PASCANI: Lenguaje de dominio específico para especificar y ejecutar pruebas automatizadas, componibles y rastreables en sistemas de software basados en componentes SCA

Miguel Ángel Jiménez Achinte - miguel.jimenez@correo.icesi.edu.co
Fabián Andrés Caicedo Cuellar - fabian.caicedo@correo.icesi.edu.co

Trabajo de grado

Directores:

Angela Villota Gómez, MSc
apvillota@icesi.edu.co

Gabriel Tamura Morimitsu, PhD
gtamura@icesi.edu.co

Universidad Icesi

Facultad de Ingeniería - Departamento de TIC
Ingeniería de Sistemas
Cali, Colombia

2013

1. Tabla de Contenido

2. Introducción

Uno de los objetivos de décadas de investigación y aprendizaje en el desarrollo de software ha sido encontrar procesos y metodologías que sean sistemáticas, predecibles y repetibles; esto con el fin de mejorar la productividad en el desarrollo y la calidad del producto de software [1]. Con base en este conocimiento adquirido, se han planteado modelos, metodologías y filosofías de desarrollo de software. Cada una de estas con diferentes propuestas y enfoques.

Una de las etapas más importantes consiste en monitorear los procesos de ingeniería de software y métodos para asegurar la calidad. La obtención de un producto de software de calidad, implica la utilización de metodologías o procedimientos estándar para la elicitación de requerimientos, análisis, diseño, implementación y pruebas de software [2]. Estos procedimientos permiten uniformar una metodología de trabajo en aras de lograr un conjunto de atributos deseables en el software, que se refieren a su calidad; es decir, un conjunto de requerimientos funcionales y no funcionales que apuntan al proceso de construcción y se ven reflejados en el software.

Se llama aseguramiento de la calidad (SQA) al conjunto de actividades planificadas y sistemáticas, necesarias para asegurar que el producto de software va a satisfacer los requisitos de calidad, como la correctitud de la solución, la escalabilidad, la estabilidad, el desempeño, entre otros.

Uno de los puntos clave en el contexto del aseguramiento de la calidad, es la etapa de pruebas de software. En la actualidad, es crucial verificar aspectos determinantes en la correctitud del software y en su correspondencia con el cumplimiento de los objetivos del negocio. Al realizar diferentes tipos de pruebas sobre el producto, los ingenieros de software se aseguran de reducir el riesgo que representa el descubrimiento de posibles fallos o comportamientos inesperados en un ambiente de producción. La aparición de dichas fallas podría generar un alto impacto sobre las organizaciones, y acarrear pérdidas de dinero y tiempo, además de daños en la reputación de los negocios; incluso lesiones físicas o la muerte, en sistemas críticos como software de control de tráfico aéreo, control de luces de señalización de tráfico vehicular, procesamiento de imágenes con fines de diagnóstico y planeación de tratamientos médicos, entre muchos otros.

El proceso que abarca el conjunto completo de pruebas, entre otros conceptos, se denomina Validación y Verificación (V&V). Este es un proceso de soporte que está estrechamente vinculado al proceso SQA, de manera tal que ambos se consideran complementarios. V&V Consiste en determinar si un producto intermedio satisface: (i) el conjunto de requerimientos especificados en el proceso de elicitación de requerimientos, y (ii) las necesidades reales del cliente y/o los usuarios [4].

Sin embargo, es necesario conocer de qué se trata Validación y Verificación para así poder establecer en qué se diferencian y cuál es el enfoque de cada una. En el primer caso, el interrogante que se intenta resolver es ¿estamos construyendo el producto correctamente?, y su principal objetivo es comprobar que el software cumple los requerimientos funcionales y los no funcionales. En el segundo caso, se debe

plantear el interrogante ¿estamos construyendo el producto correcto?, donde el objetivo es comprobar que se cumple las expectativas del cliente[1].

Actualmente, el papel de V&V en los sistemas computacionales tiene lugar durante las etapas de diseño e implementación, es decir que hace referencia a un conjunto de etapas que van de la mano con el proceso de desarrollo de software [7]; en otros términos, hasta antes de que el producto final salga de la etapa de desarrollo. Sin embargo, un enfoque diferente plantea el uso de V&V para garantizar que los cambios arquitectónicos realizados a un sistema, en tiempo de ejecución, no afectan de ninguna manera las propiedades externamente visibles ni los requerimientos funcionales y no funcionales especificados inicialmente; el desarrollo de nuevos métodos de V&V para garantizar el logro de los objetivos, en este tipo de sistemas de software, representa uno de los principales desafíos en el campo de la investigación.

Los cambios arquitectónicos mencionados en el párrafo anterior corresponden, en general, a sistemas de software con altos niveles de adaptación y autonomía, cuya característica principal es mantener satisfechos los objetivos cambiantes en las organizaciones [8]. Para que estos objetivos sean alcanzados, el software debe estar en capacidad de adaptar uno o varios de sus componentes[1] de manera autónoma. De acuerdo a [8], los sistemas de software que se auto-adaptan deben desempeñar cuatro tareas principales: monitorear los nuevos requerimientos del negocio, tanto en el contexto externo como en el interno, analizar los datos de entrada para identificar síntomas y necesidad de adaptación, planear un conjunto de acciones de adaptación y ejecutar esas acciones sobre el sistema.

Uno de los principales retos de la auto-adaptación, es crear la habilidad en el sistema para razonar sobre sí mismo y sobre su contexto [8]; este es el principio que le permite tomar decisiones sobre las adaptaciones que el sistema debe desempeñar. Por lo tanto, este debe conocer las salidas de sus acciones y usarlas como retroalimentación, para tomar la decisión correcta a ejecutar [8]. Sin embargo, el uso de la frase *decisión correcta* sugiere que el sistema es regido por un conjunto de lineamientos que limitan sus acciones, o más bien las *conducen*; entonces, debería existir un mecanismo de validación y verificación que actúe como juez de los planes de adaptación.

Este tipo de comprobación requiere procedimientos de pruebas no estáticas, debido a la naturaleza cambiante de los sistemas auto-adaptativos. Por lo tanto, dado que los cambios arquitectónicos (correspondientes al plan de adaptación) son implementados y puestos en marcha durante la ejecución del sistema, los procedimientos de pruebas propuestos deben proporcionar mecanismos dinámicos de especificación y ejecución, mientras el sistema está en ejecución.

[1] Es la abstracción de una función de negocio determinada

3. Planteamiento del problema

El desarrollo de software obedece actualmente a la planeación y ejecución de un conjunto de etapas en las cuales los ingenieros de software agregan valor al producto de software y se aseguran de que este cumpla con los objetivos para el cual fue diseñado. Estos objetivos no son más que un conjunto de requerimientos funcionales y no funcionales que nacen de la preocupación de uno o varios actores en el contexto de la organización.

Debido a la naturaleza cambiante de los objetivos del negocio y las constantes variaciones en el contexto de los productos de software, la comunidad científica [8, 14] ha incursionado en el desarrollo de nuevos sistemas de software capaces de evolucionar en el tiempo, de acuerdo a las presiones que ejerce el contexto sobre las propiedades externamente visibles en el software o la demanda de nuevos requerimientos; este tipo de sistemas recibe el nombre de auto-adaptativo, y trae consigo una serie de retos que se deben afrontar con el fin de garantizar su correcto funcionamiento, entre los cuales podemos mencionar: reconfiguración dinámica del sistema que preserve los contratos de calidad de servicio [14], V&V en tiempo de ejecución [7], mecanismos de retroalimentación que permitan identificar oportunamente la necesidad y/o síntomas de adaptación [8], entre otros.

Una de las propuestas para el cubrimiento de nuevos requerimientos en el sistema se lleva a cabo mediante la ejecución de tareas de reconfiguración estructural en la arquitectura del software [14]. Estos cambios son posibles gracias a la especificación de contratos de calidad de servicio y se realizan mientras el sistema se encuentra en ejecución. Por lo tanto, esto último sugiere la existencia de mecanismos de pruebas no sólo durante el período de producción del software, sino también en un ambiente de operación; dado que el plan inicial de pruebas ya no puede garantizar la correctitud del software y la correspondencia con los requerimientos. Sin embargo, una propuesta de solución a esta necesidad debe tener en cuenta los componentes principales de los sistemas auto-adaptativos, con el fin de aprovechar de la mejor manera los recursos del sistema y no interrumpir ni entorpecer el cumplimiento de los objetivos del negocio.

Diseñar planes de pruebas para un sistema que se auto-adapta de acuerdo a los cambios en su contexto es todo un reto para los ingenieros de software, dado que esto podría generar un número infinito de configuraciones en la estructura de su arquitectura. Por lo tanto, cuando se habla de ejecución de pruebas mientras el sistema está en funcionamiento (a lo que se le conoce como V&V en tiempo de ejecución) se debe hablar también de un mecanismo de limitación de lo que se considera correcto y de lo que no, en un cambio arquitectónico.

Como respuesta a la necesidad de validar y verificar que los cambios arquitectónicos que planea un sistema auto-adaptativo están enmarcados en un conjunto de lineamientos que garantizan el cumplimiento de los objetivos del negocio, y a la falta de una propuesta, se propone este proyecto.

4. Objetivos

4.1. Objetivo general

Desarrollar e implementar un prototipo funcional de validación y verificación en tiempo de ejecución, capaz de verificar que los cambios estructurales en la arquitectura de un sistema de software auto-adaptativo objetivo cumplen con un conjunto de requerimientos funcionales especificados.

4.2. Objetivos específicos

- Proponer un conjunto de reglas léxicas, sintácticas y semánticas que permitan especificar requerimientos funcionales para generar y ejecutar clases de prueba, en tiempo de ejecución.

- Implementar un prototipo funcional de un componente de software que permita realizar validación y verificación de un sistema auto-adaptativo en tiempo de ejecución, empleando un conjunto de especificaciones usando las reglas léxicas, sintácticas y semánticas del lenguaje propuesto.

- Integrar el prototipo funcional propuesto al caso de estudio *A variability model for generating SCA-based matrix-chain multiplication software products.*

5. Marco Teórico

5.1. Aseguramiento de la calidad mediante pruebas

El aseguramiento de la calidad (SQA, por su nombre en inglés *Software Quality Assurance*) está compuesto de un conjunto de actividades entre las cuales se involucran procesos de diseño, codificación, pruebas y mantenimiento. Las pruebas de software se descomponen en los siguientes niveles: fundamentos de las pruebas de software, niveles de prueba, técnicas de pruebas, medidas de las pruebas y por último el proceso de pruebas. El primer nivel, fundamentos de pruebas de software, abarca las definiciones básicas del área de pruebas. El segundo nivel, relativo a los niveles de prueba, se refiere al nivel de profundidad con el que se realizan las pruebas al software, y está estrechamente ligado al modelo de desarrollo en V, que se explicará más adelante. El tercer nivel, las técnicas de pruebas, es el marco de referencia que guía al desarrollador de pruebas mediante un método o metodología al diseñar las pruebas. El cuarto nivel, las medidas de las pruebas, abarca la etapa de medición posterior a la evaluación de las pruebas en el software. Y finalmente, en el proceso de pruebas, que pertenece al quinto nivel, se revisan las consideraciones previas y actividades referentes a las pruebas.

5.2. Técnicas de pruebas que se realizan al software

El conjunto de las pruebas de software está compuesto por varias categorías, en las que se encuentran las pruebas de caja negra y pruebas de caja blanca. En la primera se hace referencia a casos de prueba que se basan solamente en el comportamiento de la entrada y la salida de datos, y la segunda está basada en la información acerca de como se ha diseñado el programa.

Sin embargo, esta no es la única clasificación que existe para las pruebas de software, hay otras categorías alternas que se centran en la base usada para la planeación y ejecución de los casos de prueba. Entre esas se encuentran:

- Pruebas basadas en la intuición o experiencia: estas pruebas están relacionadas directamente con la persona designada a su realización; es decir que según el conocimiento de ésta frente al desarrollo de la aplicación, se crean los posibles casos de prueba o la persona misma decide, según su experiencia y conocimiento, cómo los desarrolla.
- Técnicas basadas en la especificación: permiten definir un conjunto de reglas o dominios con los cuales los casos de prueba que se escogieron interactúan para su ejecución.
- Técnicas basadas en el código: para el desarrollo de este tipo de pruebas se toman segmentos de código o fragmentos que se van a probar por medio de la entrada y salida de datos.
- Técnicas basadas en errores: estas pruebas son usadas para encontrar posibles problemas o conflictos que se pueden presentar mientras el sistema está en ejecución.
- Técnicas basadas en la naturaleza del software: este conjunto de pruebas depende

directamente de los expertos que soportan el buen funcionamiento del software en su contexto, debido a su experticia. Por ejemplo, se necesita el conocimiento de un experto en cardiología para probar una aplicación referente al comportamiento del corazón, debido a que él es el único con el conocimiento requerido para crear las pruebas a partir de su experiencia.

- Técnicas de pruebas basadas en el uso: se tiene en cuenta la probabilidad de uso de la aplicación, mientras el usuario interactúa con ella para encontrar posibles fallos y ver con qué frecuencia suceden.
- Técnicas de selección y combinación: esta técnica da libertad de usar o combinar las mencionadas anteriormente; es decir, según el comportamiento del software y de su estructura, es posible combinar técnicas para garantizar el aseguramiento de la calidad.

5.3. Pruebas de regresión

En general, las pruebas de regresión son pruebas cuya ejecución ocurre de manera repetitiva en diferentes momentos; es decir, que el conjunto (o subconjunto) de pruebas se ejecuta sobre el software ya había sido ejecutado anteriormente. Esto con el fin de asegurar que las modificaciones o el desarrollo de nuevos requerimientos no han propagado efectos colaterales no esperados.
Adicionalmente, las pruebas de regresión son decisivas para determinar puntos de estabilidad en el software, de tal manera que al presentarse un efecto no deseado los desarrolladores puedan devolver (en ambientes de producción) el software a un estado anterior.

En el contexto de los sistemas auto-adaptativos, las pruebas de regresión pueden jugar un papel importante al determinar si un plan de adaptación, ya ejecutado o por ejecutar, cumple con el conjunto de requerimientos funcionales y no funcionales especificados inicialmente al desplegar el sistema.

5.4. Validación y Verificación

Validacion y Verificacion (V&V) agrupa un conjunto de procesos de comprobación y análisis que aseguran que el software se produce de acuerdo a las especificaciones dadas y cumple con las necesidades de los clientes.
Es importante tener en cuenta que los términos validacion y verificacion se refieren a diferentes perspectivas. En el primer caso, los ingenieros de software deben asegurarse de construir el software correctamente, y su principal objetivo es comprobar que éste cumple los requerimientos funcionales y los no funcionales. En el segundo caso, debe asegurarse que se está construyendo el producto que el cliente espera [1].

El proceso de validar y verificar el software tiene cabida durante cada una de las etapas del ciclo de vida del desarrollo [5]. En general, las fases de validación y verificación se clasifican, según el modelo de desarrollo de software en V, en: análisis de requerimientos, diseño del sistema, diseño de alto nivel o arquitectónico, diseño de bajo nivel o detallado y por último implementación [5] [6]. En este modelo,

dichas etapas se realizan de forma secuencial, por lo que el inicio de las etapas de prueba inicia con la última y se conoce como pruebas unitarias. Después, se diseñan y realizan las pruebas de integración, de sistema y de aceptación.

Como se puede observar en la Fig. 1, la propuesta de este modelo consiste en identificar un conjunto de puntos de control entre las fases del desarrollo, y plantear planes de ejecución de pruebas. Cada rectángulo de la imagen hace referencia a una etapa del ciclo de vida del software.

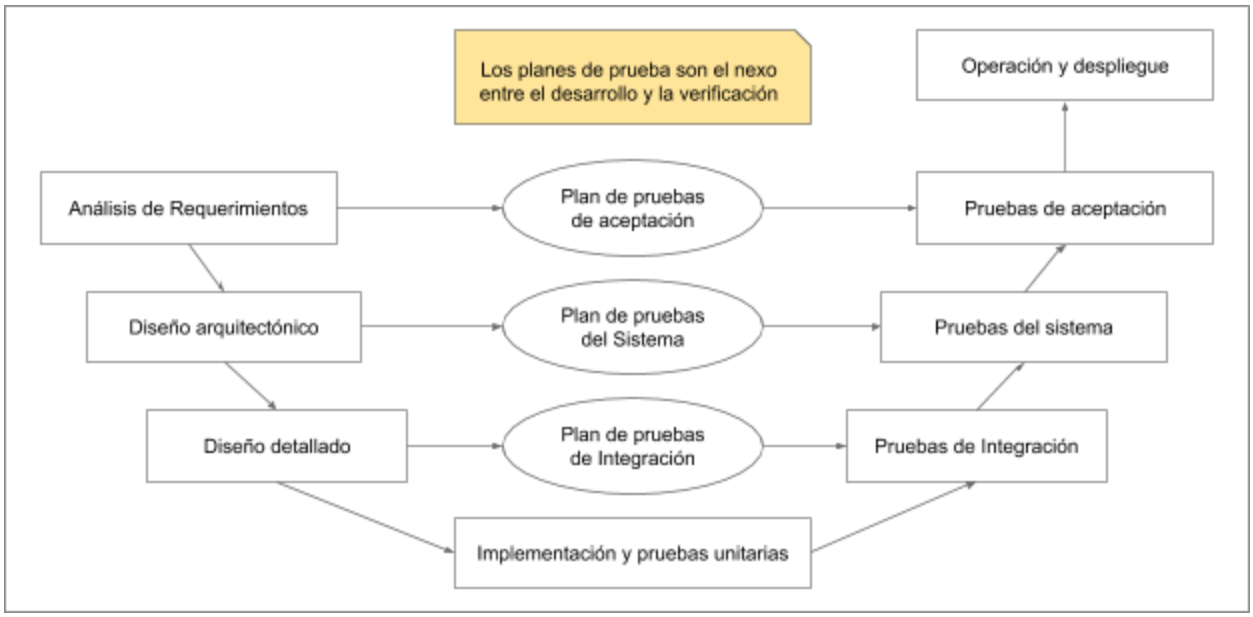


Figura No. 1. Modelo tradicional de desarrollo en V. Adaptado de [10].

