VALIDADOR DE CERTIFICADOS DE PROPIEDADES DE SEGURIDAD DE CODIGO FUENTE JAVA

LUIS FERNANDO GRISALES BADILLO

INGENIERO DE SISTEMAS Y TELECOMUNICACIONES

MAESTRÍA EN GESTIÓN Y DESARROLLO DE PROYECTOS DE SOFTWARE

UNIVERSIDAD AUTÓNOMA DE MANIZALES

MANIZALES, NOVIEMBRE DE 2016

VALIDADOR DE CERTIFICADOS DE PROPIEDADES DE SEGURIDAD DE CODIGO FUENTE JAVA

LUIS FERNANDO GRISALES BADILLO

INGENIERO DE SISTEMAS Y TELECOMUNICACIONES

Tutor

MAURICIO FERNANDO ALBA CASTRO, Ph.D.

MAESTRÍA EN GESTIÓN Y DESARROLLO DE PROYECTOS DE SOFTWARE

UNIVERSIDAD AUTÓNOMA DE MANIZALES

MANIZALES, NOVIEMBRE DE 2016

CONTENIDO

LISTA DE FIGURAS

Figura 19. Resultado validación código fuente y certificado de seguridad Adulterado.

Figura 20. Código fuente y Certificado Prueba funcionalidad completitud certificado.

Figura 21. Validación inicial completitud certificado.

Figura 22. Paralelo entre certificado seguridad adulterado y certificado original prueba completitud.

LISTA DE TABLAS

RESUMEN

En el trabajo desarrollado en (Alba, 2011) se diseñó una metodología de certificación de propiedades de seguridad de software escrito en código fuente Java, que modela los programas Java –su semántica- como un sistema de transición de estados y realiza un análisis del programa para determinar la alcanzabilidad de estados considerados como “no seguros”; como resultado final, si el programa es seguro –i.e. no alcanza estados inseguros- la metodología de certificación entrega un certificado de seguridad en el que se incluye la demostración formal de que el programa es seguro, la cual es una inferencia lógica que corresponde a una computación en programación lógica en el lenguaje Maude; este certificado lo puede obtener un consumidor de código para así tener una evidencia de que el código que va a utilizar es seguro, pero actualmente los posibles usuarios del código no tienen cómo comprobar estos certificados, es decir que no pueden comprobar si el certificado es válido (corresponde con el programa, y con la semántica del lenguaje Java).

La metodología de certificación se basa en la especificación de la semántica abstracta de Java en el lenguaje de programación declarativa y lógica Maude; el certificado es básicamente la traza de ejecución abstracta del programa Maude que hace la verificación de la no alcanzabilidad de estados inseguros mediante la interpretación del programa; esta traza es una inferencia lógica en lógica de reescritura y por esto puede utilizarse como demostración formal.

El trabajo presente en este documento muestra el desarrollo del subsistema de validación de los certificados generados por herramientas que usen la metodología de certificación de propiedades de seguridad desarrollada en el trabajo mencionado anteriormente. Básicamente lo que busca el subsistema de validación es garantizar tres cosas; 1) Que los certificados de seguridad entregados a un consumidor de código corresponden con el código fuente del programa del que se dice que pertenecen; 2) Que los cambios de estado del programa suministrados en el certificado, se correspondan con la aplicación de las reglas de reescritura de la semántica abstracta; 3) Que en el certificado se evidencie que se aplicaron todas las reglas de reescritura de la semántica abstracta Java que se podían aplicar en la interpretación del código fuente. La semántica abstracta de Java además de reglas de reescritura contiene funciones cuya aplicación también

significa cambios de estado en la interpretación del programa, pero el trabajo solo incluye la validación de la aplicación de las reglas.

La metodología de certificación y validación se enmarca en el paradigma PCC (*Proof-Carrying Code*) que tiene como objetivo la ejecución segura de código móvil, la arquitectura que plantea esta técnica contempla elementos del lado del productor y del consumidor del código, en el marco de esta técnica, la metodología de certificación realiza las actividades de parte del “productor” y el subsistema de validación está concebido para apoyar la parte del “consumidor”.

ABSTRACT

In the work developed in (Alba, 2011) was designed a certification methodology for security properties of Java code, which is responsible to make a reachability analysis of states considered as "unsafe", as a final result the certification methodology delivers a security certificate that includes all tests done by the reachability analysis, A code consumer can get this certificate in order to have a evidence that the code is safe to use; the certification methodology is based on an abstract Java semantics specification written in Maude, Actually this methodology does not have any element that facilitates the certificates validation from the side of potential code users.

This paper present the design of a validation methodology for certificates generated by tools that use the certification methodology for security properties developed in the work mentioned previously. The first validation methodology objective is to ensure that safety certificates delivered to a consumer of code correspond to the source code that says they belong to, the others objectives are to verify that rewrite steps provided in the certificate are correct and that the certificate has applied all possible rules to the Java source code, the validator design makes use of Java abstract semantics used by the certification methodology.

The PCC technique (*Proof-Carrying Code*) tries to guarantee the secure execution of mobile code, the architecture of this technique take into account elements for both producer and consumer of the code, in this context the certification methodology performs the activities part of the "producer" in PCC, the validation methodology is designed to support the "consumer" activities in PCC.

INTRODUCCIÓN

En la tesis de doctorado (Alba, 2011) se implementó la metodología JavaPCC para verificar propiedades de seguridad y políticas de confidencialidad (no interferencia y otras) de programas escritos en código fuente Java; según (Zdancewic, 2004) la no interferencia es la propiedad principal de los flujos de información, y el cumplimiento de esta propiedad debe garantizar que los datos y entradas privados o secretos del usuario no deben de afectar –interferir- los datos y salidas públicas del sistema; (Smith, 2006) realiza una clasificación de variables según su confidencialidad diferenciando las que pueden ser conocidas públicamente (de baja confidencialidad **L**) y las variables con valores privados o secretos (de alta confidencialidad **H**) y define que la no interferencia consiste en que un observador de los valores de las variables (**L**) no pueda inferir ni concluir ningún valor de las variables (**H**), como tal, la no interferencia busca que los datos secretos o privados de los usuarios solo sean conocidos y manipulados por ellos y no por "usuarios externos", lo cual apunta al cumplimiento de la confidencialidad de la información.

La metodología JavaPCC (Alba, 2011) incluye la especificación de la semántica operacional abstracta de Java realizada en Maude; en lugar de los valores concretos (por ejemplo valores numéricos enteros) de las variables del programa (por ejemplo de tipo **int**) utiliza valores abstractos de ciertos dominios abstractos (por ejemplo los niveles de confidencialidad **L**, **H** etc, o su paridad **Par, Impar**) y en lugar de las operaciones concretas (suma: 2 **+** 3 = 5, resta, etc) aplica operaciones abstractas correspondiente (suma: **Par + Impar** = **Impar**, etc). La abstracción permite reducir el número de estados posibles del programa y hace factible el análisis de alcanzabilidad que considera el flujo de información en todos los caminos posibles de ejecución del programa Java, y obtiene todos los estados finales de la computación partiendo de un estado inicial. Si los estados finales son estados seguros desde la perspectiva de la confidencialidad, es decir estados en los que no hay ninguna variable de baja confidencialidad L con valores que vienen o se derivan de valores de variables de alta confidencialidad H, el programa es seguro y la traza de ejecución del programa lógico en Maude que implementa la semántica operacional abstracta, que es la interpretación del programa Java, es una inferencia en lógica de reescritura, es decir una demostración formal; esta traza es la que se entrega como certificado de que el programa Java satisface la política de confidencialidad que asigna niveles de confidencialidad a las variables del programa.

Este trabajo trata sobre la implementación de una metodología de validación de algunos de los certificados generados por la metodología de verificación y certificación mencionada previamente; así se completaría la metodología JavaPCC (ver Figura 1) en el marco del paradigma de PCC, puesto que se requiere la herramienta que permita analizar los certificados (el **Certificate Checker**) y brinde la certeza de que cada certificado corresponde con el código

fuente Java del programa, además que cada certificado use solo reglas y ecuaciones de la semántica Java, y finalmente que el certificado incluya la aplicación de toda regla aplicable de la semántica Java .

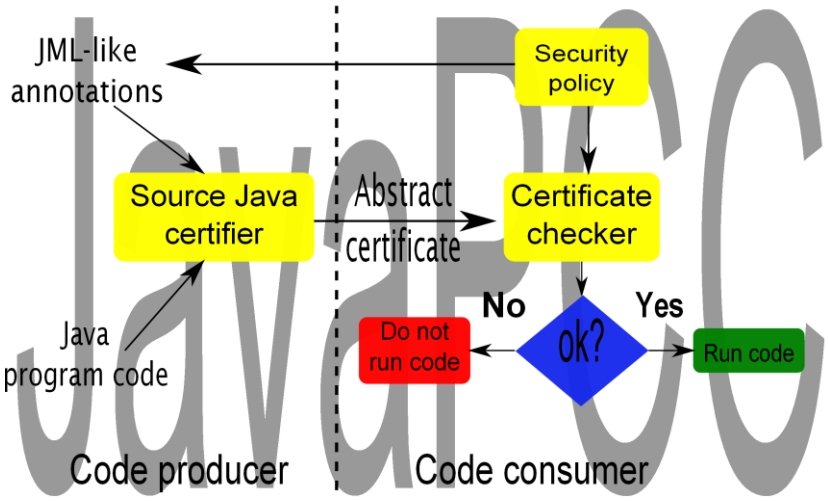


Figura 1 JavaPCC: PCC para Java

Actualmente la herramienta JavaPCC (Alba, 2011) genera tres tipos de trazas/certificados que varían en tamaño, tiempo de generación y tiempo de validación: i) el certificado completo (traza completa) con la aplicación de las reglas y ecuaciones de la semántica (*Full Certificate*), ii) el certificado con solo las reglas aplicadas (*Reduced Rules Certificate*) que excluye las ecuaciones usadas,y iii) el certificado que solo contiene las etiquetas de las reglas utilizadas (*Reduced Label Certificate*). Sin embargo el validador que se desarrollaró en este trabajo solo contempla los certificados con solo reglas, el *Reduced Rules Certificate*.

