaz de provisión:

$$C_{EF} = r_E p_f$$

Ecuación 2.4. Conector entre E y F.

De esta forma, al enviar el puerto de la interfaz de provisión a través del puerto de la interfaz de requerimiento se inicia la interacción entre las interfaces conectadas, ya que la interfaz de requerimiento una vez recibe el puerto, lo utiliza para enviar la ubicación en la cual es requerida el servicio, y la interfaz de provisión al recibir por su puerto dicha ubicación, envía el servicio a esta.

Ahora, desde la perspectiva de la semántica operacional del cálculo, la aplicación reducida es la expresión que activa la interacción entre las expresiones, ya que es posible aplicar la regla de reducción $A_{\rho arq}$ entre esta (conector) y la abstracción (interfaz de requerimiento) que recibe a través del puerto de su interfaz, dando como resultado una aplicación reducida que interactua ($A_{\rho arq}$) con la replicación que define la interfaz de provisión. Finalmente, se genera una aplicación reducida que interactua ($A_{\rho arq}$) con la abstracción que incorpora el servicio al interior del componente (Interfaz de requerimiento).

A manera de ejemplo, para introducir la interfaz de conexión definida dentro de un contexto válido, se definirán dos componentes E y F, donde E estará compuesto de una interfaz de requerimiento y F de una interfaz de provisión cuyos puertos son compatibles. Así, en el cálculo ρ_{arq} E y F estarían definidos por las siguientes ecuaciones:

$$E = \exists l_E[(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})]$$
$$F = p_f : x/xs_f$$

Ecuación 2.5. Definición de E y F.

Luego, si se define un sistema que conecte estos componentes usando el conector definido previamente, se tendría un sistema como el siguiente:

$$S = \mathrm{E} \wedge \mathrm{F} \wedge \mathrm{C}_{ef}$$

Ecuación 2.6. Definición del sistema con los componentes E y F.

Con este sistema, podría visualizarse una ejecución arquitectural definida por la siguiente secuencia de la aplicación de la regla $A_{\rho arq}$. Así, reescribiendo la ecuación del sistema con la definición de cada componente se iniciaría dicha secuencia así:

$$S = \exists l_E[(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})] \wedge p_f{:}\, x/xs_f \ \wedge r_E p_f$$
$$\xrightarrow{A_{\rho arq}} S = \exists l_E[(p_f l_E) \wedge (l_E :: i_E/E^{(int)})] \wedge p_f{:}\, x/xs_f$$
$$\xrightarrow{A_{\rho arq}} S = \exists l_E[(l_E :: i_E/E^{(int)})] \wedge l_E s_f$$
$$\xrightarrow{A_{\rho arq}} S = [s_f/i_E]/E^{(int)}$$

Ecuación 2.7. Flujo de ejecución del sistema S.

2.1.19.2 Control de ejecución Arquitectural

El cálculo ρ_{arq} provee dentro de sus herramientas sintácticas, la posibilidad de describir control en la ejecución de una arquitectura, y definir componentes manejadores de errores a través de la entidad On succes Of, la cual permite la observación de la ejecución de un componente y a partir del resultado de dicha ejecución, incluir expresiones que permitan manejar el error o continuar con el flujo diseñado por el arquitecto.

La observación puede escribirse de la siguiente manera [58]:

$$[s_f/i_E]E^{(int)} \wedge \ [OSO(E)$$
$$do\ (activar\ conectores\ de\ ensamble)$$
$$else(activar\ conector\ de\ manejo\ de\ errores)]$$

Ecuación 2.8. Definición de observación.

Así, cuando la ejecución del componente E, luego de recibir un servicio, es exitosa, la observación activa conectores (aplicaciones reducidas) que permiten la continuación de la ejecución arquitectural; de lo contrario, si la ejecución de E no fue exitosa, se activan conectores para conducir el error hacia los componentes definidos para tal manejo.

De esta forma, si se quisiera conectar un componente manejador de errores a E, usando el cálculo ρ_{arq} se tendría lo siguiente:

El sistema estaría definido por un componente mas denominado para ejemplo R, el cual se encargará de manejar fallos de ejecución de E, este nuevo componente tendrá una interfaz de requerimiento para poder conectarse con E y a su vez E tendrá una interfaz de provisión para enviar la información del error.

$$S = \mathrm{E} \wedge \mathrm{F} \wedge \mathrm{R} \wedge \mathrm{C}_{ef}$$

Ecuación 2.9. Sistema S con componente para manejo de errores.

Sin embargo, para permitir que E y R puedan conectarse se debe definir un conector entre dichos componentes usando las interfaces diseñadas para su comunicación, pero dicho conector solo debe ser habilitado en caso de una ejecución no exitosa de E, por tanto el sistema estaría definido así:

$$S = \mathrm{E} \wedge \mathrm{F} \wedge \mathrm{R} \wedge \mathrm{C}_{ef} \wedge OSO(E) do\ \tau\ else\ C_{ER}$$

Ecuación 2.10. Sistema con observación para el manejo de errores.

De esta forma si la ejecución de E luego de su comunicación con F no es exitosa, la entidad observación liberará en la ecuación del sistema el conector C_{ER} y E iniciaría su comunicación con el componente R.

$$S = \mathrm{E} \wedge \mathrm{R} \wedge (OSO(E) do\ \tau\ else\ C_{ER}) \wedge E^{\top}$$
$$\xrightarrow{Ejec_{\tau}} S = \mathrm{E} \wedge \mathrm{R} \wedge C_{ER}$$

Ecuación 2.11. Flujo de ejecución de sistema con observación.

2.1.20 Utilidad pragmática de lenguajes de descripción arquitectónica.

A continuación se presenta una relación que resume la funcionalidad ofrecida por cada uno de los LDA descritos en términos del modelado estructural y el modelado dinámico.

LDA	¿Modelado estructural?	¿Modelado dinámico?
WRIGHT	Describe la interacción de componentes vía tipos de conectores.	Usa anotaciones basadas en CSP [25] de Hoare para modelar interacción.
UNICON	Aporta al diseño composicional de arquitecturas de software concentrándose en los tipos de conectores.	No propone un modelo formal para el análisis dinámico; deja esto a herramientas externas.
RAPIDE	Describe alambrado entre componentes basado en especificaciones de módulos, reglas de conexión y restricciones para identificar patrones legales e ilegales de ensamble de módulos.	Usa un modelo de ejecución basado en eventos que se denomina POSETS (Partially Ordered Events Sets).
ENF. SINTÁCTICO	Usa la teoría de conjuntos para modelar conexiones y tipos de nodos en una arquitectura. Permite modelar parcialmente arquitecturas o subarquitecturas desde el punto de vista estático.	No lo aborda.
UML	Desde su versión 2.0 aporta abstracciones de componente, interfaces de provisión y requerimiento de servicios, conectores de ensamble que facilitan el modelado estático de	El uso de unidades lingüísticas para modelar comportamiento (Diagramas de interacción y máquinas de estado) dan algunas posibilidades

LDA	¿Modelado estructural?	¿Modelado dinámico?
	arquitecturas de software.	semiformales. Para mayor formalidad debe extenderse el metamodelado para que soporte análisis de propiedades dinámicas de la arquitectura bajo estudio, ejemplo de este enfoque son algunos los trabajos del Grupo de Investigación ARQUISOFT al respecto [59][60].
AO-ADL	Soporta la definición de componente, conector y restricciones funcionales para las conexiones entre componentes. Las interfaces de componentes deben ser conformes a los aspectos de interés al momento de interactuar.	No son claros los mecanismos para el modelado dinámico. Establece un contrato protocolario a cumplir por los puntos de conexión basado en la especificación de aspectos a cumplir.
ACME	Modela los aspectos estructurales de una arquitectura de software y permite anotaciones para especificar propiedades estructurales o restricciones adicionales.	En sus últimas versiones se cuenta con una librería (AcmeLib) que permite manipular los modelos de manera programática. Dada su naturaleza extendible, otro camino para incorporar análisis comportamental es asociar una especificación formal a los elementos estructurales a través de las anotaciones.
DARWIN	Usa representación textual canónica que describe las interfaces de los componentes y su alambrado. Permite modelado jerárquico de los componentes.	Usa el cálculo-π para modelar formalmente y lograr el chequeo de propiedades. Permite el modelado de arquitecturas dinámicas (Que cambian en tiempo de ejecución)
xADL	Se enfoca a la definir las estructuras básicas de arquitecturas prescriptivas: Componentes, conectores, interfaces, enlaces y agrupamientos	No lo aborda
Weaves	Modela estructura y configuración conforme a su estilo arquitectónico	No lo aborda; sin embargo, respeta las restricciones del estilo arquitectónico, expresadas de manera general en este documento

LDA	¿Modelado estructural?	¿Modelado dinámico?
CHAM	Sintaxis propia para describir estructura y configuración usando la analogía de moléculas y soluciones químicas	Posee sistema de re-escritura de expresiones basado en analogía del concepto de reacción química
ARCHIMATE	Lo mismo que UML con conceptos adicionales de servicio y función que potencian el modelado arquitectónico en relación con el diseño de arquitecturas empresariales	Aplica lo mismo que UML
KOALA	Captura la estructura, configuración e interfaces de componentes en el dominio de dispositivos electrónicos empotrados. Hereda las características del lenguaje Darwin	Por ser una evolución de Darwin, usa el cálculo - PI para modelar formalmente y lograr el chequeo de propiedades. Permite el modelado de arquitecturas dinámicas (Que cambian en tiempo de ejecución)
ADML	Es una especialización de ACME y adiciona el uso de meta-propiedades	Igual que ACME
ASDL	Usa esquemas del lenguaje Z para establecer dominios de estructuras posibles y caracterizar restricciones de las configuraciones	Usa el cálculo CSP [25] de Hoare para establecer posibles flujos de datos entre nodos en la arquitectura
AADL	Permite análisis de conformidad sintáctica y consistencia de datos entre interfaces para valoración rigurosa de atributos de calidad [53]	Usando representaciones formales y extensiones especializadas se pueden analizar propiedades comportamentales [53].
Pi-ADL	Permite la descripción formal de arquitecturas haciendo uso del cálculo Pi [54].Utiliza componentes y conectores, los cuales son descritos utilizando el concepto puerto y definiendo un comportamiento interno.	Ofrece una herramienta para la verificación y chequeo automático de algunas de las propiedades de la arquitectura.

Capítulo 3
Construcción de la solución

En este capítulo se describirán todos los aspectos considerados para construir una herramienta de software capaz de interpretar expresiones del cálculo ρ_{arq} que describen arquitecturas de software basadas en componentes usando las reglas de su semántica operacional para mostrar el flujo de ejecución arquitectónico. Dentro de estos aspectos se encuentran las tecnologías disponibles que apoyan la transformación de modelos, las consideraciones de análisis y diseño y la arquitectura de la aplicación desarrollada.

3.1 EVALUACIÓN DE TECNOLOGÍAS PARA LA TRANSFORMACIÓN E INTERPRETACIÓN DE MODELOS

El núcleo del problema de este proyecto está constituido en la necesidad de transformar la descripción realizada de una arquitectura con el cálculo ρ_{arq}, mapeando cada configuración arquitectónica obtenida en cada paso de reescritura a la notación gráfica de UML con el objetivo de representar el flujo de ejecución de la arquitectura descrita. Dentro de la revisión de tecnologías que podrían apoyar dicha labor, fueron encontradas tres alternativas que podrían contribuir en la solución de la necesidad planteada. A continuación se describen brevemente dichas alternativas.

3.1.1 MDA (Model-driven Architecture)

Es una alternativa para la especificación de sistemas de software basada en un conjunto de modelos que permiten la definición desde tres puntos de vista o niveles de abstracción. El primer nivel de abstracción permite concebir un sistema desde la perspectiva del negocio o del dominio, describiendo los aspectos funcionales del sistema pero ocultando toda la información relacionada con tecnología, estos modelos son denominados Modelos Independientes de la Computación (En adelante MIC) y

generalmente describen los procesos de negocio que manejará el sistema a implementar.

Esta alternativa también provee un nivel de abstracción desde la perspectiva de los modelos independientes de plataforma (MIP en adelante), los cuales permiten especificar un sistema incluyendo aspectos tecnológicos, pero sin relacionarlos directamente con una plataforma específica para su implementación. Y en el tercer nivel de abstracción se presentan los modelos específicos de plataforma (MEP en adelante), en los cuales se define el sistema en términos de los aspectos tecnológicos requeridos para su implementación sobre una plataforma definida.

Una especificación hecha bajo la filosofía de MDA consiste de modelos definitivos, independiente de la plataforma (MIP) y basado en UML e independiente de la computación (MIC), más uno o más modelos específicos de plataforma (MEP) y conjuntos de definición de interfaces [61].

MDA se enfoca principalmente en la funcionalidad y comportamiento de un sistema y no en la tecnología en la cual será implementada. Considera los detalles de implementación y las funciones de negocio en dos niveles diferentes. De esta forma, no es necesario realizar los modelos del mismo sistema cada vez que una nueva tecnología sea desarrollada, ya que el comportamiento y la funcionalidad son modelados una sola vez y la labor de mapear dichos modelos a los modelos específicos de plataforma es realizado por herramientas a través de los métodos de transformación.

El proceso de transformación en MDA consiste en la conversión de un MIP, combinado con otra información, en un MEP. Existen dos tipos de transformación la transformación vertical que corresponde a la conversión de un MIP a un MEP, o de un MEP a código fuente. También puede haber transformación horizontal, en la cual se puede convertir la especificación de un MEP a otro MEP.

Para realizar estas transformaciones MDA emplea el concepto de mapeos (mappings), estos son los encargados de describir las especificaciones y reglas de transformación entre MIP y MEP.

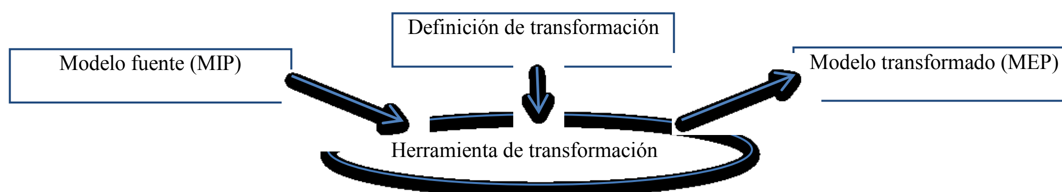

Figura 3.1. Transformación de modelos. Tomado de [61]

Para realizar dichas transformaciones, tal como se observa en la Figura 3.1, es necesaria una herramienta de transformación; sin embargo, no hay una herramienta estándar para realizar dicha transformación, así que existen diversas alternativas para definir la transformación y ejecutarla. Así por ejemplo, en el trabajo de Wei Zuo, Jinfu Feng y Jiaqiang Zhang [62] se emplea el lenguaje ASL (Action Specification Language) para definir el método de transformación y otros trabajos definen sus propios lenguajes para establecer dichas transformaciones [63].

3.1.2 ANTLR (Another tool for language recognition)

Esta herramienta, propuesta por Terrence Parr [64], provee un marco de trabajo para la construcción de reconocedores, compiladores y traductores desde descripciones gramaticales que contienen reglas semánticas.

ANTLR genera desde una gramática formal un programa que determina cuales oraciones son conformes con el lenguaje definido e incorpora un conjunto de herramientas de desarrollo que permiten al reconocedor comportarse como un traductor o un intérprete.

La herramienta puede ser utilizada como base para la construcción de intérpretes o de traductores que permitan la transformación desde un lenguaje a otro. Para esto emplea la estrategia de interpretación LL que analiza gramáticas libres de contexto empleando autómatas finitos determinísticos para realizar un preanálisis revisando desde la raíz del autómata hacia las hojas; es decir, analizando expresiones de izquierda a derecha. Esta estrategia de interpretación consiste en la construcción de un autómata finito determinístico que define los posibles estados en la interpretación de expresiones, las transiciones adecuadas y los estados finales de la gramática en análisis. Básicamente, el autómata define un conjunto de caminos establecidos por las posibles frases descritas en las reglas gramáticales que conforman expresiones bien formadas según el lenguaje en análisis, así, cada transición define la adición de una palabra a la expresión hasta llegar a predecir el estado final definido por el conjunto de palabras que establecen una expresión válida. De esta forma, para un lenguaje definido por las siguientes reglas gramaticales,

```
s: ID
 | ID '=' expresion
 | 'unsigned'* 'int' ID
 | 'unsigned'* ID ID;
```

Código 3.1. Definición de lenguaje de ejemplo para análisis LL(*). Tomado de [64]

Es posible construir el siguiente autómata finito determinístico:

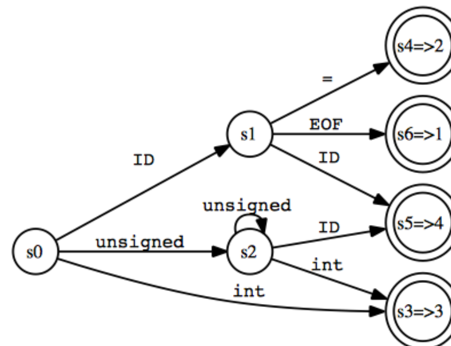

Figura 3.2. Autómata finito determinístico para análisis LL(*). La notación Sn => i indica "El estado predice la alternativa i". Tomado de [64]

En el cual se puede observar un estado s0 que determina el inicio del análisis y como cada transición adiciona una palabra a la frase que se está interpretando para cambiar entre estados y finalmente determinar una posible alternativa válida si el estado del autómata corresponde a uno de los estados finales o nodos hoja (s3, s4, s5, s6).

La entrada para ANTLR es una gramática libre de contexto aumentada con predicados sintácticos y semánticos y la posible adición de comportamientos fijos programados en java para dichos predicados. Los predicados sintácticos permiten un pre-análisis (*lookahead*) de la gramática mientras los predicados semánticos permiten construir un estado de los predicados para dirigir su interpretación. La programación java que puede incluirse en la definición de la gramática corresponde a la compatibilidad con versiones anteriores de la herramienta; sin embargo, el autor recomienda que los comportamientos requeridos en los traductores o intérpretes sean incluidos haciendo uso de la extensión de los patrones de programación provistos en la herramienta.

La versión de esta herramienta al momento de la implementación de este proyecto en el lenguaje Java es la 4.2.1. y se encuentra bajo la licencia de Apache en su versión 2.0.

3.1.3 DUALLy

DUALLy es un marco de trabajo construido como *plug-in* del ambiente de desarrollo integrado Eclipse. Permite la interoperabilidad entre lenguajes y herramientas de descripción arquitectural. Fue propuesto por Ivano Malavolta, Henry Muccini, Patrizio Pelliccione y Damien Andrew Tamburri [8] con el objetivo de ofrecer interoperabilidad a través de técnicas de transformación automatizadas entre modelos. Permite la transformación de conceptos de un modelo arquitectural en conceptos semánticamente equivalentes de otro modelo arquitectural. Para esto DUALLy trabaja en dos niveles de abstracción: a nivel de metamodelo y a nivel de modelo.

En el nivel de metamodelo la herramienta requiere que se provea una especificación del lenguaje arquitectural en términos de su metamodelo o de su perfil UML, luego se debe definir un conjunto de mapeos que relacionen semánticamente los conceptos arquitecturales entre los metamodelos.

A nivel de modelo, la herramienta recibe las especificaciones arquitecturales definidas en un lenguaje de descripción arquitectural que sea conforme con los metamodelos definidos en la aplicación. De esta forma, la herramienta permite la generación automática de transformaciones modelo a modelo.

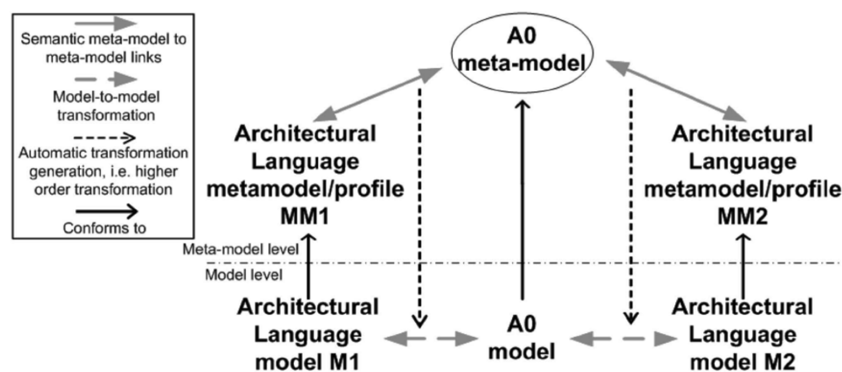

Figura 3.3. Vista conceptual de DUALLy. Tomado de [8]

3.2 CRITERIOS DE SELECCIÓN PARA LA TECNOLOGÍA BASE

La labor fundamental de esta investigación consistió en la transformación de configuraciones arquitectónicas especificadas con el cálculo ρ_{arq} a la notación visual de componentes usando UML 2.x. Cada configuración arquitectónica puede resultar de la aplicación de una regla de reescritura (Semántica operacional) del cálculo formal, así, aplicando sucesivamente reglas semánticas sobre una arquitectura es posible obtener la visualización del flujo de ejecución arquitectónica y al realizar la transformación de cada estado de la arquitectura, dicho flujo puede ser representado gráficamente. Por este motivo, se realizó una revisión de las tecnologías que permiten realizar transformaciones entre modelos o lenguajes y evaluar la contribución de cada una de ellas en este proyecto.

Dicha búsqueda partió de la necesidad de describir aspectos de configuración y comportamiento manejados con el cálculo ρ_{arq} en UML sin perder información de dichos aspectos durante el proceso de transformación. Así, la herramienta capaz de realizar dicha transformación sin tener pérdida de aspectos en el proceso sería la indicada para implementar en el proyecto.

La primera tecnología revisada fue MDA (Model-driven Architecture) por su enfoque de transformación de modelos; sin embargo, durante dicha revisión se encontró que MDA es una filosofía para la definición de metamodelos independientes y específicos de plataformas tecnológicas y algunas de las herramientas [65] que han seguido dicho paradigma se han enfocado en transformaciones específicas de modelos en su mayoría específicos de plataforma, desde luego ninguna de ellas contemplaba el cálculo ρ_{arq}. Adicionalmente, en aquellas herramientas de código abierto no fue fácil identificar la metodología de transformación que empleaban en sus propuestas, ya que de ser posible la utilización de dichos procedimientos, gran parte del trabajo en este proyecto de investigación habría sido solucionado. Sin embargo, al notar la dificultad para comprender las metodologias de transformación empleadas dicha revisión fue finalizada.

Por otro lado, en la revisión de DUALLy se logró identificar una aplicación de transformación genérica entre modelos, en la cual se considera no solamente el modelo generado para la descripción sino también su metamodelo, permitiendo generar reglas de conversión en ambos niveles y permitiendo la posibilidad de identificar la información que no es posible describir usando el metamodelo de destino.

Esta herramienta brinda muchas utilidades que facilitarían la implementación del proyecto y solo bastaría realizar una extensión de UML para que soporte la descripción del estado de las interfaces durante el flujo de ejecución de la arquitectura, de tal forma que sea posible definir en las reglas de mapeo entre modelos el momento en el cual interactúan las interfaces de dos componentes y su respectiva representación en la extensión de UML.

Una vez visualizada esta alternativa se procedió a la revisión de sus capacidades de integración con otras herramientas, desafortunadamente esta herramienta es deficiente en cuanto a no proveer algún tipo de API o documentación de su código fuente, motivo por

el cual su integración con la herramienta que se pretendía desarrollar incluiría un esfuerzo adicional.

Por otro lado, ANTLR no es una herramienta orientada a la transformación de modelos, esta se enfoca en la interpretación de gramáticas basada en la definición de una sintáxis y un conjunto de reglas gramaticales. Esta herramienta permite la generación automática de intérpretes capaces de identificar los elementos sintácticos empleados en una descripción textual basándose en una definición gramatical.

El uso de ANTLR en este proyecto brindaría un apoyo fundamental en la tarea de interpretar las expresiones del cálculo ρ_{arq} y en la generación de estructuras que permitirían la identificación y manejo de dichas expresiones; ofrece las herramientas suficientes para la implementación de traductores y un API con documentación que permitiría la manipulación de las estructuras generadas.

Finalmente, teniendo en consideración los aspectos mencionados anteriormente, se determinó que ANTLR es la herramienta que puede soportar el desarrollo de este proyecto, dado que a pesar de no tener herramientas de transformación brinda las bases y las utilidades necesarias para la implementación de procedimientos que permiten la interpretación y la construcción de un transformador desde expresiones formales a una notación visual con animación.

Uno de los aspectos principales que contribuyeron a la selección de ANTLR es la posibilidad que ofrece para la implementación de la semántica operacional que incluye el cálculo ρ_{arq}, dado que al tener una estructura con los elementos sintácticos de las expresiones es fácil identificar los patrones que generan la aplicación de dicha semántica. De esta forma, fue posible realizar la reescritura de dichas expresiones y obtener los estados de la arquitectura a través de su flujo de ejecución.

3.3 INCORPORACIÓN DE ANTLR

Para poder hacer uso pleno de las funcionalidades ofrecidas por ANTLR primero se definió la sintaxis y las reglas gramaticales (Tabla 2.2. Sintaxis del cálculo ρ_{arq}. Tomado de [58].) que definen el lenguaje. Dicha definición fue realizada haciendo uso de la notación Backus-Naur extendida con las siguientes adiciones requeridas por ANTLR para obtener el funcionamiento esperado en este proyecto.

- *Nombrado de expresiones alternativas:* En la definición de un tipo de expresión en la forma Backus Naur, es posible establecer varias alternativas para su definición a través del operador "|". Sin embargo, para permitir el control del intérprete para el manejo específico de comportamientos de reescritura en cada alternativa, es posible nombrar cada una de ellas y en la generación de la API de control son generadas operaciones para manejar cada alternativa. Esto se realiza haciendo uso del símbolo '#' seguido del nombre de la alternativa finalizando la misma línea donde se encuentra la definición así:

```
...
| ID '::' ID '/' expresion                    #Abstraccion
...
```

Código 3.2. Nombrado de expresiones alternativas.

- *Salto de expresiones:* ANTLR permite mediante la especificación de las reglas gramáticales de un lenguaje, identificar símbolos que no deben ser tenidos en cuenta en el proceso de interpretación de expresiones en el lenguaje descrito. Para esto ofrece la herramienta *skip*, la cual indica al análizador léxico que los simbolos indicados deben ser descartados del análisis.
  Así, para la definición de las reglas gramaticales del cálculo ρ_{arq}, se han establecido los símbolos para nueva línea, tabulación y retorno de carro como símbolos que no se deben considerar en el análisis de la construcción de palabras.

```
WS : [ \t\r\n]+ -> skip;
```

Código 3.3. Definición de símbolos a descartar en el análisis léxico.

Durante la caracterización de la gramática del cálculo ρ_{arq} se identificaron dos problemas fundamentales para la interpretación del cálculo:

Representación Simbólica: Cada una de las herramientas sintácticas provistas por el cálculo ρ_{arq}, emplea un conjunto de símbolos que representan y permiten identificar los componentes de una expresión y sus operadores. Para la escritura de dichos símbolos en un medio de almacenamiento, se requiere de un lenguaje especializado que permita definir cada símbolo y permita incluir tal definición dentro de una expresión.

Identificación de componentes en expresiones reducidas: La expresión reducida es un elemento sintáctico provisto en el cálculo ρ_{arq} (propuesta en [66]) que permite la definición de una aplicación con la posterior ejecución de un componente vacío. Es decir, este elemento permite la definición del envío de una variable a través de un canal. Dado que la notación de este elemento sintáctico se compone de un identificador del canal seguido del identificador de la variable a enviar, computacionalmente no es posible establecer hasta que letra de la expresión se debe abarcar para identificar el nombre del canal y de la variable de envío.

Para solucionar estos aspectos fue utilizado el lenguaje para la edición de textos científicos LATEX, el cual a través de sus herramientas permite la representación simbólica con la tipografía adecuada del cálculo ρ_{arq} (solucionando el primer aspecto mencionado); así mismo, al emplear este lenguaje es posible identificar inequívocamente cada una de las palabras que componen la sintaxis del cálculo, ya que LATEX ofrece un conjunto de etiquetas textuales para representar cada símbolo, lo cual simplifica la tarea de identificación y la reduce a identificación de etiquetas.

LATEX en términos generales ofrece para la herramienta de interpretación la alternativa de representación simbólica de la sintaxis propuesta en el cálculo ρ_{arq}; de tal manera, que pueda ser interpretada por ANTLR. Adicionalmente, al emplear esta

herramienta el problema de la representación de la aplicación reducida se soluciona, ya que es posible a través de las etiquetas que ofrece el lenguaje separar la variable de envío y el canal para que sean claramente identificables, cosa que de lo contrario sería un inconveniente en la interpretación de las expresiones de una arquitectura. Al utilizar LaTeX en la representación de una aplicación reducida se separa fácilmente la variable que representa el dato a enviar de la variable que representa el canal a través del cuál se envía, ya que el dato a enviar estará contenido dentro de la etiqueta \overline{} de LaTeX y el texto que precede a la etiqueta puede ser considerado como el canal de envío. Usando la forma Backus-Naur la manera de escribir esta expresión es la siguiente:

```
ID '\\overline{' ID '}'
```

Código 3.4. Definición de aplicación reducida.

3.3.1 Definición de la sintaxis del cálculo ρ_{arq} con representación LATEX

Una vez establecida la alternativa de representación fue necesario expresar la sintaxis definida para el cálculo ρ_{arq} usando los símbolos que provee la notación LATEX.

ANTLR permite el reconocimiento de lenguajes teniendo como base su representación usando la forma Backus-Naur extendida, para esto, provee una API que permite la interpretación de expresiones y el manejo de cada elemento sintáctico identificado.