Además de la clasificación por etapas del ciclo de vida, las pruebas diseñadas para validar y verificar el sistema se pueden clasificar en (i) inspecciones de software y (ii) pruebas de software [5]. Las primeras nacen de la preocupación por descubrir problemas analizando representaciones estáticas del sistema, lo que se conoce como verificación estática [5]. Y las segundas, se centran en la observación y el análisis del comportamiento cuando se opera el software con datos conocidos, llamada verificación dinámica [5]. Ambos enfoques pueden ser muy efectivos revelando fallos y evaluando la confiabilidad del sistema de software, pero no proporcionan evidencia sobre la ausencia de fallos [7]. Otras estrategias más efectivas y rigurosas para razonar sobre la correctitud del software emplean model checking, técnicas basadas en grafos, y otros métodos de pruebas basados en modelos formales [7].

5.5. Sistemas Autonómicos

La explosión simultánea de información, integración y nacimiento de nuevas tecnologías, y la continua evolución de los sistemas intensivos de software requieren nuevos e innovadores enfoques para construcción, ejecución y manejo de sistemas de software [11]. Una consecuencia que se desprende de esta necesidad, es que el software debe volverse más versátil, flexible, seguro, recuperable, y

configurable permitiendo la adaptación a cambios en su entorno operacional, ambiente o características del sistema.

De acuerdo a [12], el contexto puede ser definido como cualquier información que caracterice el estado de las entidades que afectan la ejecución, el mantenimiento y la evolución de los sistemas. Para los sistemas de software, el contexto es poco predecible debido a los constantes cambios que se presentan en las organizaciones; por lo tanto, sus sistemas de apoyo deben adaptarse para cumplir esos nuevos objetivos. Además, esta condición de frecuente adaptación exige a los ingenieros de software monitorear y analizar el sistema constantemente en busca de nuevas fallas. No obstante, llevar a cabo procesos de revisión y prevención contra fallas, en el software, hace parte de un costoso presupuesto de tecnologías de la información (IT). En consecuencia, toda propuesta que apunte solucionar el problema de la variabilidad en la correspondencia entre los cambios del entorno y la funcionalidad del software, debe considerar la reducción de costos mediante el uso de sistemas auto-administrables y auto-gestionables.

Con el fin de hacer esto, los sistemas deben tener las siguientes características [8, 13]:

- El sistema debe conocerse a sí mismo; por tanto, los componentes deben tener identidad.
- El sistema debe conocer el contexto que rodea su actividad.
- El sistema debe coexistir con componentes desconocidos; es decir, operar con estándares abiertos.
- El sistema debe conocer su propósito y los objetivos del negocio que persigue.

Con el fin de cumplir este comportamiento de autoadministración, los sistemas autonómicos deben implementar cuatro características fundamentales: auto-configuración, auto-recuperación, auto-protección y auto-optimización [8].

5.5.1. Autoconfiguración

La auto-configuración de un sistema se refiere a la capacidad de adaptarse a un entorno cambiante, de acuerdo con políticas de alto nivel alineadas con los objetivos de negocio propuestos por los administradores del sistema.

5.5.2. Autorecuperación

Se refiere a la capacidad de un sistema para recuperarse después de una interrupción en la ejecución. Además, se refiere a la minimización de las interrupciones para mantener el sistema de software disponible en el tiempo.

5.5.3. Autoprotección

La autoprotección de un sistema, es la capacidad para predecir, detectar, reconocer y protegerse contra ataques maliciosos y fallas no planificadas.

5.5.4. Auto Optimización

La auto-optimización se refiere a la capacidad del sistema de optimizar sus propios parámetros internos de funcionamiento, con el fin de maximizar o minimizar el cumplimiento de una función objetivo.

5.6. Especificación e Implementación de lenguajes de dominio específico

Un compilador, también llamado traductor, es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a una representación de significado equivalente en otro lenguaje, el lenguaje objeto [15].

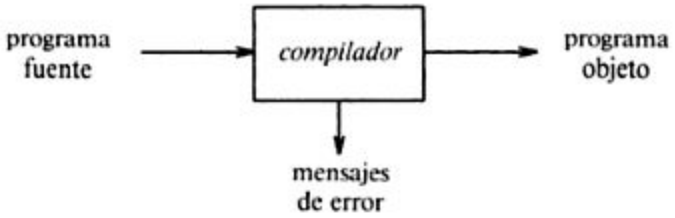


Figura No. 2.Proceso de la compilación

El proceso para construir un compilador se encuentra dividido en cuatro etapas [15]:

- Análisis léxico: transforma el programa fuente en tokens[2].
- Análisis sintáctico: construye un árbol sintáctico a partir de las secuencias de tokens.
- Análisis semántico: realiza el chequeo de tipos y la anotación semántica sobre los árboles sintácticos construidos.
- Generación de código: genera código en lenguaje objeto a partir de los árboles sintácticos chequeados y anotados semánticamente.

[2] Cadena de caracteres que tiene un significado coherente en cierto lenguaje de programación. Por ejemplo: Identificadores, palabras reservadas, operadores, y signos.

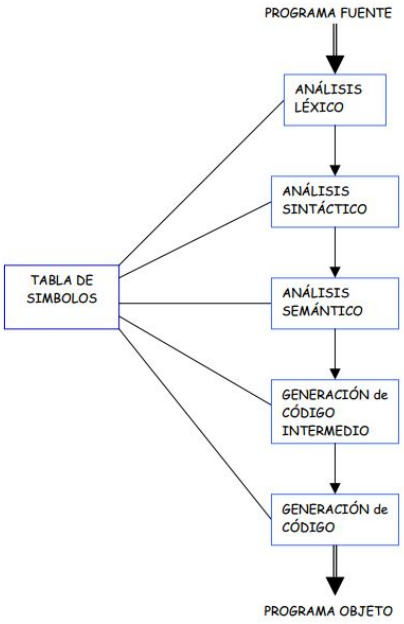


Figura No. 3. Etapas de un compilador

5.6.1. Análisis léxico

El análisis léxico es también conocido como escáner. Un analizador léxico lee caracteres del programa fuente, uno a uno, desde el archivo de entrada y va formando grupos de caracteres con alguna relación entre sí (*tokens*). Cada token es tratado como una única entidad, constituyendo la entrada de la siguiente fase del compilador [15].

El analizador léxico cumple con las siguientes tareas:

- Reconocer los *tokens* en la entrada.
- Eliminar los espacios en blanco, los caracteres de tabulación, los saltos de línea y de página y otros caracteres propios del dispositivo de entrada, según la definición de la gramática del lenguaje fuente.
- Eliminar los comentarios de entrada.
- Detectar errores léxicos.
- Reconocer los identificadores de variable, tipo, constantes, etc. y en coordinación con el analizador sintáctico guardarlos en la tabla de símbolos.
- Relacionar los mensajes de error del compilador con el lugar en el que aparecen en el programa fuente (línea, columna, etc.).

5.6.2. Análisis sintáctico

El analizador sintáctico, también llamado parser, recibe como entrada los tokens que genera el analizador léxico y comprueba si estos tokens van llegando en el orden correcto. El funcionamiento del analizador sintáctico del generador obtiene una cadena de componentes léxicos del analizador léxico y comprueba si la cadena puede ser generada por la gramática del lenguaje dado por el usuario.

5.6.3. Parser LALR

Un parser LALR o analizador sintáctico LALR es un tipo de parser definido como *Look-Ahead LR parser*, este tipo de parser se encuentra entre los LR(*L -> Left to right scanning, R -> Right-most derivation*), dicho parser hace reconocimiento bottom-up, es decir, reconoce de izquierda a derecha, primero las hojas y luego el nodo padre [15]. Los tipos de parser LR funcionan con una tabla de parseo y un stack, donde en el stack mantienen a los símbolos de la gramática.

Entre las ventajas de implementar el analizador LALR esta la de poder ser utilizado para definir varios lenguajes de programación gracias a la gramática LALR. Estos parsers son generalmente pequeños, rápidos y lineales en cuanto a la rapidez y la recuperación de errores. Los mensajes de error y el árbol sintáctico pueden estar incluidos previamente en el parser.

6. Análisis del problema

En esta sección se especifica la lista de requerimientos funcionales y no funcionales que debe satisfacer la solución al problema anteriormente descrito. Partiendo de los requerimientos, se analizan los stakeholders y sus preocupaciones más importantes a través de los casos de uso principales y de dos escenarios QAW (Quality Attribute Workshop). Y finalmente se presenta el diagrama de casos de uso contextual que representa las principales funciones de los roles Desarrollador de pruebas y Desarrollador de la especificación.

6.1. Requerimientos

Dada la descripción del problema, en este proyecto se concluyó que una solución versátil debe permitir a los desarrolladores de pruebas describir dos cuestiones importantes: i) la configuración de componentes actual del sistema y ii) la estrategia de pruebas. De tal manera que la estrategia de pruebas sea aplicable a diferentes estados del sistema (configuraciones de componentes diferentes) según corresponda. Para esto, y en concordancia con el primer objetivo específico, se propuso diseñar una especificación (reglas léxicas, sintácticas y semánticas) que le permitan al desarrollador de pruebas establecer los componentes y servicios del sistema, y diseñar casos de prueba que verifiquen el cumplimiento de los requerimientos.

Con base en lo anterior, este proyecto se desarrolló bajo los siguientes requerimientos:

1. La solución debe proveer una especificación que permita al desarrollador de pruebas diseñar una estrategia de pruebas sobre un sistema basado en componentes.
2. La especificación propuesta debe contener instrucciones que permitan describir la arquitectura del sistema a través de sus componentes y los servicios que éstos proveen y/o requieren.
3. La especificación propuesta debe considerar reglas sintácticas y semánticas en el dominio de las pruebas de software, en especial a sistemas de software basados en componentes.
4. Las instrucciones propias de la especificación propuesta deben reducir el trabajo del desarrollador de pruebas agrupando tareas atómicas en instrucciones ricas en significado (deben ser expresivas). Adicionalmente debe ser clara la separación de preocupaciones en la estructura sintáctica de la especificación.
5. La solución debe generar automáticamente las clases Java necesarias para ejecutar la estrategia de pruebas especificada por el desarrollador de pruebas.
6. La clases Java generadas deben estar escritas bajo el estándar Service Component Architecture (SCA).
7. La solución debe proveer un mecanismo que ejecute las clases Java generadas (modelo de ejecución) y exporte los resultados de todas las pruebas evaluadas a un archivo XML.
8. El modelo de ejecución debe ser compatible con el OW2 FraSCAti middleware.
9. El modelo de ejecución se limita a la ejecución de componentes de prueba a nivel local, no distribuido.

10. El modelo de ejecución debe contemplar dos tipos de ejecución: (i) cuando los componentes del sistema a probar ya han sido desplegados en la máquina virtual de FraSCAti y ii) cuando no han sido desplegados. En el segundo caso, el modelo de ejecución debe encargarse del despliegue antes de ejecutar los componentes de prueba.
11. La solución debe proveer al desarrollador de pruebas una interfaz gráfica que le permita generar las clases Java, ejecutarlas y visualizar los resultados obtenidos de la estrategia de pruebas.

6.2. Escenarios QAW

6.2.1. Stakeholders

Desarrollador de pruebas

El desarrollador de pruebas es el encargado de diseñar la estrategia de pruebas; es decir, crear módulos y escenarios de prueba para cada componente.

Preocupaciones principales:

- La solución debe generar clases Java sin errores, y el resultado de la ejecución de las pruebas debe reflejar la realidad del sistema.
- El tiempo de generación de las clases de prueba debe ser similar al tiempo de compilación del compilador de Java.
- El lenguaje debe ser lo suficientemente expresivo como para escribir cualquier tipo de prueba a los componentes del sistema.
- La especificación debe ser modular.

Desarrollador de la especificación

El desarrollador de la especificación es el encargado de desarrollar la solución y mantenerla.

Preocupaciones principales:

- La implementación de la solución debe ser mantenible.

6.2.2. Escenario No.1

Escenario	La especificación permite diseñar estrategias de pruebas para el 100% de los componentes y servicios del sistema.

Objetivo del negocio	Garantizar el cumplimiento de los requerimientos funcionales y no funcionales del sistema.
Atributos de calidad relevantes	Expresividad
Estímulo	Ejecución del generador de componentes y clases (compilador)
Fuente de estímulo	Desarrollador de pruebas
Entorno	Sistema operativo
Artefacto	Analizador sintáctico
Respuesta	Se generan los componentes SCA y las clases Java
Medida de respuesta	La solución genera componentes SCA y clases java sin errores a partir de cualquier diseño basado en la especificación.
Preguntas	En caso de que la especificación no provea una estructura de datos específica al sistema, ¿Cómo se garantiza que se pueden diseñar las pruebas?
Problemas	

6.2.3. Escenario No.2

Escenario	La compilación de las estrategias de pruebas debe resolverse en cuestión de milisegundos.
Objetivo del negocio	Generar nuevas estrategias de pruebas, de manera oportuna, que permitan garantizar los requerimientos del sistema y detectar fallas y/o errores.
Atributos de calidad relevantes	Desempeño
Estímulo	Ejecución del generador de componentes y clases (compilador)
Fuente de estímulo	Desarrollador de pruebas
Entorno	Sistema operativo
Artefacto	Generador de Código
Respuesta	Se generan los componentes SCA y las clases Java
Medida de respuesta	El tiempo de compilación se realizó en menos de un segundo,

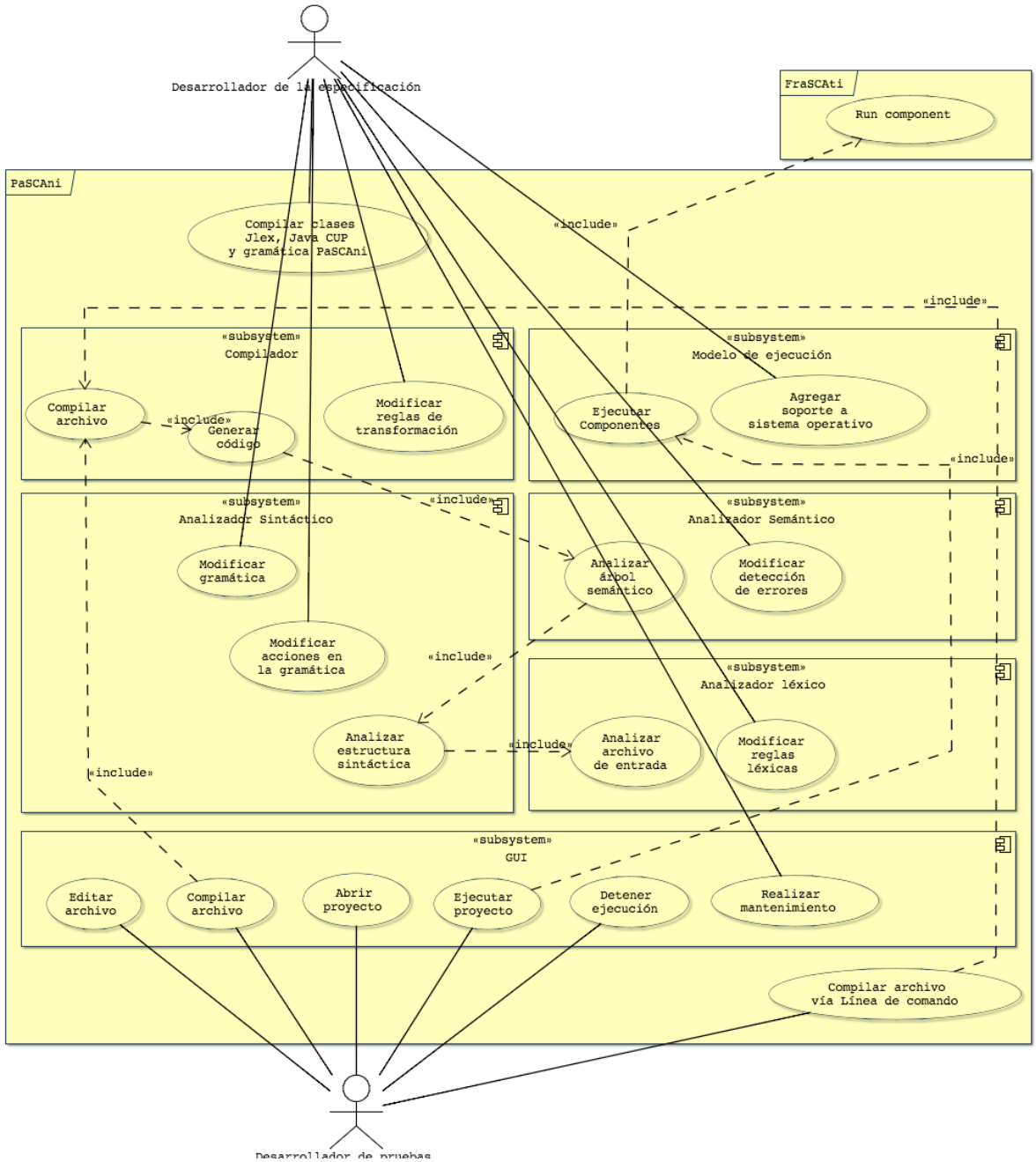
	teniendo en cuenta que el tiempo está ligado al número de archivos y líneas en la especificación (puede ser mayor para proyectos grandes)
Preguntas	¿Cuál es la relación entre el número de archivos y líneas de código que determinan el tiempo de ejecución?
Problemas	Si la estrategia de pruebas contiene errores léxicos, sintácticos o semánticos no se realiza la generación de componentes SCA y clases Java.

De los escenarios y requerimientos anteriores se tuvieron en cuenta dos drivers principales para la arquitectura: i) la especificación debe ser modular y ii) la especificación debe permitir diseñar estrategias de pruebas para el 100% de los componentes y servicios del sistema.

6.3. Diagrama de Casos de uso contextual

El siguiente diagrama de casos de uso presenta los principales casos de uso tanto como para el desarrollador de PaSCAni como para el desarrollador de pruebas que hace uso de la implementación propuesta en este proyecto.

Diagrama de Casos de uso contextual



6.4. Diseño de la solución

En la actualidad existen diversas herramientas para el desarrollo de varios de los elementos necesarios para satisfacer los requerimientos planteados en el capítulo anterior, unas más robustas que otras. En esta sección se explican los criterios de diseño y de selección de la herramienta, de manera general, se hace una descripción detallada de la construcción de la solución.
La solución del problema se encuentra en el dominio de pruebas a componentes SCA, para ampliar la idea; en consecuencia, las alternativas que se seleccionan deben ser compatibles con este dominio.