1. REFERENTE CONTEXTUAL

1.1 Descripción del Área Problemática

La inseguridad o no fiabilidad del código que usan consumidores y usuarios de programas de computador en sus desarrollos y aplicaciones informáticas, aun cuando se apliquen mecanismos para el control de acceso (que evitan el ingreso de usuarios no autorizados) y mecanismos de encriptación de la información.

Como ejemplos de código no fiable, tenemos el llamado software malicioso (por ejemplo los programas virus informáticos como los troyanos), pero también el código producido por un proveedor cualquiera, con defectos relativos a la seguridad, sin que necesariamente haya mala intención o malicia por parte del desarrollador.

Según (*US Department of justice*, 2008), el 67% de las más de 8000 empresas encuestadas tuvo incidentes relacionados con la seguridad informática, el 90% de las empresas que tuvieron incidentes sufrió pérdidas económicas por delitos informáticos, y entre los incidentes destacados están los ataques con virus y los robos de información confidencial de las personas y de las empresas. Por otra parte, según (Computer Forensics recruiter, 2007), aunque el presupuesto que las empresas dedicaron a la seguridad informática en el año 2007 fue mayor al del año 2006, el 90% de los encuestados informó que su empresa sufrió un incidente de seguridad informática en los últimos 12 meses; entre los ataques con más impacto económico están en segundo lugar los virus, gusanos y troyanos con pérdidas económicas de 8,4 millones de dólares en ese año 2007. Según el estudio (Forrester Consulting, 2012) en el que se encuestaron 240 empresas y personas influyentes vinculadas con el desarrollo de software y su seguridad, la mayoría de las organizaciones encuestadas no tienen un enfoque estratégico o holístico para la seguridad de las aplicaciones, 51% al menos han tenido un incidente en sus aplicaciones web desde inicios de 2011 y el 18% experimentó perdidas por lo menos de $500,000; entre los afectados, la reputación profesional del 50% fue desmejorada, y en el 56% de los casos se afectó negativamente la confidencialidad de los clientes; los desarrolladores no han encontrado herramientas y tecnologías adecuadas para la seguridad de las aplicaciones, ya que las herramientas actuales son muy complejas y que requieren un grado alto de experiencia en seguridad.

Según (Sabelfeld y Myers, 2003), la confidencialidad (que es uno de los pilares de la seguridad informática) de la información no es garantizada por los sistemas computacionales actuales, los mecanismos estandard de seguridad son insuficientes para proteger la confidencialidad. Mecanismos como

firewalls, cifrado y antivirus son útiles para proteger la confidencialidad de la información, sin embargo estos mecanismos no brindan una solución completa. Los antivirus están basados en detectar patrones previamente conocidos como "comportamientos maliciosos", debido a esto ofrecen protección limitada contra nuevos ataques, los escaneos de antivirus no proveen la certeza de que un programa verificado cumpla con las políticas de confidencialidad (Sabelfeld y Myers, 2003). La forma estándar de proteger la confidencialidad es el control de acceso, donde para acceder a información confidencial se requieren privilegios, pero este mecanismo solo pone restricciones al momento de entregar la información pero no para propagarla o difundirla. Una manera de atacar el problema de la falta de confidencialidad, es el uso del análisis de los flujos de información cuando los programas son ejecutados; este análisis debe mostrar que la información no debe de llegar a lugares donde la política de confidencialidad es violada.

Debido a que los mecanismos convencionales para la protección de la confidencialidad son insuficientes, nace un nuevo mecanismo para controlar los flujos de información y así mejorar la problemática con la confidencialidad, los cuales son mecanismos basados en lenguajes de programación (Sabelfeld y Myers, 2003), este enfoque posee dos variantes principales, la primera es analizar el código antes de su ejecución (análisis estático) y el segundo es analizar el código en tiempo de ejecución (análisis dinámico).

(Bavera et al., 2009) mencionan los riesgos que presentan en la actualidad la ejecución de código proveniente de fuentes externas, ya que no se sabe si este código tenga fines maliciosos y así la ejecución de este podría vulnerar la seguridad del equipo que lo esta corriendo, también mencionan que existen 3 tipos de técnicas para "mitigar" el riesgo de ejecutar código que proviene de fuentes externas: Autoría Personal, Verificación en Tiempo de Ejecución y Fortalecimiento de la Semántica del Lenguaje Fuente. El tema central de (Bavera et al., 2009) es una técnica llamada PCC (*Proof-Carrying Code*) cuyo fin es garantizarle a un "consumidor" de un código específico que este se comportará bajo los parámetros de seguridad establecidos, esto se logra por medio de una "prueba formal" que el productor adjunta al código y el consumidor realiza las verificaciones pertinentes para saber si lo ejecuta o no.

Actualmente la metodología de certificación de propiedades de seguridad de código fuente Java –JavaPCC- (la cual realiza un análisis estático del código) no cuenta con una metodología de validación de los certificados y su implementación correspondiente.

1.2 Antecedentes

El paradigma *Proof Carrying Code* PCC surge en los 90's propuesto por George Necula para el código móvil escrito en lenguajes de bajo nivel –ensamblador, bytecode, máquina- y para propiedades de seguridad de bajo nivel (manejo seguro de la memoria, operaciones aritméticas seguras –sin desbordes-, etc) (Necula, 1997). Sin embargo PCC se puede aplicar a cualquier código, en cualquier lenguaje de programación, y a diferentes propiedades de seguridad. En este paradigma el consumidor del código recibe el código que contiene la demostración formal de que el programa satisface la política de seguridad especificada por él; la demostración es el certificado de que el código satisface la política; el productor del código entrega su código junto con el certificado al consumidor o usuario del código; el consumidor comprueba este certificado formal de manera automática utilizando un validador (ver Figura 1). En (Bavera et al., 2009) explican varios aspectos de la técnica PCC (*Proof-Carrying Code*), esta técnica nos brinda una arquitectura cuyo fin es garantizarle a un "consumidor" de código que éste código se comportará bajo unos parámetros o unas políticas de seguridad establecidas por el mismo "consumidor", esta arquitectura estipula dos elementos del lado del consumidor que según (Alba, 2011) hacen parte de la base de código fiable TCB[1] (*trusted code base*), el VCGen y el verificador de los certificados. PCC se ha aplicado a diferentes lenguajes de programación y diferentes políticas de seguridad así como para el desarrollo de compiladores certificantes. Estos compiladores verifican el código fuente y generan código objeto fiable que incluye el certificado. La mayoría de propuestas se basan en sistemas de tipos que son extendidos para incluir propiedades de seguridad, de manera que la verificación del programa se reduce a comprobar sus tipos de datos, y la validación del certificado también consiste en la comprobación de si el certificado es un término que pasa la comprobación de su tipo (Alba, 2011).

JavaPCC ya cuenta con el verificador y el certificador (el ***Java Source Certifier*** de la Figura 1), pero no con el validador de los certificados. En el proceso de validación es clave reducir los términos del código fuente a su forma canónica (donde no son aplicables más ecuaciones, esto se logra con el comando "*reduce"* del interprete Maude), luego de tener los términos en esta forma se puede usar la reescritura de términos para obtener un camino posible camino de ejecución del código fuente (Esto se logra aplicando el comando "*rewrite"* del interprete Maude); pero en caso de necesitar obtener todos los posibles caminos de ejecución del código fuente, es necesario aplicar la búsqueda sobre el código fuente (se logra mediante el comando "*search"* del interprete Maude). Este validador se puede implementar tanto en el lenguaje Maude como en cualquier lenguaje de programación. En este trabajo el validador se implementará en el lenguaje C/C++ para lograr mayor eficiencia en el proceso de validación.

[1] Se asume que esta base de código es fiable, a diferencia del código que se certifica que se asume no fiable.

1.3 Justificación

La seguridad informática en general y la confidencialidad de la información en particular son importantes debido al impacto negativo que puede tener el uso de software no fiable con consecuencias como fugas de información confidencial, pérdidas económicas, demandas legales y pérdida de clientes por mala reputación (US Department of justice, 2008), (Forrester Consulting, 2012).

La herramienta JavaPCC elimina la dificultad de uso por parte de los grupos de desarrollo ya que no se requiere de un grado alto de conocimientos en seguridad ni en métodos formales para utilizarla, lo que la hace una alternativa interesante para determinar la seguridad en el código analizando el código fuente Java, de manera independiente del proceso y de la metodología de desarrollo específica que se haya utilizado. Sin embargo hay que resaltar que actualmente la herramienta de certificación carece del validador, por lo que se necesita una metodología de validación (y su implementación respectiva) de los certificados generados.

La metodología de validación y su implementación permitiría que los consumidores del código puedan comprobar automáticamente de manera fácil y rápida que los certificados son los correspondientes al código fuente del programa y a las políticas de seguridad especificadas en las anotaciones del código, evitando que se tenga que hacer una comprobación manual por parte de expertos en lógica de reescritura. En particular, se impide que se usen certificados correspondientes a otros programas -seguros bajo ciertas políticas de confidencialidad- como certificados de programas que son inseguros, tratando así de engañar al consumidor del código.