Como resultado de este proceso se obtuvo la siguiente definición para el cálculo ρ_{arq}:

```
grammar RhoArqV1;

//Gramática
//Inicio de documento
documento :  ENCABEZADO (LATEX* ecuacion+ LATEX*)+ '\\end{document}';

//Identificador de ecuaciones latex
ecuacion : '\\begin{equation*}' definicion '\\end{equation*}'
         | '\\begin{equation}' definicion '\\end{equation}';

//Definiciones
definicion : ID '=' expresion;

//Expresion Rho-Arq
expresion: '(' expresion ')'                            #Grupo
                | '[' expresion ']'                     #Grupo2
                | expresion '\\wedge' expresion         #Composicion
                | '\\exists' ID expresion               #Declaracion
                | ID '::' ID '/' expresion              #Abstraccion
                | ID ':' ID '/' expresion               #Replicacion
                | aplicacionreducida '/'  expresion     #Aplicacion
                | '\\tau/' expresion                    #ReaccionInterna
                | interior                              #InteriorDeComponente
                | '[' ID '/' ID ']' expresion           #Reemplazo
                | ejecucionExitosa                      #EjecExitosa
                | ejecucionNoExitosa                    #EjecNoExitosa
                | aplicacionreducida                    #AppReducida
                | observacion                           #ExpObservacion
                | combinador                            #ExpCombinador
                | observacionRepetida                   #ObsRepetida
                | NULO                                  #Nulo
                | ID                                    #Identificador
                ;
```

```
//Operadores
aplicacionreducida: ID '\\overline{' ID '}';
interior: ID '^{(int)}';
ejecucionExitosa: ID '^{\\top}';
ejecucionNoExitosa: ID '^{\\bot}';
observacion: OSO '(' ID ')' DO expresion ELSE expresion;
observacionRepetida: OSOREP '(' ID ')' DO expresion ELSE expresion;
combinador: IF clausula+ (ELSE ID)?;
clausula: '(' guarda 'then' ID ')';

//Operadores logicos
restriccionEcuacional: ID '=' ID;
conjuncion: '(' conjuncion ')'
                 | '[' conjuncion ']'
                 | conjuncion '\\dot{\\wedge}' conjuncion
                 | guarda '\\dot{\\wedge}' guarda;
cuantificadorExistencial: '\\dot{\\exists}' ID guarda;
guarda: VERDAD
      | FALSEDAD
      | restriccionEcuacional
      | cuantificadorExistencial
      | ID
      ;
//Lexico
IF: 'if';
OSOREP:'!OSO';
OSO: 'OSO';
DO: 'do';
ELSE: 'else';
NULO: '\\top';
VERDAD: '\\dot{\\top}';
FALSEDAD: '\\bot';
ENCABEZADO : '\\documentclass' ('['.*?']')? '{' .*? '}';
FINENCABEZADO: '\\end{document}';
LATEX: ('\\usepackage'|'\\begin'|'\\author'|'\\title') ('['.*?']')? '{' .*? '}';
ID : ([a-zA-Z_,.]+ [0-9]*)+
   | ([a-zA-Z_{},.]+ [0-9]*)+ '(' [a-zA-Z_,.]+ ')'
   | ([a-zA-Z,.]+ [0-9]*)+'_{'([a-zA-Z,.]+ [0-9]*)+'}';
WS : [ \t\r\n]+ -> skip;
```

Código 3.5. Definición de la sintaxis del cálculo ρ_{arq} usando la representación del lenguaje LATEX

En esta definición se pueden observar dos partes, la primera que establece la gramática con la cual se describirán las arquitecturas haciendo uso de LATEX y la segunda que establece el léxico o las palabras que se permitirán en la definición de las ecuaciones de los sistemas descritos. En dichas secciones se define la sintaxis propuesta para el cálculo ρ_{arq}, describiendo la manera como se deben escribir cada una de las expresiones (Tabla 2.2. Sintaxis del cálculo ρ_{arq}. Tomado de [58].) que propone el cálculo. Adicionalmente se incorpora una nueva expresión propuesta en [58] y que corresponde al reemplazo, con el objetivo de representar el momento en el cual se ha identificado una interacción entre una abstracción o replicación y una aplicación. Así, una relación entre los términos propuestos en el cálculo ρ_{arq} y su representación simbólica en LaTeX estaría definida como se muestra en la siguiente tabla:

Término ρ_{arq}	**Tipografía**	**LaTeX**
Composición	A ^ B	A \wedge B
Null	⊤	\top
Parte Interna	$A^{(int)}$	A^{(int)}
Combinador de selección	$if(C_1 \dots C_n)\ else\ A$	$if(C_1)\ (C_2) \dots (C_n)\ else\ A$

condicionada		Con $C_K :=$ "$Restricción$" $then\ B$
Abstracción	$x :: \bar{y}/E$	$x :: y/E$
Aplicación	$x\bar{y}/E$	`x \overline{y}/E`
Declaración	$\exists wE$	`\exists w E`
Replicación de abstracción	$x : \bar{y}/E$	$x : y/E$
Ejecución exitosa	$E^{\top}$	`E^{\top}`
Ejecución no exitosa	$E^{\bot}$	`E^{\bot}`
Observación	OSO(E) do F else G	OSO(E) do F else G
Observación repetida	!OSO(E) do F else G	!OSO(E) do F else G
Verdad lógica	$\dot{\top}$	`\dot{\top}`
Falsedad lógica	$\bot$	`\bot`
Restricción ecuacional	$x = y$	$x = y$
Conjunción de restricciones	$\phi \dot{\wedge} \psi$	`\phi\dot{\wedge}\psi`
Cuantificador existencial	$\dot{\exists} x\phi$	`\dot{\exists}x\phi`

Tabla 3.1. Correspondencia de elementos del cálculo y notación LaTeX. Fuente Autor.

Dentro del léxico se puede observar la definición de las etiquetas de inicio y finalización de ecuaciones en LaTex, las cuales fueron utilizadas para identificar las ecuaciones empleadas en la descripción de una arquitectura; sin embargo, estas no corresponden a la definición explícita de elementos sintácticos del cálculo ρ_{arq}.

```
//Identificador de ecuaciones latex
ecuacion : '\\begin{equation*}' definicion '\\end{equation*}'
         | '\\begin{equation}' definicion '\\end{equation}';
```

Código 3.6. Definición de inicio y fin de ecuaciones.

Asi mismo en la parte gramatical, se pueden observar tres reglas que corresponden a la estructura propia de un documento LATEX y a la conformación genérica de un conjunto posible de etiquetas que pueden ser empleadas en el encabezado del documento LaTex y que no son consideradas en la interpretación de arquitecturas, pero, que fueron consideradas en la estructura del documento para permitir compatibilidad con un documento LaTex bien formado.

```
ENCABEZADO : '\\documentclass' ('['.*?']')? '{' .*? '}';
FINENCABEZADO: '\\end{document}';
LATEX: ('\\usepackage'|'\\begin'|'\\author'|'\\title') ('['.*?']')? '{' .*? '}';
```

Código 3.7. Definición de etiquetas LaTex no consideradas en interpretación de arquitecturas.

3.3.2 Construcción del API para interpretación del cálculo ρ_{arq}

ANTLR provee una herramienta automática que permite a partir de la definición de una sintáxis la generación de clases que permiten la manipulación de las expresiones interpretadas y la generación de analizadores léxicos y sintácticos. En este proyecto se ha empleado dicha herramienta utilizando como entrada la definición sintáctica expresada en

el Código 3.5, para generar dichos analizadores se empleó la herramienta java de ANTLR en su versión 4.1, mediante la siguiente línea de comando:

java -jar antlr-4.1-complete.jar RhoArqV1.g4 -listener -visitor

Con este comando ANTLR genera un conjunto de clases que facilitan la manipulación de las expresiones escritas bajo la sintaxis del cálculo. Como resultado de su ejecución, ANTLR genera las clases *RhoArqV1Lexer* y *RhoArqV1Parser* para realizar el análisis léxico y sintáctico respectivamente. Dentro del analizador sintáctico, se generan un conjunto de clases descendientes de la clase *ParserRuleContext* (Propia del API de ANTLR) y que serán empleadas para la representación de las estructuras gramaticales, adicionalmente, al indicar las opciones –listener y –visitor, ANTLR genera dos interfaces *RhoArqV1Listener* y *RhoArqV1Visitor*, las cuales permitirán la manipulación y análisis sobre las estructuras identificadas en las expresiones analizadas mediante el uso de los patrones comportamentales Observador y Visitador.

Dado que durante la implementación de la herramienta propuesta en este proyecto, las clases de mayor utilidad fueron las provistas para la manipulación y análisis de las expresiones, a continuación una descripción detallada de dichas clases.

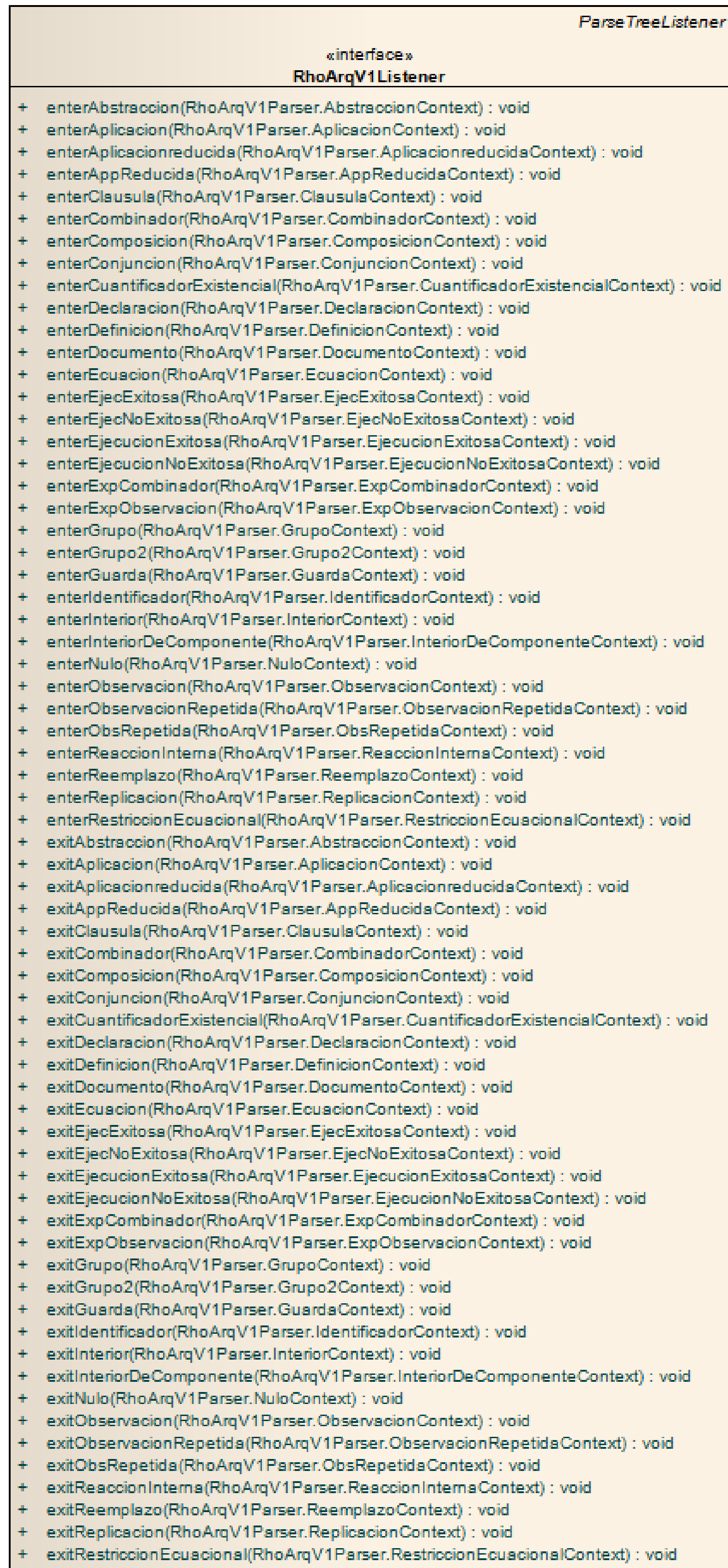

Figura 3.4. Interfaz para el análisis de expresiones mediante el patrón de comportamiento observador.

Como se puede observar en la gráfica anterior, la interfaz está provista por un conjunto de métodos que pueden clasificarse en dos tipos, métodos de entrada y métodos de salida. Dado que esta interfaz fue diseñada para comportarse de acuerdo al patrón comportamental Observador, los métodos en el objeto de la clase realizadora de la interfaz serán invocados por el objeto de tipo `ParseTreeWalker` (propio de ANTLR) en los momentos del recorrido de las expresiones en los que se identifique se ha iniciado o finalizado la revisión de un tipo de estas, los métodos de entrada serán invocados en el inicio de la expresión y los métodos de salida en el momento de finalizar. De esta manera, cada tipo de expresión cuenta con un método para el inicio y uno para la finalización, métodos empleados en el proyecto para la identificación de expresiones.

El patrón observador es una estructura que permite la consistencia entre un conjunto de clases que cooperan entre sí, de tal forma que cuando el estado de una de ellas cambia, todas aquellas que requieren conocer dicho cambio son notificadas y actualizadas automáticamente. En este caso, las clases que requieren ser notificadas son los *listeners* creados y registrados en la clase que recorre el árbol de expresiones. Estas clases serán notificadas cuando al recorrer un árbol, se inicie o se finalice la revisión de un tipo específico de expresión de tal forma que las clases conozcan la expresión revisada y realicen las acciones requeridas para procesar dicha expresión. Este patrón es implementado en la API provista por ANTLR, por tanto en este proyecto solamente se hace uso de la estructura implementando la interfaz provista y creando tantos *listeners* como sean necesarios. Así, los observadores creados para el proyecto se muestran en la siguiente figura:

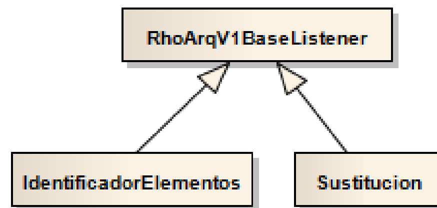

Figura 3.5. Observadores de expresiones. Fuente autor.

Cada observador se interesa en un tipo de expresión específica con la cual será notificado, así el observador `IdentificadorElementos` solamente le interesa que se le notifique cuando se esté examinando las expresiones de tipo ecuacion, pues son ellas quienes definen elementos de una arquitectura y a partir de ellas es posible identificarlos. Por otro lado, el observador `Sustitucion` solamente es notificado cuando en el árbol de expresiones se ha encontrado una expresión de tipo identificador, ya que es posible que los identificadores sean susceptibles de una sustitución en la expresión, cambiando dicho identificador por la expresión que lo define; al detectar esto, el observador `Sustitucion` se encarga de reemplazar en el árbol de expresiones dicho identificador.

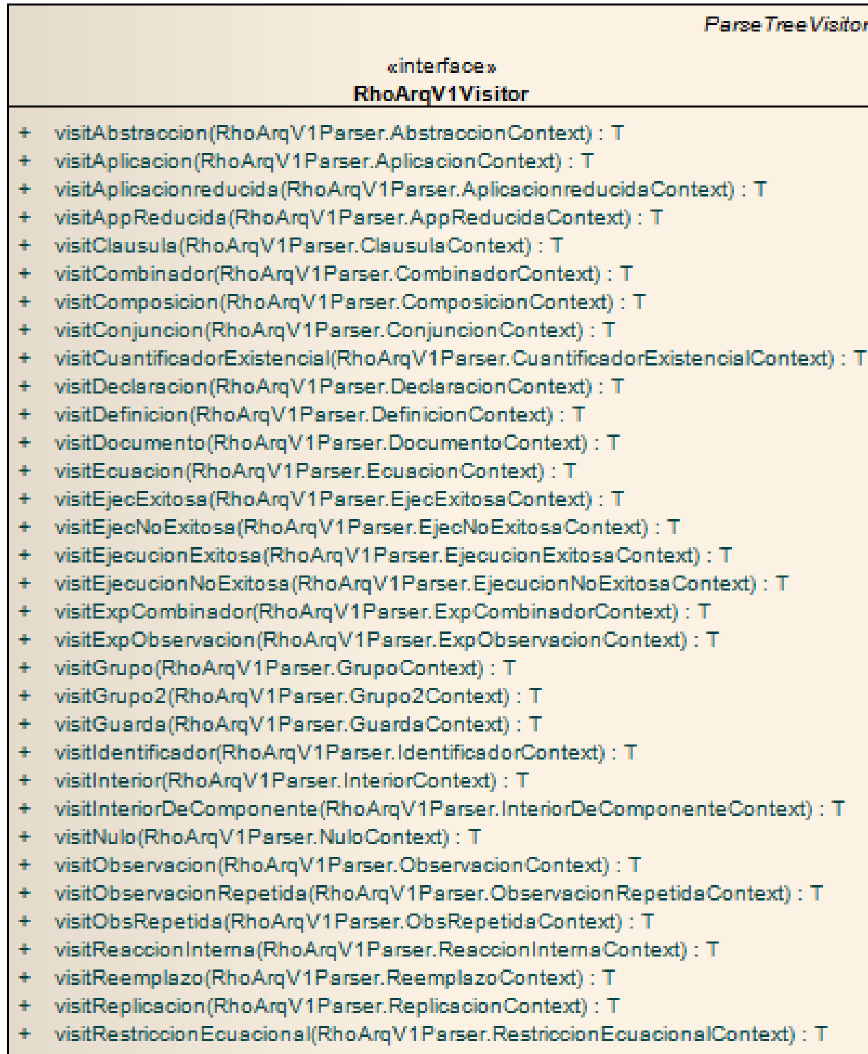

Figura 3.6. Interfaz para el análisis de expresiones mediante el patrón de comportamiento visitador.

La interfaz `RhoArqV1Visitor` provee un método por cada tipo de expresión definida en la gramática del cálculo, el cual es invocado en el objeto de la clase que la realice cuando en la revisión del árbol de expresiones (llamado a la operación `visit(ParseTree):<<T>>`) realizada por el analizador sintáctico se ha identificado una expresión del tipo específico. Dichos métodos fueron empleados en el proyecto para realizar las tareas de reescritura y la aplicación de la semántica operacional del cálculo. La interfaz adicionalmente cuenta con una plantilla que especifica un tipo genérico <<T>> para el retorno de las operaciones y así permitir un uso más amplio dentro de las tareas de análisis requeridas.

El patrón visitador es una estructura que permite la adición de operaciones o comportamientos a una clase predefinida, utilizado en los escenarios en los cuales dicho comportamiento depende de la naturaleza de las clases concretas. Su configuración permite que sea aplicado en objetos de un mismo tipo cuya estructura no cambia pero sus comportamientos son diferentes. Para este caso particular, cada visitador implementará la forma como se deben tratar los tipos de expresiones que se encuentren en el árbol de las ecuaciones analizadas y generar una nueva expresión ya sea para reescribir o para

considerar en los posteriores análisis. Los visitadores implementados en este proyecto tienen como objetivo realizar reescritura de ciertas expresiones del árbol, de acuerdo a la semántica operacional del cálculo ρ_{arq}, como resultado de dichas reescrituras, las operaciones implementadas en los visitadores retornan, además de indicadores de si hubo o no reescritura, la nueva expresión definida por la interacción entre expresiones. A continuación un diagrama que representa los visitadores implementados en este proyecto.

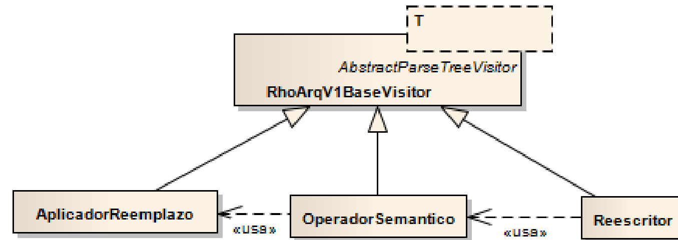

Figura 3.7. Visitadores implementados en el proyecto. Fuente autor.

En la siguiente sección se expondrá de una manera más específica la forma en la cual estas interfaces fueron implementadas dentro del diseño propuesto para la herramienta.

3.4 DEFINICIÓN DE LA ARQUITECTURA DE LA SOLUCIÓN

El modelo funcional definido para la herramienta presenta un conjunto de casos de uso que brindan algunos de los servicios que requiere un arquitecto para hacer uso del cálculo ρ_{arq} de una manera sencilla.

La Figura 3.8 muestra los casos de uso base implementados en la herramienta, los cuales permiten el inicio de la aplicación, la carga de archivos con especificaciones en calculo ρ_{arq} de arquitecturas, el inicio y la pausa de la ejecución de arquitecturas y la exportación a XMI de la estructura.

Para la construcción de una herramienta que permita comprobar la hipótesis planteada en este proyecto y proveer los casos de uso, fue necesario pensar en cinco módulos, cada uno con una función específica y que en conjunto permitieran la representación gráfica del flujo de ejecución de una arquitectura de software representada con el cálculo ρ_{arq}. Como resultado del proceso de diseño, se logró obtener la configuración arquitectónica de la Figura 3.9.

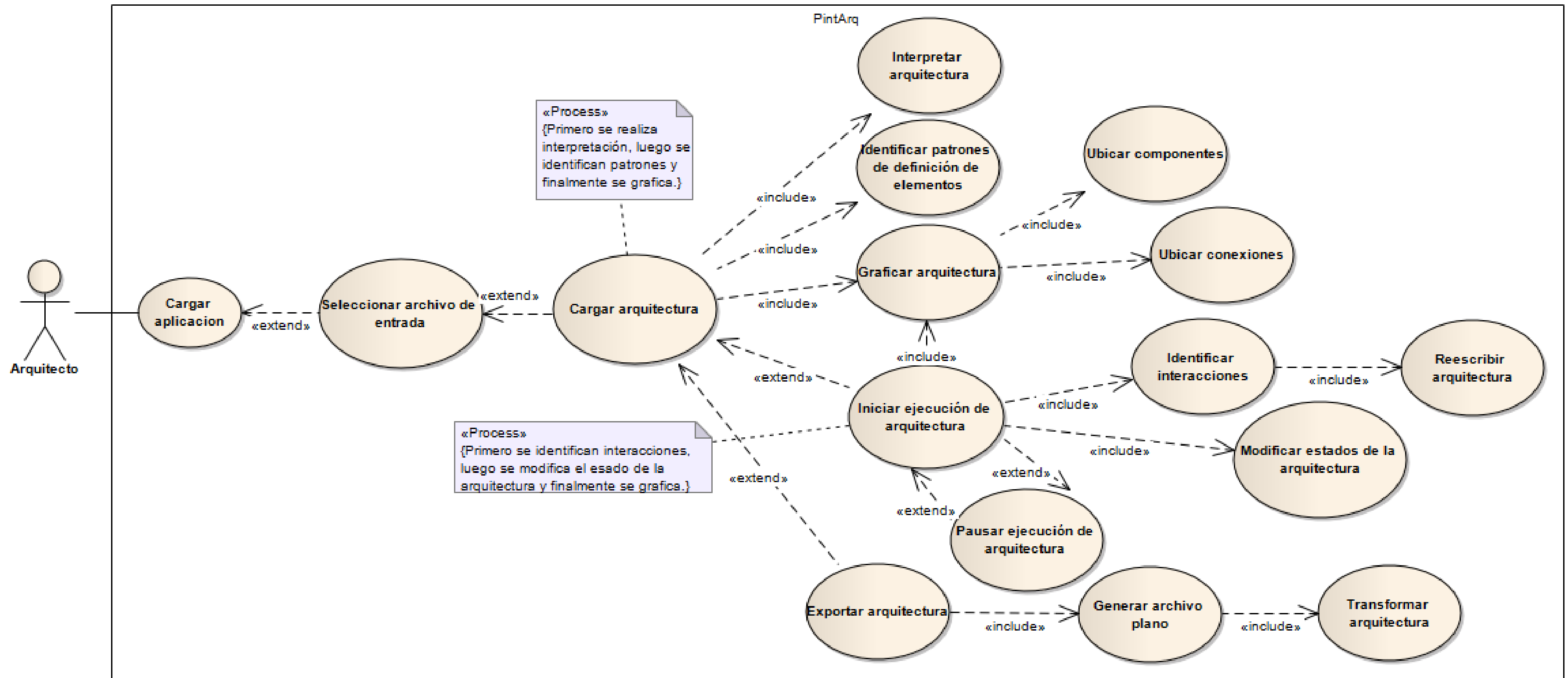

Figura 3.8. Casos de uso PintArq.

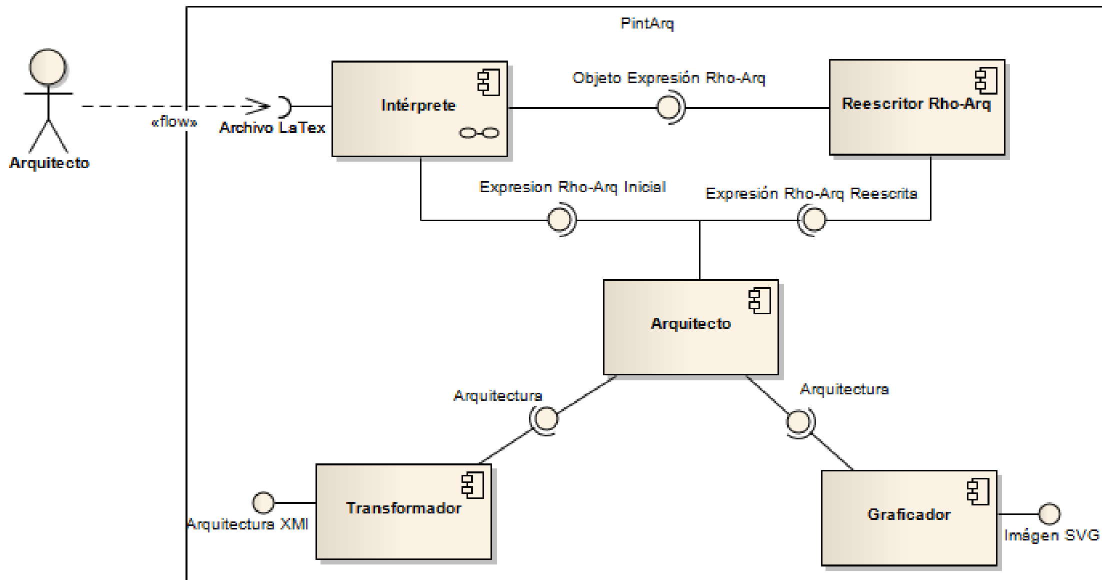

Figura 3.9. Arquitectura de la solución. Fuente autor.

En esta solución se plantean cinco módulos de software que en términos generales realizan las tareas de interpretación de expresiones, aplicación de la semántica operacional del cálculo ρ_{arq}, identificación de la estructura definida para las arquitecturas descritas, representación de la arquitectura en XMI y representación visual de la arquitectura y su flujo de ejecución. La comunicación entre dichos módulos es con base a objetos de software definidos por ANTLR para el manejo de expresiones, y objetos definidos por el autor del proyecto para representar computacionalmente una arquitectura de software.

A continuación se describe en detalle cada uno de estos módulos.

3.4.1 Intérprete

La primera tarea que se debe realizar con el objetivo de poder analizar una arquitectura es la identificación de las expresiones con las cuales dicha arquitectura está siendo descrita, en gran parte dicha labor es realizada por la API provista por ANTLR, la cual construye un árbol de expresiones donde los nodos del árbol indican el tipo de expresión y las hojas indican cada uno de los *tokens* empleados en la descripción.

El intérprete basado en ANTLR es el encargado de aplicar la identificación y clasificación de las expresiones que constituyen la especificación arquitectónica en cálculo ρ_{arq}, posteriormente extrae la ecuación que define el sistema descrito y todas las ecuaciones e identificadores empleados en la descripción de la arquitectura. Lo anterior permitirá establecer en tareas de otros módulos cuales de dichos identificadores son componentes y cuales de ellos variables.

3.4.1.1 Casos de uso implementados

Este módulo implementa uno de los casos de uso planteados para la herramienta, denominado "Interpretar arquitectura". Él se encarga de realizar la interpretación del cálculo ρ_{arq} en busca de patrones tanto estructurales como comportamentales que permitan establecer elementos arquitecturales o interacciones definidas en la arquitectura.

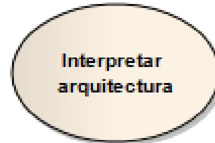

Figura 3.10. Caso de uso implementado por el intérprete.

3.4.1.2 Modelado estructural

Para lograr la ejecución de las tareas de este módulo fue necesaria la implementación de dos clases, la clase principal denominada `Reescritor` y la clase `IdentificadorElementos`.

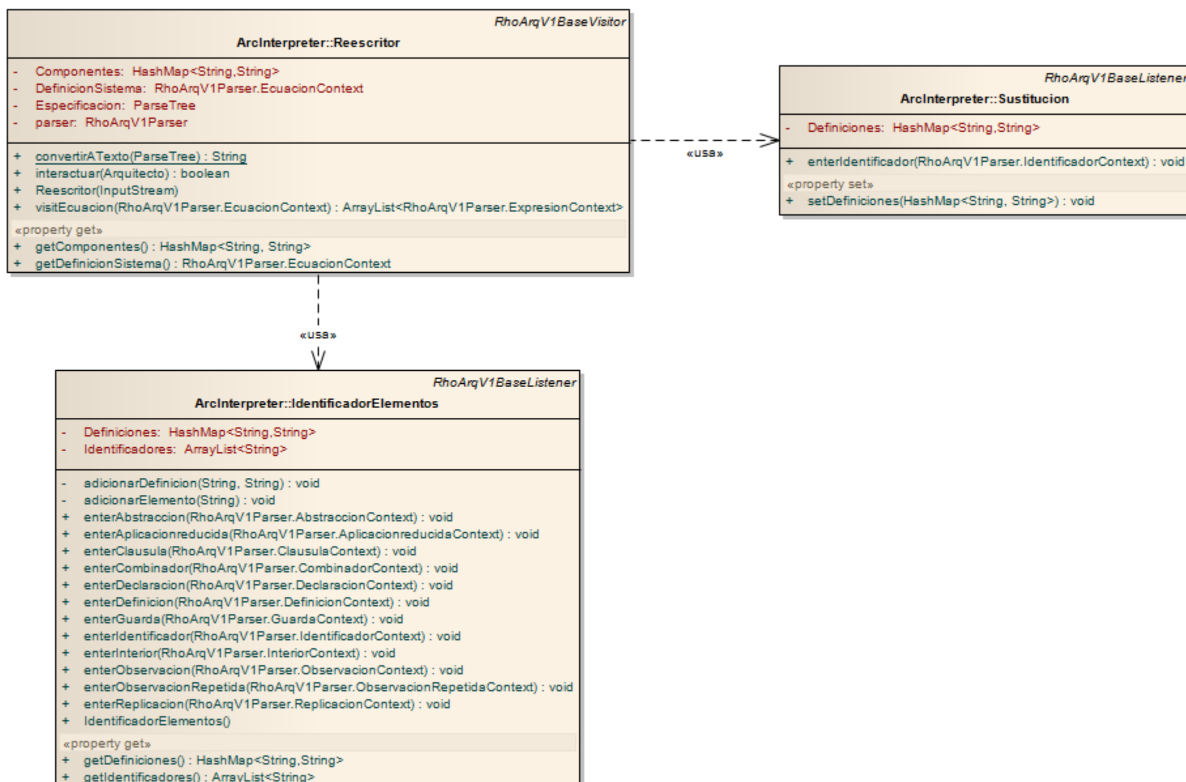

Figura 3.11. Clases definidas para el módulo intérprete.

La clase `Reescritor` fue empleada como la clase principal de la aplicación, a través de la cual se realizan las tareas de alto nivel. En este módulo dicha clase inicia con la invocación de las operaciones de interpretación de lenguajes que realiza ANTLR, con el

objetivo de construir un árbol de expresiones que clasifica cada uno de los elementos incluidos en el documento Latex que describe la arquitectura a analizar.

```
ANTLRInputStream Arquitectura = new ANTLRInputStream(DefinicionRhoArq);
RhoArqV1Lexer lexer = new RhoArqV1Lexer(Arquitectura);
CommonTokenStream tokens = new CommonTokenStream(lexer);
parser = new RhoArqV1Parser(tokens);
Especificacion = parser.documento();
```

Código 3.8. Preparación y utilización de ANTLR para construcción del árbol de expresiones.