Adicionalmente, la sección describe las diferentes tecnologías analizadas y presenta el planteamiento de la solución que consta del diseño y especificación del lenguaje propuesto como solución al problema planteado en este trabajo de grado.

7. Alternativas de solución

7.1. Trabajos relacionados

Se exploraron dos alternativas para la generación automática de los analizadores léxico y sintáctico: Xtext [17] y Java cup/JLex [19, 20], dos tecnologías que se configuran y/o especifican para generar las respectivas partes de un compilador; adicionalmente, se exploraron lenguajes de dominio específico para especificar pruebas (e.g., TTCN-3 [18]). Como resultado de dicho análisis, se obtuvieron características relevantes para el diseño de este proyecto. Cada alternativa tiene características interesantes en el dominio en que se enfocan, en el caso de Xtext, la facilidad que tiene éste para integrarse con Eclipse IDE permite generar un editor con funciones extendidas que facilitan la escritura y la detección de errores sintácticos. En el caso de *Java CUP/JLex*, éstos son herramientas que permiten el desarrollo de compiladores de manera integral, permitiendo así que el analizador éxico se comunique de forma automática con el analizador sintáctico, de acuerdo a un conjunto de reglas gramaticales.

Nuestro análisis determinó que TTCN-3 está fuera del dominio en que se debe enfocar la solución al problema planteado. Sin embargo, fue posible a partir de TTCN-3 abstraer conceptos como la definición de módulos de prueba y el control de la ejecución de las pruebas, que se explica más adelante.
Otras herramientas que hicieron parte del análisis son Apache JMeter y JUnit. Ambas librerías se encuentran en contextos diferentes a nuestro enfoque, la Arquitectura de Componentes basada en Servicios (SCA). Por un lado, la primera permite realizar diferentes tipos de pruebas, entre las más comunes, pruebas de carga; Por otro lado, JUnit permite especificar y ejecutar aserciones sobre unidades independientes de código.

Para la generación de código del compilador, se exploran librerías que permiten el uso de *templates*, inicialmente estuvo presente la alternativa de crearlos a mano, sin el uso de alguna librería como medio

de apoyo, pero se explora y se define *Google Closure Templates* que permite de manera intuitiva especificar una plantilla de acuerdo a unas reglas que se escriben en un archivo.

7.1.1. JLEX

Un analizador léxico usa expresiones regulares para reconocer los símbolos del lenguaje. Sin embargo, hoy en día no resulta eficaz escribir analizadores léxicos desde cero, por lo cual los ingenieros de software usan herramientas que se encargan de generar automáticamente el código necesario para realizar el análisis léxico, a partir de una especificación dada por el programador.

JLex es un generador de analizadores léxicos que produce un programa Java a partir de una especificación. Por cada tipo de símbolo, la especificación contiene una expresión regular y una acción. La acción, un fragmento de código Java, comunica el tipo del símbolo a la fase del análisis sintáctico..

El lenguaje de especificación del generador está descrito por medio de reglas que prescriben la estructura léxica del mismo. Esta descripción se realiza utilizando gramáticas de lenguajes regulares.

7.1.2. Java CUP

Al igual que en el caso de los analizadores léxicos, la escritura de los analizadores sintácticos es hecha, comúnmente, haciendo uso de una herramienta de software. Con el fin de implementar el generador del analizador sintáctico se utilizó *Java CUP*.

Java CUP (*Java Based Constructor of Useful Parser*) es un sistema de software para generar parsers LALR de especificaciones simples. Java CUP, como su nombre indica, está escrito en Java y usa especificaciones que incluyen código de Java embebido para, finalmente, producir parsers implementados en Java.

7.1.3. Especificación Java CUP

Una especificación CUP se divide en cuatro partes principales. La primer parte contiene declaraciones preliminares para especificar cómo el parser será generado suministrando partes del código perteneciente al runtime del parser. La segunda parte de la especificación declara terminales y no terminales de la gramática y asocia clases de objetos con cada una. La tercer parte específica la precedencia y asociatividad de los terminales. La última parte de la especificación contiene la gramática[15].

7.1.4. Xtex

Xtext proporciona un editor especializado para la creación de los lenguajes de programación y de la gramática de los DSL[3], un lenguaje de programación dedicado a un problema de dominio en particular, como ejemplo se puede mencionar las expresiones regulares. A partir de eso, se genera un *API* para manipular mediante programación: instancias, analizadores y formateadores de lectura y escritura [17].
Para especificar un lenguaje, un usuario tiene que escribir una gramática en el lenguaje de la gramática *Xtext*. Esta genera varios artefactos automáticamente a partir de una gramática EBNF[4], esta gramática describe cómo un modelo *Ecore*[5] se deriva de una notación textual. A partir de esa definición, un generador de código deriva un analizador ANTLR*,* quien genera un programa que determina si una sentencia o palabra pertenece a dicho lenguaje; y las clases del modelo de objetos.

6.1.5. TTCN-3(Testing and Test Control Notation version 3)

Es un lenguaje de especificación de pruebas, que puede aplicarse en variedad de dominios pero dentro de componentes electrónicos. Incorpora un rico sistema de tipificación y poderosos mecanismos de pareo, apoyando la comunicación basada en mensajes tanto basadas en procedimientos, temporizador de manejo, configuración de prueba dinámica incluyendo las pruebas concurrentes de comportamiento, el concepto de veredictos y resolución de veredictos y mucho más [18].
La definición abstracta de los casos de prueba que es fundamental para TTCN-3 permite especificar un sistema de prueba no propietaria, que es independiente tanto de la plataforma y del sistema operativo [18].
TTCN-3 separa por módulos la especificación de las pruebas permitiendo probar y especificar un control para detallar el orden de de ejecución de estas.

7.2. Selección de tecnologías

Para el desarrollo de la solución se seleccionó Java CUP/JLex y no Xtext, esto no quiere decir que Xtext no sea una herramienta robusta para el desarrollo, pero en cuanto al manejo del analizador léxico y el analizador sintáctico Java CUP/JLex permite la modificación de la salida de acuerdo a unas reglas de especificación, contrario a Xtex que tiene una capa más abstracta que no permite la modificación de algunos aspectos de la salida de cada analizador; como por ejemplo, el uso de los *tokens* en el analizador léxico o la configuración del árbol abstracto de sintaxis del analizador sintáctico.

De las otras alternativas exploradas se abstrajeron características que fueron de gran utilidad para el desarrollo del lenguaje, en el caso de TTCN-3 se obtuvo el enfoque modular y la definición del control, en cuanto a las librerías Apache JMeter y JUnit se abstrajo el uso de aserciones para probar expresiones, pero estas no entran en el dominio de componentes SCA por esta razón se descarta su uso. Para el compilador en la fase de generación de código se opta por usar *Google Closure Templates* por la facilidad de crear *templates*, es decir, plantillas para la generación de código; funciona mediante unas

[3] DSL: Lenguajes de dominio específicoo.
[4] Las gramáticas EBNF se utilizan para definir la sintaxis de los lenguajes de programación, representa una forma compacta de escribir gramáticas independientes de contexto.
[5] Permite definir meta-modelos, llamados también modelos de dominio.

reglas de especificación en un archivo, donde se dan los parámetros indicados y a partir de estos parámetros se crea el código.

7.3. Planteamiento de la solución

Después de analizar las alternativas descritas anteriormente, la solución elegida es diseñar e implementar un lenguaje para especificar pruebas con las tecnologías seleccionadas. A continuación, se introduce conceptualmente la definición del lenguaje, luego el modelo de traducción a SCA, y el modelo de ejecución de los componentes de prueba generados. Los detalles de implementación se presentarán en el siguiente capítulo.

Para el desarrollo del lenguaje, que hemos denominado PaSCAni, se requiere del diseño de un compilador y de un modelo de ejecución.

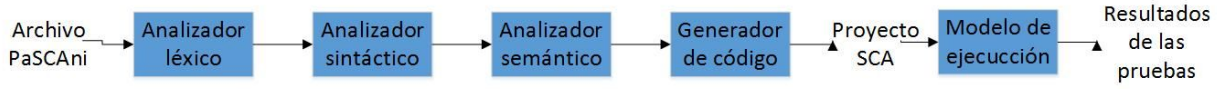


Figura No. 4. Proceso de compilación de pascani.

El compilador de PaSCAni esta compuesto de 4 partes:

- Analizador léxico: se encarga de recibir un archivo PaSCAni de entrada y producir una salida compuesta de *tokens*, para esto se usa *JLex,* quien permite generar un analizador léxico a partir de la especificación de unas reglas y de la definición de las palabras reservadas del lenguaje.

- Analizador sintáctico: se usa *java cup* quien genera un analizador sintáctico a partir de la especificación de la gramática, creando un árbol de sintaxis abstracta del lenguaje. La salida que produce el analizador léxico es la entrada del sintáctico.

- Analizador semántico: realiza el análisis del significado del archivo PaSCAni, como ejemplo la revisión del alcance de las variables.

- Generador de código: para la construcción del código que se desea generar, en este caso un proyecto SCA, que contiene las pruebas a realizar, se usan plantillas que se especifican primero en un archivo de configuración y luego mediante *Google Closure Templates* se llenan dichas plantillas con la información correspondiente de acuerdo al árbol de sintaxis abstracto y se genera el proyecto SCA.

A continuación, se ilustra la composición del lenguaje y la estructura para especificar pruebas. Para especificar pruebas, se debe crear un archivo que contiene la extensión *.pascani.*

La estructura del archivo de especificación esta compuesto de tres secciones, como se muestra en la Figura 5:

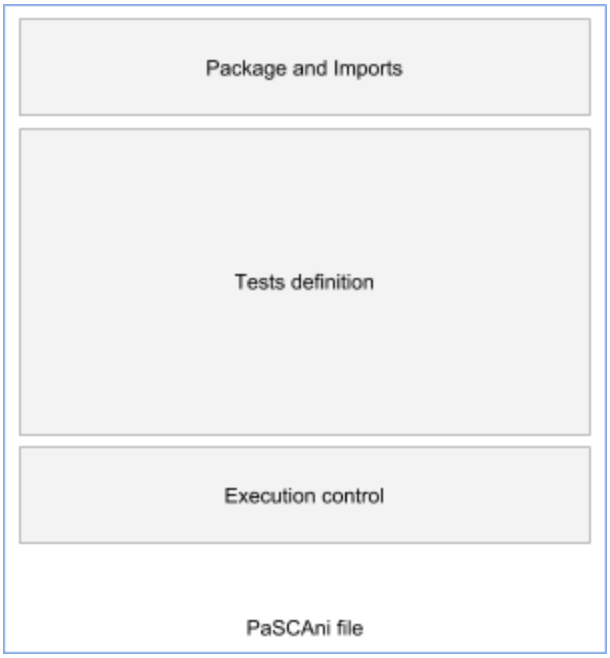


Figura No. 5. Definición de un archivo del lenguaje.

El *package* es el paquete donde se encuentra la especificación y los *imports* pueden ser de distintas maneras, se pueden importar clases java y módulos de otro módulo *PaSCAni*.
Para definir un *import* de una clase *java* se sigue la siguiente estructura:

Figura No. 6. Definición de un *import* de una clase java.

Al importar un módulo, se puede hacer uso de lo que el usuario ha especificado en él. Para importar un módulo, se debe seguir la siguiente estructura:

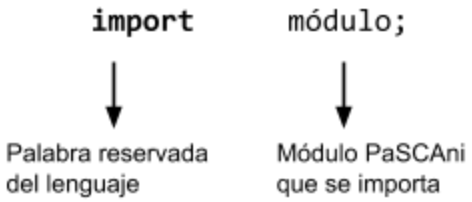


Figura No. 7. Definición de un *import* tipo módulo

Un módulo de pruebas agrupa las definiciones de un conjunto de casos de prueba y un bloque de control de ejecución.

En el interior de un módulo de pruebas se especifican los componentes de prueba, se pueden especificar dos tipos diferentes de *composites*[6]: conocido y anónimo. El tipo de componente anónimo es aquel composite que ya se encuentra en ejecución, mientras que el otro debe ser puesto en ejecución previo las pruebas. Para definir un *composite* anónimo se deberá usar la palabra reservada *anonymous* y seguir la siguiente estructura:

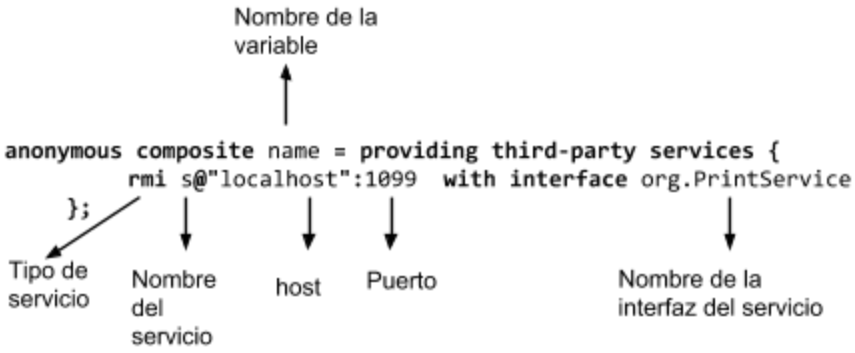


Figura No. 8. Definición de un *composite* anónimo.

Para definir un componente de tipo conocido, se hace siguiendo la estructura de la figura 9:

[6] Un composite puede ser visto como un componente cuya implementación no es código, pero una agregación de uno o más componentes que cooperan para proporcionar servicios de nivel superior.[23]

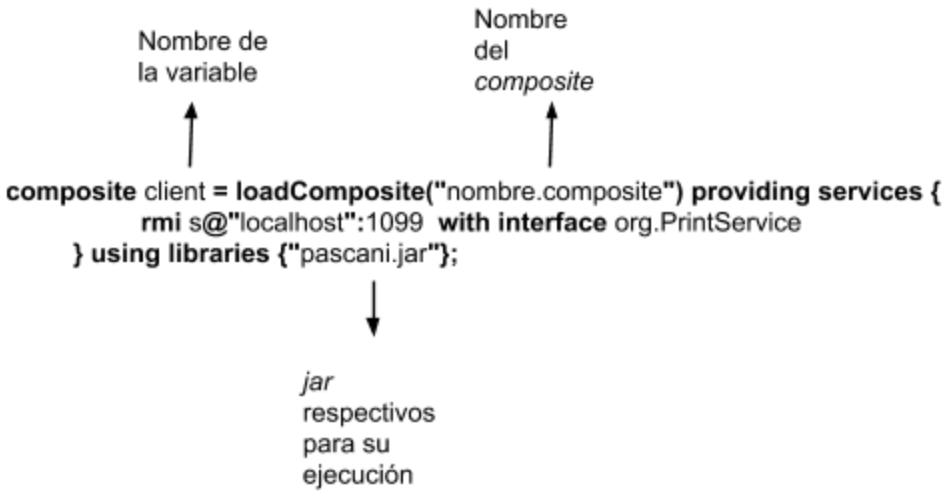


Figura No. 9. Definición de un *composite* de tipo conocido.

En el interior del módulo de pruebas que se observa en la Figura 10, se definen los casos de prueba, que en PaSCAni se denominan *testsuite*, dentro de ellos se especifican las pruebas de los servicios. En el lenguaje se define un *testsuite* de la siguiente forma:

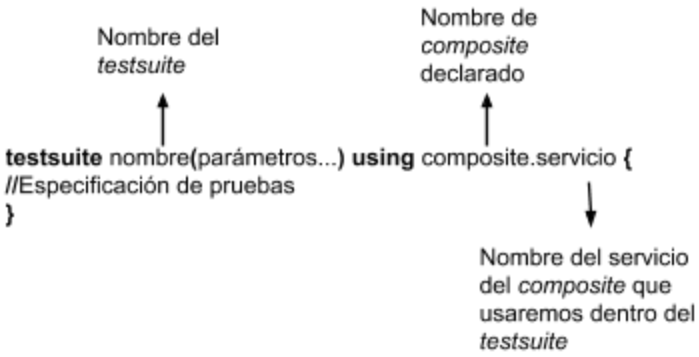


Figura No. 10. Definición de un *testsuite*.

Dentro del *testsuite* que se observa en la Figura 10 se especifican los casos de prueba, por cada uno de estos existe un veredicto[7]. Del conjunto de *test* especificados en el *testsuite* se retorna también un veredicto. Un veredicto puede tomar los siguientes valores: *failed, passed, inconclusive y exception*.

En un testsuite se pueden especificar otro tipo de estructuras que mencionaremos en la parte siguiente que habla de correspondencia, aqui solo se pretendemos mostrar la estructura general del archivo PaSCAni.

Para terminar, el Modelo de ejecución que está en la Figura 4, permite que se especifique el orden de ejecución de los *testsuite*, en otras palabras, el orden de ejecución de las pruebas. Un ejemplo general

[7] Valor resultante de la prueba

de la estructura se presenta a continuación en la figura 11:

```
package       . . .;
java-import   . . .;
import . . .;

module {

        composite . . . = loadComposite(. . .) providing
        services {
                // Especificación de servicios.
        } using libraries { };

        anonymous composite . . . = providing third-party
        services {
                // Especificación de servicios.
        };

        testsuite (. . .) using . . . {
                // Especificación de tests
        }
        control {
                // Especificación del modelo de ejecución
        }
}
```

Figura No. 11. Ejemplo general de un archivo de *PaSCAni*.

7.4. Mapeo de la estructura de un archivo PaSCAni a componentes SCA implementados con Java

Cada parte especificada en un archivo *.pascani* genera su código Java correspondiente después del proceso de compilación. Sin embargo, surgen varias soluciones respecto a cómo generar ese código bajo el estándar SCA y clases. A continuación se plantean 2:

i). Exigir al programador que los *composites* a probar ya hayan sido desplegados en un proceso de *Frascati* (al momento de ejecutar la compilación en *PaSCAni*). Al tener los composites a los que se les hará pruebas ya desplegados en *frascati*, se debe hacer los *binding*[8] a cada servicio para permitir que en la ejecución del *composite*, *frascati* pueda

[8] Un binding se utiliza como un medio de comunicación entre los servicios y maneja los protocolos.[23]

conectar las referencias del componente de prueba con los servicios provistos por el componente objeto de pruebas. Esto significa que en el archivo *.pascani* se debería escribir el *binding* que referencia al servicio del composite al que se le van a hacer sus respectivas pruebas.