1.4 Marco Teórico:

Lógica de reescritura

En (Clavel et al., 2000) definen la lógica de reescritura como una “lógica de cambios concurrentes” que pueden modelar estados y computaciones concurrentes altamente no determinísticas; En (Mesenger, 1992) la definen como una lógica que sirve para razonar sobre los cambios en un sistema concurrente cuyos modelos son precisamente “sistemas concurrentes”; En (Alba et al., 2008) definen la lógica de reescritura como un *Framework* lógico en el cual un rango amplio de “lógicas” puede ser representado. El principio operacional de la lógica de reescritura es la “reescritura de términos”, que funciona mediante el reemplazo de una parte de una expresión dada por otra expresión utilizando una ecuación o una regla de reescritura.

Maude

Maude es un lenguaje de declarativo de alto nivel y un sistema de alto desempeño, soporta computaciones de lógica de ecuaciones y de reescritura para un amplio rango de aplicaciones (Clavel et al., 2000), según (McCombs, 2003) Maude es un lenguaje de programación que sirve para modelar sistemas y las acciones dentro de estos.

En (Clavel et al., 2011) se mencionan 3 características importantes del diseño de Maude:

- Simplicidad: los programas deben ser tan simples como sea posible y tener un significado claro.
- Expresividad: Las aplicaciones se deben de expresar naturalmente, desde un sistema secuencial y determinístico hasta los altamente concurrentes no determinísticos, desde las aplicaciones pequeñas hasta los grandes sistemas, y desde las implementaciones concretas hasta las especificaciones abstractas.
- Desempeño: Maude es un sistema de alto desempeño que puede ser usado para muchas aplicaciones.

(Clavel et al., 2011) menciona aspectos Maude como la capacidad de definir módulos funcionales y de sistema; en los módulos funcionales se pueden definir tipos de datos y las operaciones sobre estos mediante funciones formando teorías ecuacionales; en los módulos de sistema se especifican tipos de datos y las operaciones sobre estos mediante funciones y reglas de reescritura formando teorías de reescritura. Las reglas de reescritura especifican desde el punto de vista computacional transiciones concurrentes en un sistema mientras que las funciones especifican transiciones no concurrentes. Desde el punto de vista lógico la computación en Maude es una inferencia lógica, en lógica de reescritura; desde el punto de vista del modelamiento de un sistema, con las ecuaciones se modelan los aspectos determinísticos del sistema y con las reglas de reescritura se modelan los aspectos no determinísticos (como lo es la concurrencia). Desde el punto de vista operacional una computación en Maude es la reescritura de una expresión inicial (un término) aplicando las funciones y reglas del programa hasta que no se puedan aplicar más. La reescritura es el reemplazo de una parte de una expresión dada, es decir, una parte de la expresión dada se ajusta a la parte izquierda de una función o regla de reescritura, por lo tanto la parte de la expresión (que se ajusta a la parte izquierda de la regla) se reemplaza por la parte derecha de la función o regla de reescritura (las reglas tienen la estructura “parte izquierda” -> “parte derecha”). Si tenemos la función que define el factorial de un número natural: factorial: Nat -> Nat .

(1) factorial(0) = 1 .

(2) factorial(N:Nat) = N * factorial (N – 1) .

La expresión “23 + factorial(10)” puede ser reescrita reemplazando “factorial(10)” por “10 * factorial(10 -1)” usando la ecuación (2). De esta manera, la expresión queda reescrita así: “23 + 10 * factorial(10 -1)”.

Maude cuenta con varios comandos que nos permiten analizar términos sobre módulos que hayamos creado, destacamos los siguientes comandos:

- *reduce*: Según (Clavel et al., 2011) El termino especificado en este comando es reducido o simplificado mediante ecuaciones hasta su forma canónica (donde no hay manera de aplicar más ecuaciones).

- *rewrite*: Según (Clavel et al., 2011) el termino especificado en este comando es reescrito usando reglas de reescritura y ecuaciones, este comando opera haciendo uso primero de las ecuaciones hasta llevar el término a su forma canónica, y posteriormente(si hay lugar) aplica reglas de reescritura; en este comando es posible especificar un límite al cual

llegar, por defecto las reglas son aplicadas estilo Top-Down (dando oportunidad de aplicar todas las reglas posibles) y se dejan de aplicar cuando se llegue al límite que se ha especificado, este comando solo produce una secuencia de reescritura.

- *search*: Según (Clavel et al., 2011) este comando ejecuta una búsqueda primero-En profundidad de secuencias de reescritura, Tomando como punto de partida el parámetro especificado como "subject" hasta el estado final que se ajusta con el parámetro "pattern", con este comando obtenemos todos los posibles caminos mediante los cuales desde el estado inicial se puede llegar al estado final.

La metodología implementada en (Alba, 2011) usa ecuaciones para modelar el comportamiento secuencial o determinístico del código fuente, y usa reglas de reescritura para modelar el comportamiento no determinístico del código fuente (estos comportamientos se dan cuando se ejecutan hilos concurrentes o cuando se generan excepciones); también usa el comando *search* de Maude con el fin de obtener todos los posibles caminos abstractos de ejecución de los programas Java, que en fin representan todos los caminos de ejecución concreta del programa Java.

Matching **(Ajuste)**

Para que una parte de una expresión o término pueda ser reescrito con una función o con una regla, debe esa parte –subtérmino- ser una instancia de la parte izquierda de una función o regla. Esto es que al sustituir adecuadamente las variables de la parte izquierda de la función o regla se obtiene la parte de la expresión que se quiere reescribir. En el ejemplo de arriba esto ocurre al sustituir la variable “N” por el valor “10” en la ecuación (2) cuando se obtiene “factorial (10)”. Cuando se encuentra la sustitución adecuada de las variables que iguala el subtérmino de la expresión con la parte izquierda de una función o regla se dice que el subtérmino hace ajuste (*Matching*) con la parte izquierda de la función o regla.

Definición de ajuste (*Matching*) (Clavel et al, 2011, página 77) : Un término T hace ajuste con otro término L si existe una sustitución de las variables de L denominada σ, tal que T = σ (L).

Semántica Abstracta de Java

En la Universidad de Illinois se desarrolló una semántica operacional concreta (FSL, 2013) que especifica gran parte del lenguaje Java versión 1.4, elementos como el manejo de multi-hilos, herencia, polimorfismo, referencia a objetos y ubicación dinámica de objetos son especificados en esta semántica, en (Alba, 2011) se especifica una semántica operacional abstracta de Java en lógica de reescritura –Maude- basada en la semántica desarrollada por la Universidad de Illinois; esta última es presentada como una "teoría de reescritura.

Maude es considerado un “lenguaje algebraico”, y las semánticas especificadas en este lenguaje están formadas por un lado por ecuaciones (las cuales permiten especificar los aspectos determinísticos del lenguaje) y por otro lado por reglas de reescritura para especificar los aspectos no determinísticos del lenguaje; a estas semánticas se les puede aplicar las herramientas para realizar análisis formales del lenguaje Maude.

La semántica abstracta de (Alba, 2011) se logra tomando ventaja de la “modularidad” de la teoría de reescritura implementada en esta; la semántica concreta Java cuenta con un tipo genérico llamado “Value” y la extensión consta de la creación de “dominios abstractos” como subtipos de “Value”. La semántica abstracta reemplaza los valores concretos de los datos por valores simbólicos –abstractos. Por ejemplo en el caso de analizar la propiedad aritmética de la paridad, los números enteros que son los valores concretos, son reemplazados por dos valores abstractos que describen una partición de los enteros, el valor **par**, y el valor **impar**. La abstracción se completa cuando se reemplazan las operaciones concretas por operaciones abstractas como la que indica que al sumar dos números pares se obtiene un número **par**, al sumar un **par** y un **impar** se obtiene un **impar**, etc.

1.5 Formulación del Problema

En la tesis (Alba, 2011) se propuso e implementó una metodología para verificar y certificar propiedades de seguridad y políticas de confidencialidad (no interferencia y borrado) de programas escritos en código fuente Java; esta metodología se basa en la especificación de una semántica operacional abstracta de Java en Maude. Los certificados son trazas de la computación Maude resultantes de la ejecución del comando *search* con programa Maude - de la semántica abstracta de Java- que interpreta el programa Java. Estas trazas de ejecución Maude, un lenguaje declarativo, son inferencias lógicas en lógica de reescritura y por esto pueden utilizarse como demostraciones formales de que el programa cumple la propiedad de seguridad.

La metodología cuenta con una implementación - prueba de concepto - como parte de una herramienta web, que entrega como resultado (si la verificación es exitosa) un certificado que incluye el comando *search* en sí con el código fuente Java transformado en términos Maude, y la traza de ejecución que incluye todos los caminos de ejecución del programa Java a partir de un estado inicial abstracto, con los posibles estados finales.

Actualmente la metodología genera certificados de tres tipos: i) el tipo "*Reduced Rules Certificate*" que solo incluye las reglas de la semántica usadas y los estados intermedios además del estado inicial y los estados finales, ii) los certificados tipo "*Full Certificate*", que contienen todas las funciones y reglas usadas además de todos los estados; y iii) el tipo "*Reduced Labels certificate*" que solo incluye las etiquetas de las reglas usadas, el estado inicial y los estados finales.

Certificado tipo "*Reduced Rules Certificate*"

El validador solo contemplará los certificados tipo "Reduced Rules Certificate", para la generación de este tipo de certificado la herramienta de certificación (Alba, 2011) aplica ecuaciones y reglas de reescritura sobre el código fuente Java, sin embargo en el certificado quedan incluidas solamente las reglas de reescritura aplicadas y los cambios de estado presentados por la aplicación de estas reglas.