El código muestra la ejemplificación de objetos de ANTLR para interpretar el archivo que contiene una especificación con cálculo ρ_{arq} de una arquitectura. Para dicha interpretación, se debe cargar en memoria el contenido del archivo, lo cual se realiza con la variable *Arquitectura* de tipo *ANTLRInputStream*, luego se crea un objeto de tipo *RhoArqV1Lexer* que hará las veces de analizador léxico de la arquitectura cargada, después se crea el objeto *tokens* de tipo *CommonTokenStream* que separa la arquitectura cargada en términos de tokens o expresiones, seguido de la creación del analizador gramatical *parser* de tipo *RhoArqV1Parser*, para finalmente realizar el análisis gramatical de la arquitectura y construir un árbol de expresiones que muestra los elementos y su composición, si la arquitectura está bien formada de acuerdo a la sintáxis del Código 3.5.

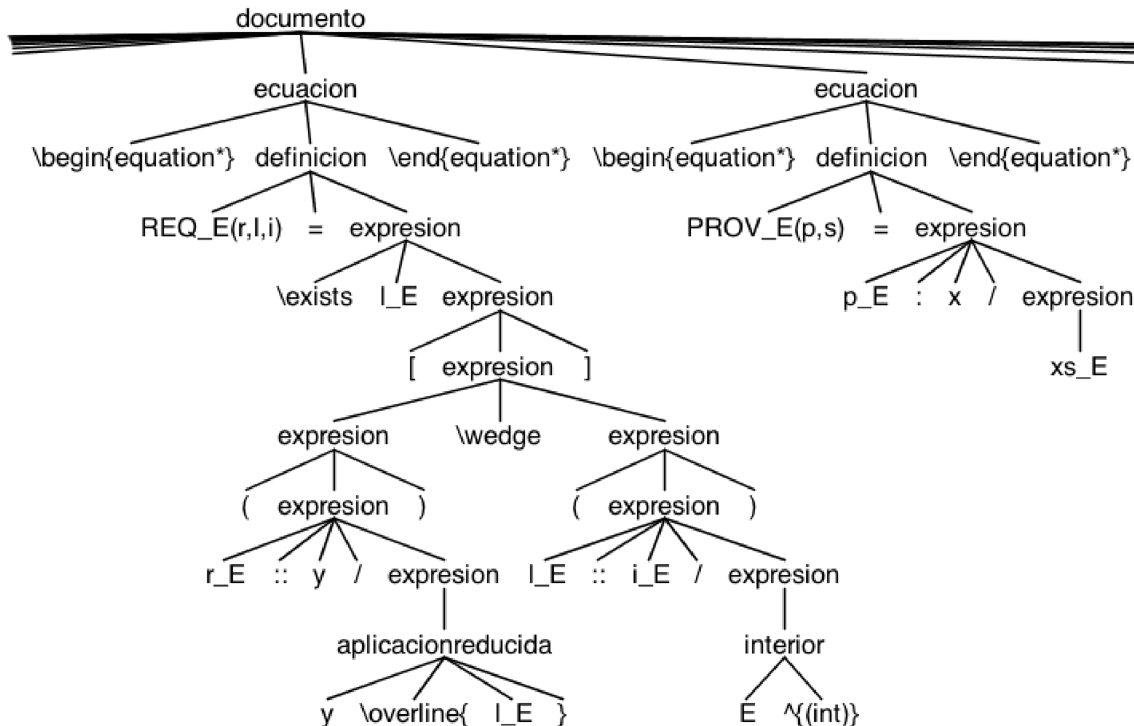

Figura 3.12. Ejemplo de árbol de expresiones construido por ANTLR.

En la Figura 3.11 se puede observar como la clase Reescritor hereda operaciones de la clase *RhoARqV1BaseVisitor*, específicamente la operación *visitEcuacion(EcuacionContext):ArrayList<ExpresionContext>*, cuya labor es la identificación de la ecuación que describe el sistema a analizar. Dicha operación identifica la primera ecuación incluida en la descripción de una arquitectura y la asume como la

ecuación que define al sistema descrito.

```
@Override public ArrayList<RhoArqV1Parser.ExpresionContext> visitEcuacion(@NotNull
RhoArqV1Parser.EcuacionContext ctx){
   if(this.DefinicionSistema==null)
   {
        this.DefinicionSistema = ctx;
   }
   return null;
}
```

Código 3.9. Identificación de la ecuación del sistema.

Posteriormente, se procederá a la detección de identificadores haciendo uso de la clase *IdentificadorElementos*, la cual hereda operaciones de la clase *RhoArqV1BaseListener*. Las operaciones heredadas permiten la identificación de todas las expresiones cuya composición sintáctica contiene un identificador, esto con el objetivo de almacenar en la propiedad denominada *Identificadores* todos los identificadores empleados en la descripción y permitir que los demás módulos puedan conocer las variables o componentes empleados en la descripción. Para poder emplear las funcionalidades de este identificador se requiere de la clase *ParseTreeWalker*, la cual realiza la tarea de recorrer todo el árbol de expresiones construido con base en la definición de la arquitectura y de acuerdo a los tipos de expresión notifica a la clase registrada, para que sea esta quien realice las acciones adecuadas para cada tipo de expresión, en este caso establecer los identificadores en la definición arquitectónica.

```
ParseTreeWalker walker = new ParseTreeWalker();
IdentificadorElementos Identificador = new IdentificadorElementos();
walker.walk(Identificador, Especificacion);
this.Componentes =Identificador.getDefiniciones();
```

Código 3.10. Identificación de elementos en el árbol de expresiones.

Las actividades mencionadas anteriormente se realizan en el constructor de la clase *Reescritor*, ya que hacen parte de la preparación para proceder a la reescritura de expresiones.

3.4.1.3 Modelado dinámico

El módulo intérprete es mayormente utilizado en la carga inicial de los archivos que contienen la especificación de la arquitectura a analizar, y es el insumo para definir la estructura de dicha arquitectura. En la Figura 3.13 se muestra en un diagrama de secuencia las interacciones de los objetos definidos en la herramienta para interpretar y definir en memoria una arquitectura. El diagrama corresponde a la implementación del caso de uso “Interpretar arquitectura”.

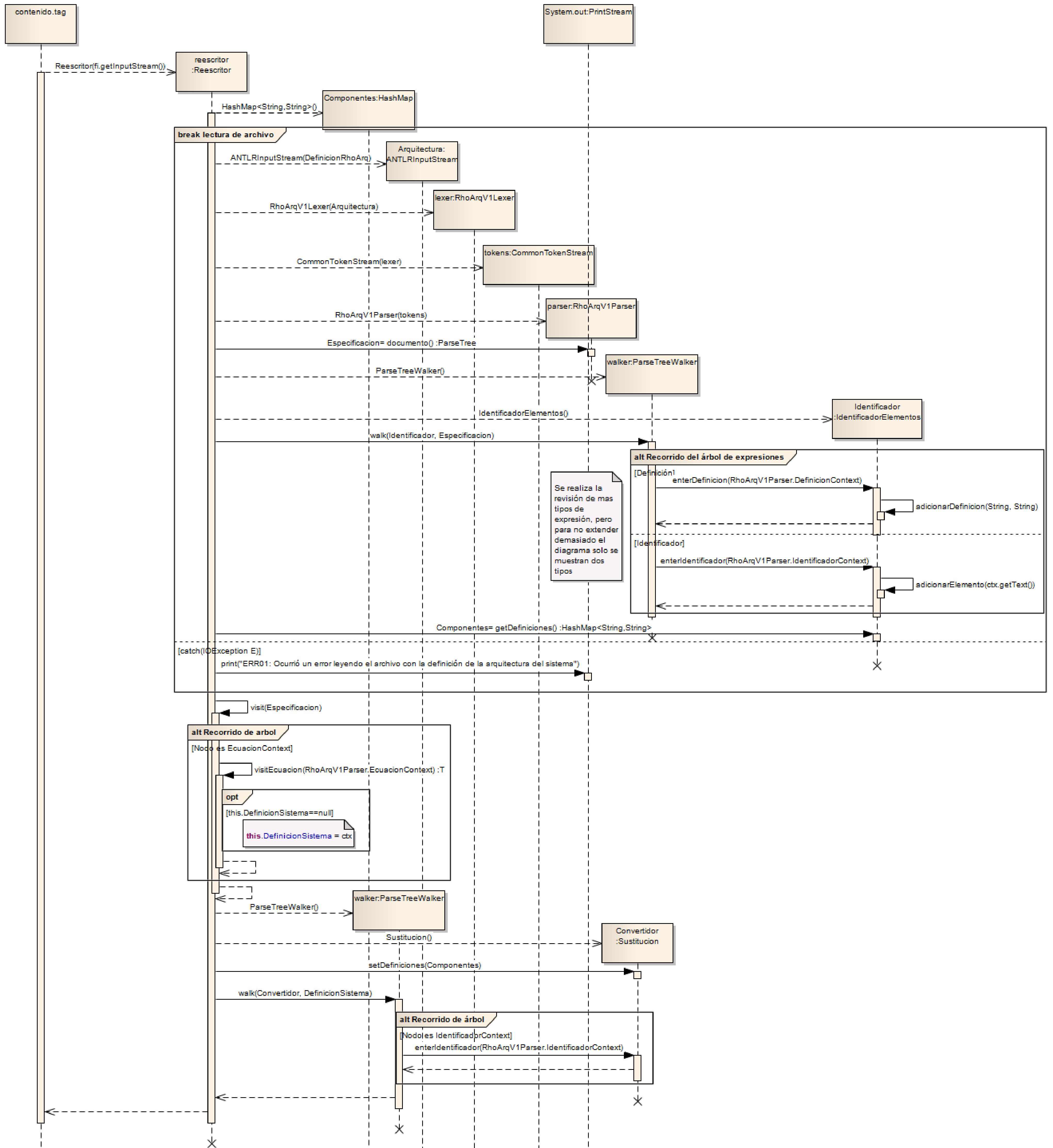

Figura 3.13. Modelo dinámico caso de uso "Interpretar arquitectura".

3.4.2 Reescritor ρ_{arq}

El módulo reescritor es el encargado de aplicar las reglas de reducción y la semántica operacional propuesta para el cálculo ρ_{arq} para cada arquitectura cargada en el sistema con el objetivo de identificar los posibles estados de ejecución de dichas arquitecturas.

3.4.2.1 Casos de uso implementados

Los casos implementados por el módulo son "Identificar interacciones", "Reescribir arquitectura" y "Modificar estados de la arquitectura". Este conjunto de casos de uso permiten establecer los momentos en los cuales los elementos de una arquitectura pueden interactuar y generar un cambio de estado, así, en los casos de uso se puede identificar la interacción, generar el proceso de reescritura para establecer la nueva ecuación del sistema y luego mapear dicho estado en la arquitectura manejada en memoria.

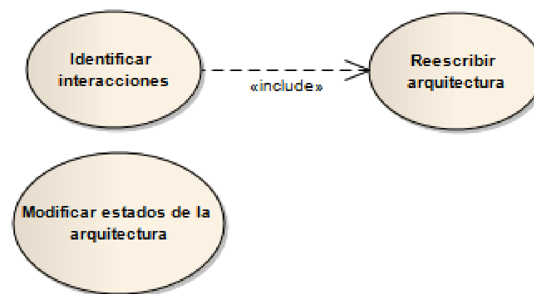

Figura 3.14. Casos de uso implementados en el módulo Reescritor ρ_{arq}.

3.4.2.2 Modelado estructural

Este módulo fue diseñado como lo muestra el diagrama de clases de la Figura 3.15, donde la clase *Reescritor* es quien realiza la organización de las tareas de reescritura utilizando las operaciones de las clases *Sustitucion* y *OperadorSemantico*.

La primera actividad realizada por el reescritor corresponde a la sustitución de identificadores en la ecuación del sistema por sus respectivas definiciones y así, obtener la ecuación del sistema con todos los elementos descritos en términos de las variables que serán intercambiadas entre componentes. Esta tarea es realizada por la clase *Sustitucion* y dado que es una labor de preparación para la aplicación de la semántica operacional, fue incluida en el constructor de la clase *Reescritor*.

```
ParseTreeWalker walker = new ParseTreeWalker();
Sustitucion Convertidor = new Sustitucion();
Convertidor.setDefiniciones(Componentes);
walker.walk(Convertidor, DefinicionSistema);
```

Código 3.11. Sustitución de identificadores por definiciones.

La clase *Sustitucion* es una implementación del observador para expresiones provisto por la API de ANTLR (El detalle de la implementación es explicado en [67] sección 7.2) para realizar la interpretación del cálculo ρ_{arq}. Esta implementación emplea el

modelo PUSH en los eventos en que se recorra un árbol de expresiones y el nodo que se evalúe sea de tipo identificador, así, cuando se recorra un árbol de expresiones y se le informe a este observador el hallazgo de un identificador, él podrá hacer el reemplazo por su definición. La clase está enfocada exclusivamente en aquellas expresiones de tipo identificador, ya que todos aquellos identificadores empleados en la ecuación del sistema, deben ser reemplazados por sus definiciones respectivas. De esta forma, para cada identificador utilizado en la definición del sistema, se buscará dentro de la descripción de la arquitectura ecuaciones que tengan una definición para dicho identificador y en el árbol de expresiones de la ecuación del sistema se reemplazará el identificador por el árbol de expresiones con la definición de dicho componente.

```
/*Cargue del archivo con la definición Rho-arq de la arquitectura*/
ByteArrayInputStream Archivo = new ByteArrayInputStream(Expresion.getBytes());
ANTLRInputStream Arquitectura = new ANTLRInputStream(Archivo);
/*Creación del analizador léxico y de los tokens del lenguaje*/
RhoArqV1Lexer lexer = new RhoArqV1Lexer(Arquitectura);
CommonTokenStream tokens = new CommonTokenStream(lexer);
/*Creación del analizador sintáctico*/
RhoArqV1Parser parser = new RhoArqV1Parser(tokens);
/*Análisis de la definición y creación del árbol de expresiones*/
ParseTree NuevoArbol = parser.expresion(0);
/*Si el elemento raíz del árbol es del tipo ExpresionContext se realiza el reemplazo de la
variable por su definición*/
if (NuevoArbol instanceof ExpresionContext ){
   Padre.children.add(Indice, NuevoArbol);
   ((ExpresionContext) NuevoArbol).parent = Padre;
}
/*Se realiza un recorrido en el árbol de la definición de la variable, para realizar en
dicha definición los reemplazos necesarios*/
ParseTreeWalker walker = new ParseTreeWalker();
walker.walk(this, NuevoArbol);
```

Código 3.12. Reemplazo de identificadores en árbol de expresiones.

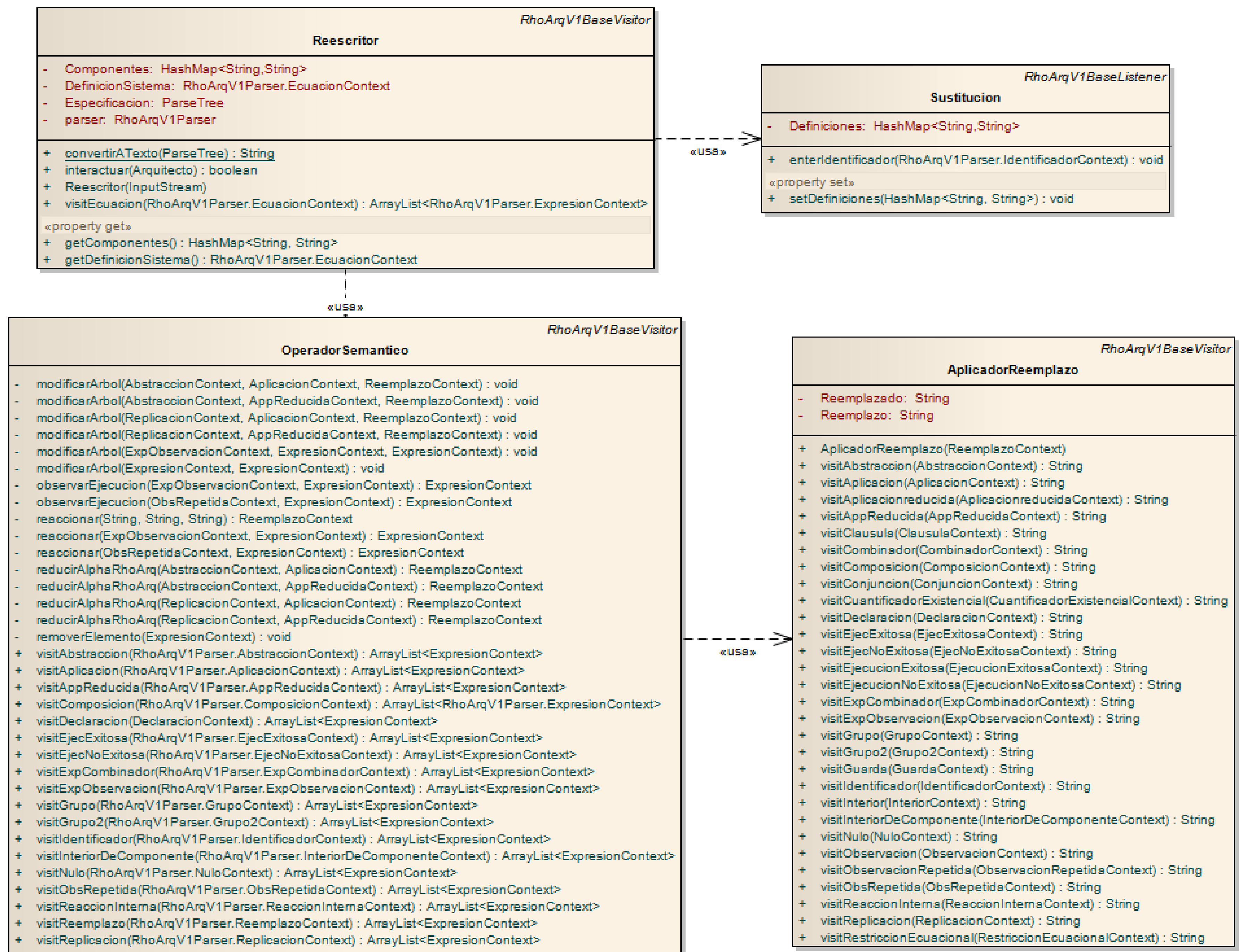

Figura 3.15. Diagrama de clases Reescritor ρ_{arq}.

En el Código 3.12 se puede observar como se construye un árbol de expresión nuevo para la definición del identificador encontrado y como es insertado reemplazando el nodo del identificador en el árbol de expresiones de la ecuación del sistema. El segmento de código corresponde a la operación *enterIdentificador(ctx:IdentificadorContext):void* de la clase *Sustitucion*, operación implementada para hacer uso del patrón observador provisto por ANTLR y recibir las notificaciones de los eventos en los cuales se encuentre en los nodos del árbol una expresión de tipo *IdentificadorContext*. En él se analiza la expresión que define el identificador observado para crear un árbol con dicha definición.

Una vez construido el árbol de expresiones que define la ecuación del sistema sin tener identificadores por reemplazar, es posible iniciar su análisis para aplicar la semántica operacional del cálculo ρ_{arq}, esta labor es realizada por la clase *OperadorSemantico*, la cual es una implementación de un visitador de expresiones. Esta clase es utilizada para visitar y analizar cada uno de los nodos del árbol de expresiones, en especial aquellos que contienen expresiones que cumplen con los patrones de interacción especificados en las reglas de reducción del cálculo presentadas en la Tabla 2.4.

El operador semántico está en capacidad de identificar los patrones de expresiones definidos por las reglas $A_{\rho_{arq}}$ y $Ejec_{\tau}$, en las cuales se especifican interacciones entre componentes y observación de ejecución exitosa respectivamente, modificando el árbol de expresiones de acuerdo a la interacción definida en cada regla. Así, para $A_{\rho_{arq}}$ cuando se ejecutan concurrentemente una aplicación y una abstracción, se generará un reemplazo de la variable enviada por la aplicación sobre la variable dispuesta en la abstracción para recibir un dato. La ejecución concurrente se identifica al tener en una expresión de tipo composición dos expresiones con posibilidad de interacción (*ExpresionesIzquierda* y *expresion* en el Código 3.13), así, el visitador al estar revisando un nodo del árbol de expresiones cuyo tipo es *ComposicionContext* revisa las expresiones que hacen parte de dicha composición para identificar si son candidatas a la aplicación de reglas de reducción.

```
if(ExpresionesIzquierda.get(Indice) instanceof AbstraccionContext && expresion instanceof
AplicacionContext){
   AbstraccionContext Abstraccion = (AbstraccionContext)ExpresionesIzquierda.get(Indice);
   AplicacionContext Aplicacion = (AplicacionContext)expresion;
   Reemplazo = this.reducirAlphaRhoArq(Abstraccion, Aplicacion);
}
if(ExpresionesIzquierda.get(Indice) instanceof AplicacionContext && expresion instanceof
AbstraccionContext){
   AplicacionContext Aplicacion=(AplicacionContext)ExpresionesIzquierda.get(Indice);
   AbstraccionContext Abstraccion = (AbstraccionContext)expresion;
   Reemplazo = this.reducirAlphaRhoArq(Abstraccion, Aplicacion);
}
if (Reemplazo != null){
   ArrayList<RhoArqV1Parser.ExpresionContext> Retorno = new
   ArrayList<RhoArqV1Parser.ExpresionContext>();
   Retorno.add(null);
   Retorno.add((ExpresionContext)Reemplazo);
   Retorno.add(expresion);
   Retorno.add(ExpresionesIzquierda.get(Indice));
   return Retorno;
}
```

Código 3.13. Aplicación de regla $A_{\rho_{arq}}$ para abstracción y aplicación

De tal forma, al identificar una abstracción y una aplicación que posiblemente puedan comunicarse entre sí, se procede a la creación de la expresión Reemplazo, en la cual la variable enviada reemplaza la variable de recepción en la abstracción si el canal de comunicación es el mismo.

```
String CanalRecibo = Abstraccion.ID(0).toString();
String CanalEnvio = Aplicacion.aplicacionreducida().ID(0).toString();
ReemplazoContext Reemplazo = null;
if(CanalRecibo.equals(CanalEnvio)){
     Reemplazo = this.reaccionar(Aplicacion.aplicacionreducida().ID(1).toString(),
     Abstraccion.ID(1).toString(), Abstraccion.expresion().getText());
     this.modificarArbol(Abstraccion, Aplicacion, Reemplazo);
}
```

Código 3.14. Identificación de interacción entre Abstracción y Aplicación

```
ReemplazoContext ExpresionReemplazo = null;
ReemplazoBuilder Builder = new ReemplazoBuilder(Reemplazo, Reemplazado, Expresion);
try {
   ExpresionReemplazo = (ReemplazoContext)Builder.construirExpresion();
} catch (IOException e) {
   System.out.println("ERR04: Ocurrió un error al generar reemplazo para una interacción");
}
```

Código 3.15. Creación de expresión Reemplazo.

La regla $A_{\rho_{arq}}$ es igualmente aplicable para las interacciones Abstracción - Aplicación reducida, Replicación - Aplicación y Replicación - Aplicación reducida, considerando para las interacciones en las cuales participa una replicación, que dicha expresión debe permanecer en el árbol de expresiones, ya que por su naturaleza se debe mantener la recepción de información a través del canal provisto de manera permanente.

La identificación del patrón de expresiones para la regla $Ejec_{\tau}$ se realiza a partir de la ejecución concurrente de una expresión observación y una ejecución exitosa de un componente, en donde el componente observado y el componente con ejecución exitosa hacen referencia al mismo componente. Dadas estas condiciones, la revisión del patrón de interacción inicia también con la revisión de una composición donde las expresiones que hacen parte de ella son una observación y una ejecución exitosa.

```
if(expresion instanceof ExpObservacionContext || expresion instanceof EjecExitosaContext){
     ExpresionContext Reemplazo = null;
     for(int Indice=IndiceExpresion+1;Indice<ExpresionesIzquierda.size();Indice++){
        if(ExpresionesIzquierda.get(Indice) instanceof EjecExitosaContext && expresion
           instanceof ExpObservacionContext){
           EjecExitosaContext EjecucionExitosa =
           (EjecExitosaContext)ExpresionesIzquierda.get(Indice);
           ExpObservacionContext Observacion = (ExpObservacionContext)expresion;
           Reemplazo = this.observarEjecucion(Observacion, EjecucionExitosa);
        }
        if(ExpresionesIzquierda.get(Indice) instanceof ExpObservacionContext && expresion
           instanceof EjecExitosaContext){
           ExpObservacionContext Observacion =
           (ExpObservacionContext)ExpresionesIzquierda.get(Indice);
           EjecExitosaContext EjecucionExitosa = (EjecExitosaContext)expresion;
           Reemplazo = this.observarEjecucion(Observacion, EjecucionExitosa);
        }
        if (Reemplazo != null){
           ArrayList<RhoArqV1Parser.ExpresionContext> Retorno = new
           ArrayList<RhoArqV1Parser.ExpresionContext>();
           Retorno.add(null);
           Retorno.add(Reemplazo);
           Retorno.add(expresion);
           Retorno.add(ExpresionesIzquierda.get(Indice));
           return Retorno;
```

```
        }
     }
}
```

Código 3.16. Aplicación de regla $Ejec_{\tau}$ para observación y ejecución exitosa.

Una vez se identifica el patrón de expresiones, se procede a especificar si hacen referencia al mismo componente y a verificar el tipo de interacción detectada, ya que puede ser por una ejecución exitosa o una ejecución no exitosa, esto se realiza a través de la operación *observarEjecucion(Observacion: ExpObservacionContext, Ejecucion: ExpresionContext): ExpresionContext*.

```
String ComponenteEnObservacion = Observacion.observacion().ID().getText();
String ComponenteEjecutado = "";
if(Ejecucion instanceof EjecExitosaContext)
        ComponenteEjecutado =
        ((EjecExitosaContext)Ejecucion).ejecucionExitosa().ID().getText();

        ...

ExpresionContext Expresion = null;
if(ComponenteEnObservacion.equals(ComponenteEjecutado)){
   Expresion = this.reaccionar(Observacion, Ejecucion);
   this.modificarArbol(Observacion, Ejecucion, Expresion);
}
```

Código 3.17. Definición del resultado de la observación de ejecución exitosa.

```
String ComponenteEnObservacion = Observacion.observacion().ID().getText();
String ComponenteEjecutado = "";

…

if(Ejecucion instanceof EjecNoExitosaContext)
        ComponenteEjecutado =
        ((EjecNoExitosaContext)Ejecucion).ejecucionNoExitosa().ID().getText();
ExpresionContext Expresion = null;
if(ComponenteEnObservacion.equals(ComponenteEjecutado)){
   Expresion = this.reaccionar(Observacion, Ejecucion);
   this.modificarArbol(Observacion, Ejecucion, Expresion);
}
```

Código 3.18. Definición del resultado de una observación ejecución no exitosa.

Finalmente, se modifica el árbol reemplazando la ejecución exitosa o no exitosa por lo indicado en la observación.

Este mismo comportamiento es realizado para la expresión observación repetida, dejando en el árbol de expresiones dicha observación pues su naturaleza indica una observación permanente durante la ejecución de la arquitectura.

Como resultado de la ejecución de la regla $A_{\rho_{arq}}$ se incluye una expresión de tipo reemplazo en la ecuación del sistema, la cual también debe ser reescrita. Para realizar esta labor se implementa la clase *AplicadorReemplazo*, encargada de reescribir las expresiones requeridas realizando el reemplazo de variables producto del paso de información desde una interfaz de provisión hacia una interfaz de requerimiento. Esta clase realiza el reemplazo de todos los identificadores cuyo nombre es igual al identificador a ser reemplazado con el identificador de reemplazo. El reemplazo es realizado mediante la implementación de métodos de un visitador de expresiones, en donde para cada identificador utilizado en las expresiones se verifica si debe ser reemplazado.

La ejecución de esta labor de reemplazo, se realiza desde el operador semántico a través de la operación *visitReemplazo(ctx: ReemplazoContext): ArrayList<ExpresionContext>*, allí se sabe cuales son los identificadores involucrados en el reemplazo y la expresión en la cual se llevará a cabo el reemplazo, así, desde allí es en donde se realiza el proceso de recorrer el árbol de la expresión sujeta al reemplazo y el cambio de identificadores es realizado.

```
AplicadorReemplazo Aplicador = new AplicadorReemplazo(ctx);
if(ctx.expresion() instanceof InteriorDeComponenteContext)
{
    Arreglo.add(ctx.expresion());
    return Arreglo;
}
String NuevaExpresion = Aplicador.visit(ctx.expresion());
```

Código 3.19. Aplicación de reemplazo de identificadores.

3.4.2.3 Modelado dinámico

La implementación de los casos de uso de este módulo fue realizada como lo muestran los siguientes diagramas de secuencia:

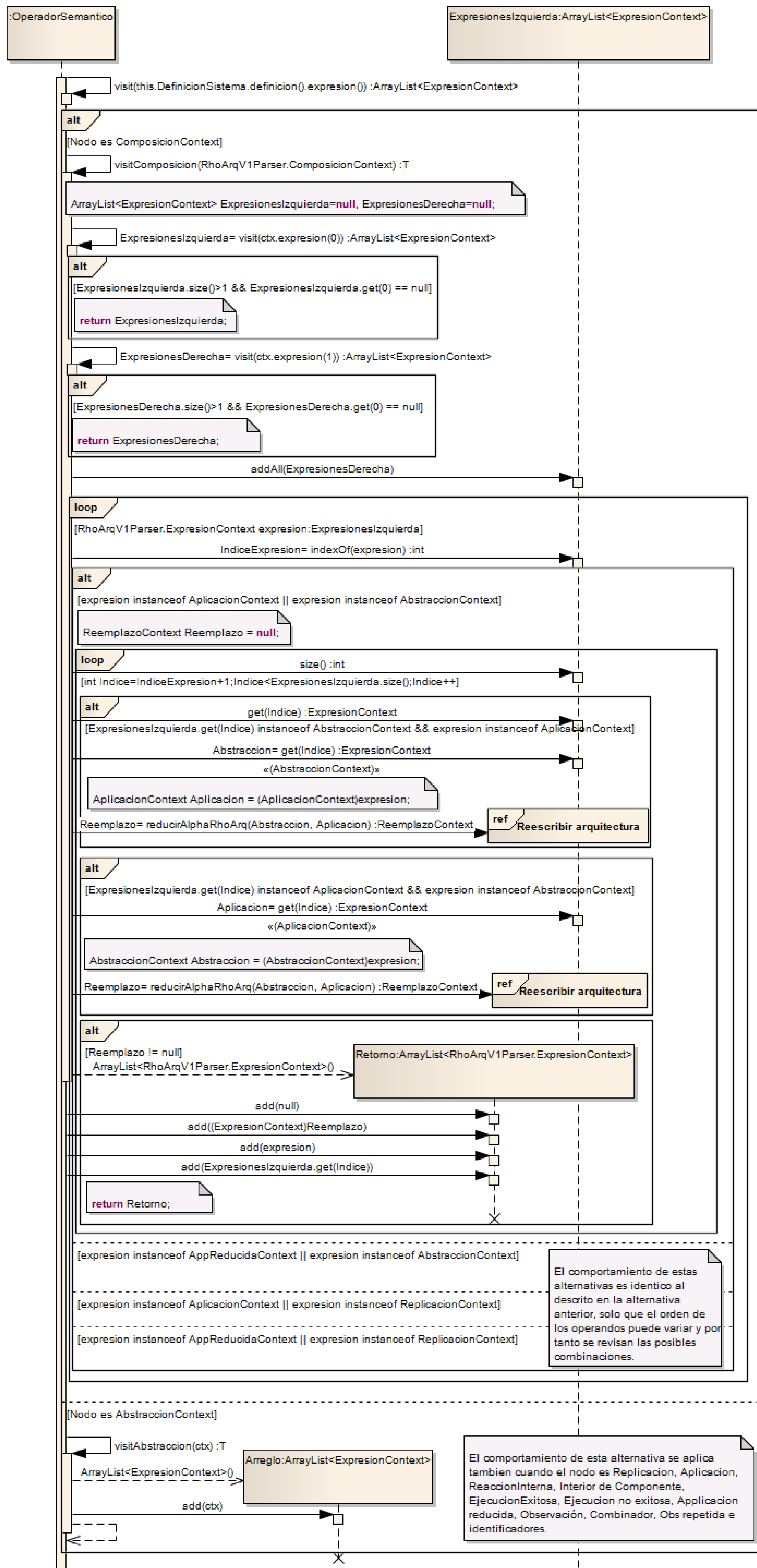

Figura 3.16. Modelo dinámico caso de uso "Identificar interacciones".