La figura 12 muestra el Composite M1 que es generado a partir del archivo *.pascani*, este *composite* aun no se encuentra en ejecución en *frascati*. Las pruebas se generan a partir del archivo *.pascani* y se encuentran dentro del composite M1. Por esta razón, se debe incluir el *binding* en la especificación del módulo de pruebas. Por otro lado, desplegar componentes manualmente, previo a la ejecución de las pruebas PaSCAni, sólo se carga un *composite* a *frascati*, el M1, porque los otros ya están en ejecución.

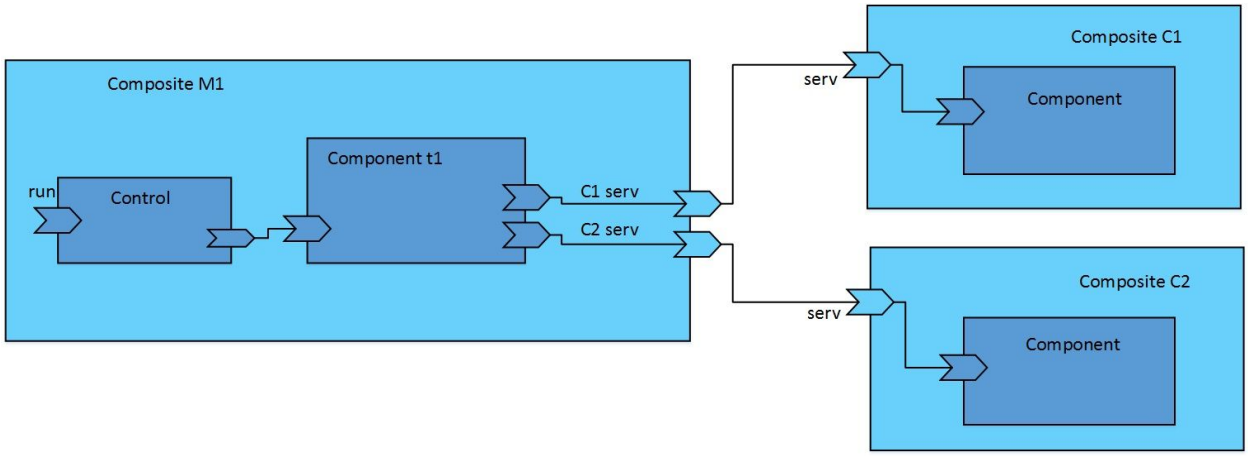


Figura No. 12. Diagrama de despliegue para componentes de prueba PaSCAni.

ii). Realizar su ejecución de manera automática desde *PaSCAni*. En esta alternativa todos los *composites* se encuentran en ejecución directamente en *frascati*, tanto los composites a los que se les ejecutarán pruebas como los generados a partir del archivo. Es decir, al momento de ejecutar las pruebas de estos composites no hay ninguno en ejecución, de ser así, se crea un *composite* que agrega a los otros en el. Esta agrupación permitirá especificar un *wire*[9] entre los *composites* que se encuentran dentro del *composite* que los agrupa, en lugar de hacer un *binding* que debe estar especificado en el archivo *.pascani*. En la figura 13, el *composite* que los agrupa es llamado MM1, el *composite* M1 hace referencia mediante un *wire* a los *composite*

[9] La relación entre una referencia y un servicio se demuestra típicamente a través de una línea en unos diagramas de SCA y se conoce como un wire. [23]

C1 y C2, al poner en ejecución esto en *frascati* solo ejecutará el composite MM1 que agrupa a los demás.

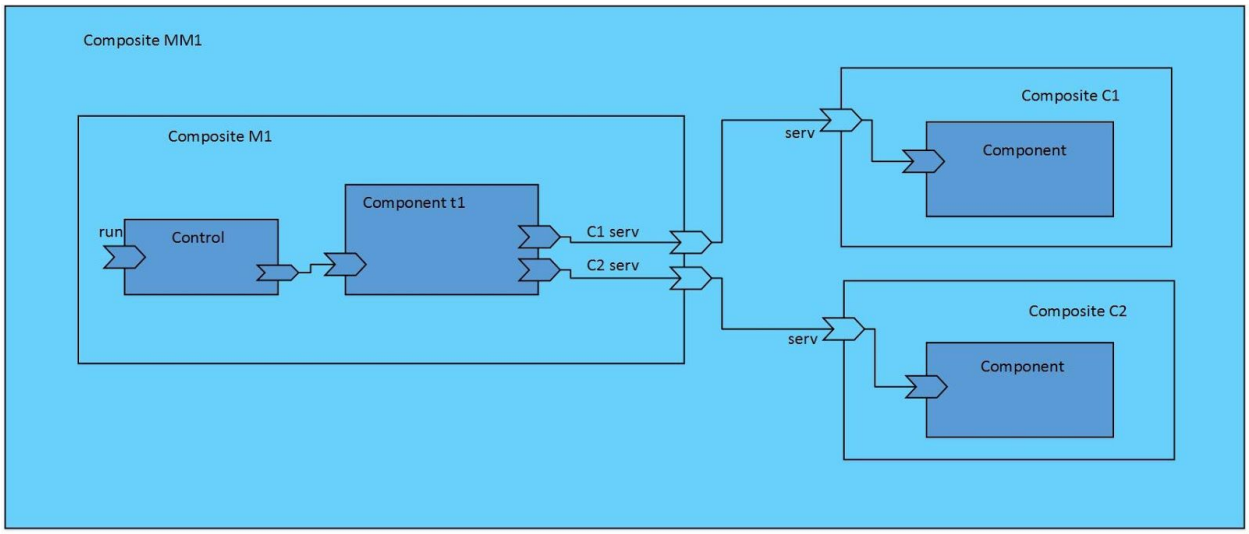


Figura No. 13. Muestra la agrupación de los *composites* en uno.

Dado que ambos casos son comunes y pueden requerirse para probar aplicaciones basadas en componentes, ambas alternativas de soluciones planteadas fueron implementadas en el lenguaje.

A continuación se describe la estructura del Composite M1 mostrado en la figura 14. A partir del archivo .*pascani* especificado correctamente, se debe generar un *composite* que corresponde al *module* que está en dicho archivo, dado que cada *module* contiene uno o varios *testsuites* con pruebas a diferentes servicios, cada *testsuite* corresponde a un componente.
En el siguiente ejemplo, se crea un *composite* para el *module* llamado *prueba* que referencia al servicio del *composite comp*, al que se le están haciendo pruebas. Dentro del *composite pruebas* deberá haber un componente que corresponde al *testsuite testUno*, así mismo, dentro del *testuite* existen varios *tests* que definen diferentes pruebas y entre ellas se define o se da como resultado un veredicto final; vale aclarar que por cada *test* hay un veredicto, y en el *statement return* el retorno que se hace en la clase correspondiente al *testsuite* es un *test*.

```
package pruebas;

module prueba {
	composite comp = loadComposite(...) providing services {...} using
libraries{...};
	testsuite testUno(int a, int b) using comp.serv {
		int valor = comp.serv(a, b);
		test p = valor > 0
```

```
                    labeled “Prueba”
                    message when failed: “Falló”
                                exception: “Ocurrió Excepción”;
            return p;
        }
}
```

Figura No. 14. Ejemplo de un archivo *.pascani.*

Para continuar, se crea el *composite* con referencia a los servicios del *composite* al cual se le ejecutan pruebas, en este caso sería una referencia al servicio *serv* que está en el *composite* que ya ha sido puesto en ejecución con el identificador *comp*. Como es bien sabido, se debe tener la interfaz del *composite* del servicio para poder realizar llamados a sus métodos, además se deben ajustar los puertos y la dirección IP, en este caso *localhost* (dado que se ha decidido en el alcance de este proyecto realizar pruebas concurrentes no distribuidas).

La figura 15 muestra la correspondencia, donde se puede observar que el *module* prueba en el diagrama es el *Composite Prueba* y el *testsuite testUno* es el *Component test*, dentro del testsuite se elaboran pruebas al servicio *serv*.

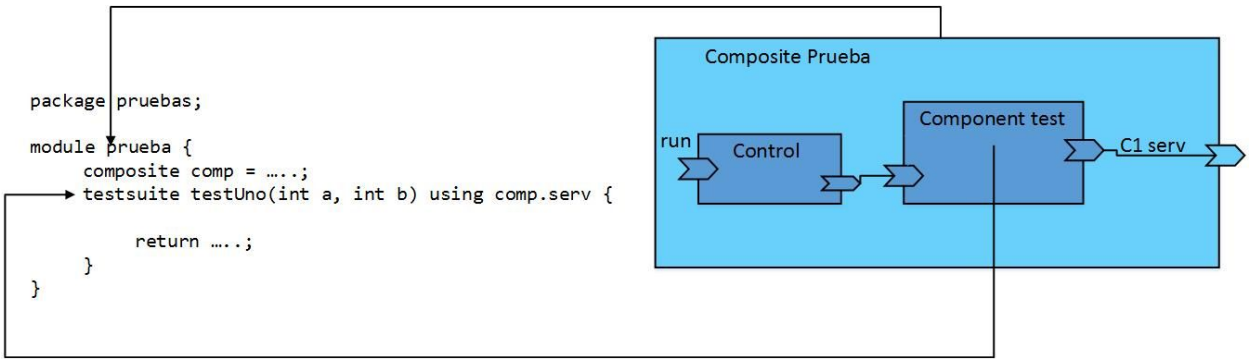


Figura No. 15. Correspondencia de *module* y *testsuite*

Una vez definido lo anterior, a un testsuite le corresponde un componente cuya implementación es una clase Java, a continuación se muestra el correspondiente del componente *Control* de la Figura 15:

```
package prueba;
import org.osoa.sca.annotations.Reference;
….
@Service
public class RunnerImpl implements java.lang.Runnable {
	@Reference
	private RunnableControl run_control;
	@Override
	public void run() {
		// Definición del control
	}
}
```

Figura No. 16. Correspondencia del componente *control*.

El componente *Control* que se muestra en la Figura 15 ofrece un servicio llamado *run* que corresponde al método que se muestra en la Figura 16. Donde se especifica el orden de la realización de las pruebas y donde se puede especificar cambios del archivo donde se muestran las pruebas(*XML*), a este archivo se hace referencia ya que al final de las pruebas siempre se mostrará al usuario los resultados, este fichero se genera mediante una clase, donde agrupa por *testsuite* las respuestas de los *test* para mostrarlos al usuario. Este archivo se visualiza en la Figura 17.

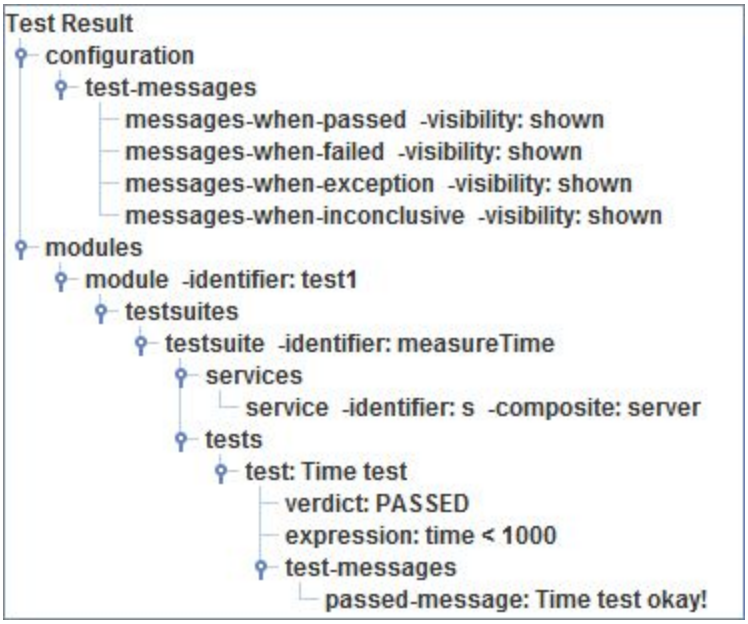


Figura No. 17. Ejemplo de como se ve un *XML* en la *GUI.*

Se debe tener en cuenta que el usuario tiene la opción de elegir lo que desea ver, por ejemplo, si a él sólo le interesan las pruebas que fallaron puede especificarlo en *PaSCAni* así:

```
String[] messages = {"passed", "failed", "inconclusive", "exception"};
set("show_messages", messages);
```

Cualquier cambio que se desee hacer al archivo final de los resultados de las pruebas(*XML*) se hace en el "control".

Anteriormente se explica que a cada testsuite le corresponde un componente cuya implementación es una clase Java, sin embargo al componente *Control* que se muestra en la Figura 6. también le corresponde un componente pero este no es igual que un componente de un *testsuite*, por tanto cabe explicar cómo es la estructura correspondiente de él. En la Figura 7. se muestra la correspondencia del componente *Control*, implementa la interfaz `java.lang.Runnable` que nos permite sobreescribir el método *run,* donde esta el especificado el orden de las pruebas.

Cada componente asociado a su *testsuite* tiene las referencias respectivas a los servicios que va a probar e implementa la interfaz `RunnableTestsuite`, al implementar esta interfaz implementa el método *run*(que es totalmente diferente al de la interfaz `java.lang.Runnable`) en cada componente que corresponde a un *testsuite* y dentro de este método se hacen las

pruebas que han sido especificadas por el usuario, esto es con el fin de hacer que la referencia entre el *control* y los servicios de los componentes que son asociados a los *testsuites* se hagan por medio de un *wire*, permitiendo que el control sea independiente de los servicios que va a usar cada testsuite, de esta manera solo se llamará el método *run* de cada componente para correr las pruebas.

El método *run*(de la interfaz `RunnableTestsuite`) del componente recibe la cantidad de parámetros necesaria para su ejecución, según como estén especificadas las pruebas. Siempre que se vayan a ejecutar las pruebas de un *testsuite* se deberá invocar en el componente *control* el método *run* del componente que está asociado a ese *testsuite*.
Para el *testsuites* mencionado anteriormente, se crea un componente dentro de cada *composite*, este componente debe tener la referencia al servicio que se quiere probar como se muestra en la Figura 18. Donde posee una referencia al servicio *servicio_test* permitiendo que se le puedan efectuar pruebas.

```
package lib;
….
@Service(RunnableTestsuite.class)
@Scope("CONVERSATION")
public class TestSuiteGenerate implements RunnableTestsuite {
	private Hashtable<String, Test> testsResults = new Hashtable<String, Test>();
	private Integer verdict = Verdict.INCONCLUSIVE;
	@Reference
	private ServicioSuma servicio_test;
	@Override
	public Integer run(Object ... parametros) {
		// Se especifican las pruebas respectivas
	}
	@Override
	public Hashtable<String, Test> getTestsResults() {
		return this.testsResults;
	}
	@Override
	public Integer getVerdict() {
		return this.verdict;
	}
}
```

Figura No. 18. Ejemplo de la correspondencia de un *testsuite*.

Al implementar la interfaz que se muestra en la Figura 18. se implementan no solo el método *run* sino también dos métodos más *getResults* que permite obtener los veredictos asociados a cada test que se encuentra especificado dentro de un testsuite y el *getVerdict* que me devuelve el veredicto final de todo el *testsuite*.

Dentro de un *testsuite* se agrupan muchos *test* como se mencionó anteriormente, cada *test* corresponde a un bloque de código que se muestra en la Figura 19.

```
Ejemplo de un test:
test lessThanASecond = time < 1000 labeled "Prueba de tiempo" message
                       when passed: "Pasó";

Corresponde a:
Hashtable<Integer, String> messages = new Hashtable<Integer, String>();
messages.put(Verdict.PASSED, "Pasó la prueba de tiempo");
Test lessThanASecond = new Test("Prueba de tiempo", messages);
try{
        boolean result = time < 1000;
        if(result)
                lessThanASecond.setVerdict(Verdict.PASSED);
        else
                lessThanASecond.setVerdict(Verdict.FAILED);
} catch(Exception e){
        if(e instanceof TimeoutException)
                lessThanASecond.setVerdict(Verdict.INCONCLUSIVE);
        else
                lessThanASecond.setVerdict(Verdict.EXCEPTION);
}
```

Figura No. 19. Ejemplo de correspondencia de un *test*.

En la Figura 19. se observa como para un *test* corresponde su veredicto final, en este caso es el *result* y en el *HashTable* guardo los mensajes asociados al *test*.

Ahora bien, cabe la posibilidad de que el usuario desee iterar varias veces un servicio, como lo pretende la utilización del statement *for*. En la Figura 20 se muestra un ejemplo de cómo se debe especificar un *for* en el archivo *.pascani*.

```
for(StackDescriptorTest test in cases){
            int obj = knowledge.serv(test);
            test p= obj < 1000 labeled "Prueba Valores" message when passed:
            "Pasó la prueba";
}
```

Figura No. 20. Especificación de un *for* en un archivo *.pascani*

Para iniciar, cabe resaltar que en *Java* el *for* es secuencial, es decir, si se está ejecutando una línea dentro del *for* se debe terminar la ejecución de esa línea para continuar con la siguiente si existe. Se pretende en un archivo *.pascani* que el *for* permita probar diversos *tests* de manera secuencial. Para el fragmento de código de la Figura 20, la idea es ejecutar muchos casos de prueba, esperando que se ejecute uno después de otro. El código correspondiente en Java se puede observar en la Figura 21.

```
for(StackDescriptorTest test : cases){

      Hashtable<Integer, String> messages = new Hashtable<Integer,
String>();
      Test p = new Test("", messages);

      try{
             int obj = knowledge.serv(test);
                       boolean result =  obj < 1000;
                       if(result)
             p.setVerdict(Verdict.PASSED);
                       else
             p.setVerdict(Verdict.FAILED);

               }catch(Exception e){
                       if(e instanceof TimeoutException)
             p.setVerdict(Verdict.INCONCLUSIVE);
                       else
             p.setVerdict(Verdict.EXCEPTION);
               }
}
```

Figura No. 21. Correspondencia en Java de un *for* en PaSCAni.

7.5. Modelo de ejecución

Finalizada la tarea del compilador se genera un proyecto SCA con un archivo de configuraciones de despliegues de los componentes (SCA) que el usuario ha especificado con anterioridad en el archivo *PaSCAni*, este archivo contiene las configuraciones necesarias para iniciar el despliegue. Dentro de dicho fichero se encuentran las dependencias de los componentes y la información del proyecto SCA que incluye el nombre del servicio, el directorio donde se encuentra, el método y el nombre del *composite*.