Para la ilustración del certificado se usa el siguiente ejemplo (Figura 3) de código fuente Java con anotaciones de confidencialidad ***//@*** **setLabel** (que establecen el nivel de confidencialidad **High**; si la variable no tiene anotación se asume de nivel **Low**) y anotaciones de borrado parcial ***//@*** **up** (que establecen a **High** el borrado parcial y a **Top** el borrado total) borrado tomado de (Alba, 2011):

```
class Safe1Erasure1p1 {
        static  Testclass t = new Testclass(3, -3, 6);
       public static void main(String[] args)
          {          t.mE1();          }
 }
class Testclass  {
        int xh;  //@ setLabel(xh, High);
        int yl;          int zl;

        public Testclass(int xp, int xy, int zp)  {
          //@ setLabel(xp, High);
          xh = xp;
          yl = xy;
          zl = zp;
         }

       public void mE1() {
          // erasure specification:
          /*@ up(zl, Low, High);  @*/
             xh = xh + yl + zl;
             yl = yl + 2;
             zl = 0;
       }
 }
```

Figura 2. Ejemplo de programa Java con anotaciones

Para explicar claramente el contenido del certificado, se va a dividir "lógicamente" en tres partes: i) la parte de "inicialización" (Figura 3, ii) la parte de la regla usada en el cambio de estado (Figura 4), y iii) la parte del cambio de estado en sí (Figura 5).

```
(1)  search in PGM-SEMANTICS-LABELED :
(2)  java((preprocess(
(3)  (default class t( 'Safe1Erasure1p1) imports nil extends Object implements none   {
(4)  (default     static) (t('Testclass) d('t) = new t('Testclass) < (i(3),- I (3)),i(6) >) ;
(5)  (public static) void 'main(t('String)[] d('args))throws(noType) {
(6)  3 @ ('t .     'mE1 < noExp > ;)}}

(7)  default class t('Testclass) imports nil extends Object   implements none {(((
(8)  default (int d('xh)) ; default (int d('yl)) ;) default     (int d('zl)) ;)

(9)  public t('Testclass)(((int d('xp)),(int d('xy))),(int d(   'zp)))throws(noType)
(10){((nop 11 @ ('xh = 'xp ;))
(11)12 @ ('yl = 'xy ;))
(12)13 @ (    'zl = 'zp ;)})

(13)public void 'mE1(noPara)throws(noType) {
(14)(((16 @ ('eraseH <   'zl > ;))
(15)17 @ ('xh = ('xh + 'yl) + 'zl ;))
(16)18 @ ('yl = 'yl + i(2) ;))
(17)19 @ ('zl = i(0) ;)}}))

(18)noType . 'main < new string [i(0)] > noVal))
      =>!
(19)JS:JavaState .S
```

Figura 3. Estructura certificado seguridad, parte 1: la inicialización.

En la parte inicial del certificado de seguridad (Figura 4) se evidencia una invocación al comando *search* de Maude en la línea (1), seguido del código fuente Java en términos Maude que va desde la línea (3) hasta la línea (17) inclusive. Note entre las líneas (7) y (17) el código Maude que corresponde a la clase Java **TestClass**, y las líneas (13) a (17) el código del método **mE1**. En la línea (2) se invoca la función de la semántica "java()" que crea como estado inicial del programa un estado "vacío" (memoria vacía, las pilas de control de ejecución vacías –pilas de métodos, bucles, y excepciones, y sin seguros) e invoca la función que crea el ambiente (las variables en memoria) de la clase principal.

********** REGLA
rl
t(**TC**:ThreadCtrl id(**I**) k(#(**L**:Location) -> **K**:Continuation)) store(**st:**Store [
L:Location,**Value**:Value,-1])
=>
t(k(Value:Value -> K:Continuation) TC:ThreadCtrl id(I)) store(st: Store
[L:Location,Value:Value,-1]) [label SACC01] **.**

********** SUSTITUCIÓN

TC:ThreadCtrl --> obj(o(f(onil) curr(t('Safe1Erasure1p1)) orig(t(
'Safe1Erasure1p1)))) fstack(noItem) xstack(noItem) lstack(noItem)
finalblocks(noItem) env(['args,l(8)]) holds(nil) lenv(Low, nopol)
I --> 0
L:Location --> l(7)
K:Continuation --> 'mE1 < noExp > -> ; -> e(['args,l(8)]) -> stop
st --> [l(1),High,nopol,-1] [l(2),Low,nopol,-1] [l(3),Low,nopol,-1]
[l(4),High, nopol,-2] [l(5),Low,nopol,-2] [l(6),Low,nopol,-2]
[l(8),a(string, anil),0]
Value:Value --> < o(f([t(t('Testclass)),f(['xh,l(4)] ['yl,l(5)] ['zl,l(6)])])
curr(t('Testclass)) orig(t('Testclass))),Low,nopol >

Figura 4. Estructura certificado seguridad, parte 2: la regla y sustitución de un cambio de estado.

En la segunda parte del certificado (Figura 4 se muestra la regla usada en el cambio de estado, cuya parte izquierda está entre la palabra clave **rl** y la flecha **=>,** resaltada en azul, y cuya parte derecha está entre la flecha **=>** y el punto que termina la regla, resaltada en verde. Esta regla permite acceder al valor almacenado en la memoria -el **store-** en la posición señalada por el valor de la variable **L**. Las variables en la regla tienen un tipo y son de la forma **NombreVariable: Tipo.** Tenemos entonces en la parte izquierda de la regla las variables **TC**, **I**, **L**, **K**, **st** y **Value**. También se muestra la sustitución de esas variables de la parte izquierda de la regla que hace ajuste con un subtérmino del estado de la computación. Las sustituciones se muestran así, NombreVariable -> valor, e indican que por ejemplo la variable K toma el valor "'mE1 < noExp > -> ; -> e(['args,l(8)]) -> stop", y que la posición de memoria que se accederá es la posición almacenada en la variable L, es decir la posición 7.

```
(1)  out(noOutput) in(noVal)
(2)  t(id(0) k(#(l(7)) -> 'mE1 < noExp > -> ; -> e(['args,l(  8)]) -> stop)
(3)    obj(o(f(onil) curr(t('Safe1Erasure1p1)) orig(t(    'Safe1Erasure1p1))))
(4)    fstack(noItem) xstack(noItem)  lstack(noItem)
(5)    finalblocks(noItem) env(['args,l(8)]) holds(nil) lenv(Low, nopol))

(6)  store([  l(1),High,nopol,-1] [l(2),Low,nopol,-1] [l(3),Low,nopol,-1] [l (4),High,
     nopol,-2]                  [l(5),Low,nopol,-2] [l(6),Low,nopol,-2]
(7)  [l(7),< o(f([t(t(     'Testclass)),f(['xh,l(4)] ['yl,l(5)] ['zl,l(6)])]) curr(t ('Testclass))
(8)  orig(t('Testclass))),Low,nopol >,-1]
(9)  [l(8),a(string, anil),0])

(10)code(( default class t('Safe1Erasure1p1) extends Object implements none
     {(default  static) (t('Testclass) d('t) = new t('Testclass) < i(3),- i(3),i
     (6) >) ; (  public static) void 'main(string[] d('args))throws(noType) {3 @
     ('t . 'mE1  < noExp > ;)} public t('Safe1Erasure1p1)(noPara)throws(noType)
     super(  noExp){nop}} default class t('Testclass) extends Object implements
     none {  default (int d('xh)) ; default (int d('yl)) ; default (int d('zl))
     ; public  void 'mE1(noPara)throws(noType) {(16 @ ('eraseH < 'zl > ;)) (17 @
     ('xh = ( 'xh + 'yl) + 'zl ;)) (18 @ ('yl = 'yl + i(2) ;)) 19 @ ('zl = i(0) ;)}
(11)public t('Testclass)((int d('xp)),(int d('xy)),(int d('zp)))throws(noType)
(12)super(noExp){nop (11 @ ('xh = 'xp ;)) (12 @ ('yl = 'xy ;)) 13 @ ('zl = 'zp     ;)}}))
     static([t(t('Safe1Erasure1p1)),f(['t,l(7)])] [t(t ('Testclass)),f(     noEnv)])
(13)busy(noObj)  nextLoc(8) nextTid(1)
```

Figura 5. Estructura certificado seguridad, parte 3: Un cambio de estado, el estado anterior.

Después de la sustitución, se muestran el estado anterior y el estado siguiente de la computación, separados por la flecha "--->". El estado de la computación (Figuras 5 y 6) utiliza diferentes constructores en su representación y tiene dos partes, el estado local de cada hilo, y el estado global del programa. En el estado global tenemos los constructores **out** e **in** que representan los datos de salida y entrada de los mecanismos estándar, el constructor **store** que representa la memoria y su contenido – los valores, el código del programa (**code**), la siguiente posición de memoria disponible (**nextLoc**), el siguiente identificador de hilo disponible (**nextTId**) y el conjunto de seguros ocupados por todo el programa (**busy**).