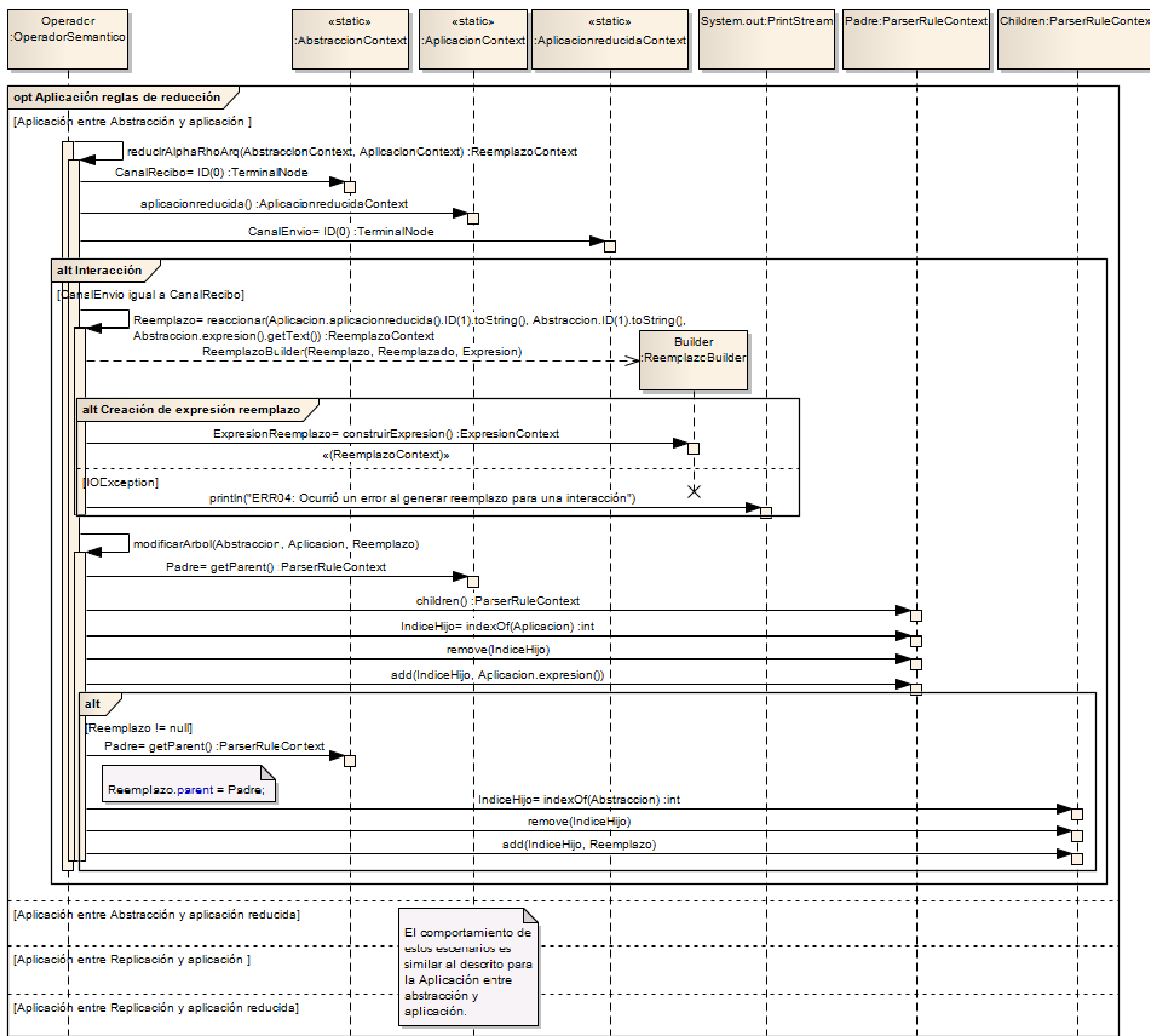

Figura 3.17. Modelo dinámico caso de uso “Reescribir arquitectura” para el caso de aplicaciones.

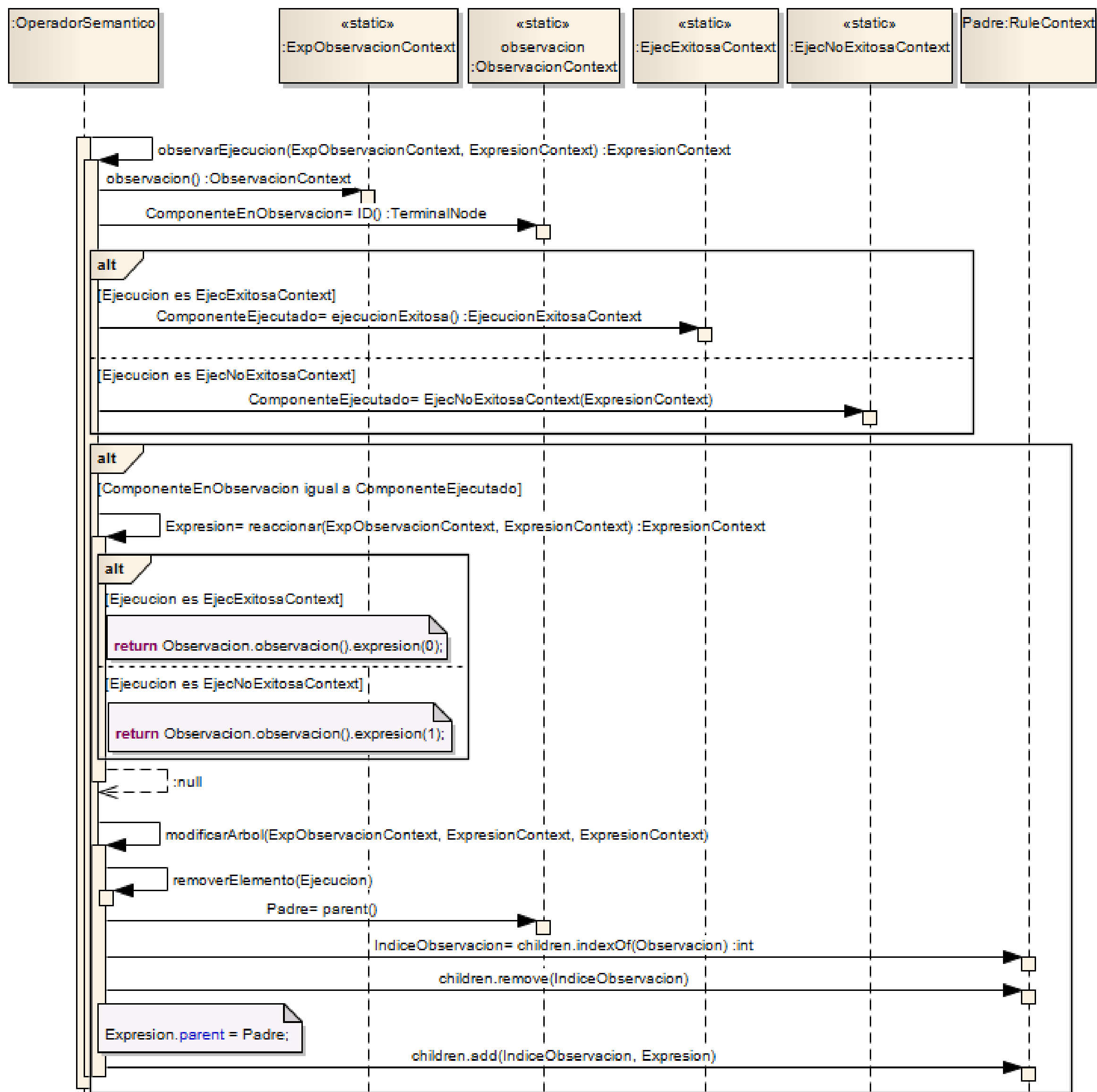

Figura 3.18. Modelo dinámico caso de uso “Reescribir arquitectura” para el caso de observaciones.

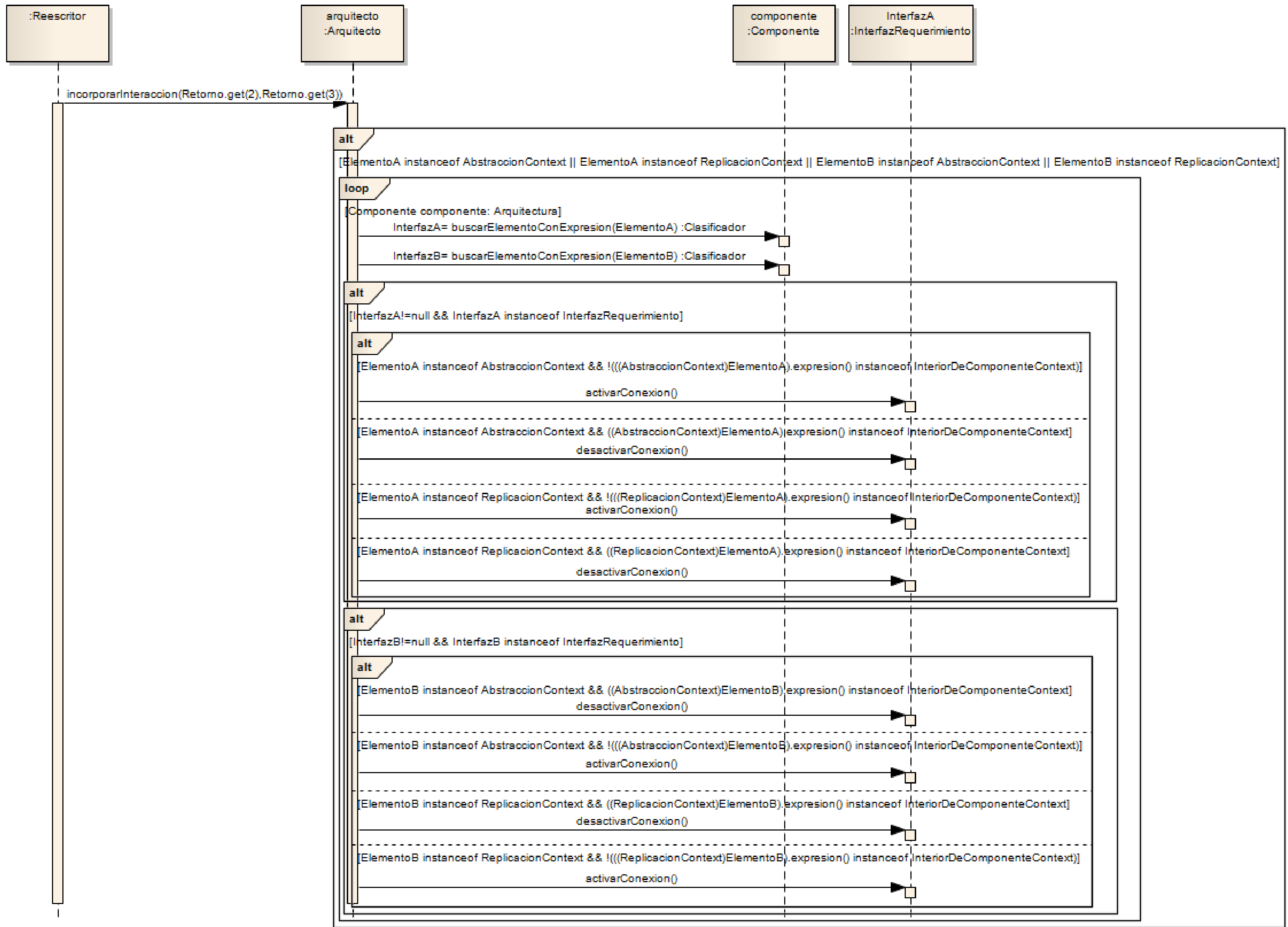

Figura 3.19.Modelo dinámico caso de uso “Modificar estados de la arquitectura”.

3.4.3 Arquitecto

Este módulo es el encargado de identificar la configuración de la arquitectura descrita con cálculo ρ_{arq} y mantener los estados de dicha configuración durante su flujo de ejecución. Para esto, se planteó un modelo de clases que permitiera reflejar la configuración de dichas arquitecturas, en dicho modelo, se plantea una clase denominada clasificador, la cual representará a cada clasificador empleado para describir arquitecturas, en este caso, componentes, interfaces de provisión e interfaces de requerimiento.

3.4.3.1 Casos de uso implementados

Para este módulo se pensó en un caso de uso el cual se encarga de identificar la configuración de la arquitectura descrita con cálculo ρ_{arq} a partir de la búsqueda de patrones en las expresiones empleadas en la definición. El caso de uso fue denominado “Identificar patrones de definición de elementos” y en él, es representado utilizando el modelo estructural de clasificadores la arquitectura que será analizada.

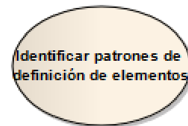

Figura 3.20. Caso de uso implementado en el módulo Arquitecto.

3.4.3.2 Modelado estructural

Para la implementación de las funcionalidades del Arquitecto, fue necesario definir una estructura de clases que permitiera representar en memoria los elementos estructurales de una arquitectura (Componentes e interfaces), para ello se definieron en un patrón *composite* las estructuras que definen una arquitectura agrupadas bajo la clase abstracta `Clasificador` como lo muestra la Figura 3.21 .

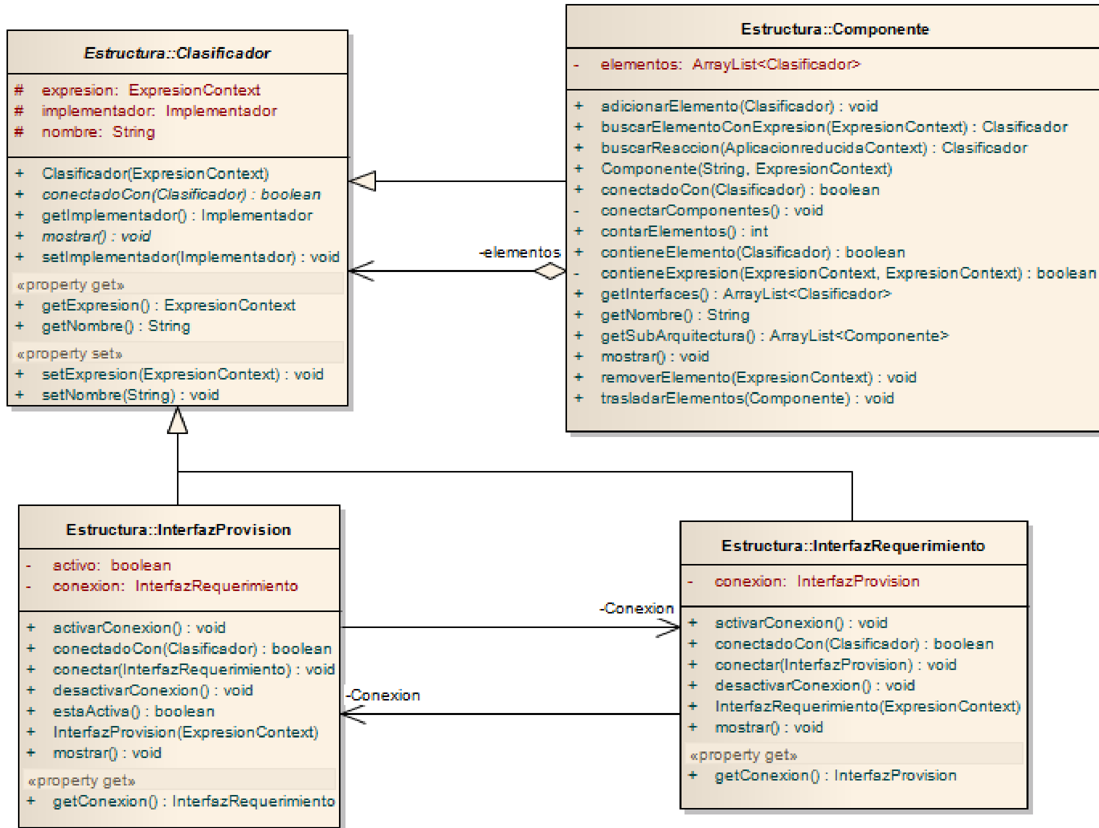

Figura 3.21. Diagrama de clases para la configuración de arquitecturas.

Adicionalmente, para construir los objetos que representan la configuración de la arquitectura descrita con cálculo ρ_{arq} se construyó la clase Arquitecto, encargada de identificar los patrones de expresiones que se emplean para describir interfaces y componentes.

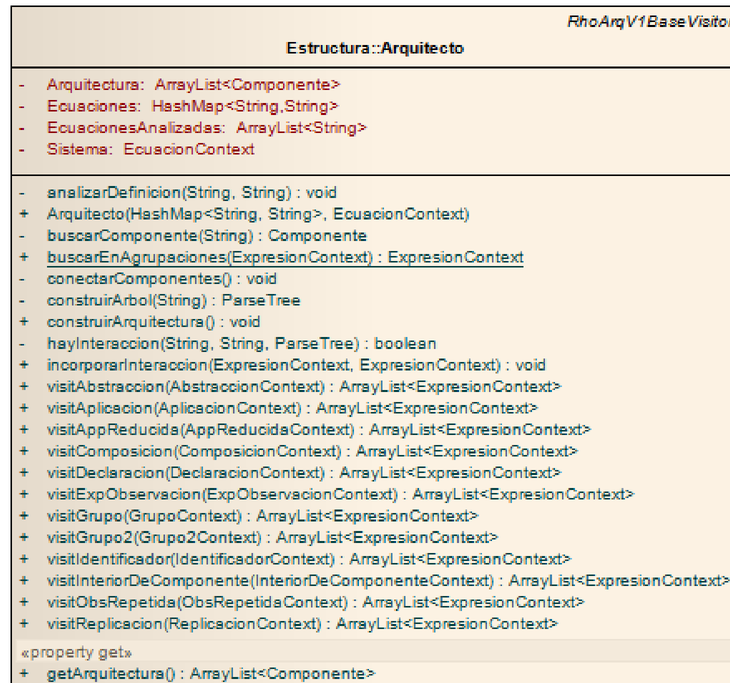

Figura 3.22. Clase Arquitecto

Esta clase analiza las ecuaciones empleadas para la descripción de una arquitectura en busca de todos los clasificadores que describen la configuración, una vez identificados, procede a establecer cuales de ellos se encuentran conectados o componen la estructura de otros clasificadores (composición jerárquica).

```
Set<String> Ecuaciones = this.Ecuaciones.keySet();
this.EcuacionesAnalizadas = new ArrayList<String>();
for(String Elemento:Ecuaciones)
     if(!Elemento.equals(Sistema.definicion().ID().getText()))
        analizarDefinicion(Elemento, this.Ecuaciones.get(Elemento));
this.conectarComponentes();
```

Código 3.20. Identificación de la configuración arquitectural.

El arquitecto implementa algunos métodos de un visitador de expresiones con el objetivo de identificar los patrones que definen los elementos de una arquitectura en cada ecuación descrita, así:

- Las variables que tienen una definición serán consideradas como candidatas a ser componentes.
- Todas las expresiones de tipo `AbstraccionContext` o `ReplicacionContext` serán consideradas como interfaces de provisión.
- Las expresiones de tipo `DeclaracionContext` que cumplan con los patrones de envío de una ubicación y de un servicio (que contengan una composición de abtracciones) serán consideradas como interfaces de requerimiento.

```
ParseTree Arbol = this.construirArbol(Definicion);
ArrayList<ExpresionContext> Elementos = this.visit(Arbol);
ArrayList<String>Identificadores = new ArrayList<String>();
Componente componente = new Componente(Llave, (ExpresionContext)Arbol);
for(ExpresionContext Elemento:Elementos){
     Clasificador clasificador=null;
```

```
    if(Elemento instanceof DeclaracionContext)
       clasificador = new InterfazRequerimiento(Elemento);
    if(Elemento instanceof ReplicacionContext)
       clasificador = new InterfazProvision(Elemento);
    if(Elemento instanceof IdentificadorContext)
       Identificadores.add(Elemento.getText());
    if(clasificador != null)
       componente.adicionarElemento(clasificador);
}
```

Código 3.21. Identificación de elementos de la arquitectura

Una vez establecidos los clasificadores empleados en la arquitectura, el arquitecto procede a identificar las conexiones existentes entre ellos, esto a partir de las expresiones de tipo aplicación reducida y de las interfaces identificadas para cada componente.

```
for(Componente A:Arquitectura){
    for(AppReducidaContext Conector:Conectores)    {
       Clasificador InterfazA = A.buscarReaccion(Conector.aplicacionreducida());
       if(InterfazA != null){
          Iterator<Componente> Indice = Arquitectura.iterator();
          while(Indice.hasNext()&&Indice.next()!=A){}
          while(Indice.hasNext()){
              Componente B = Indice.next();
              Clasificador InterfazB = B.buscarReaccion(Conector.aplicacionreducida());
              if(InterfazB != null){
                 if(InterfazA instanceof InterfazProvision && InterfazB instanceof
                 InterfazRequerimiento){
       ((InterfazProvision)InterfazA).conectar((InterfazRequerimiento)InterfazB);
       ((InterfazRequerimiento)InterfazB).conectar((InterfazProvision)InterfazA);
                 }
                 if(InterfazB instanceof InterfazProvision && InterfazA instanceof
                 InterfazRequerimiento){
       ((InterfazProvision)InterfazB).conectar((InterfazRequerimiento)InterfazA);
       ((InterfazRequerimiento)InterfazA).conectar((InterfazProvision)InterfazB);
                 }
              }
          }
       }
    }
}
```

Código 3.22. Identificación de conexiones

Con todos los elementos identificados, el arquitecto crea un listado de componentes en memoria que definen la arquitectura descrita en el cálculo ρ_{arq}, donde cada componente tiene definidas sus interfaces y cada una de ellas se encuentra conectada de acuerdo a la descripción de dicha arquitectura.

3.4.3.3 Modelado dinámico

La implementación del caso de uso definido para este módulo se muestra en el siguiente diagrama de secuencia.

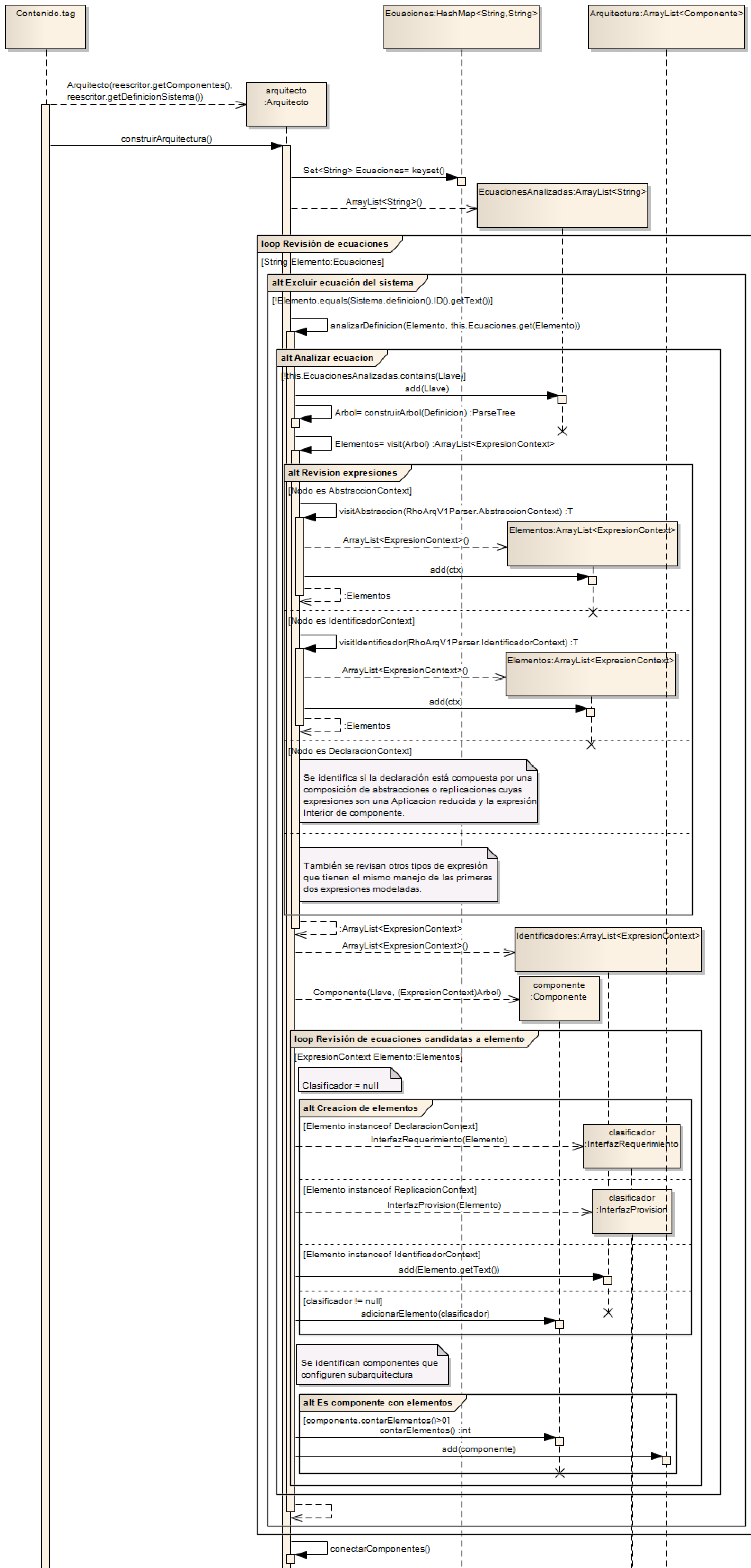

Figura 3.23. Modelo dinámico caso de uso “Identificar patrones de definición de elementos”.

3.4.4 Transformador

El módulo transformador es el encargado de escribir en el formato XML para intercambio de modelos, la configuración de la arquitectura analizada.

3.4.4.1 Casos de uso implementados

Para este módulo fueron implementados tres casos de uso que en conjunto se encargan de representar la estructura de la arquitectura definida por el modulo Arquitecto en el formato XMI. Estos casos de uso realizan la transformación de cada elemento a XMI (Transformar arquitectura), escritura del archivo XML (Generar archivo plano) y entregar dicha representación XMI al usuario (Exportar arquitectura).

Figura 3.24. Casos de uso implementados en el módulo Transformador.

3.4.4.2 Modelado estructural

Para la construcción de este módulo, fue implementado el patrón puente (*bridge*) para extender la funcionalidad de los clasificadores que describen la configuración de una arquitectura, como lo muestra el siguiente modelo:

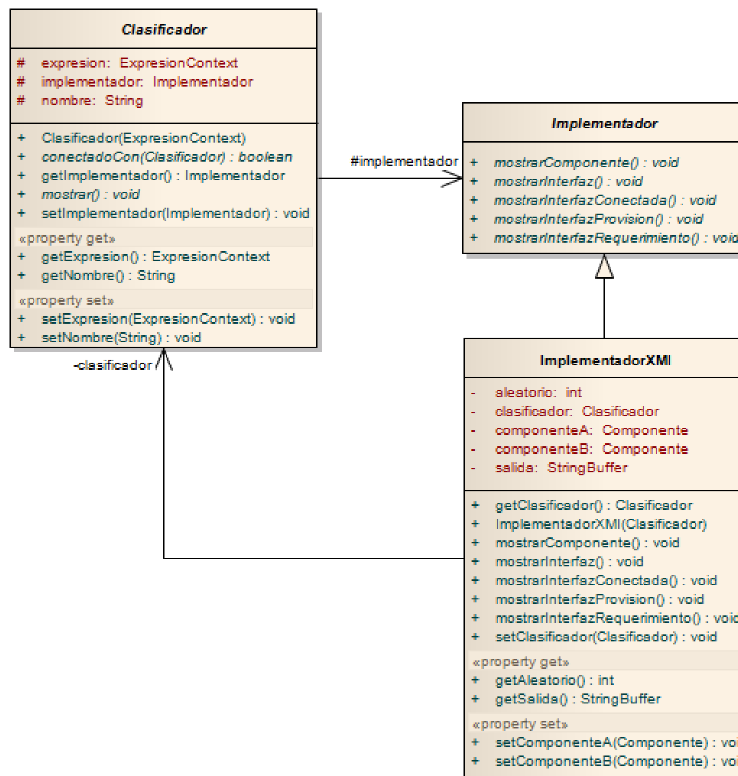

Figura 3.25. Patrón puente para transformación de configuración a XMI.

De esta forma, cada elemento identificado en la arquitectura tendrá la capacidad para ser representado usando XML de acuerdo al tipo de elemento arquitectural. Cada clasificador tendrá un implementador asociado para generar su representación ya sea en XMI o SVG, para este caso ya que se habla del módulo de transformación se explicará el implementador para XMI.

La clase `ImplementadorXMI` es la encargada de traducir en términos de XML la estructura de cada uno de los clasificadores, así, realiza la implementación de las operaciones para representar usando XML, según sea un componente, una interfaz de requerimiento, una interfaz de provisión o una interfaz conectada.

```
public void mostrarComponente() {
        if(this.clasificador instanceof Componente)
        {
                Componente componente = (Componente)this.clasificador;
                if(componente.contarElementos()==0)
                {
                        salida.append("<ownedMember xmi:type=\"uml:Component\"
xmi:id=\"PINTARQID_comp_"+this.clasificador.getNombre().hashCode()+"\"
name=\""+this.clasificador.getNombre()+"\" visibility=\"public\"/>"+"\n");
                        salida.append("<ownedMember xmi:type=\"uml:Class\"
xmi:id=\"PINTARQID_CLASS_"+this.clasificador.getNombre().hashCode()+"\"
name=\"$help://componentdiagram.htm\" visibility=\"public\"/>"+"\n");
                }
                else
                {
                        salida.append("<ownedMember xmi:type=\"uml:Component\"
xmi:id=\"PINTARQID_comp_"+this.clasificador.getNombre().hashCode()+"\"
name=\""+this.clasificador.getNombre()+"\" visibility=\"public\">"+"\n");
                        for(Clasificador C: componente.getInterfaces())
                        {
                                if(C.getImplementador() != null && C.getImplementador()
```

```
instanceof ImplementadorXMI)
                                        {
                                                C.getImplementador().mostrarInterfaz();
                                salida.append(((ImplementadorXMI)C.getImplementador()).getSalida());
                                        }
                                }
                                salida.append("</ownedMember>"+"\n");
                        }
                }
}
```

Código 3.23. Operación mostrarComponente():void de ImplementadorXMI.