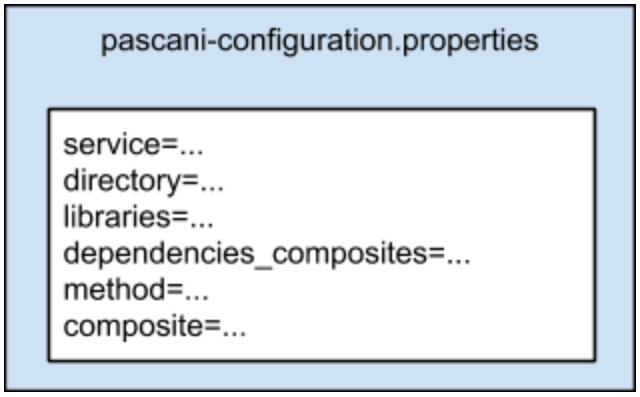


Figura No. 22. Estructura del archivo de configuraciones.

La función del modelo de ejecución es desplegar los componentes que no se encuentran actualmente en ejecución en la máquina virtual de FraSCAti. Esta tarea se realiza mediante el archivo de configuraciones. Como primera instancia se analizan las dependencias de componentes que se deben desplegar y se ponen en ejecución directamente en la máquina virtual de frascati, sin embargo pueden haber *composites* que ya están en ejecución en la máquina virtual de *FraSCAti* permitiendo realizar pruebas a componentes que están también en ejecución, luego se debe ejecutar el composite generado para así generar el resultado de las pruebas.

En la Figura 23. el *Composite C2* se encuentra en ejecución en la máquina virtual de *FraSCAti*, mientras que el *C1* no, en este caso hay una dependencia, pues primero se debe ejecutar el composite *C1* antes del *Composite generado*, si no es así, el *Composite generado* no encontraría al *Composite C1* para ejecutar sus pruebas.

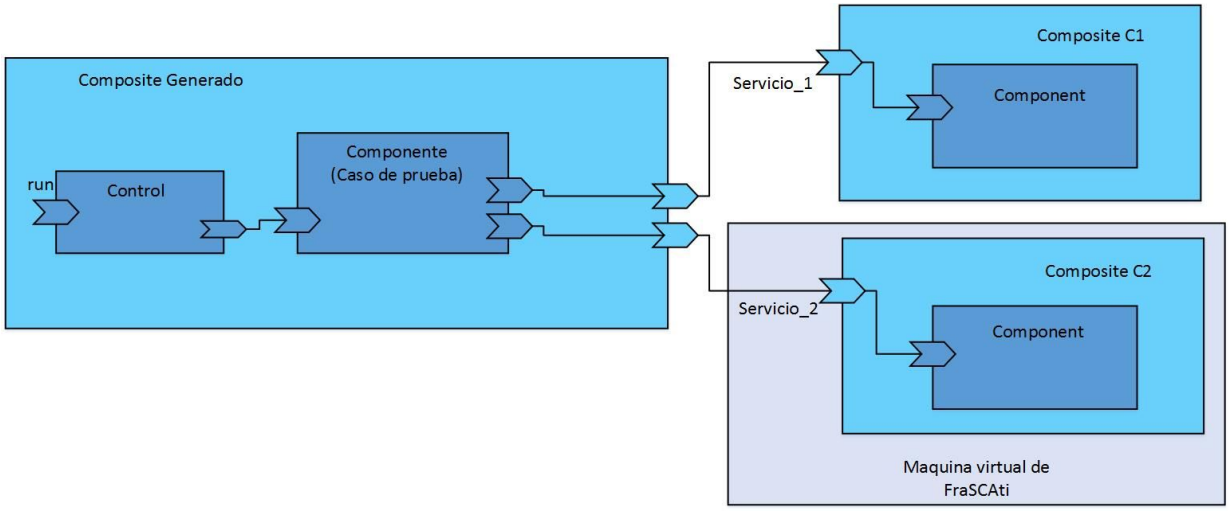


Figura No. 23. Muestra el despliegue de componentes en la máquina virtual de FraSCAti.

Se debe tener en cuenta que si un componente depende de otro componente y este a su vez de otro y estos deben ser desplegados, es necesario especificarlo en el archivo *PaSCAni* para evitar conflictos en el momento del despliegue en la máquina virtual de *FraSCAti*.

8. Implementación

8.1. Introducción a la sección

Esta sección describe los paquetes que se crean para la implementación del compilador y del modelo de ejecución. La implementación del compilador consta de varios paquetes distribuidos en cuatro partes: analizador léxico, sintáctico y semántico junto con el generador de código; mientras que la implementación del modelo de ejecución sólo consta de uno.

Además de la implementación del compilador y del modelo de ejecución se implementa una herramienta visual llamada *PaSCAni Explorer* que requiere de la creación de dos paquetes descritos en la sección. Las clases principales creadas en los paquetes de la implementación se muestran en los respectivos diagramas de clase a los que pertenecen con una breve descripción.

8.2. Descripción de paquetes

La implementación de PaSCAni esta compuesta por el compilador y el modelo de ejecución, la figura 24 permite visualizar la distribución de los paquetes creados.

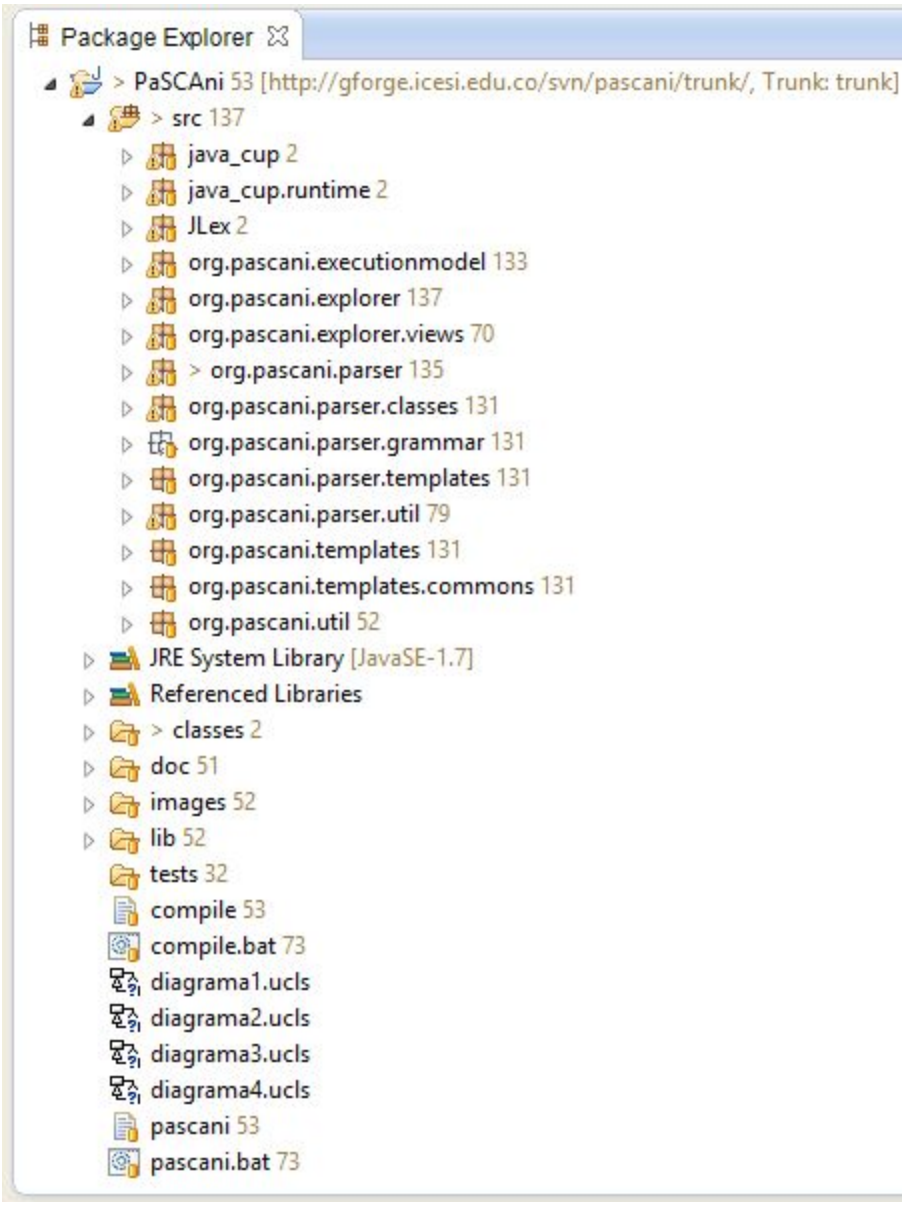


Figura No. 24. Paquetes del proyecto Java

Como se mencionó anteriormente el compilador está compuesto de 4 partes:

- Analizador léxico: se encuentra implementado en el paquete *JLex*, este se encuentra compuesto por una clase llamada *Main.java* que se encarga de generar el analizador léxico a partir de un archivo de especificación que se encuentra en el paquete *org.pascani.parser.grammar* y es llamado *PaSCAni.lex*, este archivo contiene las reglas de especificación que requiere JLex para poder generar el analizador léxico.

- Analizador sintáctico: el analizador sintáctico se encuentra en varios paquetes, el paquete *java_cup* y el *java_cup.runtime* donde estos se encargan de generar el analizador sintáctico a partir de la especificación de un conjunto de reglas y de la gramática independiente del contexto, esta especificación se encuentra en un archivo dentro del paquete *org.pascani.parser.grammar* con el nombre *PaSCAni.cup*, dentro de este se encuentra definida la gramática y la salida del árbol abstracto de sintaxis.

- Analizador semántico: se encuentra implementado en varios paquetes, el paquete *org.pascani.parser,* quien contiene la implementación que genera y anota semánticamente el *Java CUP* del árbol abstracto de sintaxis en el paquete *org.pascani.parser.classes* con un conjunto de 57 clases que se llenan a partir del árbol abstracto de sintaxis, que en sí son la abstracción de la información que contiene dicho árbol, por otro lado, se encuentra la implementación de la tabla de símbolos en el paquete *org.pascani.parser.util* que se usa para el análisis semántico.

- Generador de código: para generar el proyecto SCA se usa *Google Closure Templates*, donde se deben definir las plantillas que se encuentran en los paquetes *org.pascani.templates.commons* y el archivo de especificación de las reglas, el uso de estas plantillas se encuentra en el paquete *org.pascani.templates*, pero dentro del paquete *org.pascani.parser* se encuentran 2 clases que se encargan del proceso de generación, estas clases son *CodeGenerator.java* y *GeneratedFile.java*.

Para el modelo de ejecución se encuentra un paquete llamado *org.pascani.executionmodel*, compuesto por un conjunto de 3 clases que manejan el despliegue de cada *Composite* de acuerdo a la especificación del desarrollador de pruebas.

Adicionalmente, esta implementado *PaSCAni Explorer*, una herramienta visual que puede hacer el proceso de compilación y de ejecución, además permite visualizar el resultado de las pruebas, esto se encuentra dentro de los paquetes *org.pascani.explorer.views*(desarrollo de la parte visual de la ventana) y *org.pascani.explorer*(desarrollo de los eventos de la ventana), dentro del paquete *org.pascani.util* hay una clase que *DocumentUtilities.java* que permite la modificación de un archivo *PaSCAni* mediante la herramienta visual.

8.3. Diagrama de clases

En esta sección se explican las partes más importantes del diagrama de clases con las que se implementa PaSCAni.

- Paquete *org.pascani.executionmodel*: encargado del despliegue de componentes SCA.

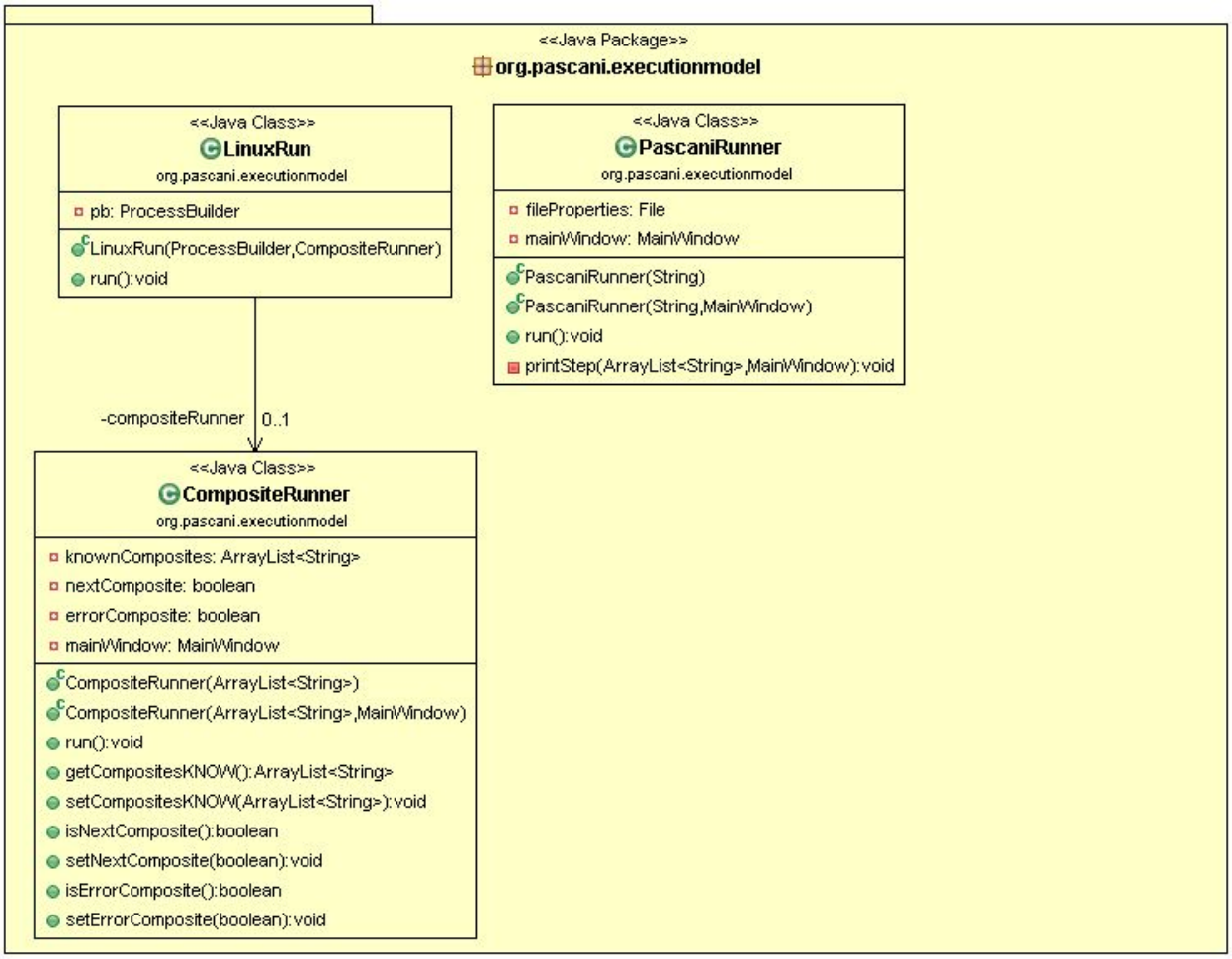


Figura No. 25. Paquete *org.pascani.executionmodel*

Dentro del paquete *org.pascani.executionmodel* se encuentran 3 clases que se describen a continuación:

- *PascaniRunner.java*: permite validar el archivo de configuración que es generado después del proceso de compilación.

- *CompositeRunner.java*: es el encargado del orden de ejecución de los componentes SCA, y prepara la ejecución de cada componente de acuerdo al sistema operativo.

- *LinuxRun.java*: se usa solo cuando se encuentra en un sistema operativo Linux para desplegar los componentes.

- Paquete *org.pascani.explorer*: encargado de los eventos de las ventanas que usa el explorador.

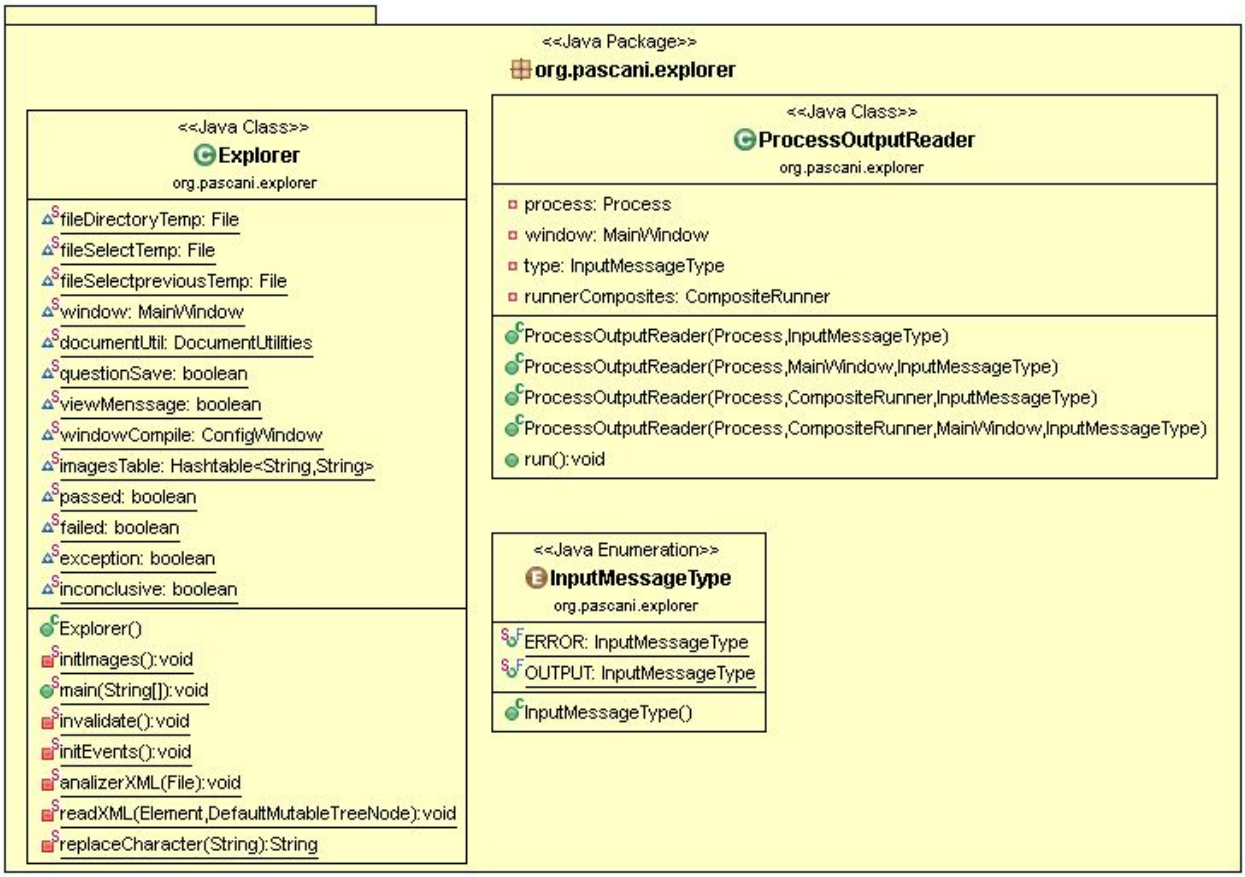


Figura No. 26. Paquete *org.pascani.explorer*

Dentro del paquete *org.pascani.explorer* se encuentran 2 clases que se describen a continuación:

- *Explorer.java*: es el encargado de los eventos y validaciones respectivas de la ventana.