En el estado local se tiene el constructor **t** que representa todo el estado local de cada hilo de ejecución. Cada hilo tiene su identificador (**id**), su continuación (**k**[2]), el objeto activo del hilo (**obj**), las 3 pilas para controlar la ejecución de métodos (**fstack**), la ejecución de bucles (**lstack**) y la ejecución de las excepciones (**xstack**), su ambiente (**env**), los seguros usados por el hilo (**holds**) y el nivel de confidencialidad (**lenv**) que es uno de los elementos extendidos en la semántica abstracta. Note en la Figura 5–línea (5)- que en el ambiente **env** solo está la variable **args**, cuyo valor nulo está almacenado en la posición 8 (**l(8)**) del store –línea (9). Note también –línea (2)- que en el

[2] La continuación **k** es una pila de acciones pendientes ordenada de izquierda a derecha y separadas por "→". La siguiente acción pendiente es la primera a la izquierda.

extremo izquierdo de la continuación **k** la primera acción que se debe hacer es obtener el valor del **store** almacenado en la posición **7.** En azul (Figura 5) se resalta el subtérmino del estado anterior que hace ajuste matching con la parte izquierda de la regla de la Figura 4

out(noOutput) **in**(noVal)
code(…)

busy(noObj) **nextLoc**(8) **nextTid**(1))

t(**k**(< o(f([t(t('Testclass')),f(['xh,l(4)] ['yl,l(5)] ['zl,l(6)])]) curr(t('Testclass'))
orig(t('Testclass'))),Low,nopol > -> 'mE1 < noExp > -> ; -> e(['args,l(8)]) -> stop)
(**obj**(o(f(onil) curr(t('Safe1Erasure1p1)) orig(t('Safe1Erasure1p1))))
fstack(noItem) **xstack**(noItem) **lstack**(noItem) **finalblocks**(noItem) **env**(['args,l(8)])
holds(nil) **lenv**(Low, nopol))

id(0))
store(([l(1),High,nopol,-1] [l(2),Low,nopol,-1] [l(3),Low,nopol,-1] [l(4),High,nopol,-2]
[l(5),Low,nopol,-2] [l(6),Low, nopol,-2] [l(8),a(string, anil),0]) [l(7),< o(f([t(t('Testclass')),f(['xh,l(
4)] ['yl,l(5)] ['zl,l(6)])]) curr(t('Testclass')) orig(t('Testclass'))),Low, nopol >,-1])

Figura 6. Estructura certificado seguridad, parte 3: Un cambio de estado, el estado siguiente.

Del estado siguiente (Figura 6) no se muestra el código –constructor **code**- dado que es el mismo del estado anterior. No cambian el **store** ni otros elementos del estado global, y solo cambia la continuación **k** del estado local **t**. Note que en el extremo izquierdo de la continuación queda el valor que está en la posición 7 del **store**, que es un objeto de la clase **Testclass**. En azul se resalta el subtérmino del estado que es instancia de la parte derecha de la regla de la Figura 4 y que reemplaza al subtérmino que hizo ajuste con la parte izquierda de la regla.

Con base a lo anterior se plantea la siguiente pregunta:

¿Cómo garantizar que los certificados de seguridad tipo "*Reduced Rules Certificate*" que recibe un consumidor de código corresponden con las políticas de seguridad, con el código fuente del programa Java y con la semántica abstracta Java?

1.6 Objetivos

1.6.1 Objetivo General:

- Desarrollar el validador de los certificados "*Reduced Rules certificate*" generados por la herramienta de certificación.

1.6.2 Objetivos específicos:

- Diseñar una metodología de validación de los certificados generados por la herramienta, que contemple los certificados de tipo "*Reduced rules certificate"*.

- Realizar validación sobre la metodología diseñada.

- Implementar el validador de los certificados tipo "*Reduced rules certificate*" que genera la herramienta basado en la metodología diseñada y usando el lenguaje C++.

- Realizar validación sobre la implementación del validador (coherencia con metodología diseñada).

- Validar sobre resultados esperados.

- Experimentar para determinar el desempeño y la eficiencia de la validación con base en los ejemplos disponibles actualmente.

1.7 Alcance

La metodología e implementación del validador propuesta en este trabajo está concebida para ser un complemento del trabajo desarrollado en (Alba, 2011), por consiguiente éste validador solo se podrá usar para certificados generados por herramientas que hagan uso de la metodología de verificación desarrollada en (Alba, 2011).

En (Alba, 2011) la metodología de certificación estipula tres tipos de certificados (Reduced Label Certificate, Reduced Rules Certificate, Full Certificate), el diseño y la implementación del validador propuesto en este trabajo solo contemplará los certificados de tipo Reduced Rules Certificate, los otros dos tipos de certificados (Reduced Label Certificate, Full Certificate) no se manejaran en esta versión del validador, también el diseño del validador estipula el uso de la extensión de la semántica abstracta Java usada en (Alba, 2011).

1.8 Resultados Esperados

• Implementación del validador de Certificados.

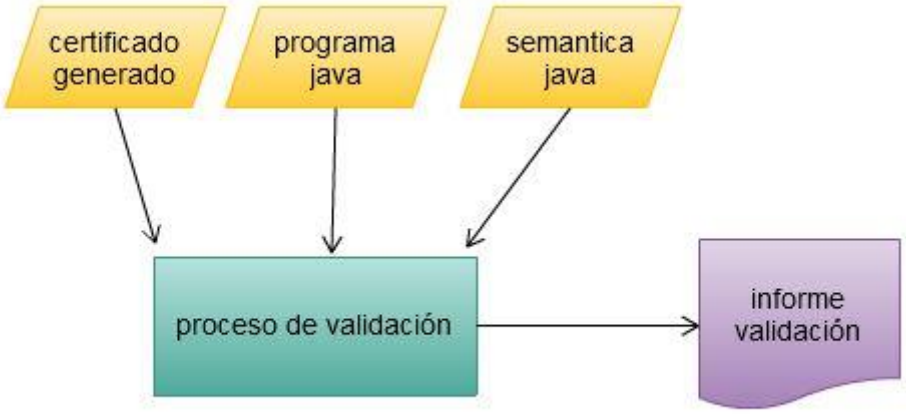
• Informes técnicos de la implementación del validador.

• Experimentos con medición del tamaño del programa, del certificado y del tiempo de generación y de validación de los certificados.

2. ESTRATEGIA METODOLOGICA

2.1 Metodología

Figura 7. Esquema general del validador



La Figura 1 muestra el esquema general del validador, en el cual existen varias entradas (certificado generado, programa Java, semántica Java), las cuales pasan por un “proceso de validación”, al final se entrega un informe de la validación realizada.

Pasos del proceso de validación:

1) Validar certificado vs programa Java:

- Este subproceso se realiza con el fin de garantizar que el certificado recibido fue generado basándose en el código Java que también recibimos inicialmente, y esto lo logramos validando la parte inicial del certificado con el comando *search* que contiene el código Java en términos Maude.

  La primera actividad de este subproceso es obtener el código Java en términos Maude, para ello utilizamos la clase JavaWrapper del programa JavaRL, que lo transforma en un término Maude.

Figura 8. Conversión de código Java a términos Maude

- Para la segunda actividad utilizamos el certificado que se recibió como entrada inicial, tomamos el código Java transformado en un término Maude que viene con el comando *search* de este certificado (Figura 3) y lo comparamos con el programa Java términos Maude que obtuvimos en la actividad anterior.

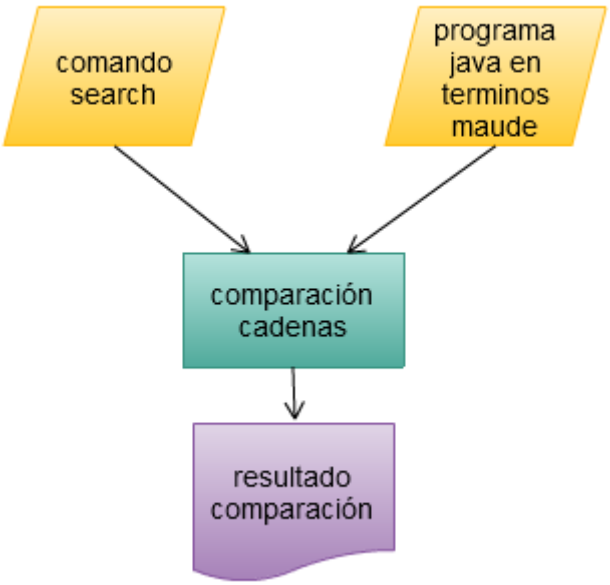


Figura 9. Comparación de la parte inicial del certificado con el código Java en términos Maude.

Tanto el estado inicial del certificado como el programa Java en términos Maude se pueden tratar como dos cadenas de texto, por lo tanto estos dos elementos se compararán en forma de cadenas de texto.

2) Verificar reglas del certificado en la semántica Java:

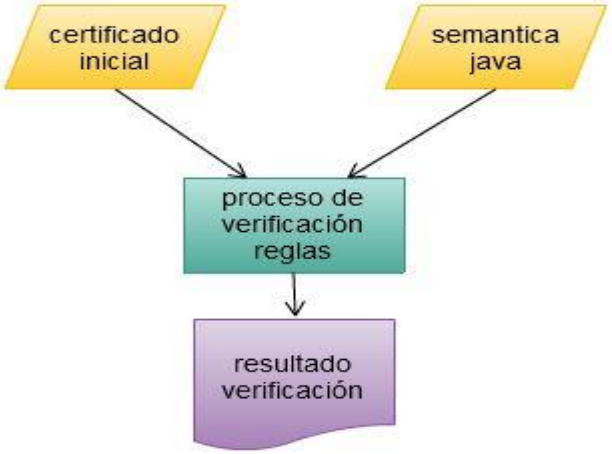


Figura 10. Verificación de las reglas del certificado en la semántica abstracta de Java.

- El certificado de seguridad *Reduced Rules Certificate* está constituido por el comando *search* (que incluye el código fuente Java en términos Maude) y las reglas aplicadas sucesivamente junto con el cambio de

estado correspondiente. La aplicación de cada regla tiene su correspondiente sustitución y cambio de estado en el certificado.

- Para validar que las reglas del certificado sean correctas se comprueba cada paso de reescritura (cambio de estado) usando las reglas. i) Se comprueba que el estado anterior a la aplicación de la regla sea alcanzable por el programa mediante pasos de reescritura que correspondan solo a la aplicación de funciones con base en el estado inicial o el anterior del anterior); ii) se comprueba la regla, inicialmente verificando que la regla de reescritura esté incluida en la semántica abstracta de Java, luego que la parte izquierda de la regla haga ajuste –*matching*- con un subtérmino del estado anterior, y después que la sustitución y el nuevo estado obtenidos sean los mismos del certificado.