La operación `mostrarComponente():void` es el punto de inicio para la generación del código XML de la arquitectura, ya que el componente es el elemento que genera las etiquetas contenedoras de la configuración de sus interfaces. Esto permite que sea posible recorrer la arquitectura (como se muestra en el Código 3.24) y para cada componente se pueda invocar esta operación, asegurando que todos los elementos que hacen parte del componente sean representados en XML.

```
public StringBuffer exportarArquitectura()
{
        StringBuffer SalidaXMI = new StringBuffer();
        for(Componente c:Arquitectura)
        {
                c.getImplementador().mostrarComponente();
                SalidaXMI.append(((ImplementadorXMI)c.getImplementador()).getSalida());
        }
        return SalidaXMI;
}
```

Código 3.24. Operación exportarArquitectura():StringBuffer de la clase Exportador.

La operación `mostrarComponente():void` revisa los elementos que hacen parte del componente y luego para cada una de las interfaces invoca la operación del `ImplementadorXMI mostrarInterfaz():void`.

```
public void mostrarInterfaz(){
        if(this.clasificador instanceof InterfazProvision)
                if(((InterfazProvision)this.clasificador).getConexion()==null)
                        this.mostrarInterfazProvision();
        if(this.clasificador instanceof InterfazRequerimiento)
                if(((InterfazRequerimiento)this.clasificador).getConexion()!=null)
                {
                        if(componenteA != null && componenteB != null)
                                this.mostrarInterfazConectada();
                }
                else
                        this.mostrarInterfazRequerimiento();
}
```

Código 3.25. Operación mostrarInterfaz():void de la clase ImplementadorXMI.

En esta operación, se identifica el tipo de Interfaz que será desplegada y adicionalmente se identifica si la interfaz está conectada, esto último solo para las interfaces de requerimiento, dadas las reglas sintácticas definidas para la representación XMI. Dependiendo de dichas características se procede a invocar las operaciones `mostrarInterfazProvision():void`, `mostrarInterfazRequerimiento():void` o `mostrarInterfazConectada():void`.

```
public void mostrarInterfazProvision() {
        if(this.clasificador.getNombre()!=null)
                salida.append("<provided
xmi:id=\"PINTARQID_prov_"+this.clasificador.getNombre().hashCode()+"\"
```

```
name=\""+this.clasificador.getNombre()+"\"/>\n");
        else
        {
                this.aleatorio = (int)Math.random()*100000;
                salida.append("<provided
xmi:id=\"PINTARQID_prov_"+(String.valueOf(aleatorio)).hashCode()+"\" name=\"\"/>\n");
        }
}
```

Código 3.26. Operación mostrarInterfazProvision():void de la clase ImplementadorXMI.

```
public void mostrarInterfazRequerimiento() {
        if(this.clasificador.getNombre()!=null)
                salida.append("<required
xmi:id=\"PINTARQID_req_"+this.clasificador.getNombre().hashCode()+"\"
name=\""+this.clasificador.getNombre()+"\"/>\n");
        else
        {
                this.aleatorio = (int)Math.random()*100000;
                salida.append("<required
xmi:id=\"PINTARQID_req_"+(String.valueOf(aleatorio)).hashCode()+"\" name=\"\"/>\n");
        }
}
```

Código 3.27. Operación mostrarInterfazRequerimiento():void de la clase ImplementadorXMI.

```
public void mostrarInterfazConectada() {
        if(componenteA != null && componenteB != null)
        {
                if(this.clasificador.getNombre()!=null)
                {
                        salida.append("<ownedConnector xmi:type=\"uml:Connector\"
xmi:id=\"PINTARQID_CONEXION_"+this.clasificador.getNombre().hashCode()+"\"
name=\""+this.clasificador.getNombre()+"\" visibility=\"public\" kind=\"assembly\">\n");
                        salida.append("\t<end xmi:type=\"uml:ConnectorEnd\"
xmi:id=\"PINTARQID_CONEXION_"+this.clasificador.getNombre().hashCode()+"\"
role=\"PINTARQID_comp_"+this.componenteA.getNombre().hashCode()+"\"/>\n");
                        salida.append("\t<end xmi:type=\"uml:ConnectorEnd\"
xmi:id=\"PINTARQID_CONEXION_"+this.clasificador.getNombre().hashCode()+"\"
role=\"PINTARQID_comp_"+this.componenteB.getNombre().hashCode()+"\"/>\n");
                        salida.append("</ownedConnector>\n");
                }
                else
                {
                        aleatorio = (int)Math.random()*100000;
                        String Nombre = (String.valueOf(aleatorio));
                        salida.append("<ownedConnector xmi:type=\"uml:Connector\"
xmi:id=\"PINTARQID_CONEXION_"+Nombre.hashCode()+"\"
name=\""+this.clasificador.getNombre()+"\" visibility=\"public\" kind=\"assembly\">\n");
                        salida.append("\t<end xmi:type=\"uml:ConnectorEnd\"
xmi:id=\"PINTARQID_CONEXION_"+Nombre.hashCode()+"\"
role=\"PINTARQID_comp_"+this.componenteA.getNombre().hashCode()+"\"/>\n");
                        salida.append("\t<end xmi:type=\"uml:ConnectorEnd\"
xmi:id=\"PINTARQID_CONEXION_"+Nombre.hashCode()+"\"
role=\"PINTARQID_comp_"+this.componenteB.getNombre().hashCode()+"\"/>\n");
                        salida.append("</ownedConnector>\n");
                }
        }
}
```

Código 3.28. Operación mostrarInterfazConectada():void de la clase ImplementadorXMI.

Para obtener la definición XMI del modelo arquitectural completo se implementó la clase `Exportador`, la cual como se mostró con el Código 3.24 recorre cada componente y agrupa todas las representaciones XMI en un único documento XML.

```
public StringBuffer exportarArchivo()
{
```

```
        this.asignarImplementadores();
        StringBuffer SalidaArquitectura = this.exportarArquitectura();
        StringBuffer SalidaXMI = new StringBuffer();
        SalidaXMI.append("<?xml version=\"1.0\" encoding=\"windows-1252\"?>\n");
        SalidaXMI.append("<xmi:XMI xmi:version=\"2.1\"
xmlns:uml=\"http://schema.omg.org/spec/UML/2.0\"
xmlns:xmi=\"http://schema.omg.org/spec/XMI/2.1\">\n");
        SalidaXMI.append("\t<xmi:Documentation exporter=\"PintArq\"
exporterVersion=\"1.0\"/>\n");
        SalidaXMI.append("\t<uml:Model xmi:type=\"uml:Model\"
name=\"Modelo_"+this.NombreArchivo+"\" visibility=\"public\">\n");
        SalidaXMI.append("\t\t<ownedMember xmi:type=\"uml:Package\"
xmi:id=\"PintArqPK_"+this.NombreArchivo.hashCode()+"\" name=\"Arquitectura\"
visibility=\"public\">\n");
        SalidaXMI.append(SalidaArquitectura);
        SalidaXMI.append("\t\t</ownedMember>");
        SalidaXMI.append("\t</uml:Model>");
        SalidaXMI.append("</xmi:XMI>");
        return SalidaXMI;
}
```

Código 3.29. Operación exportarArchivo():StringBuffer de la clase Exportador.

La operación `exportarArchivo():StringBuffer` construye el documento XML con las etiquetas de encabezado y de raíz del documento y luego inserta la representación XML de cada elemento de la Arquitectura. Esta clase esta asociada con un servlet denominado `ExportadorServlet`, que se encarga de recibir las peticiones de exportación a XMI desde la interfaz de usuario y entregar en forma de descarga el documento XML generado por el exportador.

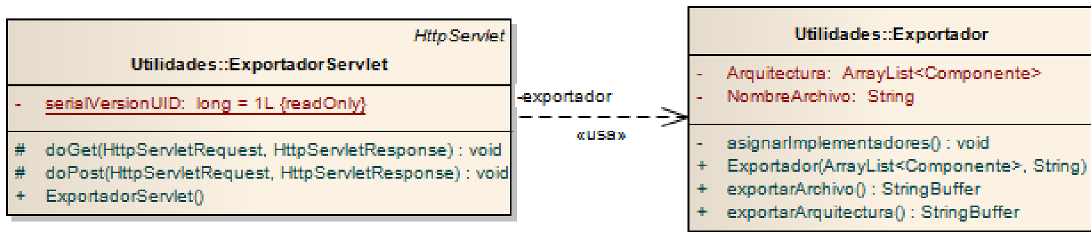

Figura 3.26. Modelo de clases para Exportación XMI de una arquitectura.

3.4.4.3 Modelado dinámico

Cada uno de los fragmentos de código expuestos están asociados a la implementación de los casos de uso del módulo, a continuación en los diagramas de secuencia presentados se podrá observar el contexto general del comportamiento del módulo y de los momentos en los cuales se ejecuta el código presentado.

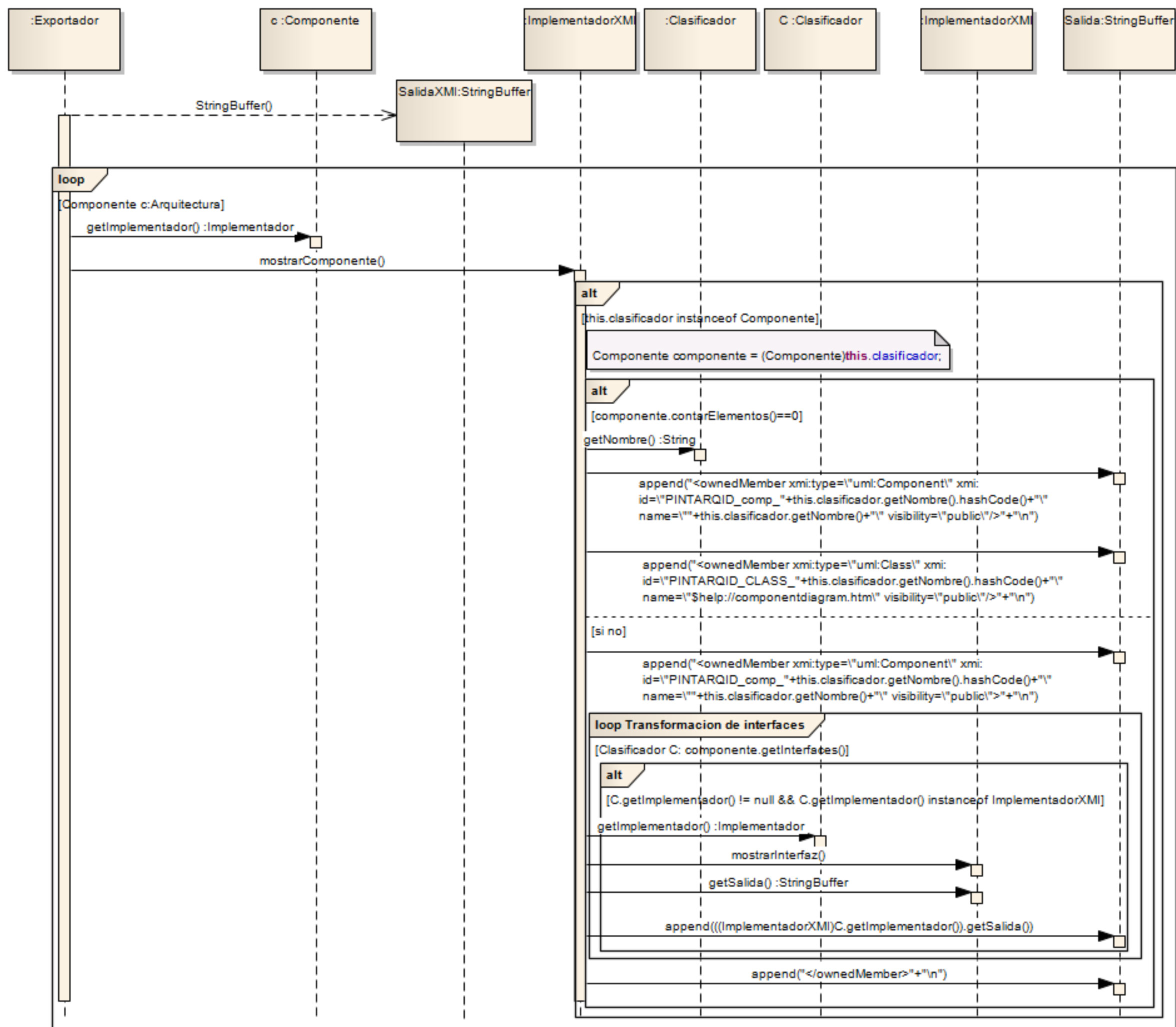

Figura 3.27. Modelo dinámico caso de uso "Transformar arquitectura".

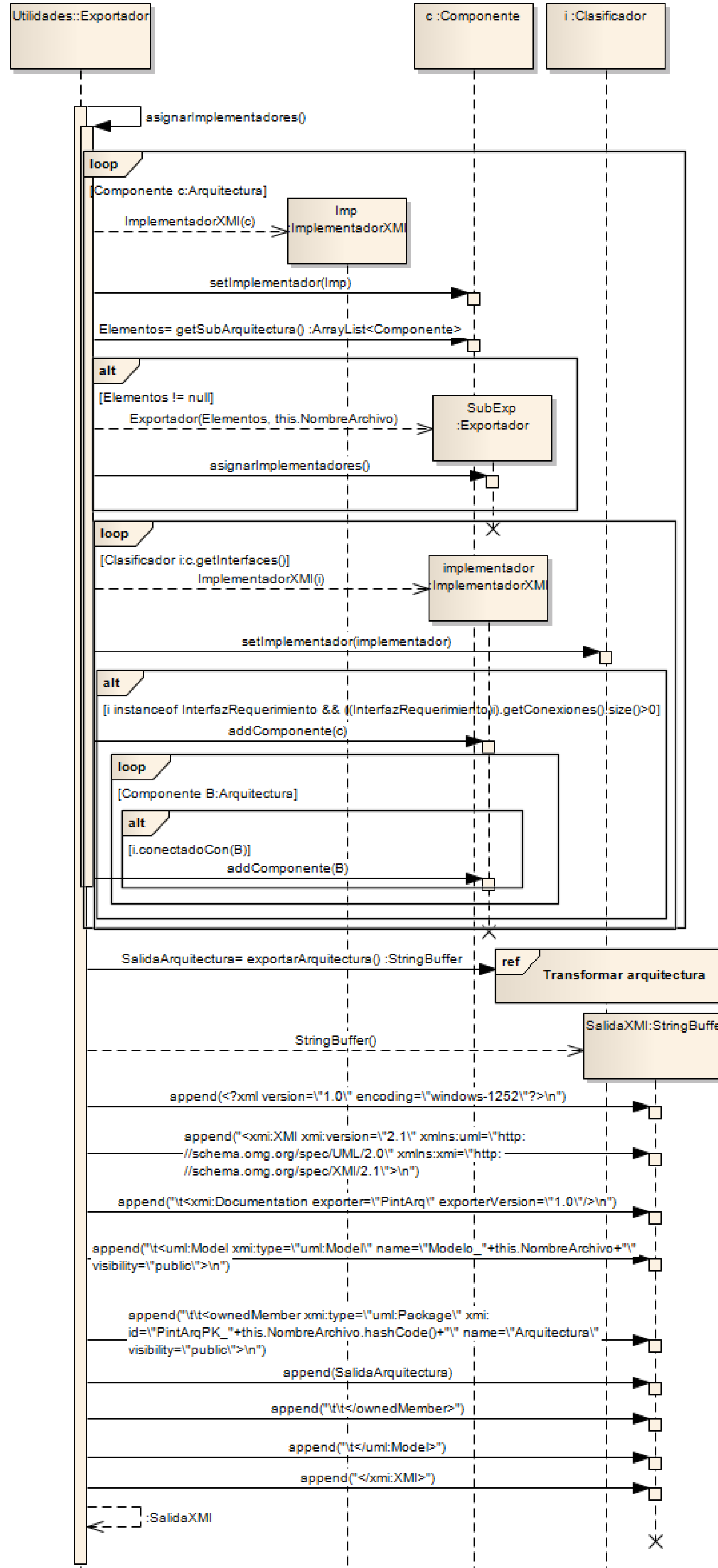

Figura 3.28. Modelo dinámico caso de uso “Generar archivo plano”.

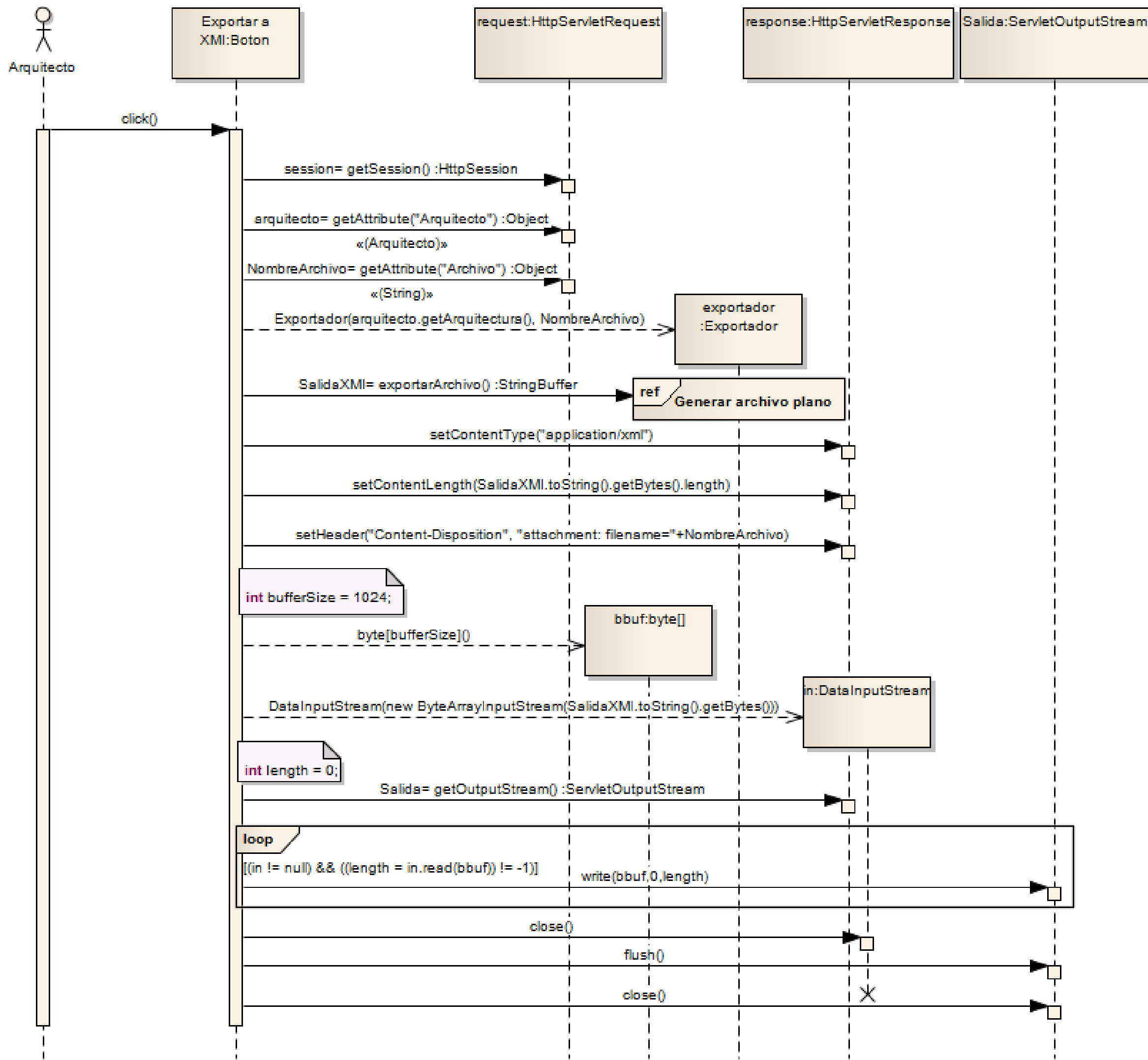

Figura 3.29. Modelo dinámico caso de uso "Exportar arquitectura".

3.4.5 Graficador

En este módulo es donde se lleva a cabo la labor de despliegue en pantalla de la configuración de la arquitectura. Aquí fueron implementados los algoritmos para organizar los elementos de la arquitectura y reflejar usando la notación gráfica de UML 2.x el flujo de ejecución de una arquitectura.

3.4.5.1 Casos de uso implementados

En este módulo fue implementado el caso de uso "Graficar arquitectura", encargado de realizar el despliegue gráfico de la configuración de la arquitectura analizada utilizando la tecnología SVG (Scalable Vector Graphics) y a partir de la configuración definida por el módulo Arquitecto. Adicionalmente, fueron implementados los casos "Ubicar componentes" y "Ubicar conexiones" que están encargados de realizar la ubicación de cada elemento de configuración en el área de despliegue de la aplicación.

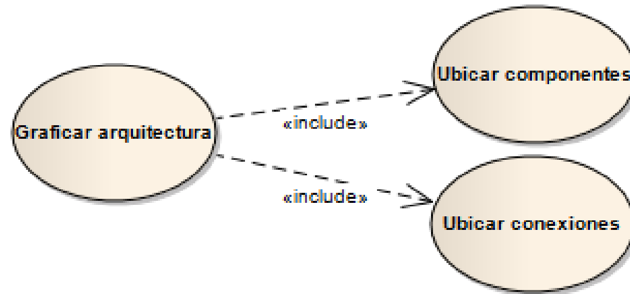

Figura 3.30. Casos de uso implementados en el módulo Graficador.

3.4.5.2 Modelado estructural

Para lograr el objetivo del despliegue gráfico, fueron implementadas un conjunto de clases usando el patrón de diseño puente (bridge) con el objetivo de extender la funcionalidad de los clasificadores e incluir operaciones que permitieran la generación de los elementos gráficos en pantalla.

Figura 3.31. Patrón puente para despliegue gráfico de una arquitectura.

Para el despliegue de los elementos gráficos se ha utilizado la tecnología SVG (Scalable Vector Graphics), en donde los elementos gráficos pueden ser descritos mediante XML y son desplegados en pantalla debido al soporte de HTML5 de los navegadores actuales. La clase `ImplementadorSVG` es la encargada de generar el código SVG necesario para el despliegue de cada elemento gráfico asociado a un clasificador que compone la arquitectura, adicionalmente almacena las coordenadas en las cuales el clasificador será desplegado.

Dado que el implementador es quien genera el código de visualización, es necesario organizar en la pantalla cada uno de los elementos que componen la arquitectura, y que de acuerdo a dicha organización sean allí desplegados. Para esto se creo la clase `Graficador`, en donde fueron implementados los algoritmos necesarios para la organización y ubicación de cada elemento gráfico que describe a la arquitectura analizada.

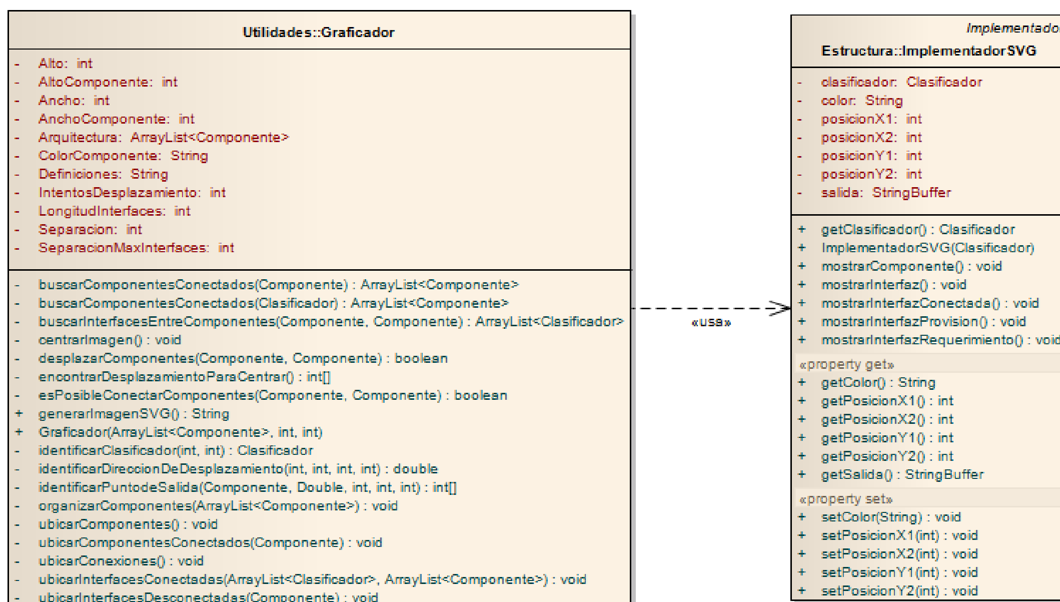

Figura 3.32. Clase graficador.

3.4.5.3 Modelado dinámico

El algoritmo propuesto por el autor para la ubicación de los elementos a partir de los componentes identificados por el módulo Arquitecto se presenta en el diagrama de actividades de la Figura 3.34 y consta a nivel general de los siguientes pasos :

1. Organizar los componentes por cantidad de interfaces de mayor a menor.
2. Para cada componente en la lista, se asigna un ImplementadorSVG (si no tiene uno ya) que permita la generación del código SVG para su representación.
3. Se ubica el componente en el centro de la región de visualización mas un desplazamiento en el eje X dependiendo de la cantidad de componentes que se hayan ubicado.
4. Se buscan los componentes conectados al componente seleccionado.
5. Organizar de mayor a menor por cantidad de interfaces los componentes conectados al componente seleccionado.

6. Distribuir alrededor del componente seleccionado empezando con el componente de mayor número de conexiones ubicado en el angulo cero y con una separación angular entre componentes de 2*π/(Número de componentes).
7. Identificar otros componentes que estén ubicados alrdedor del componente seleccionado y que eviten la ubicación de los componentes conectados, si los hay se redistribuyen los ángulos considerando dicho obstáculo para evitar intersección entre los objetos de los componentes, asi la separación angular será (2*π/(#Componentes+#Obstaculos)).
   Como se muestra en la Figura 3.33, para ubicar cuatro componentes conectados alrededor de E, estos estarian ubicados con una separación angular de 90 grados, pero si existiera el obstáculo F, la separación estaria dada por 72 grados, dejando espacio para el componente que obstaculiza la ubicación de los nuevos componentes.
8. Si alguno de los componentes conectados ya está ubicado se revisa si es posible trazar una línea de conexión, si así es, se deja en la ubicación actual; de lo contrario, se realiza el desplazamiento perpendicular a la línea de conexión de ambos componentes en la dirección que tenga menos componentes ubicados.
   a. Si es posible conectar los componentes luego de desplazarlos, se deben revisar todas las conexiones que tengan ambos componentes para realizar el mismo desplazamiento y asi evitar que dichas conexiones puedan tener intersección con otros elementos.
   b. Cada desplazamiento se realizará para una distancia especifica y si con dicho desplazamiento aún no es posible conectar los componentes, se intentará el proceso de desplazamiento cuantas veces sea necesario para lograr graficamente la conexión, sin exceder una cantidad máxima de intentos.
      En el caso que se alcance la cantidad máxima de intentos, los componentes quedarán con su ubicación original.
9. Asignar los implementadores para SVG en cada una de las interfaces conectadas del componente seleccionado.
10. Para cada pareja de componentes conectados con el componente seleccionado, se identifica la cantidad de interfaces entre ellos.
11. Se ubican las interfaces paralelas con una distancia de separación fija entre ellas.
12. Asignar implementadores para SVG en cada interfaz desconectada.
13. Determinar la separación angular para ubicar las interfaces desconectadas.
14. Identificar los ángulos ocupados por la ubicación de interfaces conectadas. Si hay, se recalcula la separación, así: (2*π/(#Interfaces+#Obstaculos)).
15. Centrar la imagen generada en la región de visualización.

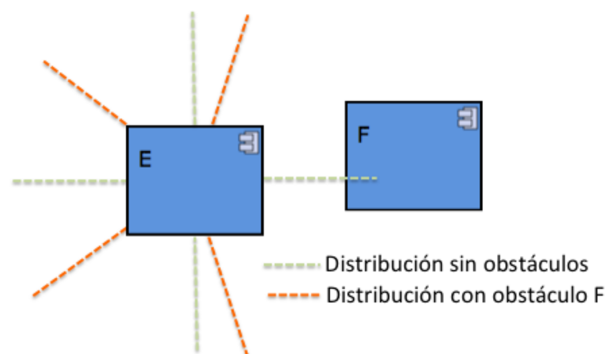

Figura 3.33. Distribución de conectores para componentes. Fuente autor.

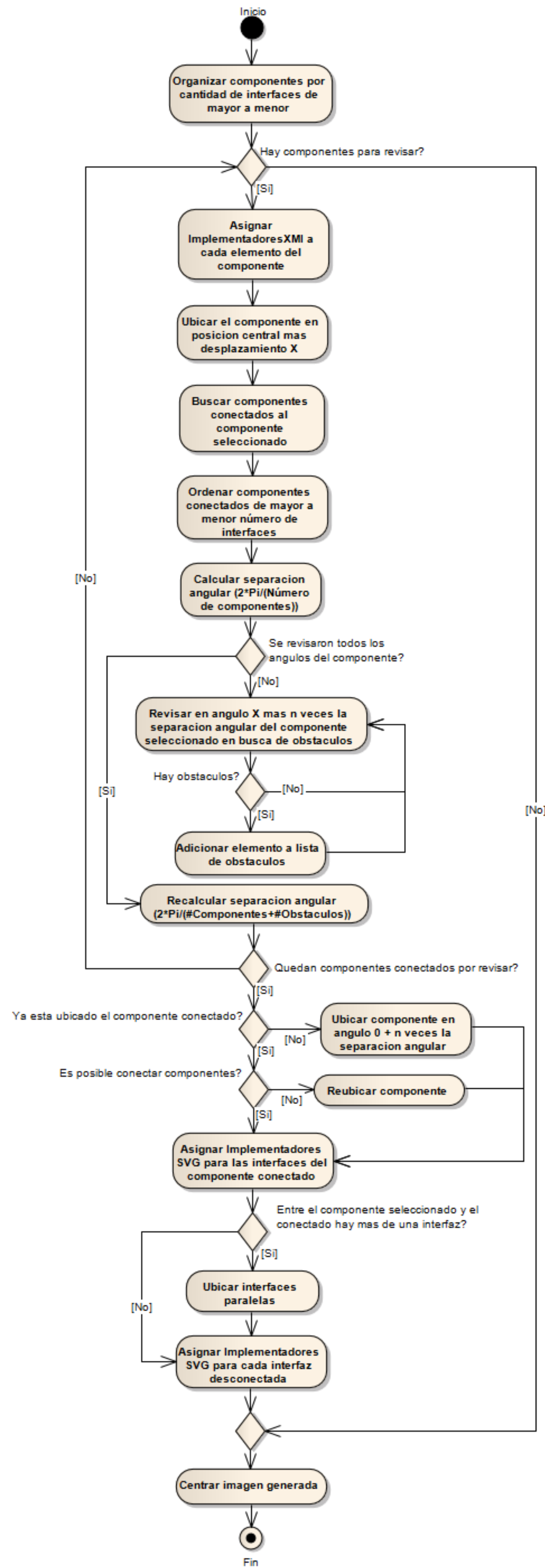

Figura 3.34. Diagrama de actividades para la ubicación de elementos de una arquitectura.