- *ProcessOutputReader.java*: captura los mensajes que se generan de acuerdo a la

ejecución en *frascati*.

- Paquete *org.pascani.parser*: se compone de las clases que son generadas por el generador del analizador sintáctico (*java cup*) y de las clases que se encargan del manejo del árbol abstracto de sintaxis y de la generación del componente *control* donde se encuentran las pruebas.

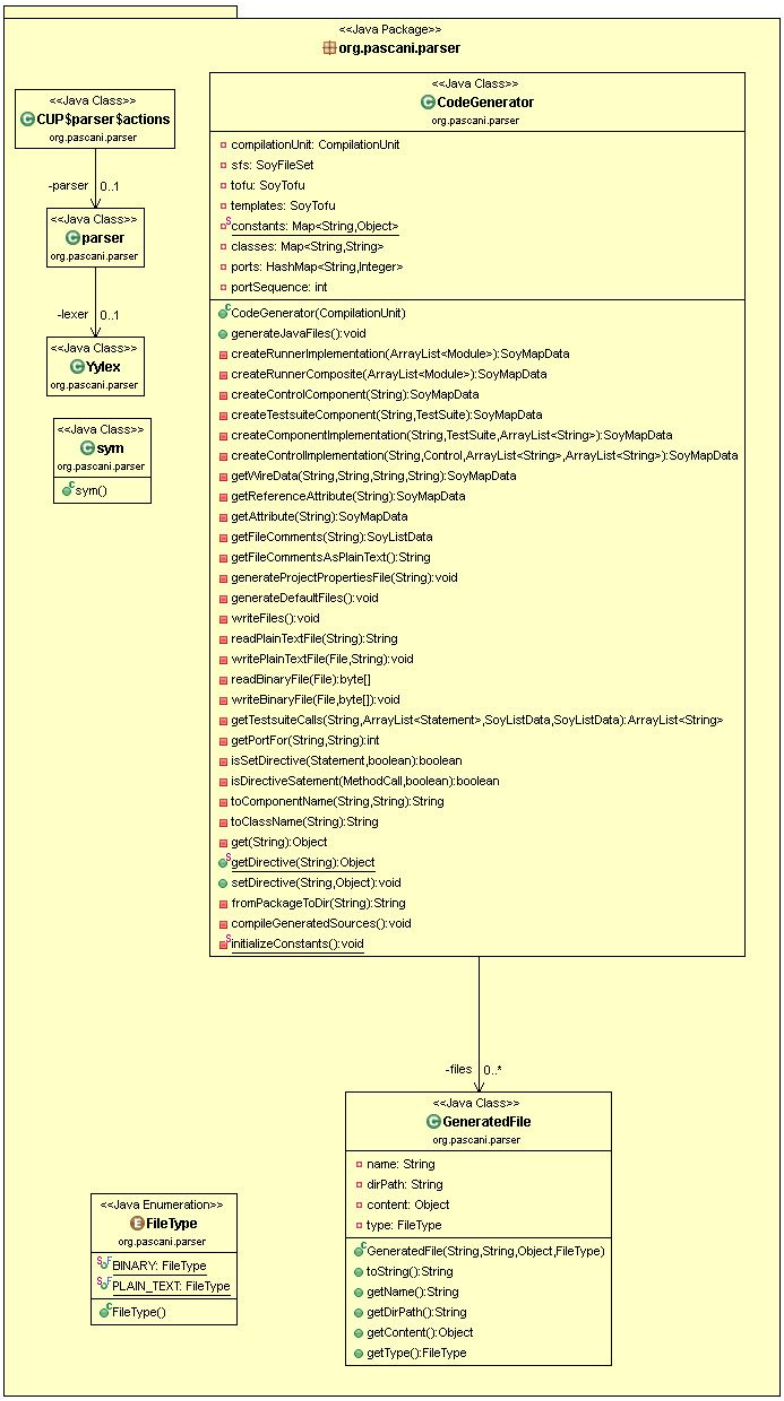


Figura No. 27. Paquete *org.pascani.parser*

Las clases *GeneratedFile.java* y *CodeGenerator.java* son el conjunto de clases que se encargan del desarrollo del árbol de sintaxis abstracto y de la generación de código, en este caso, del componente control. Las demás clases son las que genera el *java cup* a partir de la especificación de la gramática.

9. Caso de estudio

9.1. A variability model for generating SCA-based matrix-chain multiplication software products

9.1.1. Introducción

El caso de estudio seleccionado para verificar el correcto funcionamiento de PaSCAni y su pertinencia en las pruebas de sistemas de software basados en componentes, es una propuesta de componentes SCA para hacer óptima la multiplicación de matrices. Ésto aprovechando las ventajas de SCA para sistemas distribuidos.

Ésta propuesta de componentes es una línea de productos de software (SPL, por sus siglas en inglés), con la cual se pretende hallar una configuración de *assets* (componentes de software) que reduzca considerablemente el tiempo de ejecución de algoritmos ya conocidos, como el algoritmo de Volker Strassen, o el uso de un procedimiento de parentización, en su versión secuencial o paralela, entre otros. Usando diferentes estrategias de distribución adaptadas a SCA y tomadas de una propuesta basada en la arquitectura map/reduce [14].

En esta sección se presentarán los componentes principales de la SPL a través de un modelo de variabilidad, los productos de software que se pueden obtener de la SPL y un análisis detallado sobre la implementación y los resultados obtenidos con los módulos de prueba PaSCAni.

9.1.2. Modelo de variabilidad

Para la construcción de la SPL se tuvieron en cuenta las siguientes estrategias de multiplicación: algoritmo de Strassen, multiplicación por bloques o submatrices, multiplicación de una fila por una columna de bloques y multiplicación de n matrices.

Para este proyecto los componentes se describieron de forma general, sin llegar al detalle de las variaciones relacionadas al número de procesadores o la capacidad de memoria. Sin embargo, se agregó una variación al modelo relacionada al tipo de almacenamiento usado en cada estrategia (distribuido o compartido). Adicionalmente, se tuvo en cuenta un procedimiento que mejora el rendimiento de la multiplicación de más de dos matrices; la parentización propuesta en [21] y [22] mejora el rendimiento en un 50%.

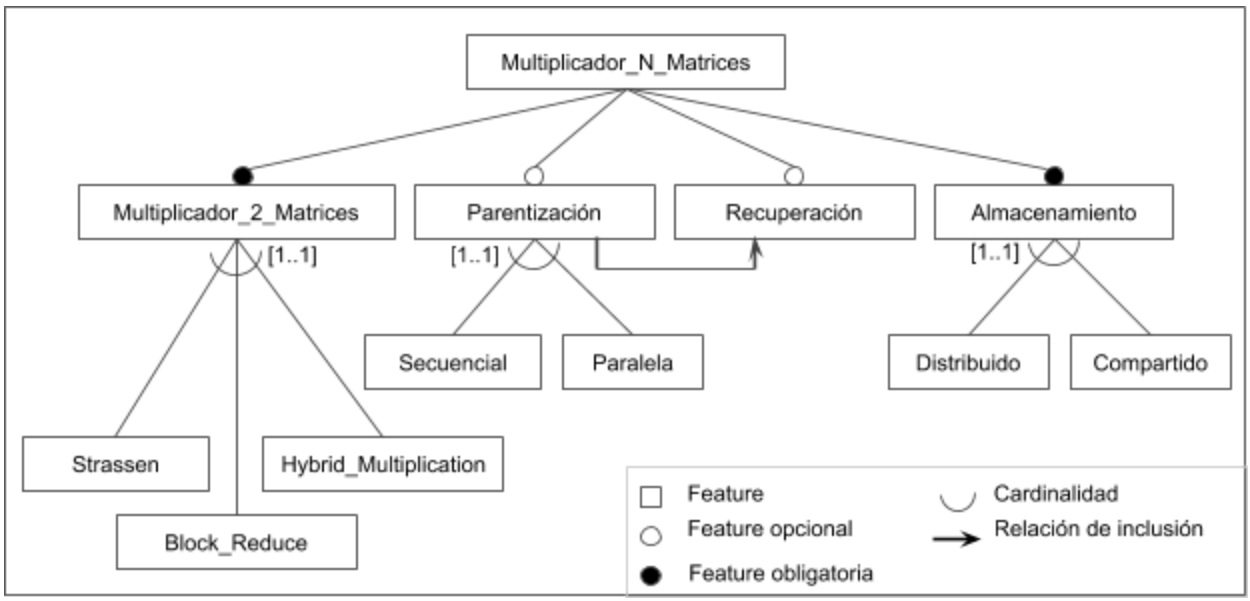


Figura No. 28: Modelo de variabilidad de la multiplicación de matrices

```
productline(L):-

L=[
    Multiplicador_N_Matrices,
    Multiplicador_2_Matrices,
    Strassen,
    Block_Reduce,
    Hybrid_Multiplication,
    Almacenamiento,
    Distribuido,
    Compartido,
    Parentizacion,
    Secuencial,
    Paralela,
    Recuperacion
],

fd_domain(
    [
        Multiplicador_N_Matrices,
        Multiplicador_2_Matrices,
        Strassen,
        Block_Reduce,
        Hybrid_Multiplication,
        Almacenamiento,
        Distribuido,
        Compartido,
        Parentizacion,
```

```
        Secuencial,
        Paralela,
        Recuperacion
    ],
    0, 1
),

Multiplicador_N_Matrices#=1,
Multiplicador_N_Matrices#<=>Multiplicador_2_Matrices,
Multiplicador_N_Matrices#<=>Almacenamiento,
Multiplicador_N_Matrices#>=Parentizacion,
Multiplicador_N_Matrices#>=Recuperacion,
(1-Parentizacion)+Recuperacion#>0,
Parentizacion+(1-Recuperacion)#>0,
1*Multiplicador_2_Matrices#=Strassen+Block_Reduce+Hybrid_Multiplication,
1*Almacenamiento#=Distribuido+Compartido,
1*Parentizacion#=Secuencial+Paralela,

fd_labeling(L).
lista(M,L):-
       productline(M),

       A=[
            'Multiplicador de n matrices',
            'Multiplicador de 2 matrices',
            'Strassen',
            'Block_Reduce',
            'Multiplicación de filas y columnas de bloques',
            'Almacenamiento',
            'Distribuido',
            'Compartido',
            'Parentización',
            'Secuencial',
            'Paralela',
            'Recuperación'
        ],

predicado(M,L,A).

predicado(M,L,A):-
       M=[],
       L=[],
       A=[].

predicado(M,L,A):-
       M=[H|T],
       L=[H1|T1],
       A=[H2|T2],
       predicado2(H,H1,H2),
       predicado(T,T1,T2).

predicado2(X,Y,W):- X >0 -> Y=W ; Y='n'.
```

Figura No. 29 Modelo de variabilidad escrito en GNU Prolog

La figura número 28 representa el modelo de variabilidad y los componentes de la SPL. Y la figura 29 muestra su representación en GNU Prolog.

A continuación se hace una breve descripción de la principal funcionalidad de cada variación, obviando las variaciones simples (e.g. selección de una estrategia para el nodo *Multiplicador de matrices*):

9.1.3. Multiplicador de n matrices

La multiplicación de más de dos matrices se implementó re-usando los componentes de software para la multiplicación de pares. Esto se hizo para aprovechar los recursos computacionales de una mejor manera; cuando un procesador se libera, se le envía otro par de matrices. Opcionalmente, en el caso de n matrices el rendimiento se ve positivamente afectado al ejecutar, previo a las multiplicaciones, un proceso de parentización con el fin de conocer el orden óptimo de multiplicación.

9.1.4. Algoritmo de Strassen

El algoritmo de Strassen es un conocido algoritmo propuesto por el matemático Volker Strassen en el año 1969, que mejoró el procedimiento estándar para la multiplicación de matrices cuadradas de orden $m2^k$. La propuesta de Strassen redujo la complejidad del algoritmo común de $O(n^3)$ a $O(n^{2.81})$ [13].

Para la SPL se desarrolló un componente SCA cuyo único servicio multiplica dos matrices utilizando el algoritmo de Strassen. Sin embargo, para los productos que implementan una estrategia híbrida (e.g. *multiplicación de bloques*) no se usó tal componente, en su lugar se importaron las clases Java directamente. Esto para evitar el tiempo adicional en envío de datos a través de la red.

9.1.5. Multiplicación de bloques de matrices

El fin último de la multiplicación particionada de matrices es encontrar el umbral entre la cantidad de datos transmitidos en la red y el tamaño de las submatrices a multiplicar, en el que la multiplicación de 2 matrices es óptima [14]. La estrategia de multiplicación, en este caso, consiste en dividir las matrices (cuadradas) en bloques de un tamaño determinado, y

multiplicarlos como si fueran una sola posición de cada matriz. El tamaño del bloque es decisivo para hallar dicho umbral; durante las pruebas ejecutadas en el Laboratorio de Arquitectura de Software (LASO) se encontró que para matrices de aproximadamente 3600x3600 elementos, el tamaño de bloque con mejores tiempos de ejecución es de 200.

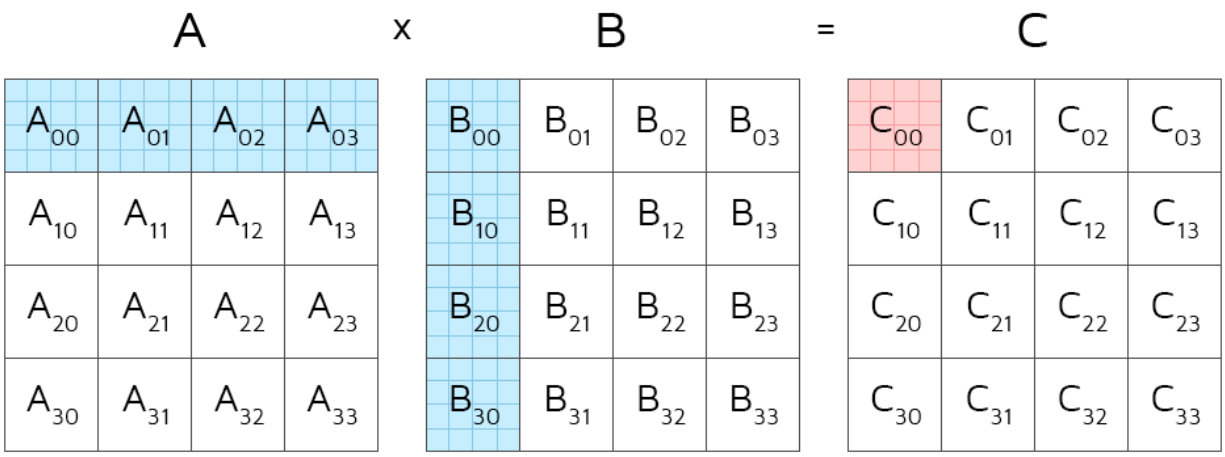


Figura No. 30 Estrategia de distribución por bloques

La figura No. 30 representa la dinámica de distribución de bloques. Cada bloque es tratado como una sola posición de la matriz correspondiente; así, para hallar el bloque C_{00} se deben multiplicar y sumar los bloques de la siguiente manera:

$$C_{00} = A_{00}*B_{00} + A_{01}*B_{10} + A_{02}*B_{20} + A_{03}*B_{30}$$

En donde cada multiplicación requiere el envío de dos bloques de elementos a través de la red.

9.1.6. Multiplicación de filas y columnas de bloques de matrices

La multiplicación de filas y columnas de bloques es una mejora, en términos de aprovechamiento de la red, a la estrategia anterior. Sin embargo, es más exigente en términos de procesamiento y memoria.

En la multiplicación de bloques, cada par de bloques debe ser enviado a un procesador tantas veces como columnas o filas de bloques hayan,[10] con el fin de hallar cada bloque en la matriz resultante; es decir, que por un solo bloque resultante, se deben hacer varios envíos. En el caso de la multiplicación por filas y columnas de bloques, se reduce considerablemente el número de envíos *atómicos*[11], ya que para hallar un bloque resultante basta con un sólo envío.

[10] Aplica solamente para matrices cuadradas
[11] Envíos de bloques o submatrices

Esta estrategia es relativamente parecida a la estrategia anterior, reduciendo el número de bloques por envío.

Otra ventaja importante que presenta la multiplicación por filas y columnas de bloques, respecto a la estrategia anterior, es que reduce el número de procesadores necesarios para calcular la matriz resultante. Aunque, como se dijo anteriormente, requiere un nivel mayor de capacidad de procesamiento y memoria.

9.1.7. Almacenamiento

La SPL considera dos formas de almacenamiento: compartido y distribuido. La ejecución de los módulos PaSCAni se realizó utilizando un dispositivo de almacenamiento conectado a la red (NAS, por su nombre en inglés *Network-attached storage*); es decir, se usó almacenamiento compartido.