- Adicionalmente se verifica si existen reglas de reescritura en la semántica abstracta Java que no estén aplicadas en el certificado y que debieron haberse aplicado en algún estado del programa. Para esto se usa el comando *search* de Maude.

2.2 Proceso de desarrollo

Debido al carácter del proyecto el proceso de desarrollo utilizado es PSP 0 que permite de manera individual un seguimiento al proceso de desarrollo, determinando entregables básicos, métricas y elementos de seguimiento en varias iteraciones.

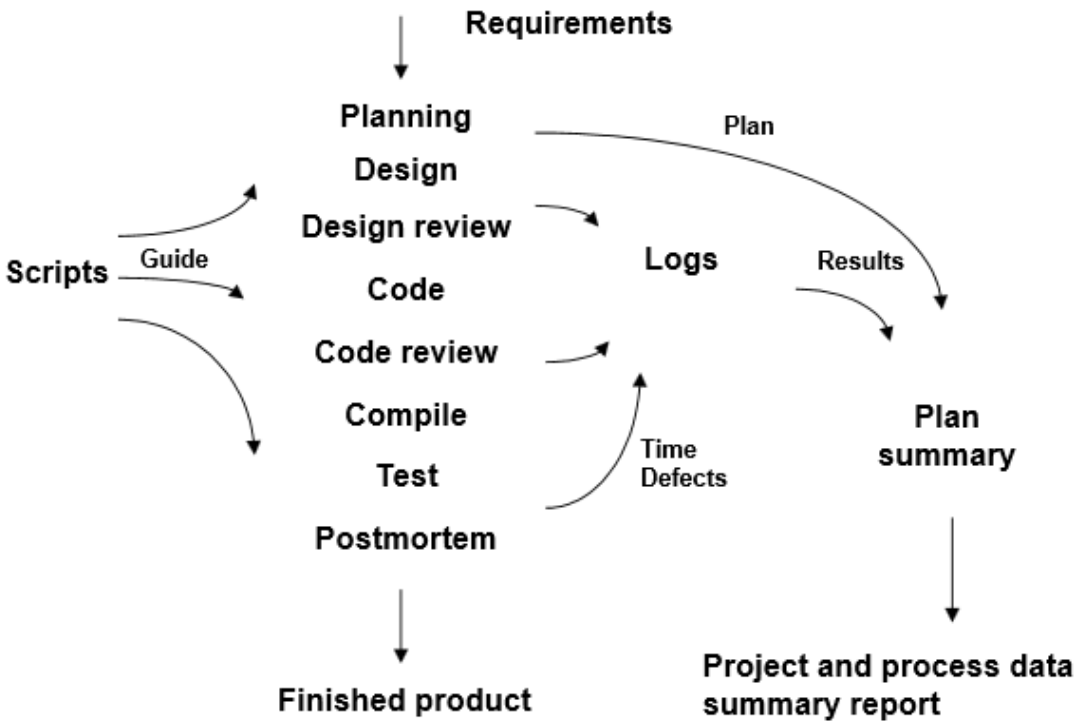


Figura 11. Flujo de Procesos de PSP0. Tomada de (SEI, 2006, página 7)

Las etapas del proceso son los siguientes:

1. **Plan:** Producir un plan para desarrollar el programa definido en los requerimientos
2. **Design:** Producir una especificación del diseño del programa
3. **Code:** Transformar el diseño en instrucciones de lenguaje de programación
4. **Compile:** Trasladar las instrucciones en código fuente al código ejecutable
5. **Test:** Verificar que el código ejecutable satisfaga los requerimientos
6. **Postmortem:** Resumir y analizar los datos del proyecto

2.3 Pruebas

Se realizaron tres tipos de pruebas para el validador, estas pruebas apuntaron a las funcionalidades de este:

1) Funcionalidad de validar la correspondencia del certificado con el código

   Se tomaron cinco certificados con su respectivo código fuente, se tomaron cinco certificados con códigos fuentes que no corresponden a estos; estos últimos eran certificados basados en un código fuente que resultó de pequeñas modificaciones de uno de los códigos del primer grupo pero que se presentaron como certificados del código origen de las variaciones.

2) Funcionalidad de validez de los pasos de reescritura en el certificado

- Se tomaron programas seguros con sus certificados de seguridad incluidos en la aplicación WEB, y se realizaron dos rondas de pruebas :

  En la primera ronda de pruebas, aleatoriamente a un grupo de los certificados se les modificó reglas de reescritura, específicamente se modificó la sustitución de la regla que corresponde con la semántica abstracta de Java por una sustitución errónea (Que no corresponda con la semántica).

  En la segunda ronda de pruebas, aleatoriamente a un grupo de los certificados se les cambió reglas que existen en la semántica abstracta de Java por reglas que no existen en la semántica; se validaron todos los certificados, tanto los modificados como los no modificados; en el resultado final el validador detecto correctamente todas la anomalías.

3) Funcionalidad de validar completitud de aplicación de reglas

- Se tomaron diez certificados, cada uno tenía mínimo dos caminos de ejecución; aleatoriamente a seis de estos certificados se les eliminaron uno de los caminos de ejecución; en el proceso de validación, el validador detectó que faltaban caminos de ejecución en los certificados modificados.

2.4 Presupuesto

2.2.1 Resumen por rubros

Tabla 1. Resumen de rubros.

RUBROS	**VALOR**
EQUIPOS DE COMPUTO	1.400.000
NOMINA	2.000.000
MATERIALES	180.000
BIBLIOGRAFIA	300.000
OTROS INSUMOS	300.000
TOTAL	**4.180.000**

2.2.2 Presupuesto detallado

EQUIPOS DE CÓMPUTO

Tabla 2. Presupuesto equipos de cómputo.

NOMBRE	**DESCRIPCION**	**CANTIDAD**	**VALOR UNITARIO**	**TOTAL**
Computador Portátil	Computador portátil para el desarrollo del proyecto de grado	1	1.400.000	1.400.000
Totales				**1.400.000**

NOMINA

Tabla 3. Presupuesto nómina.

NOMBRE	**DESCRIPCION**	**CANTIDAD/PERSONAL**	**CANT. HORAS**	**VALOR UNIT. HORA**	**TOTAL**
Profesional	Profesional encargado de desarrollar el trabajo de grado.	1	300	0	0
Asesor	Asesor del trabajo de grado.	1	250	8.000	2.000.000
Totales					2.000.000

MATERIALES

Tabla 4. Presupuesto materiales.

DESCRIPCION	JUSTIFICACION	CANTIDAD	VALOR UNITARIO	TOTAL
Papelería, fotocopias, tinta, USB, DVD, ganchos, Lapiceros Portátil	Insumos requeridos para el proyecto	1	180.000	180.000
Totales				**180.000**

BIBLIOGRAFIA

Tabla 5. Presupuesto bibliografía.

DESCRIPCION	JUSTIFICACION	CANTIDAD	VALOR UNITARIO	TOTAL
Libros y Artículos	Documentación necesaria para realización del trabajo	3	100.000	300.000
Totales				**300.000**

OTROS INSUMOS

Tabla 6. Presupuesto otros insumos.

DESCRIPCION	JUSTIFICACION	CANT. MESES	VALOR UNIT.	TOTAL

Conexión a internet	conexión para realizar consultas y diversas actividades que requieran acceso a la red	6	50.000	300.000
Totales				**300.000**

3. DESARROLLO

3.1 Validación certificado vs programa Java

Para realizar esta actividad, se toman dos archivos que se reciben como entradas de la validación, el certificado de seguridad y el código fuente java; al código fuente se le realiza un procesamiento para que este quede en términos Maude, los pasos para lograrlo (en términos generales) son los siguientes:

- Con el proyecto javaRL (FSL, 2013) se ejecuta un comando con la siguiente estructura: "java javarl.JavaRL -cp [fuente].java -maudecode -op [transformado].maude".

- Ya con el código fuente en Maude ([transformado].maude), ejecutamos este archivo con el proyecto Maude (Clavel et al., 2011); el comando ejecutado tiene la siguiente estructura: "maude [transformado].maude > [TrazaEjecucion].maude".

Los certificados de seguridad inician con la ejecución del comando "search"; el texto entre la palabra search y el texto "=>! JS:JavaState ." es el estado inicial del certificado o en otras palabras, el código fuente en términos Maude al cual le pertenece el certificado; Lo siguiente es tomar el estado inicial del certificado de seguridad y compararlo con el contenido del archivo "[TrazaEjecucion].maude".