Con dicho algoritmo se realiza la ubicación de todos los componentes que configuran la arquitectura, y haciendo uso de las operaciones de la clase `ImplementadorSVG` se genera el código SVG que define la representación gráfica del elemento y su ubicación en la región de visualización. Dicha visualización es realizada por un interprete de código SVG, el cual es brindado por los exploradores de internet con soporte para HTML 5.

En los siguientes diagramas de secuencia se muestra la generalidad del módulo graficador y algunos de los mensajes importantes implementados.

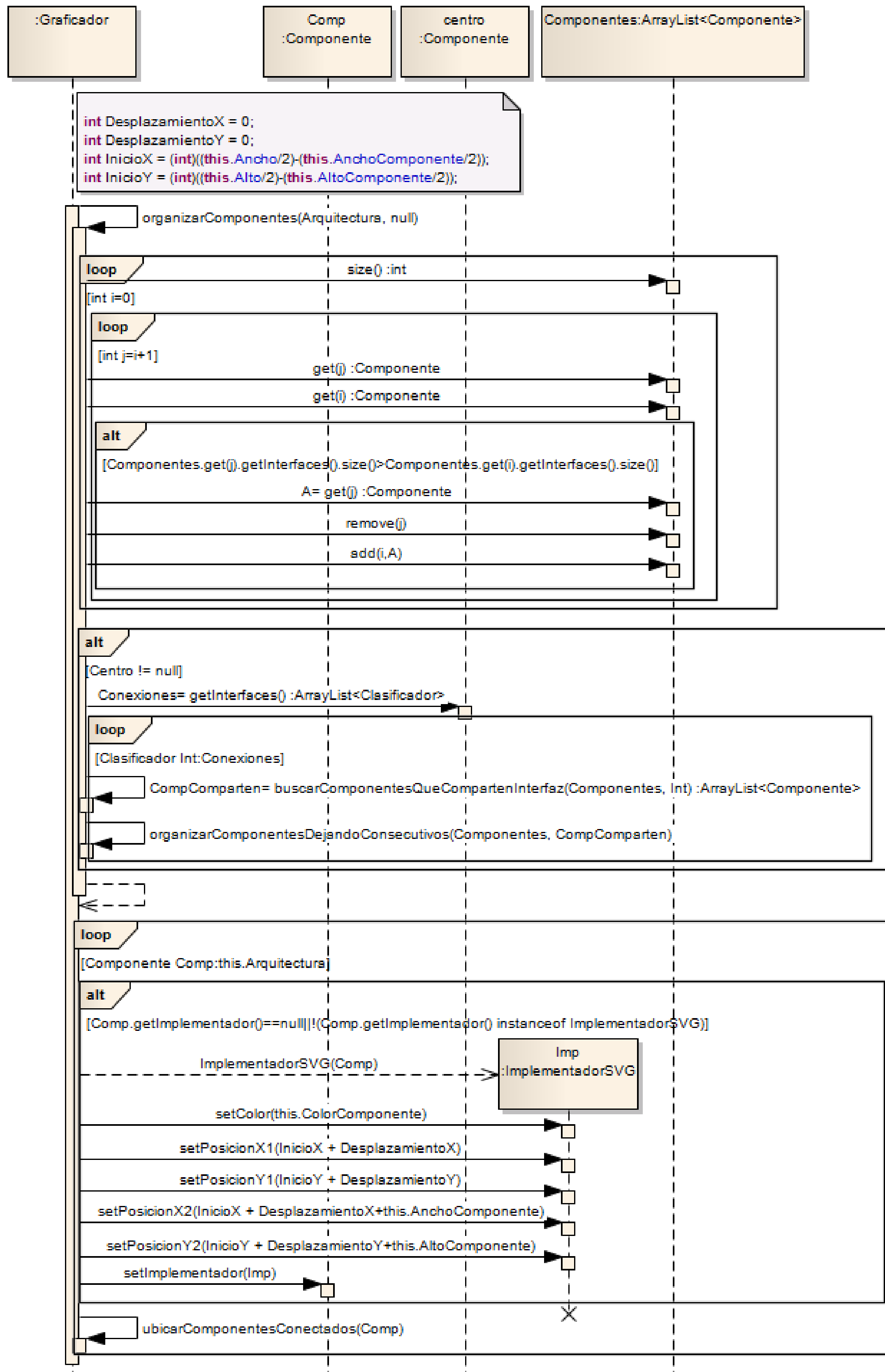

Figura 3.35. Modelo dinámico caso de uso “Ubicar componentes”.

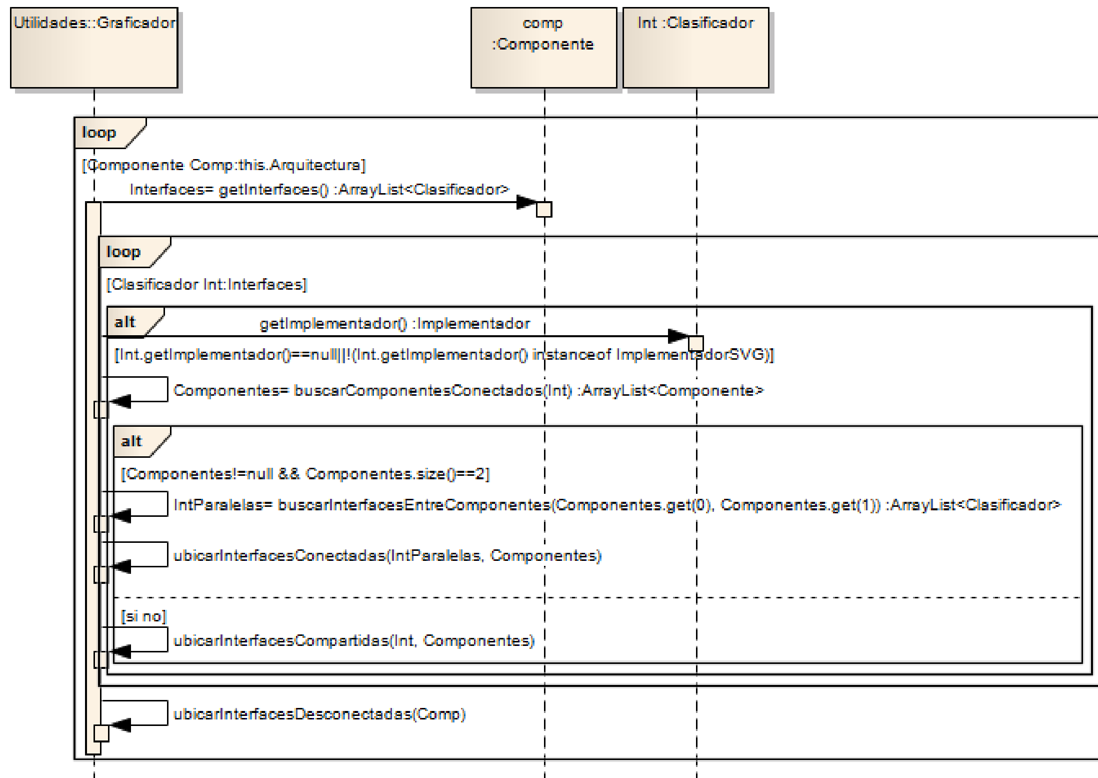

Figura 3.36. Modelo dinámico caso de uso "Ubicar conexiones".

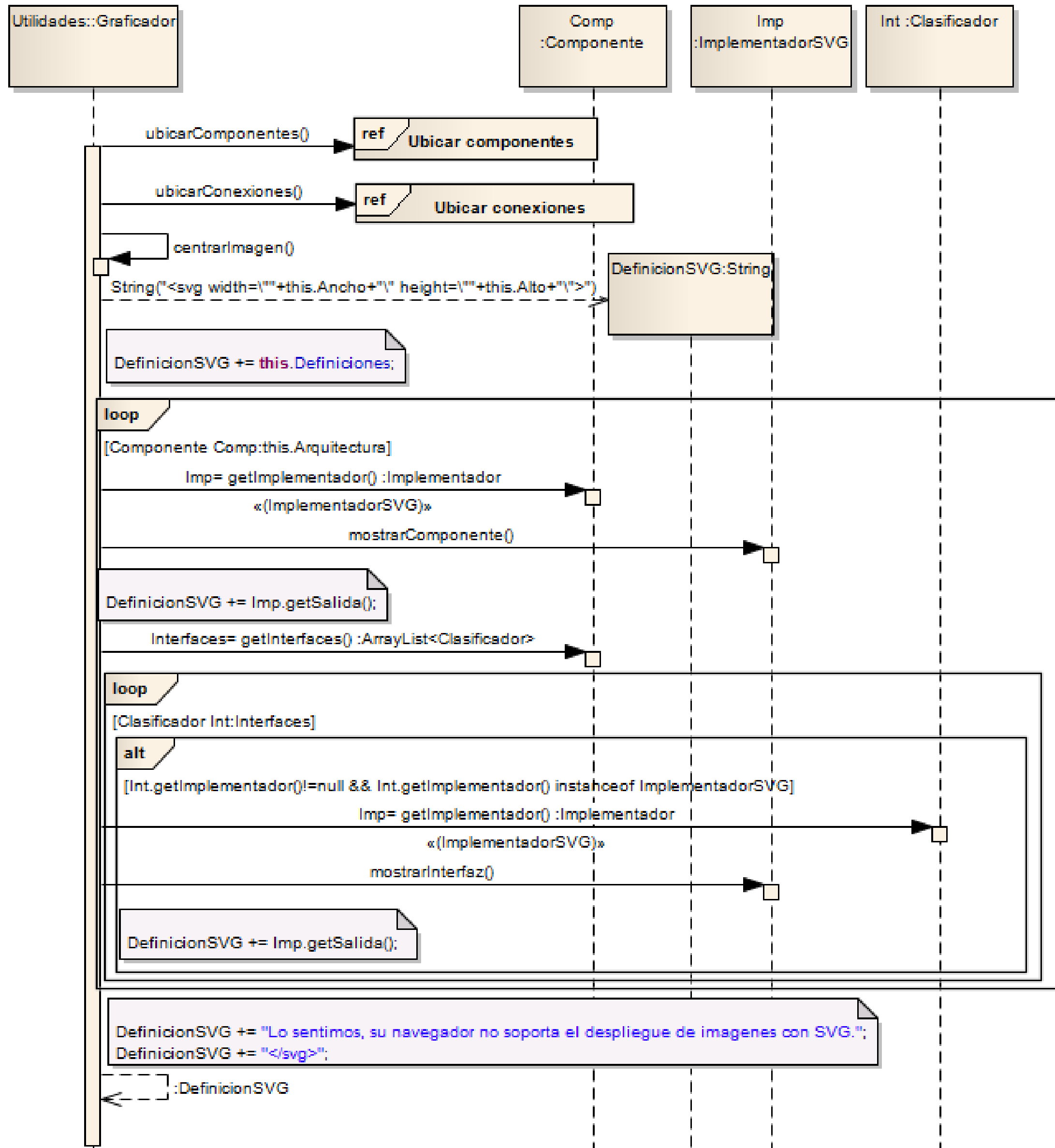

Figura 3.37. Modelo dinámico caso de uso "Graficar arquitectura".

3.5 PRUEBAS DE LA HERRAMIENTA

Para verificar el correcto funcionamiento de la herramienta construida, es necesario verificar los siguientes aspectos principales:

- Aplicación de la semántica operacional
- Mapeo de la descripción en el cálculo ρ_{arq} y la descripción en UML.
- Visualización del flujo de ejecución de una arquitectura.

3.5.1 Aplicación de la semántica operacional.

Las pruebas de semántica operacional consisten en la revisión de la aplicación de las reglas de reducción sobre expresiones donde sea posible su aplicación, así, inicialmente para probar la regla $A_{\rho_{arq}}$ se emplea la expresión:

$$S = c :: x/A^{(int)} \ \wedge \ c\overline{y}$$

con la cual se espera una reescritura de la ecuación de la siguiente manera:

$$S = [y/x]A^{(int)}$$

Como entrada para la aplicación, se prepara un documento Latex como se muestra a continuación:

```
\documentclass[10pt,letterpaper,fleqn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\title{Prueba Rho-arq}
\begin{document}
\begin{equation*}
        S=(c::x/A^{(int)}) \wedge c\overline{y}
\end{equation*}
\end{document}
```

Código 3.30. Definición en latex de ecuación de prueba para regla $A_{\rho_{arq}}$

Como se puede observar, se ha incluido la expresión abstracción entre un grupo de paréntesis para evitar que el intérprete tome la expresión composición solamente entre la expresión interior de A y la aplicación reducida.

Al ejecutar el *reescritor* e iniciar las operaciones de la clase *OperadorSemantico*, la aplicación detecta la interacción y genera la nueva expresión según lo esperado:

```
\begin{equation*}
        S = ( [ y / x ] A ^{(int)}    )
\end{equation*}
```

Código 3.31. Resultado latex de la aplicación de la regla $A_{\rho_{arq}}$.

Dicha definición corresponde textualmente a la siguiente ecuación:

$$S = ([y/x]A^{(int)})$$

También es necesario probar la aplicación sucesiva de la regla $A_{\rho_{arq}}$ dado que este escenario es el que en gran parte define el flujo de ejecución de una arquitectura al liberar expresiones que pueden interactuar posteriormente. Para este objetivo se preparó la siguiente ecuación:

$$S = \ c_1 :: x/c_2\overline{x} \ \wedge \ c_1\overline{y} \ \wedge \ c_2{:}\, z/B^{(int)}$$

En ella, se presentan dos interacciones, la primera enviando y a través del canal c_1 la variable y y en la segunda enviando la variable x que será reemplazada por y a través del canal c_2. Como resultado de la primera interacción, se realizará el reemplazo de x por y en la abstracción y como resultado de la segunda interacción se obtendrá un reemplazo de z por y en el interior de B.

Así, la expresión que representa la primera interacción es como la siguiente:

$$S' = [y/x](c_2\bar{x}) \wedge \ c_2: z/B^{(int)}$$

y la expresión final de la arquitectura será:

$$S'' = \ c_2: z/B^{(int)} \wedge [y/z]B^{(int)}$$

Para probar este escenario, se construye la siguiente definición en LaTex:

```
\documentclass[10pt,letterpaper,fleqn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\title{Prueba Rho-arq}
\begin{document}
\begin{equation*}
        S=(c_1::x/c_2\overline{x}) \wedge (c_1\overline{y}) \wedge
c_2:z/B^{(int)}
\end{equation*}
\end{document}
```

Código 3.32. Definición Latex para prueba de ejecución secuencial de reglas.

Y como resultado de la ejecución del operador semántico se obtuvo lo siguiente:

```
\begin{equation*}
        S = ( [ y / x ] c_2 \overline{ x }  )  \wedge c_2 : z / B ^{(int)}
\end{equation*}
\begin{equation*}
        S = ( c_2 : z / B ^{(int)} )  \wedge [ y / z ] B ^{(int)}
\end{equation*}
```

Código 3.33. Resultado de ejecución sucesiva de regla $A_{\rho_{arq}}$

Dichas ecuaciones corresponden al resultado esperado de la interacción del sistema y su representación textual se muestra a continuación:

$$S = ([y/x]c_2\overline{x}) \wedge c_2 : z/B^{(int)}$$

$$S = (c_2 : z/B^{(int)}) \wedge [y/z]B^{(int)}$$

La ejecución de la regla $Ejec_\tau$ se verifica al incluir en una composición una observación y una ejecución exitosa o no exitosa, la primer prueba será con ejecución exitosa, así, la ecuación para este escenario es la siguiente:

$$S = OSO(A) \ do \ B \ else \ C \wedge A^{\top}$$

En dicha ecuación se espera que la observación identifique la ejecución exitosa del componente A y libere la ejecución del componente B, dejando la ecuación del sistema como sigue:

$$S' = B$$

Para esta prueba el documento LaTex preparado es el siguiente:

```
\documentclass[10pt,letterpaper,twocolumn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\begin{document}
\begin{equation*}
        S=(OSO (A) do B else C) \wedge A ^{\top}
\end{equation*}
\end{document}
```

Código 3.34. Código LaTex para prueba de regla $Ejec_\tau$ con ejecución exitosa.

La respuesta del operador semántico para esta ecuación es:

```
\begin{equation*}
        S = ( B )
\end{equation*}
```

Código 3.35. Resultado de prueba para regla $Ejec_\tau$ con ejecución exitosa.

Ahora realizando la prueba pero con ejecución no exitosa

$$S = OSO(A)\ do\ B\ else\ C \wedge A^{\perp}$$

se esperaría que la ecuación del sistema quedará así:

$$S' = C$$

Para esto el documento latex preparado es el siguiente

```
\documentclass[10pt,letterpaper,twocolumn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\begin{document}
\begin{equation*}
        S=(OSO (A) do B else C) \wedge A ^{\bot}
\end{equation*}
\end{document}
```

Código 3.36. Código LaTex para prueba de regla $Ejec_\tau$ con ejecución no exitosa.

Y luego de ejecutar el operador semántico, este entrega como resultado la siguiente ecuación que es equivalente al resultado esperado.

```
\begin{equation*}
        S = ( C )
\end{equation*}
```

Código 3.37. Resultado de prueba para regla $Ejec_\tau$ con ejecución no exitosa

3.5.2 Mapeo de la configuración arquitectural en UML

La descripción arquitectural a partir del cálculo ρ_{arq} es realizada a partir de unos patrones de expresión que permiten definir interfaces de provisión y de requerimiento las cuales de acuerdo a las variables establecidas pueden o no interactuar entre sí. Según esto, para describir una interfaz de provisión el cálculo ofrece la expresión abstracción, por tanto toda aquella abstracción utilizada en una expresión es candidata a ser interpretada como una interfaz de este tipo. Por otro lado una interfaz de requerimiento es definida empleando la expresión declaración, la cual exterioriza del componente la ubicación a la cual debe ser enviado el servicio requerido; sin embargo, para que la interfaz de provisión conozca dicha ubicación, la interfaz de requerimiento también debe enviar dicha información, por tanto también emplea la expresión abstracción. Así, la declaración debe tener en su ámbito una composición entre dos abstracciones, una de ellas que activa el envío de la ubicación y otra que recibe el servicio requerido y lo envía al componente.

De acuerdo a lo mencionado, los patrones de expresión que la herramienta identificará están dados por las siguientes expresiones:

$$Prov = c :: x/x\bar{d}$$
$$Req = \exists l[p :: y/y\bar{l} \wedge l :: i/A^{(int)}]$$

Donde una interfaz de provisión está definida por una expresión abstracción que recibe información a través del canal c para ser incorporada en una aplicación reducida en x y enviar d a través de la ubicación recibida.

Una interfaz de requerimiento estará definida por una declaración cuyo ámbito incluye la ejecución concurrente de dos abstracciones, la primera de ellas que recibe un canal para enviar la ubicación en la cual se requiere el servicio y la segunda para recibir el servicio requerido y transferirlo al componente.

Las pruebas de mapeo consisten en la verificación de la funcionalidad del arquitecto y del graficador, de tal forma que si una de las expresiones escritas en la descripción arquitectural cumple con los patrones mencionados, estos deben ser mapeados como interfaces de requerimiento o de provisión según sea el caso.

La primera prueba realizada verifica el mapeo de un componente A definido con una interfaz de provisión y una interfaz de requerimiento, así, el documento LaTex con la descripción de dicha arquitectura es el siguiente:

```
\documentclass[10pt,letterpaper,twocolumn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\begin{document}
\begin{equation*}
        S=A
\end{equation*}
\begin{equation*}
```

```
        A=(c:x/x\overline{d}) \wedge \exists  l[(p::y/y\overline{l})\wedge
l::i/A^{(int)}]
\end{equation*}
\end{document}
```

Código 3.38. Documento LaTex para prueba de mapeo de interfaz de requerimiento y provisión.

Al procesar dicho archivo en la aplicación se obtiene el siguiente modelo en UML:

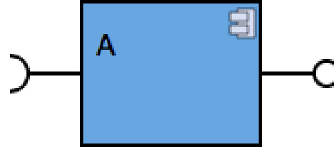

Figura 3.38. Resultado de interpretación para un componente con una interfaz de provisión y una interfaz de requerimiento.

El modelo obtenido representa de acuerdo a lo esperado la arquitectura descrita usando la notación UML de componentes.

La siguiente prueba pretende verificar que el módulo arquitecto pueda identificar las interfaces conectadas entre sí, para esto se incorporará un componente adicional con una interfaz de provisión y un conector representado por la expresión de tipo aplicación reducida. Dicho conector es el encargado de enviar a la abstracción de la interfaz de requerimiento el canal por el cual puede enviar la ubicación o puerto a la interfaz de provisión.

El documento LaTex que describe dicho escenario es el siguiente

```
\documentclass[10pt,letterpaper,twocolumn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\begin{document}
\begin{equation*}
        S=A \wedge B \wedge c_AB
\end{equation*}
\begin{equation*}
        A=(c_A:x/x\overline{f}) \wedge \exists l[(p::y/y\overline{l})\wedge
l::i/A^{(int)}]
\end{equation*}
\begin{equation*}
        B=c_B:x/x\overline{d}
\end{equation*}
\begin{equation*}
        c_AB=p\overline{c_B}
\end{equation*}
\end{document}
```

Código 3.39. Documento LaTex para prueba de conexión de componentes.

Y al procesar dicho archivo, se obtiene el siguiente modelo UML:

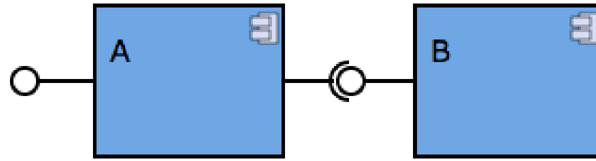

Figura 3.39. Modelo UML para conexión de componentes

Usando este mismo escenario y dado que ya se demostró que el operador semántico está en capacidad de aplicar las reglas de reducción del cálculo, se puede probar que al iniciar la aplicación de la semántica operacional, el reescritor comunica las interacciones encontradas al arquitecto para que este las incorpore en un cambio de estado de la arquitectura y así el graficador pueda representarla visualmente a través de la interfaz de provisión involucrada en la interacción.

De esta manera al observar el flujo de ejecución de esta arquitectura se puede evidenciar que existe una interacción entre el componente `A` y `B` mediante sus interfaces, de la siguiente manera.

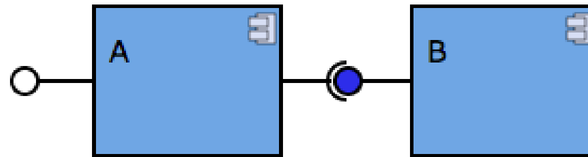

Figura 3.40. Visualización de interacción entre componentes.

Para probar la aplicación sucesiva de la regla de reducción $A_{\rho_{arq}}$ se adicionará una conexión adicional entre los componentes `A` y `B`, de tal forma que se pueda observar la interacción secuencial de las interfaces. Para esto, se define un documento latex como el siguiente:

```
\documentclass[10pt,letterpaper,twocolumn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\begin{document}
\begin{equation*}
        S=A \wedge B \wedge c_AB \wedge c_BA
\end{equation*}
\begin{equation*}
        A=(c_A:x/x\overline{d}) \wedge \exists l_A[(p_A::y/y\overline{l_A})\wedge
l_A::i/A^{(int)}]
\end{equation*}
\begin{equation*}
        B=(c_B:x/x\overline{d}) \wedge \exists l_B[(p_B::y/y\overline{l_B})\wedge
l_B::i/B^{(int)}]
\end{equation*}
\begin{equation*}
        c_AB=p_A\overline{c_B}
\end{equation*}
\begin{equation*}
        c_BA=p_B\overline{c_A}
\end{equation*}
\end{document}
```

Código 3.40. Documento LaTex para prueba de ejecuciones sucesivas.

Al procesar este archivo en la herramienta, se observa una configuración como la siguiente:

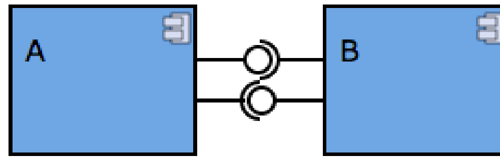

Figura 3.41. Configuración para prueba de interacción sucesiva.

Y al iniciar el flujo de ejecución de dicha arquitectura, se pueden observar los siguientes estados:

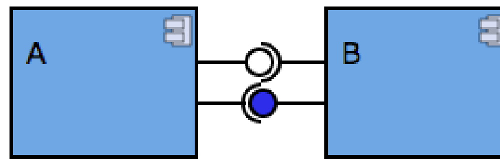

Figura 3.42. Primera interacción de la arquitectura.

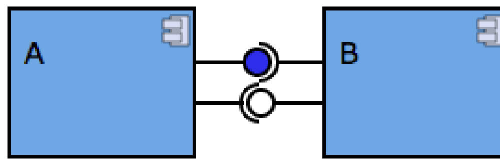

Figura 3.43. Segunda interacción de la arquitectura

3.5.2.1 Prueba de ensamble complejo de componentes

Esta prueba se basa en la configuración descrita en [58] para el sistema S_2 donde se define un componente manejador de errores. Esta configuración contiene cuatro componentes, donde uno de ellos tiene una interfaz de requerimiento que es utilizada por los otros tres para enviar la información de los errores ocurridos y este componente se encargue de dicho manejo.

Para este escenario se definirán tres componentes E,F,T y el componente manejador M. E tendrá dos interfaces de provisión y una interfaz de requerimiento, F tendrá dos interfaces de provisión, T tendrá una interfaz de provisión y una de requerimiento y M tendrá una interfaz de requerimiento. E, tendrá conexión a través de su interfaz de requerimiento con F y a través de una de sus interfaces de provisión con T, a su vez, E,F y T estarán conectados con M a través de sus interfaces de provisión restantes.

La definición ρ_{arq} para esta configuración es así:

$$S = F^{\top} \wedge [OSO(F)\ do\ F \wedge c_{FE} \wedge E\ else\ F \wedge c_{FM} \wedge M]$$
$$\wedge [OSO(E)\ do\ E \wedge\ c_{ET} \wedge T\ else\ E \wedge\ c_{EM} \wedge M] \wedge [OSO(T)\ do\ T\ else\ T$$
$$\wedge c_{TM} \wedge M]$$
$$E = P_E : x/x\overline{s_e} \wedge P_{Ee} : v/v\overline{s_{Ee}} \wedge \exists l_E [r_E :: y/yl_E \wedge l_E :: i_E/E^{(int)}]$$
$$F = p_f : z/z\overline{s_f} \wedge p_{fe} : w/w\overline{s_{fe}}$$

$T = \exists l_T[r_T: q/ql_T \wedge l_T: i_T/T^{(int)}] \wedge p_{Te}: n/n\overline{s_{Te}}$
$M = \exists l_m[r_m: y/y\overline{l_m} \wedge l_m: i_m/M^{(int)}]$
$c_{FE} = r_E\overline{p_F}$
$c_{FM} = r_m\overline{p_{Fe}}$
$c_{ET} = r_T\overline{p_E}$
$c_{EM} = r_M\overline{p_{Ee}}$
$c_{TM} = r_m p_{Te}$

Código 3.41. Configuración ρ_{arq} para sistema complejo.

Ahora la representación LaTex de estas ecuaciones es como se muestra a continuación:

```
\documentclass[10pt,letterpaper,fleqn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\title{Prueba Rho-arq}
\begin{document}
\begin{equation*}
        S= F^{\top} \wedge [OSO (F) do F \wedge c_FE \wedge E else F \wedge c_FM
\wedge M] \wedge [OSO (E) do E \wedge c_ET \wedge T else E \wedge c_EM \wedge M]
\wedge [OSO (T) do T else T \wedge c_TM \wedge M]
\end{equation*}
\begin{equation*}
        E=(p_E:x/x\overline{s_E}) \wedge (p_Ee:v/v\overline{s_Ee}) \wedge \exists
l_E[(r_E::y/y\overline{l_E}) \wedge (l_E::i_E/E^{(int)})]
\end{equation*}
\begin{equation*}
        F=(p_F:z/z\overline{s_F}) \wedge (p_Fe:w/w\overline{s_Fe})
\end{equation*}
\begin{equation*}
        M=\exists l_M[(r_M:y/y\overline{l_M}) \wedge (l_M:i_M/M^{(int)})]
\end{equation*}
\begin{equation*}
        T=(\exists l_T[(r_T:q/q\overline{l_T}) \wedge (l_T:i_T/T^{(int)})])
\wedge (p_Te:n/n\overline{s_Te})
\end{equation*}
\begin{equation*}
        c_FE = r_E\overline{p_F}
\end{equation*}
\begin{equation*}
        c_FM = r_M\overline{p_Fe}
\end{equation*}
\begin{equation*}
        c_ET = r_T\overline{p_E}
\end{equation*}
\begin{equation*}
        c_EM = r_M\overline{p_Ee}
\end{equation*}
\begin{equation*}
        c_TM = r_M\overline{p_Te}
\end{equation*}
\end{document}
```

Código 3.42. Código LaTex con descripción del sistema complejo.

Al realizar la carga de esta configuración en el sistema desarrollado, se obtiene la siguiente representación gráfica, que corresponde directamente con la configuración descrita en ρ_{arq}.

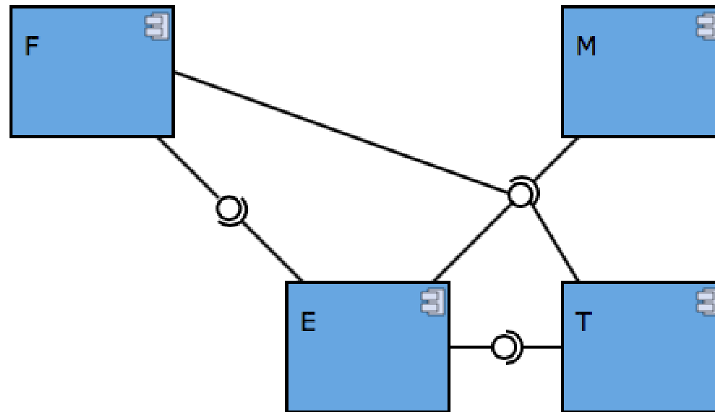

Figura 3.44. Representación gráfica de configuración compleja.