9.2. Productos generados

Los productos generados a partir del modelo de variabilidad son:

1.
    a. Multiplicador de n matrices
    b. Multiplicador de 2 matrices
    c. Multiplicación de filas y columnas de bloques
    d. Almacenamiento compartido
    e. Parentización paralela
    f. Recuperación

2.
    a. Multiplicador de n matrices
    b. Multiplicador de 2 matrices
    c. Multiplicación de filas y columnas de bloques
    d. Almacenamiento compartido
    e. Parentización secuencial
    f. Recuperación

3.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de filas y columnas de bloques
d. Almacenamiento distribuido

4.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de filas y columnas de bloques
d. Almacenamiento distribuido
e. Parentización paralela
f. Recuperación

5.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de filas y columnas de bloques
d. Almacenamiento distribuido
e. Parentización secuencial
f. Recuperación

6.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de bloques
d. Almacenamiento compartido

7.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de bloques
d. Almacenamiento compartido
e. Parentización paralela
f. Recuperación

8.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c.

d. Multiplicación de bloques
e. Almacenamiento compartido
f. Parentización secuencial
g. Recuperación

9.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de bloques
d. Almacenamiento distribuido

10.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de bloques
d. Almacenamiento distribuido
e. Parentización paralela
f. Recuperación

11.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Multiplicación de bloques
d. Almacenamiento distribuido
e. Parentización secuencial
f. Recuperación

12.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento compartido

13.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento compartido
e. Parentización paralela

f. Recuperación

14.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento compartido
e. Parentización secuencial
f. Recuperación

15.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento distribuido

16.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento distribuido
e. Parentización paralela
f. Recuperación

17.

a. Multiplicador de n matrices
b. Multiplicador de 2 matrices
c. Strassen
d. Almacenamiento distribuido
e. Parentización secuencial
f. Recuperación

9.3. Configuración de los productos

De los 17 productos anteriores, se configuraron y probaron 3. A continuación se describe brevemente cada configuración:

- ✓ Multiplicación de dos matrices (strassen)

  Un componente SCA que provee el servicio de multiplicar un par de matrices utilizando el algoritmo de Strassen.

- ✓ Multiplicación híbrida (hybrid-multiplication)

  Conjunto de componentes que provee el servicio de multiplicar un par de matrices haciendo uso de la estrategia híbrida de multiplicación.

- ✓ Multiplicación de n matrices (nmatrices)

  Conjunto de componentes que provee el servicio de multiplicar un conjunto de matrices, provenientes de un directorio de imágenes, haciendo uso de la estrategia híbrida de multiplicación.

9.4. Módulos de prueba PaSCAni

A continuación se presentan los módulos de prueba escritos en PaSCAni. Siguiente a cada módulo se presenta la estructura de archivos generada por el compilador.

Adicional a los módulos de prueba de los productos generados, se presenta un módulo de prueba para la SPL. Su función principal es integrar los módulos y comparar las estrategias de multiplicación, con el fin de encontrar la configuración óptima de componentes de software (objetivo principal de la SPL).

```
package org.driso.matrices.strassen.resources;

java-import org.driso.matrices.common.*;
java-import java.util.Arrays;

module StrassenTest {

    composite strassen = loadComposite("Strassen.composite") providing services {
```

```
        rmi multiplication@"localhost":1099 with interface
        org.driso.matrices.strassen.interfaces.MatricesMultiplicationService
    } using libraries {
        "strassen/mcm-strassen.jar",
        "lib/mcm-common.jar"
    };

    testsuite checkCorrectness(int[][] A, int[][] B) using strassen.multiplication
{

        int[][] C = strassen.multiplication.multiplicate(A, B);
        int[][] naiveC = MatrixUtils.multiplyUsingNaiveAlgorithm(A, B);

        test correctness = Arrays.deepEquals(C, naiveC)
            labeled "Multiplication's correctness"
            message when passed: "The algorithm is well implemented"
                        failed: "Wrong implementation"
                        exception: "An exception has been thrown"
                        inconclusive: "There is no conclusion";

        return correctness;
    }

    testsuite compareWithClassicStrassen(
        int[][] A, int[][] B, long classicStrassenTime
    ) using strassen.multiplication {

        long measured = exectime strassen.multiplication.multiplicate(A, B);
        Console.log("Strassen SCA measured time: " + measured);

        test isUnderExpectedTime =      measured <= classicStrassenTime
            labeled "Comparisson between Strassen SCA and Strassen POJO"
            message when passed: "Strassen SCA's execution time is less or" +
                                 "equal than Strassen POJO's execution time"
                        failed: "It does not worth executing Strassen SCA"
                        exception: "An exception has been thrown"
                         inconclusive: "There is no conclusion";

        return isUnderExpectedTime;
    }

    control {

        String pathA = "/home/sas1/app/matrices/images/A1";
        String pathB = "/home/sas1/app/matrices/images/A2";
        int blocksize = 200;

        int[][] A = ImageUtils.
             getImageData(
                ImageUtils.readImageFromLocalURL(pathA)
            );

        int[][] B = ImageUtils.
```

```
            getImageData(
                ImageUtils.readImageFromLocalURL(pathB)
            );

         MatrixUtils mutils = new MatrixUtils();
        long classicStrassenTime =
             exectime mutils.multiplicationMatrixReturn(A, B);

        compareWithClassicStrassen(A, B, classicStrassenTime);
        checkCorrectness(A, B);

       }
}
```

Figura No. 32 Módulo de prueba PaSCAni correspondiente al producto número 1

```
strassen
 |--src
   |--org
     |--pascani
       |--commons
         |--templates
           |--TestsResultsTemplate.soy
           --TestsResultsTemplateSoyInfo.java
         |--Commons.java
         --Console.java
         --Service.java
         --Test.java
         --TestsResultsGenerator.java
       |--components
         |--RunnerImpl.java
         --StrassenTest_CheckCorrectnessImpl.java
         --StrassenTest_ControlImpl.java
         --StrassenTest_CompareWithClassicStrassenImpl.java
       |--resources
         |--runner.composite
         --StrassenTest.composite
       |--services
         |--RunnableControl.java
         --RunnableTestsuite.java
 |--lib
  |--soy-latest.jar
 |--StrassenTest.jar
 --pascani-configuration.properties
 --testresults.xml
```

Figura No. 33 Estructura de archivos generada por PaSCAni para el producto No. 1

```
package org.driso.matrices.hybrid_multiplication.resources;

java-import org.driso.matrices.common.*;
```

```
java-import java.util.Arrays;

module HybridMultiplicationTest {

    composite hybrid_multiplication =
loadComposite("HybridMultiplication.composite")
    providing services {
        rmi hybridMultiplication@"localhost":1096 with interface

org.driso.matrices.hybrid_multiplication.interfaces.HybridMultiplicationService
    } using libraries {
        "hybrid-multiplication/mcm-hybrid-multiplication.jar",
        "lib/mcm-common.jar"
    };

    testsuite checkCorrectness(String pathA, String pathB, String pathC, int
blocksize)
    using hybrid_multiplication.hybridMultiplication {

        int[][] A =
ImageUtils.getImageData(ImageUtils.readImageFromLocalURL(pathA));
        int[][] B =
ImageUtils.getImageData(ImageUtils.readImageFromLocalURL(pathB));

        // Service call
        hybrid_multiplication.
            hybridMultiplication.multiply(pathA, pathB, pathC, blocksize);

        int[][] C =
ImageUtils.getImageData(ImageUtils.readImageFromLocalURL(pathC));
        int[][] naiveC = MatrixUtils.multiplyUsingNaiveAlgorithm(A, B);

        test correctness = Arrays.deepEquals(C, naiveC)
            labeled "Multiplication's correctness"
            message when passed: "The algorithm is well implemented"
                        failed: "Wrong implementation"
                        exception: "Some exception has been threw"
                        inconclusive: "There is no conclusion";

        return correctness;
    }

    testsuite checkExecutionTime(
        String A,
        String B,
        String C,
        int blocksize,
        long expectedTime
    ) using hybrid_multiplication.hybridMultiplication {

        long measured =
            exectime hybrid_multiplication.
```

```
                hybridMultiplication.multiply(A, B, C, blocksize);

        Console.log("Hybrid-multiplication measured time: " + measured);

        test isUnderExpectedTime = measured <= expectedTime
            labeled "Comparisson between Hybrid-multiplication and Strassen SCA"
            message when passed: "Hybrid-multiplication's execution time is less
or " +
                                 "equal than Strassen-SCA's execution time"
                           failed: "No vale la pena ejecutar el algoritmo hibrido
para"
                           exception: "Some exception has been threw"
                           inconclusive: "There is no conclusion";

        return isUnderExpectedTime;
    }

    control {

        String pathA = "/home/sas1/app/matrices/images/A1";
        String pathB = "/home/sas1/app/matrices/images/A2";
        String pathC = "/home/sas1/app/matrices/images/results/C_A1_A2.jpg";
        int blocksize = 200;
        long expectedTime = 30000;

        checkExecutionTime(pathA, pathB, pathC, blocksize, expectedTime);
        checkCorrectness(pathA, pathB, pathC, blocksize);
    }
}
```

Figura No. 34 Módulo de prueba PaSCAni correspondiente al producto número 2

```
hybrid-multiplication
 |--src
    |--org
       |--pascani
          |--commons
             |--templates
                |--TestsResultsTemplate.soy
                 --TestsResultsTemplateSoyInfo.java
             |--Commons.java
              --Console.java
              --Service.java
              --Test.java
              --TestsResultsGenerator.java
          |--components
             |--RunnerImpl.java
              --HybridMultiplicationTest_CheckCorrectnessImpl.java
              --HybridMultiplicationTest_CheckExecutionTimeImpl.java
              --HybridMultiplicationTest_CheckCorrectnessImpl.java
```

```
|   |   |--resources
|   |   |   |--HybridMultiplicationTest.composite
|   |   |   |--runner.composite
|   |   |--services
|   |   |   |--RunnableControl.java
|   |   |   |--RunnableTestsuite.java
|--lib  |   |
|  |--soy-latest.jar
|--HybridMultiplicationTest.jar
|--pascani-configuration.properties
|--testresults.xml
```

Figura No. 35 Estructura de archivos generada por PaSCAni para el producto No. 2

```
package org.driso.matrices.resources;

java-import org.driso.matrices.common.*;

module NMatricesTest {

    composite _control = loadComposite("Control.composite") providing services {
        rmi matrixChainMultiplication@"localhost":2100 with interface
        org.driso.matrices.interfaces.Executable
    } using libraries {
        "nmatrices/mcm-nmatrices.jar",
        "lib/mcm-common.jar"
    };

    testsuite checkExecutionTime(String imagesDirectory, long expectedTime)
    using _control.matrixChainMultiplication {
        long measured =
            exectime _control.matrixChainMultiplication.execute(imagesDirectory);

        Console.log("NMatrices measured time: " + measured);

        test isUnderExpectedTime = measured <= expectedTime
            labeled "Multiplication's performance"
            message when passed: "Execution time expectations accomplished"
                        failed: "In order to get a lower execution time, it is "
+
                                "necessary to enhance implementation"
                        exception: "An exception has been thrown"
                        inconclusive: "There is no conclusion";

        return isUnderExpectedTime;
    }

    control {

        String imagesDirectory = "/home/sas1/app/matrices/images/";
        long expectedTime = 30000;
```

```
        checkExecutionTime(imagesDirectory, expectedTime);

    }

}
```

Figura No. 36 Módulo de prueba PaSCAni correspondiente al producto número 3

```
nmatrices
 |--src
 |   |--org
 |   |   |--pascani
 |   |   |   |--commons
 |   |   |   |   |--templates
 |   |   |   |   |   |--TestsResultsTemplate.soy
 |   |   |   |   |   --TestsResultsTemplateSoyInfo.java
 |   |   |   |   |--Commons.java
 |   |   |   |   --Console.java
 |   |   |   |   --Service.java
 |   |   |   |   --Test.java
 |   |   |   |   --TestsResultsGenerator.java
 |   |   |   |--components
 |   |   |   |   |--RunnerImpl.java
 |   |   |   |   --NMatricesTest_CheckExecutionTimeImpl.java
 |   |   |   |   --NMatricesTest_ControlImpl.java
 |   |   |   |--resources
 |   |   |   |   |--NMatricesTest.composite
 |   |   |   |   --runner.composite
 |   |   |   |--services
 |   |   |   |   |--RunnableControl.java
 |   |   |   |   --RunnableTestsuite.java
 |--lib  |   |
 |   |--soy-latest.jar
 |--NMatricesTest.jar
 |--pascani-configuration.properties
 |--testresults.xml
```

Figura No. 37 Estructura de archivos generada por PaSCAni para el producto No. 3

```
package org.driso.pascani;

import org.driso.matrices.strassen.resources.StrassenTest;
import
org.driso.matrices.hybrid_multiplication.resources.HybridMultiplicationTest;
import org.driso.matrices.resources.NMatricesTest;
java-import org.driso.matrices.common.*;

module ChainMatrixMultiplication {

    testsuite compareMethods(
      int[][] A,
```

```
      int[][] B,
      String pathA,
      String pathB,
      String pathC,
      int blocksize
    ) using StrassenTest.strassen.multiplication,
          HybridMultiplicationTest.hybrid_multiplication.hybridMultiplication {

        long strassenTime =
             exectime StrassenTest.
                    strassen.multiplication.multiplicate(A, B);

        long hybridMultiplicationTime =
             exectime HybridMultiplicationTest.
                         hybrid_multiplication.
                             hybridMultiplication.
                                 multiply(pathA, pathB, pathC, blocksize);

        test betterMethod = strassenTime < hybridMultiplicationTime
            labeled "Comparisson among multiplication methods"
            message when passed: "Strassen SCA presents better performance than "
+
                                 "Hybrid-multiplication"
                         failed: "Hybrid-multiplication presents better
performance" +
                                 "than Strassen SCA"
                         exception: "An exception has been thrown"
                         inconclusive: "There is no conclusion";

        return betterMethod;
    }

    control {

        String imagesDirectory = "/home/sas1/app/matrices/images/";
        String pathA = "/home/sas1/app/matrices/images/A1";
        String pathB = "/home/sas1/app/matrices/images/A2";
        String pathC = "/home/sas1/app/matrices/images/results/C_A1_A2.jpg";
        int blocksize = 200;
        long expectedTime = 30000, expectedTimeNMatrices = 120000;

        int[][] A = ImageUtils.
                        getImageData(ImageUtils.readImageFromLocalURL(pathA));

        int[][] B = ImageUtils.
                        getImageData(ImageUtils.readImageFromLocalURL(pathB));

        StrassenTest.compareWithClassicStrassen(A, B, expectedTime);
        StrassenTest.checkCorrectness(A, B);

        HybridMultiplicationTest.
            checkExecutionTime(pathA, pathB, pathC, blocksize, 30000);
```

```
        HybridMultiplicationTest.
            checkCorrectness(pathA, pathB, pathC, blocksize);

        NMatricesTest.checkExecutionTime(imagesDirectory, expectedTimeNMatrices);

        compareMethods(A, B, pathA, pathB, pathC, blocksize);

    }

}
```

Figura No. 38 Módulo de prueba PaSCAni correspondiente a la SPL (3 productos)

```
chain-matrix-multiplication
 |--src
   |--org
     |--pascani
       |--commons
         |--templates
           |--TestsResultsTemplate.soy
           --TestsResultsTemplateSoyInfo.java
         |--Commons.java
         --Console.java
         --Service.java
         --Test.java
         --TestsResultsGenerator.java
       |--components
         |--ChainMatrixMultiplication_CompareMethodsImpl.java
         --ChainMatrixMultiplication_ControlImpl.java
         --HybridMultiplicationTest_CheckCorrectnessImpl.java
         --HybridMultiplicationTest_CheckExecutionTimeImpl.java
         --NMatricesTest_CheckExecutionTimeImpl.java
         --RunnerImpl.java
         --StrassenTest_CheckCorrectnessImpl.java
         --StrassenTest_CompareWithClassicStrassenImpl.java
       |--resources
         |--ChainMatrixMultiplication.composite
         --HybridMultiplicationTest.composite
         --NMatricesTest.composite
         --StrassenTest.composite
         --runner.composite
       |--services
         |--RunnableControl.java
         --RunnableTestsuite.java
 |--lib
   |--soy-latest.jar
 |--ChainMatrixMultiplication.jar
 --pascani-configuration.properties
 --testresults.xml
```

Figura No. 39 Estructura de archivos generada por PaSCAni para la SPL (3 productos)

9.5. Análisis de resultados

El proceso de escritura de los módulos de prueba es relativamente rápido y fácil. La integración que realiza PaSCAni entre instrucciones Java e instrucciones propias facilita al desarrollador de pruebas la escritura de los casos de prueba, usando sus propias librerías y sin preocupaciones referentes al consumo y la provisión de servicios.

Adicionalmente, el compilador de PaSCAni hace transparente la complejidad de la implementación del código en Java necesario para ejecutar las pruebas a nivel de servicio. Esto último hace de PaSCAni una buena alternativa para hacer validación y verificación (V&V) en tiempo de ejecución.

10. Guía práctica para el desarrollo de pruebas con PaSCAni

Esta guía aborda el proceso completo de generación y ejecución de componentes de prueba bajo el estándar SCA, usando la especificación de PaSCAni y el middleware FraSCAti. Adicionalmente, se incluyen los pasos para derivar automáticamente del modelo de variabilidad los productos de software usando GNU Prolog. En esta guía se usará el modelo de variabilidad presentado en la sección del caso de estudio.