Pruebas primera funcionalidad del validador:

Obtenemos un certificado de seguridad y su respectivo código fuente en java y realizamos una validación con estas dos entradas:

```
search in PGM-SEMANTICS-LABELED : java((preprocess((default class t(
    'MedWebSitelp1) imports nil extends Object implements none ((default
    static) (t('MedicalDiagnosis) d('t) = new t('MedicalDiagnosis) < noExp >) ;
    (public static) void 'main(t('String)[] d('args))throws(noType) {8 @ ('t .
    'medicalDiagnosis < noExp > ;))) default class t('MedicalDiagnosis) imports
    nil extends Object implements none ((((((((((((default (boolean d('fever))
    ; default (boolean d('malaise)) ;) default (boolean d('influenza)) ;)
    default (boolean d('userReqExit)) ;) public boolean 'getmalaise(
    noPara)throws(noType) {18 @ return 'malaise ;)) public boolean 'getfever(
    noPara)throws(noType) {19 @ return 'fever ;)) public boolean 'getinfluenza(
    noPara)throws(noType) {20 @ return 'influenza ;)) public t(
    'MedicalDiagnosis)(noPara)throws(noType) (((((nop (boolean d('f) = b(false)
    ;)) 24 @ ('malaise = 'f ;)) 25 @ ('fever = 'f ;)) 27 @ ('influenza = 'f ;)))
    29 @ ('userReqExit = b(false) ;))) public void 'getSymptons(noPara)throws(
    noType) ((((boolean d('tx) = b(true) ;) 34 @ ('malaise = 'tx ;)) 35 @ (
    'fever = 'tx ;)) 36 @ return;)) public void 'getUserReq(noPara)throws(
    noType) {40 @ ('userReqExit = b(true) ;))) public void 'appEnd(
    noPara)throws(noType) (((44 @ ('malaise = b(false) ;)) 45 @ ('fever = b(
    false) ;)) 46 @ ('influenza = b(false) ;))) public void 'exit(
    noPara)throws(noType) {nop}) public void 'diagnosis(noPara)throws(noType) {
    55 @ (if 'malaise && 'fever 53 @ ('influenza = b(true) ;) else nop fi)))
    public void 'medicalDiagnosis(noPara)throws(noType) ((((61 @ (while !
    'userReqExit 61 @ (((58 @ ('getSymptons < noExp > ;)) 59 @ ('diagnosis <
    noExp > ;)) 60 @ ('getUserReq < noExp > ;)))) 61 @ ;) 62 @ ('appEnd < noExp
    > ;)) 63 @ ('exit < noExp > ;)))))) noType . 'main < new string [1(0)] >
    noVal)) =>! JS:JavaState .
********** rule
rl t(TC:ThreadCtrl id(I) k(#(L:Location) -> K:Continuation)) store(st [
    L:Location,Value:Value,-1]) => t(k(Value:Value -> K:Continuation)
    TC:ThreadCtrl id(I)) store(st [L:Location,Value:Value,-1]) [label SACC01] .
TC:ThreadCtrl --> obj(o(f(onil) curr(t('MedWebSitelp1)) orig(t(
    'MedWebSitelp1)))) fstack(noItem) xstack(noItem) lstack(noItem)
```

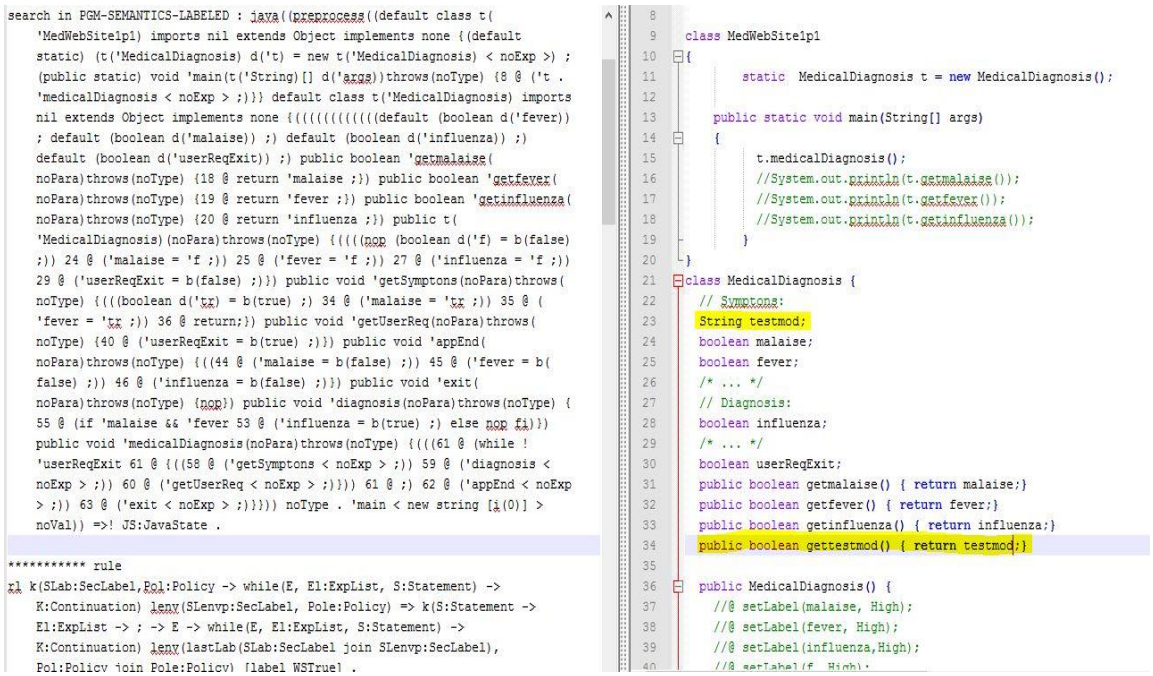
```
class MedWebSitelp1
{
    static  MedicalDiagnosis t = new MedicalDiagnosis();

    public static void main(String[] args)
    {
        t.medicalDiagnosis();
        //System.out.println(t.getmalaise());
        //System.out.println(t.getfever());
        //System.out.println(t.getinfluenza());
    }
}
class MedicalDiagnosis {
  // Symptons:
  boolean malaise;
  boolean fever;
  /* ... */
  // Diagnosis:
  boolean influenza;
  /* ... */
  boolean userReqExit;
  public boolean getmalaise() { return malaise;}
  public boolean getfever() { return fever;}
  public boolean getinfluenza() { return influenza;}

  public MedicalDiagnosis() {
    //@ setLabel(malaise, High);
    //@ setLabel(fever, High);
    //@ setLabel(influenza,High);
    //@ setLabel(f, High);
    boolean f = false;
```

Figura 12. Código fuente y Certificado Prueba 1 Concordancia.

Luego cargamos estos archivos en el validador y ejecutamos la validación:

Figura 13. Validación inicial concordancia.

Observamos que el validador encontró que el certificado de seguridad y el código fuente son coherentes, la segunda prueba que se realizó fue modificando el código fuente para observar el resultado obtenido desde el validador:



```
search in PGM-SEMANTICS-LABELED : java((preprocess((default class t(
    'MedWebSite1p1) imports nil extends Object implements none {(default
    static) (t('MedicalDiagnosis) d('t) = new t('MedicalDiagnosis) < noExp >) ;
    (public static) void 'main(t('String)[] d('args))throws(noType) {8 @ ('t .
    'medicalDiagnosis < noExp > ;)}} default class t('MedicalDiagnosis) imports
    nil extends Object implements none {((((((((((((default (boolean d('fever))
    ; default (boolean d('malaise)) ;) default (boolean d('influenza)) ;)
    default (boolean d('userReqExit)) ;) public boolean 'getmalaise(
    noPara)throws(noType) {18 @ return 'malaise ;}) public boolean 'getfever(
    noPara)throws(noType) {19 @ return 'fever ;}) public boolean 'getinfluenza(
    noPara)throws(noType) {20 @ return 'influenza ;}) public t(
    'MedicalDiagnosis)(noPara)throws(noType) {((((nop (boolean d('f) = b(false)
    ;)) 24 @ ('malaise = 'f ;)) 25 @ ('fever = 'f ;)) 27 @ ('influenza = 'f ;))
    29 @ ('userReqExit = b(false) ;)}) public void 'getSymptons(noPara)throws(
    noType) {((((boolean d('tr) = b(true) ;) 34 @ ('malaise = 'tr ;)) 35 @ (
    'fever = 'tr ;)) 36 @ return;}) public void 'getUserReq(noPara)throws(
    noType) {40 @ ('userReqExit = b(true) ;)}) public void 'appEnd(
    noPara)throws(noType) {((44 @ ('malaise = b(false) ;)) 45 @ ('fever = b(
    false) ;)) 46 @ ('influenza = b(false) ;)}) public void 'exit(
    noPara)throws(noType) {nop}) public void 'diagnosis(noPara)throws(noType) {
    55 @ (if 'malaise && 'fever 53 @ ('influenza = b(true) ;) else nop fi)})
    public void 'medicalDiagnosis(noPara)throws(noType) {((((61 @ (while !
    'userReqExit 61 @ {((58 @ ('getSymptons < noExp > ;)) 59 @ ('diagnosis <
    noExp > ;)) 60 @ ('getUserReq < noExp > ;)})) 61 @ ;) 62 @ ('appEnd < noExp
    > ;)) 63 @ ('exit < noExp > ;)})))) noType . 'main < new string [i(0)] >
    noVal)) =>! JS:JavaState .

********** rule
rl k(SLab:SecLabel,Pol:Policy -> while(E, El:ExpList, S:Statement) ->
    K:Continuation) lenv(SLenvp:SecLabel, Pole:Policy) => k(S:Statement ->
    El:ExpList -> ; -> E -> while(E, El:ExpList, S:Statement) ->
    K:Continuation) lenv(lastLab(SLab:SecLabel join SLenvp:SecLabel),
    Pol:Policy join Pole:Policy) [label WSTrue] .
```

```
8
9   class MedWebSite1p1
10  {
11          static  MedicalDiagnosis t = new MedicalDiagnosis();
12
13      public static void main(String[] args)
14      {
15          t.medicalDiagnosis();
16          //System.out.println(t.getmalaise());
17          //System.out.println(t.getfever());
18          //System.out.println(t.getinfluenza());
19      }
20  }
21  class MedicalDiagnosis {
22    // Symptons:
23    String testmod;
24    boolean malaise;
25    boolean fever;
26    /* ... */
27    // Diagnosis:
28    boolean influenza;
29    /* ... */
30    boolean userReqExit;
31    public boolean getmalaise() { return malaise;}
32    public boolean getfever() { return fever;}
33    public boolean getinfluenza() { return influenza;}
34    public boolean gettestmod() { return testmod;}
35
36    public MedicalDiagnosis() {
37      //@ setLabel(malaise, High);
38      //@ setLabel(fever, High);
39      //@ setLabel(influenza,High);
40      //@ setLabel(f, High);
```

Figura 14. Modificación código fuente entrada.