Luego, al ejecutar dicha arquitectura el reescritor del sistema identifica la interacción de las expresiones de ejecución exitosa de *F* y la expresión de observación sobre *F*, liberando la expresión que permite la interacción de *F* con *E*. Una vez son realizadas dichas reescrituras y el arquitecto identifica las interacciones, el sistema es desplegado como se muestra a continuación:

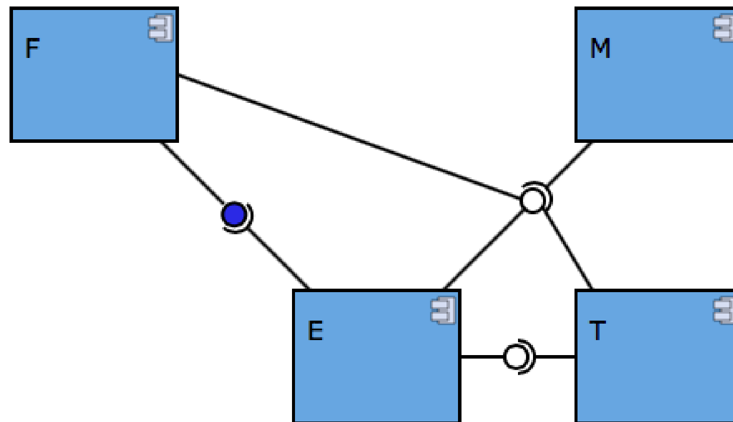

Figura 3.45. Ejecución de arquitectura compleja.

Así, el graficador representa gráficamente la interacción ocurrida entre *F* y *E* y resalta las interfaces involucradas.

Si la configuración del sistema cambia y a cambio de una ejecución exitosa de *F*, se incluye una ejecución no exitosa como se muestra en la ecuación

$$S = F^{\perp} \wedge [OSO(F)\ do\ F \wedge c_{FE} \wedge E\ else\ F \wedge c_{FM} \wedge M] \wedge [OSO(E)\ do\ E \wedge\ c_{ET} \wedge T\ else\ E \wedge\ c_{EM} \wedge M] \wedge [OSO(T)\ do\ T\ else\ T \wedge c_{TM} \wedge M]$$

Código 3.43. Definición del sistema complejo con ejecución no exitosa de F.

Para que la ecuación del sistema refleje esta configuración, es necesario modificar la definición LaTex sobre la primera ecuación del documento, dejándola como se muestra a continuación:

```
\begin{equation*}
S= F^{\bot} \wedge [OSO (F) do F \wedge c_FE \wedge E else F \wedge c_FM \wedge
M] \wedge [OSO (E) do E \wedge c_ET \wedge T else E \wedge c_EM \wedge M] \wedge
[OSO (T) do T else T \wedge c_TM \wedge M]
\end{equation*}
```

Código 3.44. Definición LaTex del sistema con ejecución NO exitosa de F.

Luego al cargar dicha definición en la herramienta construida, se obtiene la misma configuración mostrada en la Figura 3.44 debido a que la estructura de la arquitectura no fue modificada, solamente ha sido modificado un estado de ella. Así, al iniciar la ejecución de dicha arquitectura, el estado gráfico que se obtiene es el siguiente:

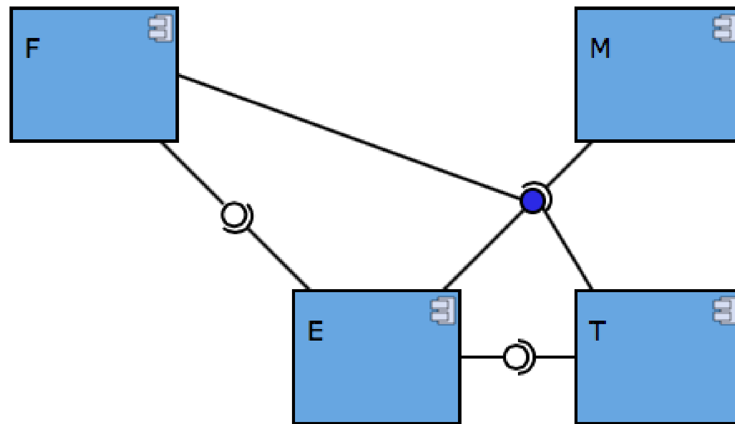

Figura 3.46. Representación gráfica para la ejecución No exitosa de F.

Esto debido a que la expresión observación al identificar una ejecución no exitosa del componente F, realiza la liberación de la expresión que permite la interacción del componente F con el componente M para que realice el manejo de errores correspondiente.

3.5.2.2 Expresiones con combinador de selección condicionada

En la implementación de la reescritura de este tipo de expresiones, dado que para la evaluación de las guardas se requiere la intervención del arquitecto, se implementó una selección aleatoria de las guardas que se interpretarán como verdaderas.

En esta prueba se ha configurado una arquitectura de tres componentes donde el combinador de selección determina cual es la interacción que debe realizarse, para esto se ha dispuesto de los componentes E, F y G, donde E tiene conexión con F y con G.

Dependiendo de la evaluación de la guarda, la arquitectura reflejará la activación de la interacción de E con F o de E con G.

$$S = E \wedge C_{EF} \wedge if (a = b then F) else G \wedge C_{EG}$$

$$PROV_E(p,s) = p_E : x/x\overline{s_E}$$

$$E = PROV_E(p,s)$$

$$F = \exists l_F[(r_F :: y/y\overline{l_F}) \wedge (l_F :: i_F/F^{(int)})]$$

$$G = \exists l_G[(r_G :: y/y\overline{l_G}) \wedge (l_G :: i_G/G^{(int)})]$$

$$C_{EF} = r_F\overline{p_E}$$

$$C_{EG} = r_G\overline{p_E}$$

Figura 3.47. Configuración de arquitectura con combinador de selección condicionada.

El documento LaTex que define dicha arquitectura es el mostrado a continuación:

```
\documentclass[10pt,letterpaper,fleqn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\title{Prueba Rho-arq}
\begin{document}
\begin{equation*}
        S=E \wedge C_{EF} \wedge if (a = b then F) else G \wedge C_{EG}
\end{equation*}
\begin{equation*}
        PROV_E(p,s)=p_E:x/x\overline{s_E}
\end{equation*}
\begin{equation*}
        E=PROV_E(p,s)
\end{equation*}
\begin{equation*}
        F=\exists l_F[(r_F::y/y\overline{l_F})\wedge(l_F::i_F/F^{(int)})]
\end{equation*}
\begin{equation*}
        G=\exists l_G[(r_G::y/y\overline{l_G})\wedge(l_G::i_G/G^{(int)})]
\end{equation*}
\begin{equation*}
        C_{EF}=r_F\overline{p_E}
\end{equation*}
\begin{equation*}
        C_{EG}=r_G\overline{p_E}
\end{equation*}
\end{document}
```

Código 3.45. Código LaTex con descripción de sistema usando combinador de selección condicionada.

Una vez se carga la definición del sistema, la herramienta muestra la configuración del sistema como en la Figura 3.48.

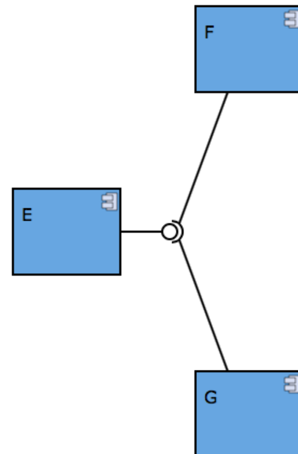

Figura 3.48. Configuración de sistema para prueba de elemento combinador.

El combinador utilizado tiene una clausula que contiene la guarda a=b, al iniciar el flujo de ejecución de la arquitectura cuando la guarda es verdadera se libera el componente F para su interacción con E, permitiendo que el estado de la arquitectura en la herramienta se muestre como en la Figura 3.49.

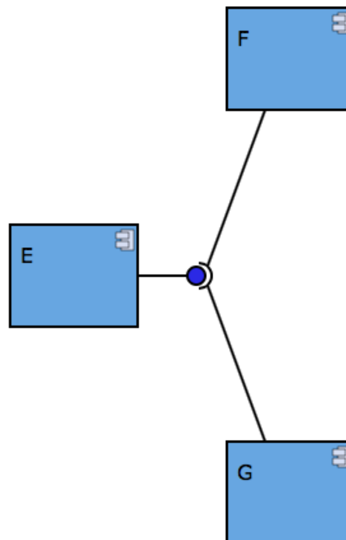

Figura 3.49. Arquitectura con valor de la guarda verdadero.

Por el contrario si el resultado de evaluar la guarda del combinador es falso, se libera el componente G para su interacción con E y el estado de la arquitectura desplegado es el mismo desplegado en la Figura 3.49.

3.5.3 Exportación a XMI

Para verificar la exportación a XMI, se emplearán las mismas arquitecturas utilizadas para probar el mapeo de la configuración arquitectural y un ejemplo mas usando la arquitectura propuesta en la Figura 3.9.

3.5.3.1 Exportación de arquitecturas expuestas

A continuación se mostrará el resultado de la exportación a XMI de cada una de las arquitecturas mostradas en el capítulo 3.5.2.

Al exportar una arquitectura compuesta de un componente con una interfaz de provisión y una interfaz de requerimiento (Figura 3.38), el sistema genera el siguiente documento XML:

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI     xmi:version="2.1"     xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="PintArq" exporterVersion="1.0"/>
<uml:Model   xmi:type="uml:Model"   name="Modelo_DefinicionRhoArqUnComponente.tex"
visibility="public">
       <ownedMember      xmi:type="uml:Package"      xmi:id="PintArqPK_1281041477"
       name="Arquitectura" visibility="public">
              <ownedMember   xmi:type="uml:Component"   xmi:id="PINTARQID_comp_65"
              name="A" visibility="public">
                     <provided xmi:id="PINTARQID_prov_2" name=""/>
                     <required xmi:id="PINTARQID_req_3" name=""/>
              </ownedMember>
         </ownedMember>
</uml:Model>
</xmi:XMI>
```

Código 3.46. Documento XML generado para exportación a XMI de arquitectura con un componente.

Y el modelo arquitectural al generar un diagrama con los objetos importados genera la siguiente configuración:

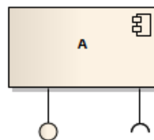

Figura 3.50. Diagrama producto de importación XMI para arquitectura de un componente.

Para la exportación de la arquitectura empleada en la prueba de conexión de componentes (Figura 3.39), se realiza la exportación y se genera el siguiente documento XML:

```
<?xml   version="1.0"   encoding="windows-1252"?>   <xmi:XMI   xmi:version="2.1"
xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="PintArq" exporterVersion="1.0"/>
<uml:Model       xmi:type="uml:Model"       name="Modelo_ConexionComponentes.tex"
visibility="public">
```

```
<ownedMember          xmi:type="uml:Package"          xmi:id="PintArqPK_742150993"
name="Arquitectura" visibility="public">
         <ownedMember     xmi:type="uml:Component"     xmi:id="PINTARQID_comp_65"
         name="A" visibility="public">
             <provided xmi:id="PINTARQID_prov_2" name=""/>
             <ownedConnector                                  xmi:type="uml:Connector"
             xmi:id="PINTARQID_CONEXION_3"      name="3"      visibility="public"
             kind="assembly">
                <end   xmi:type="uml:ConnectorEnd"   xmi:id="PINTARQID_CONEXION_3"
                role="PINTARQID_comp_65"/>
                <end   xmi:type="uml:ConnectorEnd"   xmi:id="PINTARQID_CONEXION_3"
                role="PINTARQID_comp_66"/>
             </ownedConnector>
        </ownedMember>
        <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_66" name="B"
        visibility="public">
        </ownedMember>
</ownedMember>
</uml:Model>
</xmi:XMI>
```

Código 3.47. Documento XML para arquitectura con conexión de componentes.

El cual es importado en la herramienta de modelado UML y posteriormente se construye un diagrama con los elementos importados que refleja con exactitud la misma arquitectura desplegada en PintArq:

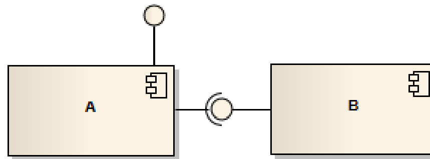

Figura 3.51. Arquitectura luego de importación XMI para arquitectura con conexión de componentes.

Para la arquitectura de la prueba de ejecución sucesiva de la regla de reducción $A_{\rho_{arq}}$(Figura 3.41), se genera el siguiente documento XML:

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI      xmi:version="2.1"      xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="PintArq" exporterVersion="1.0"/>
<uml:Model        xmi:type="uml:Model"        name="Modelo_ReduccionSucesiva.tex"
visibility="public">
<ownedMember          xmi:type="uml:Package"          xmi:id="PintArqPK_-91886430"
name="Arquitectura" visibility="public">
         <ownedMember     xmi:type="uml:Component"     xmi:id="PINTARQID_comp_65"
         name="A" visibility="public">
             <ownedConnector                                  xmi:type="uml:Connector"
             xmi:id="PINTARQID_CONEXION_2"      name="2"      visibility="public"
             kind="assembly">
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_2"
                   role="PINTARQID_comp_65"/>
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_2"
                   role="PINTARQID_comp_66"/>
             </ownedConnector>
         </ownedMember>
         <ownedMember     xmi:type="uml:Component"     xmi:id="PINTARQID_comp_66"
         name="B" visibility="public">
```

```
            <ownedConnector                              xmi:type="uml:Connector"
            xmi:id="PINTARQID_CONEXION_3"      name="3"       visibility="public"
            kind="assembly">
                  <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_3"
                  role="PINTARQID_comp_66"/>
                  <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_3"
                  role="PINTARQID_comp_65"/>
            </ownedConnector>
        </ownedMember>
</ownedMember>
</uml:Model>
</xmi:XMI>
```

Código 3.48. Documento XML generado para la arquitectura de prueba usada en la ejecución sucesiva de la regla $A_{\rho_{arq}}$.

Al importarlo en la herramienta de modelamiento UML y generar un diagrama con los objetos importados, se obiene una arquitectura idéntica a la arquitectura generada en PintArq.

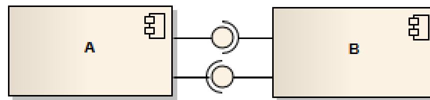

Figura 3.52. Arquitectura luego de importación XMI para ejemplo de ejecución sucesiva de la regla $A_{\rho_{arq}}$.

Para la arquitectura empleada en la configuración compleja, se genera el siguiente documento XML:

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI     xmi:version="2.1"     xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="PintArq" exporterVersion="1.0"/>
<uml:Model    xmi:type="uml:Model"    name="Modelo_DefinicionRhoArqEnsComplejo.tex"
visibility="public">
         <ownedMember     xmi:type="uml:Package"     xmi:id="PintArqPK_-443930011"
         name="Arquitectura" visibility="public">
              <ownedMember   xmi:type="uml:Component"   xmi:id="PINTARQID_comp_69"
              name="E" visibility="public">
                  <ownedConnector                          xmi:type="uml:Connector"
                  xmi:id="PINTARQID_CONEXION_48" name="null" visibility="public"
                  kind="assembly">
                        <end                        xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_48"
                        role="PINTARQID_comp_69"/>
                        <end                        xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_48"
                        role="PINTARQID_comp_70"/>
                  </ownedConnector>
            </ownedMember>
            <ownedMember   xmi:type="uml:Component"   xmi:id="PINTARQID_comp_84"
            name="T" visibility="public">
                  <ownedConnector                          xmi:type="uml:Connector"
                  xmi:id="PINTARQID_CONEXION_49" name="null" visibility="public"
                  kind="assembly">
                        <end                        xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_49"
                        role="PINTARQID_comp_84"/>
                        <end                        xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_49"
                        role="PINTARQID_comp_69"/>
                  </ownedConnector>
```

```
            </ownedMember>
            <ownedMember   xmi:type="uml:Component"   xmi:id="PINTARQID_comp_70"
            name="F" visibility="public">
            </ownedMember>
            <ownedMember   xmi:type="uml:Component"   xmi:id="PINTARQID_comp_77"
            name="M" visibility="public">
                  <ownedConnector                          xmi:type="uml:Connector"
                  xmi:id="PINTARQID_CONEXION_50" name="null" visibility="public"
                  kind="assembly">
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_50"
                        role="PINTARQID_comp_77"/>
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_50"
                        role="PINTARQID_comp_69"/>
                  </ownedConnector>
                  <ownedConnector                          xmi:type="uml:Connector"
                  xmi:id="PINTARQID_CONEXION_51" name="null" visibility="public"
                  kind="assembly">
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_51"
                        role="PINTARQID_comp_77"/>
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_51"
                        role="PINTARQID_comp_84"/>
                  </ownedConnector>
                  <ownedConnector                          xmi:type="uml:Connector"
                  xmi:id="PINTARQID_CONEXION_52" name="null" visibility="public"
                  kind="assembly">
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_52"
                        role="PINTARQID_comp_77"/>
                        <end                          xmi:type="uml:ConnectorEnd"
                        xmi:id="PINTARQID_CONEXION_52"
                        role="PINTARQID_comp_70"/>
                  </ownedConnector>
            </ownedMember>
        </ownedMember>
</uml:Model>
</xmi:XMI>
```

Código 3.49. Documento XML generado para arquitectura de ensamble complejo.

Luego al importarlo en la herramienta de modelamiento UML y generar un diagrama con los objetos generados, se obtiene la siguiente arquitectura:

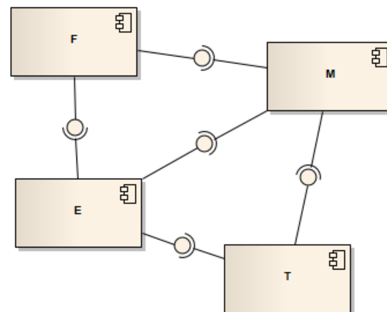

Figura 3.53. Arquitectura importada usando XMI para ejemplo de ensamble complejo.

En este escenario se puede observar que la interfaz de requerimiento del componente M se creo varias veces en la herramienta de modelado UML, una para cada conexión, dado que no es posible representar en este lenguaje interfaces de uso compartido.

3.5.3.2 *Exportación de la arquitectura de PintArq*

Para esta arquitectura, se genera el siguiente documento Latex que describe el flujo básico de PintArq:

```
\documentclass[10pt,letterpaper,fleqn]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{amssymb}
\author{Alejandro Rico}
\title{Arquitectura PintArq}
\begin{document}
\begin{equation*}
        S= Arquitecto \wedge Transformador \wedge Graficador \wedge Reescritor
\wedge Interprete \wedge (OSO (Interprete) do (C_{IR} \wedge Reescritor^{\top})
else C_{IA}) \wedge Interprete^{\top} \wedge (OSO (Arquitecto) do (C_{AG} \wedge
C_{AT}) else \tau Arquitecto) \wedge OSO (Reescritor) do (C_{RA} \wedge
Arquitecto^{\top}) else \tau Reescritor
\end{equation*}
\begin{equation*}
Interprete = RCargue \wedge PArcInicial_{ANTLR} \wedge PArquitectura_{ANTLR}
\end{equation*}
\begin{equation*}
        RCargue=\exists
l_{Interprete}[(r_{Interprete}:x/x\overline{l_{Interprete}}) \wedge
(l_{Interprete}:Archivo/Interprete^{(int)})]
\end{equation*}
\begin{equation*}
        PArcInicial_{ANTLR}=(p1_{Interprete}:y/y\overline{Arquitectura_{ANTLR}})
\end{equation*}
\begin{equation*}
        PArquitectura_{ANTLR}=(p2_{Interprete}:y/y\overline{Arquitectura_{ANTLR}}
)
\end{equation*}
\begin{equation*}
Reescritor = RArquitectura \wedge PReescritura
\end{equation*}
\begin{equation*}
        RArquitectura = \exists
l_{Reescritor}[(r_{Reescritor}:x/x\overline{l_{Reescritor}}) \wedge
(l_{Reescritor}:Arquitectura/Reescritor^{(int)})]
\end{equation*}
\begin{equation*}
        PReescritura=(p_{Reescritor}:y/y\overline{ArquitecturaReescrita})
\end{equation*}
\begin{equation*}
Arquitecto = RArquitectura_{Arquitecto} \wedge PArquitectoTrans \wedge
PArquitectoGraf
\end{equation*}
\begin{equation*}
        RArquitectura_{Arquitecto} = \exists
l_{Arquitecto}[(r_{Arquitecto}:x/x\overline{l_{Arquitecto}}) \wedge
(l_{Arquitecto}:Arquitectura/Arquitecto^{(int)})]
\end{equation*}
\begin{equation*}
        PArquitectoTrans=p1_{Arquitecto}:y/y\overline{ObjetosArquitectura}
\end{equation*}
\begin{equation*}
        PArquitectoGraf=p2_{Arquitecto}:y/y\overline{ObjetosArquitectura}
\end{equation*}
```

```
\begin{equation*}
Transformador = RArquitectura_{Transformador} \wedge PTransformador
\end{equation*}
\begin{equation*}
        RArquitectura_{Transformador} = \exists
l_{Transformador}[(r_{Transformador}:x/x\overline{l_{Transformador}}) \wedge
(l_{Transformador}:Arquitectura/Transformador^{(int)})]
\end{equation*}
\begin{equation*}
        PTransformador=p_{Transformador}:y/y\overline{ArquitecturaXMI}
\end{equation*}
\begin{equation*}
Graficador = RArquitectura_{Graficador} \wedge PGraficador
\end{equation*}
\begin{equation*}
        RArquitectura_{Graficador} = \exists
l_{Graficador}[(r_{Graficador}:x/x\overline{l_{Graficador}}) \wedge
(l_{Graficador}:Arquitectura/Graficador^{(int)})]
\end{equation*}
\begin{equation*}
        PGraficador=p_{Gracficador}:y/y\overline{ArquitecturaSVG}
\end{equation*}
\begin{equation*}
        C_{IR} = r_{Reescritor}\overline{p2_{Interprete}}
\end{equation*}
\begin{equation*}
        C_{RA} = r_{Arquitecto}\overline{p_{Reescritor}}
\end{equation*}
\begin{equation*}
        C_{IA} = r_{Arquitecto}\overline{p1_{Interprete}}
\end{equation*}
\begin{equation*}
        C_{AT} = r_{Transformador}\overline{p1_{Arquitecto}}
\end{equation*}
\begin{equation*}
        C_{AG} = r_{Graficador}\overline{p2_{Arquitecto}}
\end{equation*}
\end{document}
```

Código 3.50. Código latex con definición de la arquitectura de PintArq.

Luego al cargar el archivo en la aplicación, esta muestra la siguiente configuración arquitectural:

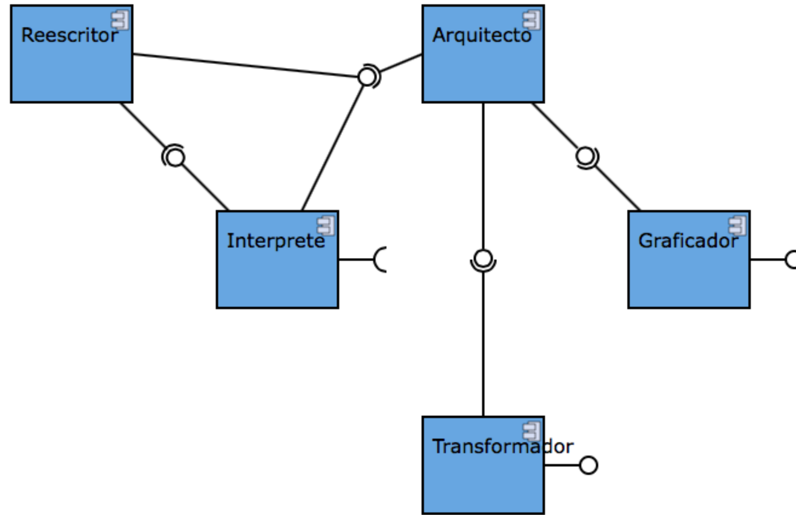

Figura 3.54. Arquitectura de PintArq

Seguido se realiza la exportación a XMI y la aplicación genera el siguiente documento XML:

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="PintArq" exporterVersion="1.0"/>
<uml:Model xmi:type="uml:Model" name="Modelo_PintArq Architecture.tex" visibility="public">

<ownedMember xmi:type="uml:Package" xmi:id="PintArqPK_-682759381" name="Arquitectura"
visibility="public">
        <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_404669720"
name="Interprete" visibility="public">
                <required xmi:id="PINTARQID_req_2" name=""/>
        </ownedMember>
        <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_-830736967"
        name="Arquitecto" visibility="public">
                <ownedConnector xmi:type="uml:Connector" xmi:id="PINTARQID_CONEXION_3"
                name="3" visibility="public" kind="assembly">
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_3"
                        role="PINTARQID_comp_-830736967"/>
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_3"
                        role="PINTARQID_comp_404669720"/>
                </ownedConnector>
                <ownedConnector xmi:type="uml:Connector" xmi:id="PINTARQID_CONEXION_4"
                name="4" visibility="public" kind="assembly">
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_4"
                        role="PINTARQID_comp_-830736967"/>
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_4"
                        role="PINTARQID_comp_758190782"/>
                </ownedConnector>
        </ownedMember>
        <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_-273483824"
        name="Graficador" visibility="public">
                <ownedConnector xmi:type="uml:Connector" xmi:id="PINTARQID_CONEXION_5"
                name="5" visibility="public" kind="assembly">
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_5"
                        role="PINTARQID_comp_-273483824"/>
                        <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_5"
                        role="PINTARQID_comp_-830736967"/>
                </ownedConnector>
                <provided xmi:id="PINTARQID_prov_6" name=""/>
        </ownedMember>
        <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_-917527918"
name="Transformador" visibility="public">
```

```
            <ownedConnector xmi:type="uml:Connector" xmi:id="PINTARQID_CONEXION_7"
            name="7" visibility="public" kind="assembly">
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_7"
                   role="PINTARQID_comp_-917527918"/>
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_7"
                   role="PINTARQID_comp_-830736967"/>
            </ownedConnector>
            <provided xmi:id="PINTARQID_prov_8" name=""/>
     </ownedMember>
     <ownedMember xmi:type="uml:Component" xmi:id="PINTARQID_comp_758190782"
     name="Reescritor" visibility="public">
            <ownedConnector xmi:type="uml:Connector" xmi:id="PINTARQID_CONEXION_9"
            name="9" visibility="public" kind="assembly">
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_9"
                   role="PINTARQID_comp_758190782"/>
                   <end xmi:type="uml:ConnectorEnd" xmi:id="PINTARQID_CONEXION_9"
                   role="PINTARQID_comp_404669720"/>
            </ownedConnector>
     </ownedMember>
</ownedMember>
</uml:Model>
</xmi:XMI>
```

Código 3.51. Código XML con definición de la arquitectura de PintArq en formato XMI.

Al tomar este documento e importarlo sobre un proyecto existente en la herramienta Enterprise Architect en su versión 7.5. se observa en el árbol de objetos que los componentes y las interfaces son creadas correctamente.

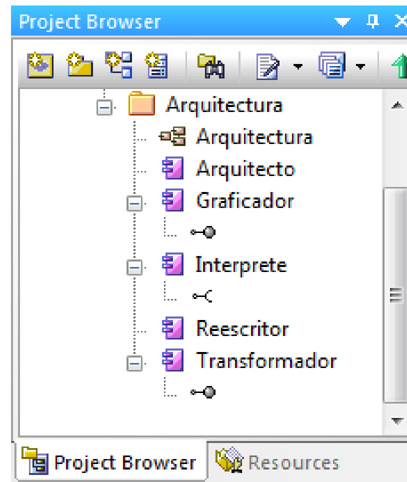

Figura 3.55. Componentes importados en Enterprise Architect.

Adicionalmente, para verificar que la configuración arquitectural sea la misma, se crea un diagrama incluyendo los componentes del árbol y se obtiene el siguiente diagrama:

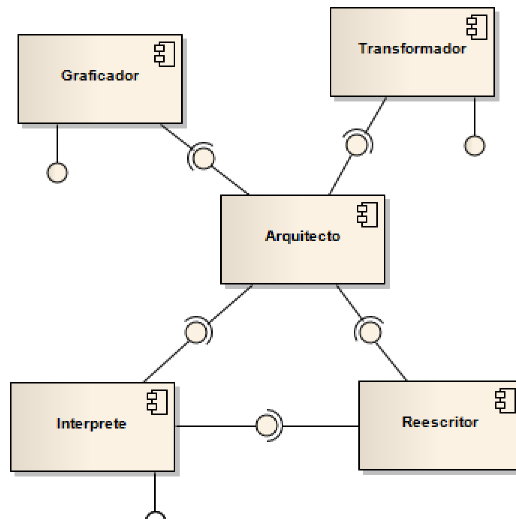

Figura 3.56. Arquitectura PintArq al importar XMI en Enterprise Architect.

Como se puede observar, la configuración tiene una diferencia respecto al diagrama presentado en la Figura 3.54 ya que allí se muestra una interfaz de requerimiento provista por el Arquitecto y la cual es utilizada por el Interprete y el reescritor, sin embargo, en UML no es posible a partir de la configuración arquitectural expresar el uso compartido de una interfaz y por tanto crea dos interfaces de requerimiento en el Arquitecto.

Capítulo 4
Conclusiones y aportes

En este capítulo se mostrarán las conclusiones obtenidas durante el proceso de investigación realizado para este proyecto al respecto de la interpretación de expresiones escritas usando el cálculo ρ_{arq} y la representación gráfica de arquitecturas basadas en componentes. Adicionalmente, se mostrarán los módulos y algoritmos aportados en este trabajo.

4.1 INTERPRETACIÓN DE EXPRESIONES

Durante el desarrollo del proyecto se identificó la importancia de la interpretación para lograr los objetivos de la investigación y por esta razón, el mayor esfuerzo del proyecto se concentró en las tareas de interpretación de expresiones definidas con el cálculo ρ_{arq}. En dicho esfuerzo, fue posible establecer que a partir de una descripción con cálculo ρ_{arq} de una arquitectura basada en componentes es posible identificar los elementos estructurales de dicha arquitectura y teniéndolos plenamente identificados y clasificados, estos pueden ser representados utilizando la notación gráfica propuesta por UML. Para identificar y clasificar adecuadamente estos elementos fue necesario establecer un mecanismo que permitiera identificar los tipos de expresión empleados en la descripción, para lo cual se empleó el editor de texto científico Latex y el motor de interpretación ANTLR, herramientas con las cuales fue posible desarrollar un conjunto de algoritmos para identificar los tipos de expresiones y a partir de estos establecer la configuración de los elementos estructurales empleados en la descripción de una arquitectura, particularmente, las conexiones entre componentes descritas a partir de aplicaciones reducidas para las cuales es posible la aplicación de la semántica operacional del cálculo. Así mismo, fue posible la identificación de los componentes al detectar las ecuaciones y con ellas los identificadores que tienen una definición asociada en la descripción de la arquitectura.