10.1. Representación del Modelo de variabilidad en GNU Prolog

A continuación se describe el código que representa el modelo de variabilidad de la figura 28, presentado en la figura 29.

```
productline(L):-

// Se crea la lista de features
L=[
    Multiplicador_N_Matrices,
    Multiplicador_2_Matrices,
    Strassen,
    Block_Reduce,
    Hybrid_Multiplication,
    Almacenamiento,
    Distribuido,
    Compartido,
    Parentizacion,
    Secuencial,
    Paralela,
    Recuperacion
],

// Se asigna el dominio (0, 1) a las variables de la lista.
fd_domain(
    [
        Multiplicador_N_Matrices,
        Multiplicador_2_Matrices,
        Strassen,
        Block_Reduce,
        Hybrid_Multiplication,
        Almacenamiento,
        Distribuido,
        Compartido,
        Parentizacion,
        Secuencial,
        Paralela,
        Recuperacion
```

```
    ],
    0, 1
),
```

Después de configurar la lista de features del modelo, deben establecerse las relaciones de obligatoriedad, opcionalidad y cardinalidad:

```
/*
* La feature principal Multiplicador_N_Matrices es obligatoria, por lo tanto siempre
* debe ser igual a 1
*/
Multiplicador_N_Matrices#=1,

Multiplicador_N_Matrices#<=>Multiplicador_2_Matrices,
Multiplicador_N_Matrices#<=>Almacenamiento,

/*
*    Dado    que    Parentización    es    opcional,    la    feature    principal Multiplicador_N_Matrices
* debe ser mayor o igual que el valor de Parentización, de tal manera que:
*
* 1 >= 0 o 1 >= 1
*
* Igual ocurre en el caso de la Recuperación.
*/
Multiplicador_N_Matrices#>=Parentizacion,
Multiplicador_N_Matrices#>=Recuperacion,

/*
* Se expresan los dos lados de la inclusión Recuperación ⇔ Parentización, que
* representa el hecho de que no puede haber Recuperación sin Parentización y
* viceversa.
*
* ¬Parentización v  Recuperación
*  Parentización v ¬Recuperación
*/
(1-Parentizacion)+Recuperacion#>0,
Parentizacion+(1-Recuperacion)#>0,

/*
* La suma de los valores de Strassen, Block_Reduce y Hybrid_Multiplication no debe
*  ser mayor que el valor de la feature padre Multiplicador_2_Matrices. De tal manera * que solamente se pueda seleccionar una de las tres. Igual ocurre en los casos
* siguientes.
*
* Cuando la feature padre no se encuentra en el producto derivado, es decir vale 0,
* las features hijas deben tener el mismo valor.
```

```
*/
1*Multiplicador_2_Matrices#=Strassen+Block_Reduce+Hybrid_Multiplication,
1*Almacenamiento#=Distribuido+Compartido,
1*Parentizacion#=Secuencial+Paralela,
```

Hasta aquí ya se ha representado completamente el modelo de variabilidad de la figura 28. En las siguientes instrucciones se configura una nueva lista que contiene los nombres descriptivos de las features y las instrucciones necesarias para generar los productos resultantes. Antes de continuar con la descripción del código, se muestra a continuación la salida de este programa:

```
X = [1,1,0,0,1,1,0,1,1,0,1,1]
Y = [
       'Multiplicador_N_Matrices',
       'Multiplicador_2_Matrices',
       n,
       n,
       'Hybrid_Multiplication',
       'Almacenamiento',
       n,
       'Compartido',
       'Parentizacion',
       n,
       'Paralela',
       'Recuperacion'
]

X = [1,1,0,0,1,1,0,1,1,1,0,1]
Y = [
       'Multiplicador_N_Matrices',
       'Multiplicador_2_Matrices',
       n,
       n,
       'Hybrid_Multiplication',
       'Almacenamiento',
       n,
       'Compartido',
       'Parentizacion',
       'Secuencial',
       n,
       'Recuperacion'
]

                                        .  .  .
```

La lista X de cada producto contiene el valor de las features en el dominio (1, 0) y la lista Y el nombre correspondiente a cada valor. La letra n representa la ausencia de la feature en el producto derivado.
Las siguientes instrucciones permiten generar el listado de nombres (lista Y) a partir de la solución (lista X) del solver, que en este caso es GNU Prolog.

```
fd_labeling(L).
lista(M,L):-
       productline(M),

       A=[
            'Multiplicador de n matrices',
            'Multiplicador de 2 matrices',
            'Strassen',
            'Block_Reduce',
            'Multiplicación de filas y columnas de bloques',
            'Almacenamiento',
            'Distribuido',
            'Compartido',
            'Parentización',
            'Secuencial',
            'Paralela',
            'Recuperación'
        ],

predicado(M,L,A).

predicado(M,L,A):-
       M=[],
       L=[],
       A=[].

predicado(M,L,A):-
       M=[H|T],
       L=[H1|T1],
       A=[H2|T2],
       predicado2(H,H1,H2), true
       predicado(T,T1,T2).

predicado2(X,Y,W):- X >0 -> Y=W ; Y='n'.
```

Todas las instrucciones anteriores deben ser escritas en un archivo .pl, que en nuestro caso se llama "representación-modelo-de-variabilidad.pl". Finalmente, desde GNU Prolog debe ejecutarse los siguientes comandos:

```
consult('representación-modelo-de-variabilidad.pl').\r
lista(X,Y).\r
```

10.2. Configuración de los productos de software

Una vez realizada la generación, se debe realizar la configuración de los assets para los productos seleccionados; en el caso de este proyecto, los assets son componentes SCA.

Los productos seleccionados a implementar son:

- ✓ Multiplicación de dos matrices (Strassen)
- ✓ Multiplicación híbrida (Hybrid_Multiplication)
- ✓ Multiplicación de n matrices (Multiplicacion_N_Matrices)

Sin embargo, esta guía se centra en el primer producto: Multiplicación de dos matrices usando el algoritmo de Strassen.

Dado el alcance de este proyecto y la motivación principal (el desarrollo de la especificación de PaSCAni), los productos se generaron de manera manual. Sin embargo, las características de los componentes SCA permiten realizar re-configuraciones a la arquitectura y por lo tanto a sus relaciones. Una propuesta más avanzada podría considerar el uso de GNU Prolog + Java para realizar la configuración automática de los productos de software (i.e. adaptar las interfases y configurar el binding de los servicios).

Posterior al diseño del modelo de variabilidad se desarrollaron los siguientes componentes SCA:

- ➔ Strassen

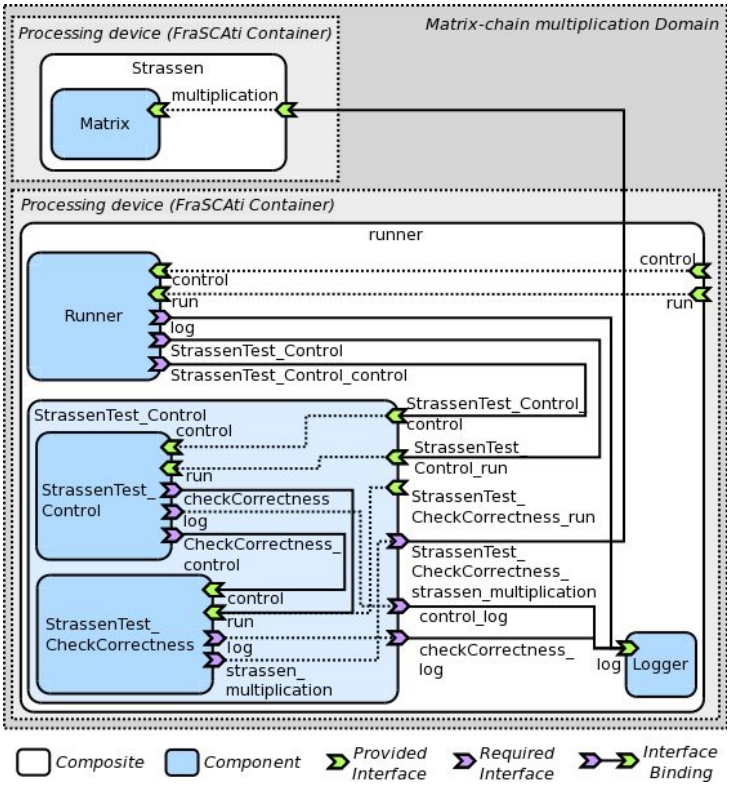


Figura No. 40. Diagrama de componentes SCA para el producto No. 1: Strassen

10.3. Diseño de la estrategia de pruebas con PaSCAni

A continuación se describe de forma general la composición del módulo de prueba del producto seleccionado. El módulo `StrassenTest` está destinado a probar la correctitud y el tiempo de ejecución del servicio de multiplicación.

```
package org.driso.matrices.strassen.resources;

/*
 * Dado que se está probando un sólo componente, no es necesario realizar imports
de
 * otros módulos de prueba.
 */
module StrassenTest {

/*
 * Se específica el componente a probar y sus servicios. Esta sección del módulo
 * está destinada a describir la arquitectura del software a probar, en términos
de
 * sus componentes.
 */
    composite strassen = . . .;
/*
 * Este testsuite comprueba la correctitud de la multiplicación para la
 * implementación de Strassen. Para esto, se reciben las dos matrices a
multiplicar
 * como parámetro y se especifica qué servicio se está usando.
 */
    testsuite checkCorrectness(int[][] A, int[][] B) using strassen.multiplication
{
        . . .
    }

/*
 * Este testsuite compara el tiempo de ejecución del producto con el tiempo de
 * ejecución de la implementación Plain Old Java Object (POJO) del algoritmo de
 * Strassen. Se recibe como parámetro las matrices a multiplicar y el tiempo de la
 * versión POJO en milisegundos. Al igual que en el caso anterior, debe
 * especificarse el servicio a probar.
 */
    testsuite compareWithClassicStrassen(int[][] A, int[][] B, long classicTime
    ) using strassen.multiplication {
        . . .
    }

/*
 * El control contiene la especificación de la estrategia de prueba; es decir, la
 * secuencia de llamados a los testsuite combinado con expresiones Java. En el
caso
 * de este módulo, las expresiones Java corresponden a la lectura de las matrices
```

```
y
 * la ejecución de Strassen POJO.
 */
    control {
        . . .
    }
}
```

El módulo completo se puede ver en la figura No. 32. Cada `testsuite` debe retornar un `test` que define el veredicto de la prueba; de la misma manera como lo hacen los métodos (no void) en Java.

10.4. Compilación del proyecto PaSCAni

PaSCAni ofrece una interfaz por línea de comando que permite compilar y ejecutar un módulo de pruebas. En el caso de las pruebas compuestas, se debe compilar el módulo cuyo control contiene la estrategia de pruebas (el control de los otros módulos será ignorado). Una vez se hayan configurado las variables de entorno JAVA_HOME, FRASCATI_HOME, PASCANI_HOME y el PATH (apuntando a los binarios de Java, FraSCAti y PaSCAni) el usuario podrá ejecutar el comando `pascani` a través de su terminal. Al hacerlo, verá el siguiente mensaje de ayuda:

```
equipo:~ Usuario$ pascani

Usage: pascani [options] [arguments]
Legal options include:
       compile<file> generate Java classes and FraSCAati composites correspon...
       run           <file> execute the specified PaSCAni project
Legal arguments include:
       -sourcepath    <directory>    specify the directory in which compiler will look...
       -classpath     <directory>    specify the directory in which compiler will look...
       -destination   <directory>    specify the directory in which generated files wi...
       -warnings             print warnings after compilation
       -version              print the version information for PaSCAni and exit
       -help                 print this message
```

Para compilar un módulo, como se explica en el mensaje de ayuda, se debe ejecutar `pascani` con los siguientes parámetros:

```
pascani compile     -sourcepath         <fuentes de módulos pascani>
                    -classpath          <fuentes de clases java>
                    -destination        <directorio para la generación de código>
```

El parámetro `classpath` permite especificar tanto carpetas de archivos como librerías jar. El `sourcepath` permite especificar carpetas de archivos que contengan módulos de prueba PaSCAni. En ambos casos, para indicar más de una fuente se debe usar un separador dependiendo del sistema operativo; en el caso de Windows debe usarse “;” y en Linux “:”.

Para la compilación del producto Strassen se usó:

```
cd /home/Usuario/.../ruta-al-producto/
```

```
pascani compile  \
-sourcepath  src/ \
-classpath
../dist/assets/strassen/mcm-strassen.jar:../dist/assets/lib/mcm-common.jar \
-destination /home/Usuario/tests/strassen
```

10.5. Ejecución de componentes SCA con FraSCAti

Una vez se haya realizado la compilación de los módulos de prueba, se deben desplegar los componentes del sistema a ser probado (previo a la ejecución de los componentes de prueba). Para realizar la ejecución de dichos componentes, diríjase a OW2 FraSCAti User Guide en el capitulo No. 3, sección The frascati command.

Con el fin de automatizar la ejecución en un grid de computadores, en este proyecto se realizó una especificación de Shell Script usando expect. Dicho archivo se especifica a continuación:

```
#!/usr/bin/expect
 eval spawn ssh -oStrictHostKeyChecking=no -oCheckHostIP=no sas1@grid1
 set prompt ":|#|\\\$"
 interact -o -nobuffer -re $prompt return
 send "contraseña\r"
 interact -o -nobuffer -re $prompt return
 send "cd /home/sas1/app/matrices/\r"
 interact -o -nobuffer -re $prompt return
 send "frascati run Strassen -libpath mcm-strassen.jar:../lib/mcm-common.jar\r"
 interact
```

Fig. # Automatización de despliegue de componentes SCA usando Expect

10.6. Ejecución de los componentes de prueba

Posterior al despliegue de los componentes del sistema a ser probado, debe realizarse el despliegue de los componentes de prueba generados por PaSCAni. En el caso del sistema operativo Windows, PaSCAni realizará esta tarea de forma automática usando el comando `pascani run` y especificando la ruta del archivo `pascani-configuration.properties` generado en la carpeta destino especificada en el momento de la compilación. En el caso de Linux, PaSCAni desplegará las instrucciones de despliegue.

11. Resultados y Conclusiones

- Desarrollamos una nueva forma de validar y verificar sistemas de software basados en componentes SCA.

- PaSCAni reduce altamente el esfuerzo del desarrollador de pruebas al escribir y ejecutar pruebas sobre el sistema.

- Estamos proponiendo una solución versátil, útil no sólo en el contexto de los sistemas de software basados en componentes sino también en los sistemas auto-adaptativos, como una línea base para, gradualmente, construir funcionalidades más grandes y complejas, como la auto-recuperación.

- A problemas específicos, soluciones específicas. Los DSL hacen transparente la complejidad de las tecnologías y permiten al desarrollador enfocarse en la solución.

- Es necesario enfrentar la carencia de semántica en las soluciones de software. El usuario debe sentir que resuelve un problema a partir de conceptos conocidos y afines al contexto correspondiente.

- Un enfoque modular en los mecanismos de validación y verificación, en tiempo de ejecución, contribuye no sólo a la realización de pruebas integrales sino también a garantizar el cumplimiento de los requerimientos en sistemas cuya arquitectura cambia constantemente.

- La ejecución de las pruebas preliminares de PaSCAni son aceptables en tiempo de respuesta. El tiempo de compilación es muy comparable a los resultados de compilación de java e igualmente los de ejecución.

12. Trabajo futuro

- Mejorar la gramática de PASCANI, en términos de expresividad.

- Desarrollar herramientas que le permitan al desarrollador de pruebas ser más productivo (editor, plugin de eclipse para PaSCAni Explorer).

- Integrar PaSCAni a un sistema de software auto-adaptativo y desarrollar las clases necesarias para realizar auto-recuperación basada en la ejecución de pruebas propuesta.

13. Referencias Bibliográficas

[1] SWEBOK executive editors, Alain Abran, James W. Moore; editors, Pierre Bourque, Robert Dupuis. (2004). *Guide to the Software Engineering Body of Knowledge - 2004 Version*. IEEE Computer Society. pp. 1–1

[2] GARCIA LEON, Delba y BELTRAN BENAVIDES, Alfa. Un enfoque actual sobre la calidad del software. ACIMED [online]. 1995, vol.3, n.3 [citado 2014-02-01], pp. 40-42 . Disponible en: <http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S1024-94351995000300005>. ISSN 1024-9435.

[3] Glenford J. Myers, Art of Software Testing, John Wiley & Sons, Inc., New York, NY, 1979

[4] Montilva, J., & Barrios, J. (10 de Octubre de 2007). DESARROLLO DE SOFTWARE EMPRESARIAL. Bogotá , Cundinamarca, Colombia.

[5] Lundqvist, K. (2002, Septiembre). *Aerospace Software Engineering*. Boston, Massachusetts, Estados Unidos de América.

[6] One Stop Testing - V model. Recuperado el 10 de Octubre de 2012, de: http://www.onestoptesting.com/sdlc-models/v-model.asp

[7] Tamura, G., Villegas, N., Müller, H., Sousa, J., Becker, B., Karsai, G., y otros. (2012). *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems.* Rocquencourt: Software Engineering for Self-Adaptive Systems 2 Springer (Ed.) (2012) 116-141.

[8] CASTAÑEDA BUENO, Lorena. A Reference Architecture for Component-Based Self-Adaptive Software Systems. Santiago de Cali, 2012, Tesis Maestría (Máster en Gestión de Informática y Telecomunicaciones concentración en Ingeniería del Software). Universidad Icesi. Facultad de Ingeniería.

[9] M. Visconti, P. Antiman and P. Rojas. Experiencia con un modelo de madurez para el mejoramiento del proceso de aseguramiento de calidad del software. Novatica, 125, (1997), 18–21.

[10] *Thomas B. Hilburn , Massood Townhidnejad, Software quality: a curriculum postscript?, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, p.167-171, March 07-12, 2000, Austin, Texas, USA.*

[11] Cheng, B.H., de Lemos, R., Giese, H., et al.: Software engineering for self-adaptive

systems: A research roadmap. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009).

[12] Villegas, N., Müller, H.: Context-driven adaptive monitoring for supporting soa governance. Proceedings of the 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA)

[13] S. Huss-lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation. In Proceedings of Supercomputing'96, pages 9-6, 1996.

[14] A MapReduce Algorithm for Matrix Multiplication. Recuperado el 13 de Septiembre de 2013, de: http://www.norstad.org/matrix-multiply/

[15] Aho, A., Sethi, R., Ullman, & J. (1998). *Compilers: Principles, Techniques and Tools*, Addison Wesley. 1979.

[16] De Castro, R. *Teoría de la computación*. Universidad Nacional de Colombia, Departamento de Matemáticas. Bogotá, Colombia.

[17] Xtext. Recuperado el 22 de Noviembre de 2013, de: http://www.eclipse.org/Xtext/

[18] TTCN-3. Recuperado el 27 de Diciembre de 2013, de: http://www.ttcn-3.org/index.php/about/introduction

[19] CUP Parser Generator for Java. Recuperado el 27 de Diciembre de 2013, de: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[20] JLex: A Lexical Analyzer Generator for Java (TM). Recuperado el 27 de Diciembre de 2013, de: http://www.cs.princeton.edu/~appel/modern/java/JLex/

[21] T. Hu and M. Shing, “Computation of Matrix Chain Products. Part I,” SIAM J. Computing, vol. 11, pp. 362-373, May 1982.

[22] T. Hu and M. Shing, “Computation of Matrix Chain Products. Part II,” SIAM J. Computing, vol. 13, pp. 228-251, May 1984.

[23] A quick guide to SCA. Recuperado el 27 de Diciembre de 2013, de: http://tuscany.apache.org/quick-guide-to-sca.html