Otro aspecto asociado al cálculo corresponde a las características

comportamentales de una arquitectura, las cuales son definidas haciendo uso de expresiones sobre las cuales es posible la aplicación de la semántica operacional provista por el cálculo ρ_{arq}. En este proyecto de investigación fue posible automatizar la aplicación de dicha semántica sobre expresiones que cumplen con las reglas sintácticas del cálculo, a través de la detección de patrones de escritura que establecen o permiten la aplicación de las reglas de reducción o reescritura. Aquí fueron establecidos patrones conformados por un conjunto de expresiones bajo una estructura específica de escritura los cuales aplican para la reescritura de expresiones y definen un cambio de estado en una arquitectura; de tal manera, que la herramienta desarrollada puede identificar dichos patrones de especificación y aplicar automáticamente las reglas de reducción para transformar la especificación del sistema en una nueva expresión que describe un nuevo estado de su flujo de ejecución, y así permitir la identificación automática de los cambios de estado en el flujo de ejecución de las arquitecturas descritas con el cálculo. Todas estas actividades fueron incluidas en un algoritmo que identifica y reescribe aquellos patrones definidos en el proyecto, automatizando las reglas de reducción del cálculo.

4.2 REPRESENTACIÓN GRÁFICA

El objetivo principal de este trabajo consistió en facilitar el análisis del flujo de una arquitectura al proveer una notación gráfica basada en UML 2.X sobre la cual el arquitecto pueda identificar fácilmente los elementos que están interactuando y de esta manera determinar si el flujo de ejecución desplegado es el flujo esperado. Para lograr esto, fue necesaria la construcción de un conjunto de algoritmos que usando SVG (Scalable Vector Graphics) despliegan la configuración de la arquitectura en análisis de manera gráfica. Dichos algoritmos además de representar los elementos estructurales de la arquitectura con la notación gráfica de UML, se encargan de considerar el espacio disponible para el despliegue del modelo, distribuyendo los elementos gráficos en dicho espacio de tal forma que los elementos de la arquitectura no queden sobrepuestos y puedan ser fácilmente reconocidos por el arquitecto.

Estos algoritmos fueron diseñados para organizar y distribuir los elementos de la arquitectura considerando las conexiones definidas por las interfaces conectadas, así, ubica inicialmente aquellos componentes con el mayor número de conexiones y alrededor de él, haciendo uso de coordenadas polares ubica los elementos con los cuales tiene algún tipo de conexión, dentro de este proceso también se logró construir un algoritmo que identificara si un componente a ubicar comparte espacio de despliegue con otro (sobreposicion) o si su interfaz tiene intersección con otro elemento, y además solucionar la sobreposición de dichos elementos teniendo en cuenta un número predeterminado de intentos de solución.

4.3 APORTES ORIGINALES

El aporte de este trabajo de investigación está basado en la contrucción de algoritmos que en conjunto permiten el despliegue gráfico de una arquitectura de software basada en componentes. Asi, a continuación un listado de los algoritmos desarrollados por el autor.

- Reescritura del cálculo ρ_{arq}.

Este algoritmo tiene la capacidad de identificar patrones de escritura que cumplan con las reglas de reducción propuestas en la semántica operacional del cálculo y transformarlas en las expresiones equivalentes definidas por dichas reglas. Esta transformación se lleva a cabo en un árbol de expresiones construido por la funcionalidad de ANTLR[67]. Este algortimo se encuentra descrito de manera general en la sección 3.4.2 con los diagramas de secuencia de las figuras Figura 3.16, Figura 3.17 y Figura 3.18.

- Identificación de elementos arquitecturales.

Para este proyecto fue desarrollado un algoritmo capaz de identificar patrones de escritura que describen elementos arquitecturales tales como componentes e interfaces tanto conectadas como desconectadas. Así mismo, este algoritmo también tiene la capacidad de identificar composición jerárquica de componentes al reconocer los elementos que definen una subarquitectura. Este algoritmo se describe de manera general con el diagrama de secuencia de la Figura 3.23.

- Distribución de elementos gráficos

Para desplegar gráficamente la arquitectura descrita haciendo uso del cálculo ρ_{arq}, fue necesaria la implementación de un algoritmo que organice y distribuya todos los elementos estructurales identificados en una arquitectura. Dicho algoritmo realiza la ubicación de los elementos dentro de un área de la pantalla de tal forma que no exista sobreposición en su despliegue. De manera general este algoritmo está descrito en la Figura 3.34.

Capítulo 5
Trabajo Futuro

A partir del trabajo realizado en este proyecto de investigación se lograron identificar tres campos de acción, los cuales tomando como base el resultado de este proyecto podrían fortalecer el objetivo principal de facilitar el uso del cálculo ρ_{arq}.

La primera posibilidad de trabajo radica en la construcción de una herramienta gráfica de edición de arquitecturas de software y escritura en cálculo ρ_{arq}, que permita de manera fácil y con una interfaz amigable incluir de manera intuitiva los elementos estructurales y de configuración de una arquitectura de software y que a su vez dependiendo de la configuración definida por el arquitecto en la interfaz, genere la definición en cálculo ρ_{arq} de la arquitectura de software establecida. De esta forma sería posible unir dicho producto con el resultado de este proyecto para que el arquitecto defina de manera gráfica una arquitectura y pueda ver de manera instantánea el flujo de ejecución sin tener que traducir o definir dicha arquitectura con el cálculo ρ_{arq}.

Otra de las posibilidades de mejora radica en el despliegue gráfico de la composición jerárquica definida en una arquitectura, ya que el proyecto es capaz de identificar los elementos (y su configuración) que hacen parte de una subarquitectura, sin embargo, al momento del despliegue solamente refleja el componente y no la subarquitectura definida para dicho componente. Al incluir esta característica en la herramienta, el arquitecto sería capaz de observar el comportamiento en conjunto del sistema y sus partes, brindando más posibilidades de análisis para el arquitecto ya que se despliegan tanto los componentes de la arquitectura principal como los de las subarquitecturas y podría verse el comportamiento de las interaccciones entre componentes de la arquitectura y de sus subarquitecturas. Por el momento en la versión actual es posible la revisión del comportamiento de dichas subarquitecturas si son cargadas independientemente como arquitecturas en la aplicaión, así el arquitecto tendrá que analizarlas por separado.

Finalmente, se identificó la posibilidad de desarrollar herramientas de análisis para identificación automática de abrazos mortales y verificación de algunas propiedades del software que podrían ser revisadas al emplear un lenguaje de especificación formal y al tener el flujo de ejecución identificado y representado simbólicamente mediante

expresiones provistas por ANTLR. Dichas funcionalidades ampliarían las posibilidades de análisis brindadas al Arquitecto e incentivaría el uso de este tipo de herramientas en la definición de arquitecturas de software.

BIBLIOGRAFÍA

[1] D. C. Luckham, “Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events.,” Stanford University, Stanford, CA, USA, 1996.

[2] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cimpan, B. Warboys, B. Snowdon, and R. M. Greenwood, “Support for evolving software architectures in the ArchWare ADL,” in *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, 2004, pp. 69–78.

[3] C. Chaudet and F. Oquendo, “pi;-SPACE: a formal architecture description language based on process algebra for evolving software systems,” in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, 2000, pp. 245–248.

[4] H. A. Diosa, “Especificación de un Modelo de Referencia Arquitectural de Software A Nivel de Configuración, Estructura y Comportamiento.,” Universidad del valle-Escuela de ingenieria de sistemas y computación, 2008.

[5] D. C. Craig and W. M. Zuberek, “Compatibility of Software Components - Modeling and Verification,” in *Dependability of Computer Systems, 2006. DepCos-RELCOMEX ’06. International Conference on*, 2006, pp. 11–18.

[6] K. S. Barber, T. Graser, and J. Holt, “Providing early feedback in the development cycle through automated application of model checking to software architectures,” in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, 2001, pp. 341–345.

[7] J. Ding, H. Zhu, H. Zhu, and Q. Li, “Formal Modeling and Verifications of Deadlock Prevention Solutions in Web Service Oriented System,” in *Engineering of Computer*

Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on, 2010, pp. 335–343.

[8] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri, "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies," *Softw. Eng. IEEE Trans.*, vol. 36, no. 1, pp. 119–140, 2010.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures," in *Proceedings of the 5th European Software Engineering Conference*, 1995, pp. 137–153.

[10] R. Agarwal and D. Umphress, "Extreme programming for a single person team," in *Proceedings of the 46th Annual Southeast Regional Conference on XX*, 2008, pp. 82–87.

[11] F. Ahmed and L. F. Capretz, "Best practices of RUP in software product line development," in *Computer and Communication Engineering, 2008. ICCCE 2008. International Conference on*, 2008, pp. 1363–1366.

[12] Q. Wang, G. Huang, J. Shen, H. Mei, and F. Yang, "Runtime software architecture based software online evolution," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, 2003, pp. 230–235.

[13] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[14] R. Kazman, L. Bass, and M. Klein, "The essential components of software architecture design and analysis," *J. Syst. Softw.*, vol. 79, no. 8, pp. 1207–1216, 2006.

[15] M. Shaw, R. DeLine, D. V Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *Softw. Eng. IEEE Trans.*, vol. 21, no. 4, pp. 314–335, 1995.

[16] A. K. Jones, "The Maturing of Software Architecture," in *Software Engineering Symposium*, 1993.

[17] B. Boehm, "Megaprogramming (abstract)," in *Proceedings of the 22nd annual ACM computer science conference on Scaling up : meeting the challenge of complexity in real-world computing applications: meeting the challenge of complexity in real-world computing applications*, 1994, p. 412–.

[18] D. Garlan and M. Shaw, "An Introduction to Software Architecture," Jan. 1994.

[19] G. Huang, H. Mei, and F. Yang, "Runtime software architecture based on reflective middleware," *Sci. China Ser. F Inf. Sci.*, vol. 47, no. 5, pp. 555–576, 2004.

[20] R. K. Len Bass Paul Clements, *Software architecture in practice*. Addison Wesley, 2003.

[21] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[22] E. W. Dijkstra, "On the role of scientific thought," *Sel. Writings Comput. A Pers. Perspect.*, pp. 60–66, 1982.

[23] R. J. Allen, "A formal approach to software architecture," Carnegie Mellon, School of computer Science, 1997.

[24] A. R and G. D, "A Formal Basis for Architectural Connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, pp. 213–249, 1997.

[25] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[26] D. Bryan, C. Luckham, J. John, L. Kenney, M. Augustin, J. Vera, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 336–355, 1995.

[27] D. C. Luckham, "The Stanford Rapide™ Project," 1998. [Online]. Available: http://complexevents.com/stanford/rapide/.

[28] T. R. Dean and J. R. Cordy, "A Sintactic Theory of Software Architecture," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 302–313, 1995.

[29] D. Garlan, R. Monroe, and D. Wile, "Acme: an architecture description interchange language," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, 1997, p. 7–.

[30] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An infrastructure for the rapid development of XML-based architecture description languages," in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 266–276.

[31] R. K. Pandey, "Architecture description languages (ADLs) vs UML: a review," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 3, pp. 1–5, May 2010.

[32] S. V. H. Gil, "Representación de la arquitectura de software usando UML," *Sist. y Telemática*, vol. 1, pp. 63–75, Jul. 2006.

[33] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum, “Integrating architecture description languages with a standard design method,” in *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, 1998, pp. 209–218.

[34] P. Inverardi and A. L. Wolf, “Formal specification and analysis of software architectures using the chemical abstract machine model,” *Softw. Eng. IEEE Trans.*, vol. 21, no. 4, pp. 373–386, 1995.

[35] G. Berry and G. Boudol, “The chemical abstract machine,” in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1990, pp. 81–94.

[36] J.-P. Banâtre, A. Coutant, and D. Le Metayer, “A parallel machine for multiset transformation and its programming style,” *Futur. Gener. Comput. Syst.*, vol. 4, no. 2, pp. 133–144, 1988.

[37] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., and J. E. Robbins, “A component- and message-based architectural style for GUI software,” in *Proceedings of the 17th international conference on Software engineering*, 1995, pp. 295–304.

[38] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, “A comprehensive approach for the development of modular software architecture description languages,” *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 2, pp. 199–245, 2005.

[39] J. Spencer, “Architecture Description Markup Language (ADML) Creating an Open Market for IT Architecture Tools,” Sep-2000. [Online]. Available: http://www.opengroup.org/architecture/adml/background.htm.

[40] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, “The Koala component model for consumer electronics software,” *Computer (Long. Beach. Calif).*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[41] M. M. Gorlick and R. R. Razouk, “Using weaves for software construction and analysis,” in *Software Engineering, 1991. Proceedings., 13th International Conference on*, 1991, pp. 23–34.

[42] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP’97 �Object-Oriented Programming*, vol. 1241, M. Aksit and S. Matsuoka, Eds. Springer Berlin / Heidelberg, 1997, pp. 220–242.

[43] M. Rong, “An Aspect-Oriented Software Architecture Description Language based on temporal logic,” in *Computer Science and Education (ICCSE), 2010 5th International Conference on*, 2010, pp. 91–96.

[44] Z. X.-Y. y Tang Zhi-Song, "A Temporal Logic-Based Software Architecture Description Language - XYZ/ADL," *J. Softw.*, vol. 14(4), pp. 713–720, 2003.

[45] J. M. Spivey, "The Z Notation: A Reference Manual," *2001*. [Online]. Available: http://spivey.oriel.ox.ac.uk/~mike/zrm/.

[46] J. R. Abrial and B. Meyer, "The Specification Language Z: Syntax and Semantics," R. M. McKeag and A. M. Macnaghten, Eds. New York, NY, USA: Cambridge University press, 1980.

[47] M. D. Rice and S. B. Seidman, "Using Z as a Substrate for an Architectural Style Description Language." Colorado State University, Fort Collins, 1996.

[48] S. B. Seidman and M. Rice, "Software Architecture," in *Computer Science Handbook*, A. B. Tucker, Ed. New Orleans: Chapman & Hall/CRC, 2004, pp. 109–1 a 109–17.

[49] M. D. Rice and S. B. Seidman, "A formal model for module interconnection languages," *Softw. Eng. IEEE Trans.*, vol. 20, no. 1, pp. 88–101, 1994.

[50] T. O. Group, "Archimate 2.0 Specification," 2012. [Online]. Available: http://www.opengroup.org/subjectareas/enterprise/archimate.

[51] B. Lewis, "Software portability gains realized with metah and ada95," *ADA Lett.*, vol. XXII, pp. 37–46, 2002.

[52] N. Medvidovic, R. N. Taylor, and E. M. Dashofy, *Software Architecture. Foundations, Theory, and Practice*. Crawfordsville (USA): John Wiley & Sons Inc., 2010.

[53] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Westford Massachusetts, United States: Addison-Wesley, 2013.

[54] R. Milner, *Communicating and mobile Systems: The π-Calculus*. New York, NY, USA: Cambridge University press, 1999.

[55] F. Oquendo, "Dynamic Software Architectures: Formally Modelling Structure and Behaviour with Pi-ADL," *2008 Third Int. Conf. Softw. Eng. Adv.*, 2008.

[56] F. Oquendo, "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 1–14, 2004.

[57] H. Cirstea, "Calcul de réécriture: fondements et applications," Université Henri Poincaré - Nancy I, 2000.

[58] H. A. Diosa, J. F. Diaz Frias, and J. F. Gaona Cuevas, “Cálculo para el modelado formal de arquitecturas de software basadas en componentes: Cálculo ρ-arq,” *Rev. científica Univ. Dist.*, vol. 12, pp. 172–184, 2010.

[59] A. M. Parra Torres, J. C. Reyes Ortega, and H. A. Diosa, “Diseño e implementación de una herramienta de traducción de XADL 2.0 a diagramas basados en componentes UML 2.0.,” Universidad Distrital Francisco José de Caldas, 2008.

[60] D. C. Donoso Casas, D. C. Serrano, and H. A. Diosa, “Diseño y desarrollo de un traductor sintáctico entre el lenguaje de descripción arquitectural XADL 2.0 y el cálculo para modelamiento formal de arquitecturas de software Rho-arq,” Universidad Distrital Francisco José de Caldas, 2008.

[61] Y. Singh and M. Sood, “Models and Transformations in MDA,” in *2009 First International Conference on Computational Intelligence, Communication Systems and Networks*, 2009, pp. 253–258.

[62] W. Zuo, J. Feng, and J. Zhang, “Model Transformation from xUML PIMs to AADL PSMs,” in *Computing, Control and Industrial Engineering (CCIE), 2010 International Conference on*, 2010, vol. 1, pp. 54–57.

[63] E. Guerra, J. de Lara, D. Kolovos, and R. Paige, “A Visual Specification Language for Model-to-Model Transformations,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, 2010, pp. 119–126.

[64] P. Terrence and K. S. Fisher, “LL(*): The Foundation of the ANTLR Parser Generator,” 2011.

[65] Q-Success, “Software pointers.” [Online]. Available: http://www.software-pointers.com/en-mda-tools.html.

[66] H. A. Diosa, J. F. Díaz Frías, and C. M. Gaona Cuevas, “Especificación formal de arquitecturas de software basadas en componentes: Chequeo de corrección con cálculo ρ-arq,” *Rev. científica Univ. Dist.*, vol. 12, pp. 156–171, 2010.

[67] P. Terence, *The definitive ANTLR 4 reference*. Dallas, Texas: The Pragmatic Programmers, LLC., 2012.

Anexo A.
Guía de uso de la herramienta

Este capítulo mostrará los requerimientos para la correcta operación de la herramienta y brindará una guía de la manera en la cual debe operarse la herramienta para observar la funcionalidad y obtener los resultados esperados de ella.

A.1. REQUERIMIENTOS TÉCNICOS

La herramienta fue implementada utilizando el lenguaje de programación Java en su versión 1.6 bajo la filosofía de aplicación web, por tanto, es requerido un servidor de aplicaciones compatible con java 1.6 o superior y para su visualización, se requiere un navegador de internet compatible con HTML 5 (las pruebas de la herramienta fueron realizadas usando Safari v.7.1.6 y Google Chrome v.43.0.2357) el cual incluye el despliegue de gráficos mediante SVG (Scalable Vector Graphics).

En el desarrollo de la funcionalidad fueron empleadas un conjunto de librerias externas que deben estar instaladas en dicho servidor, estas se listan a continuación:

- API para java de ANTLR en su versión 4.1.
- Librería Commons FileUpload del proyecto Apache en su versión 1.3.1.

A.2. INSTALACIÓN

Dado que la herramienta fue desarrollada como una aplicación web, su instalación puede variar dependiendo del servidor de aplicaciones empleado, sin embargo, en términos generales para la instalación de la aplicación solamente basta con el despliegue del paquete

de software contenido en el archivo *war* (*web archive*) en la carpeta de publicación del respectivo servidor.

Es posible que dependiendo del servidor de aplicaciones, la estructura del paquete de software requiera modificación, para lo cual se sugiere hacer referencia al procedimiento de construcción de paquetes java para el respectivo servidor de aplicaciones.

El paquete entregado con este proyecto, es un paquete *war* que puede ser instalado en cualquier servidor de aplicaciones con contenedores compatibles para Java edición empresarial. Para las pruebas fue utilizado el servidor Jetty versión 8.1.10 provisto por el IDE eclipse empleado para el desarrollo.

A.3. PANTALLA DE OPERACIÓN

La aplicación tiene una única plantilla visual que será presentada durante todo el proceso de análisis de arquitecturas. Dicha plantilla de presentación es como lo muestra la siguiente figura:

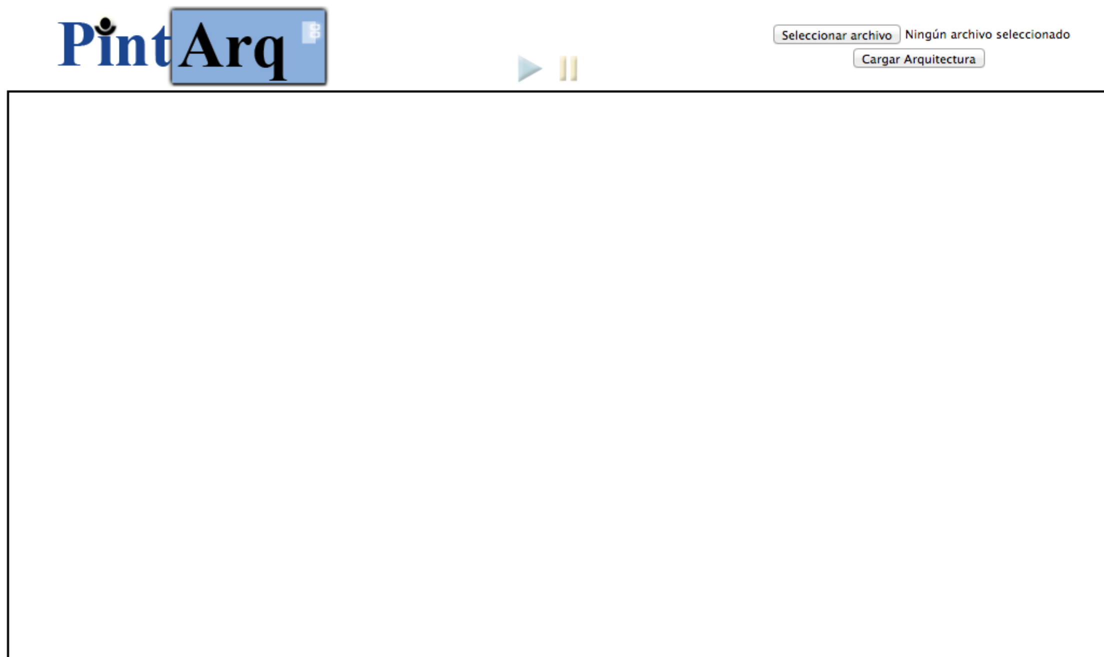

Figura A.5.1. Esquema visual de la herramienta.

La pantalla está dividida en dos zonas, la zona superior donde se despliegan los controles para la operación de las arquitecturas y la zona de despliegue donde se podrá observar la configuración de la arquitectura descrita e ingresada a la aplicación.

En la zona superior existen tres regiones, de izquierda a derecha, la zona de logo,

la zona de ejecución y la zona de cargue y exportación. En la zona de ejecución se pueden observar dos controles para la ejecución y pausa del flujo de ejecución de la arquitectura cargada, incialmente estos controles lucirán deshabilitados, sin embargo en la medida en que estos puedan ser utilizados serán habilitados para su uso.

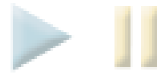

Figura A.5.2. Controles para la ejecución de arquitecturas.

Los controles de ejecución permiten el inicio y la detención del flujo de ejecución de la arquitectura en análisis. El triángulo azul una vez es presionado permite que la arquitectura interactue y cambie de estados según la descripción de esta y el simbolo con dos lineas paralelas permite detener temporalmente el flujo de ejecución y observar el estado de la arquitectura para ese momento.

La zona de cargue de arquitecturas consta de un conjunto de botones que permiten la carga de arquitecturas en la aplicación y la exportación al formato XMI.

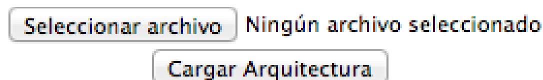

Figura A.5.3. Controles para carga de arquitecturas

Inicialmente esta zona solamente permitirá la selección de un archivo con la descripción arquitectural y la carga del archivo al sistema mediante los botones “Seleccionar archivo” y “Cargar Arquitectura” respectivamente. Una vez una arquitectura ha sido cargada en el sistema se mostrará el nombre del archivo carga y se permitirán dos acciones: Revisar otra arquitectura y exportar a XMI.

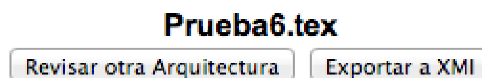

Figura A.5.4. Controles para el cambio de arquitectura y la exportación a XMI.

Con la opción “revisar otra arquitectura” se desplegarán de nuevo los controles para selección de un archivo y cargue de dicha arquitectura, con la opción exportar XMI, se iniciará la descarga de un archivo XML con la especificación del modelo visualizado.

A.4. ANÁLISIS DE ARQUITECTURAS

Para realizar el análisis de una arquitectura descrita con cálculo ρ_{arq}, es necesario contar con su descripción usando la representación en LaTex de dicha arquitectura. Con dicho archivo, se debe proceder a cargarlo en la aplicación a través de los controles provistos en la zona de selección y carga de arquitecturas.

Una vez la arquitectura es cargada en la aplicación, se procede a realizar la interpretación y el análisis de dicho archivo explicados en los módulos intérprete y arquitecto y luego mostrar en la zona de despliegue la configuración identificada de la arquitectura.

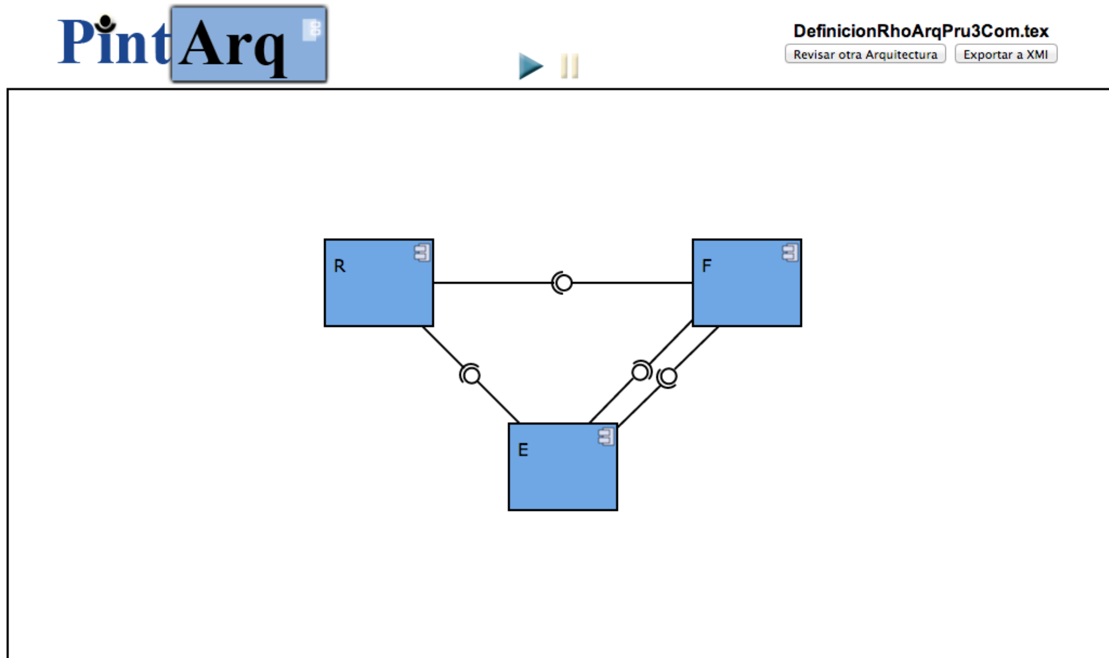

Figura A.5.5. Cargue de arquitecturas en la aplicación.

Una vez desplegada la configuración es posible iniciar el flujo de ejecución de la arquitectura cargada, para esto se deben utilizar los controles de ejecución (Figura A.2) ubicados en la parte superior de la pantalla. Al presionar el botón para el inicio de la ejecución de la arquitectura, los estados de las interfaces cambian de acuerdo al comportamiento descrito en la definición con el cálculo ρ_{arq}. Si durante la ejecución se desea detener el flujo es posible hacerlo presionando el botón de pausa, el cual es habilitado una vez inicia la ejecución de la arquitectura.

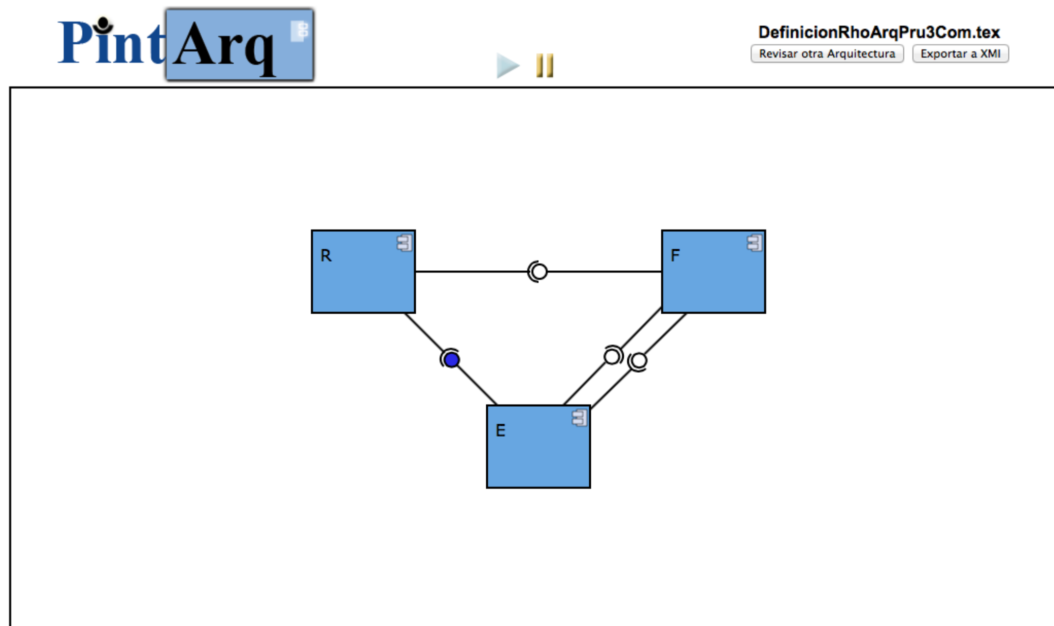

Figura A.5.6. PintArq durante la ejecución de una arquitectura

Durante la ejecución será posible la visualización de las interfaces que interactúan ya que se resalta la interfaz de provisionamiento rellenando el circulo que la representa con el color azul rey como se puede observar en la Figura A.6. Una vez finaliza la interacción entre las interfaces, el circulo de la interfaz regresa a su representación original permitiendo ver al arquitecto los momentos de interacción entre las interfaces y la secuencia que tendría el flujo de ejecución para la arquitectura descrita sin necesidad de tener que aplicar manualmente la semántica operacional del cálculo y facilitando la identificación de flujos no deseados en la arquitectura en construcción.

A.5. EXPORTACIÓN A XMI

La herramienta permite que el arquitecto pueda almacenar la configuración de la arquitectura descrita usando el lenguaje de intercambio de modelos en XML XMI, así, es posible revisar dicha configuración en herramientas compatibles con UML 2.X. que permitirán complementar la documentación o la ejecución de otros tipos de análisis.

Para llevar a cabo el proceso de exportación solo basta con presionar el botón con la etiqueta “Exportar a XMI” descrito con la Figura A.4. luego de la carga de una arquitectura. Esta acción revisará la configuración de la arquitectura cargada e iniciará el proceso de representación bajo la sintaxis XMI de los elementos empleados en la definición de la arquitectura. Por defecto el modelo quedará creado bajo un paquete denominado con el mismo nombre del archivo cargado. Luego de la construcción del archivo la herramienta iniciará una descarga la cual permitirá almacenar el archivo XMI en el computador del usuario