Figura 15. Resultado validación concordancia con código fuente modificado.

Después de finalizada la validación se observa que el validador detecto que el código fuente y el certificado no eran coherentes.

Se detectó una dificultad al momento de realizar la validación de concordancia, y fue que en el certificado recibido como entrada las instrucciones de asignación (excepto en declaración de variables), validaciones (sentencias if) y ciclos tenían unas etiquetas de la siguiente forma: número entero + carácter @ (ej. 10 @ ('System . 'out . 'println)); cuando se transformaba el código fuente recibido como entrada a términos Maude, las etiquetas generadas localmente eran distintas a las del estado inicial del certificado aunque la instrucción era igual (ej. 10 @ ('System . 'out . 'println) certificado de entrada, 8 @ ('System . 'out . 'println) certificado local), debido a esto se decidió omitir estas etiquetas de las instrucciones para realizar la validación de concordancia.

Resumen Pruebas Validación Concordancia:

Código Fuente	Modificado	Resultado Validación
MedWebSite1p1	Si	Fallido
Safe1Erasure1p1	Si	Fallido
Safe1Erasure1p4	No	Exitoso
Safe1Noninterference44	Si	Fallido
Safe1Noninterference45	No	Exitoso
Safe1Noninterference2p1	No	Exitoso
Safe1Noninterference39	Si	Fallido
Safe1Noninterference43	Si	Fallido
Safe1Noninterference30	No	Exitoso
Safe1Noninterference40	No	Exitoso

Tabla 7. Resultados de pruebas validación concordancia.

3.2 Verificación Reglas del Certificado

En esta etapa de la validación se verifican las reglas y las sustituciones de estas (Cambios de estados del código fuente) en el certificado de seguridad; las verificaciones que se hacen son las siguientes:

- Se verifican las reglas aplicadas en el certificado, se evalúa si la regla está incluida en la semántica abstracta de java mediante ajuste – *matching*.
- Se evalúa que la sustitución izquierda de la regla aplicada se ajuste con el estado anterior del certificado.
- Se evalúa que la sustitución derecha se ajusta con el estado generado con la aplicación de la regla de reescritura.

La validación descrita anteriormente se logró por medio de un algoritmo implementado en el lenguaje de programación Java, el algoritmo realiza los siguientes pasos:

1) Ejecuta el comando search de Maude sobre el estado inicial del certificado recibido como entrada, Maude nos entrega como salida las reglas aplicadas al estado inicial y sus sustituciones derecha e

izquierda; las reglas aplicadas y sus respectivas sustituciones se almacenan en una estructura de datos(una lista de objetos).

2) Del **certificado recibido como entrada** se toman las reglas aplicadas y sus respectivas sustituciones para almacenarlas en una estructura de datos; luego se compara cada regla y sus sustituciones de la estructura de datos generada en el paso “uno” contra cada regla y sus sustituciones de la estructura de datos obtenida en este paso; en caso de que alguna comparación falle, se almacena la regla y sus sustituciones en una estructura de datos auxiliar; Si ninguna validación falla, se le indica al usuario final que las reglas y sustituciones del certificado de entrada están correctas; pero si alguna validación falla, se le indica al usuario final que hay reglas o sustituciones incorrectas y se le muestra estructura de datos auxiliar para que pueda ver cuáles son las reglas y sustituciones incorrectas.

Pruebas segunda funcionalidad del validador:

Tomamos un certificado de seguridad y su código fuente correspondiente:

```
search in PGM-SEMANTICS-LABELED : java((preprocess((default class t(
    'MedWebSite1p1) imports nil extends Object implements none ((default
    static) (t('MedicalDiagnosis) d('t) = new t('MedicalDiagnosis) < noExp >) ;
    (public static) void 'main(t('String)[] d('args))throws(noType) (8 @ ('t .
    'medicalDiagnosis < noExp > ;)))) default class t('MedicalDiagnosis) imports
    nil extends Object implements none ((((((((((default (boolean d('fever))
    ; default (boolean d('malaise)) ;) default (boolean d('influenza)) ;)
    default (boolean d('userReqExit)) ;) public boolean 'getmalaise(
    noPara)throws(noType) (18 @ return 'malaise ;)) public boolean 'getfever(
    noPara)throws(noType) (19 @ return 'fever ;)) public boolean 'getinfluenza(
    noPara)throws(noType) (20 @ return 'influenza ;)) public t(
    'MedicalDiagnosis)(noPara)throws(noType) (((((nop (boolean d('f) = b(false)
    ;)) 24 @ ('malaise = 'f ;)) 25 @ ('fever = 'f ;)) 27 @ ('influenza = 'f ;))
    29 @ ('userReqExit = b(false) ;))) public void 'getSymptons(noPara)throws(
    noType) ((((boolean d('tr) = b(true) ;) 34 @ ('malaise = 'tr ;)) 35 @ (
    'fever = 'tr ;)) 36 @ return;)) public void 'getUserReq(noPara)throws(
    noType) (40 @ ('userReqExit = b(true) ;))) public void 'appEnd(
    noPara)throws(noType) (((44 @ ('malaise = b(false) ;)) 45 @ ('fever = b(
    false) ;)) 46 @ ('influenza = b(false) ;))) public void 'exit(
    noPara)throws(noType) (nop)) public void 'diagnosis(noPara)throws(noType) (
    55 @ (if 'malaise && 'fever 53 @ ('influenza = b(true) ;) else nop fi)))
    public void 'medicalDiagnosis(noPara)throws(noType) ((((61 @ (while !
    'userReqExit 61 @ (((58 @ ('getSymptons < noExp > ;)) 59 @ ('diagnosis <
    noExp > ;)) 60 @ ('getUserReq < noExp > ;)))) 61 @ ;) 62 @ ('appEnd < noExp
    > ;)) 63 @ ('exit < noExp > ;))))) noType . 'main < new string [i(0)] >
    noVal)) =>! JS:JavaState .
********** rule
rl t(TC:ThreadCtrl id(I) k(#(L:Location) -> K:Continuation)) store(gt [
    L:Location,Value:Value,-1]) => t(k(Value:Value -> K:Continuation)
    TC:ThreadCtrl id(I)) store(gt [L:Location,Value:Value,-1]) [label SACC01] .
TC:ThreadCtrl --> obj(o(f(onil) curr(t('MedWebSite1p1)) orig(t(
    'MedWebSite1p1)))) fstack(noItem) xstack(noItem) lstack(noItem)
```

```
class MedWebSite1p1
{
        static  MedicalDiagnosis t = new MedicalDiagnosis();

    public static void main(String[] args)
    {
        t.medicalDiagnosis();
        //System.out.println(t.getmalaise());
        //System.out.println(t.getfever());
        //System.out.println(t.getinfluenza());
    }
}
class MedicalDiagnosis {
  // Symptons:
  boolean malaise;
  boolean fever;
  /* ... */
  // Diagnosis:
  boolean influenza;
  /* ... */
  boolean userReqExit;
  public boolean getmalaise() { return malaise;}
  public boolean getfever() { return fever;}
  public boolean getinfluenza() { return influenza;}

  public MedicalDiagnosis() {
    //@ setLabel(malaise, High);
    //@ setLabel(fever, High);
    //@ setLabel(influenza,High);
    //@ setLabel(f, High);
    boolean f = false;
    malaise = f;
```

Figura 16. Código fuente y Certificado Prueba 1 Validez Reglas y sustituciones.

Después de cargar ambos archivos, se realiza una validación inicial de reglas y sustituciones de reescritura:

Figura 17. Resultado Validación inicial reglas y sustitución de reglas.

Para comprobar que el validador evalúa las reglas y las sustituciones, del certificado modificamos algunas reglas y sustituciones, se realiza de nuevo la validación:

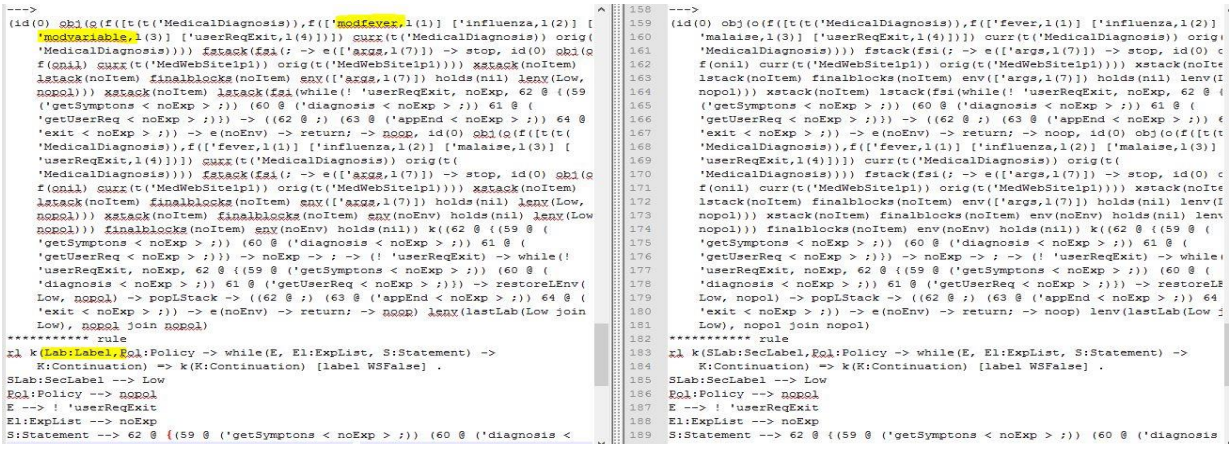


Figura 18. Paralelo entre certificado seguridad adulterado y certificado original.

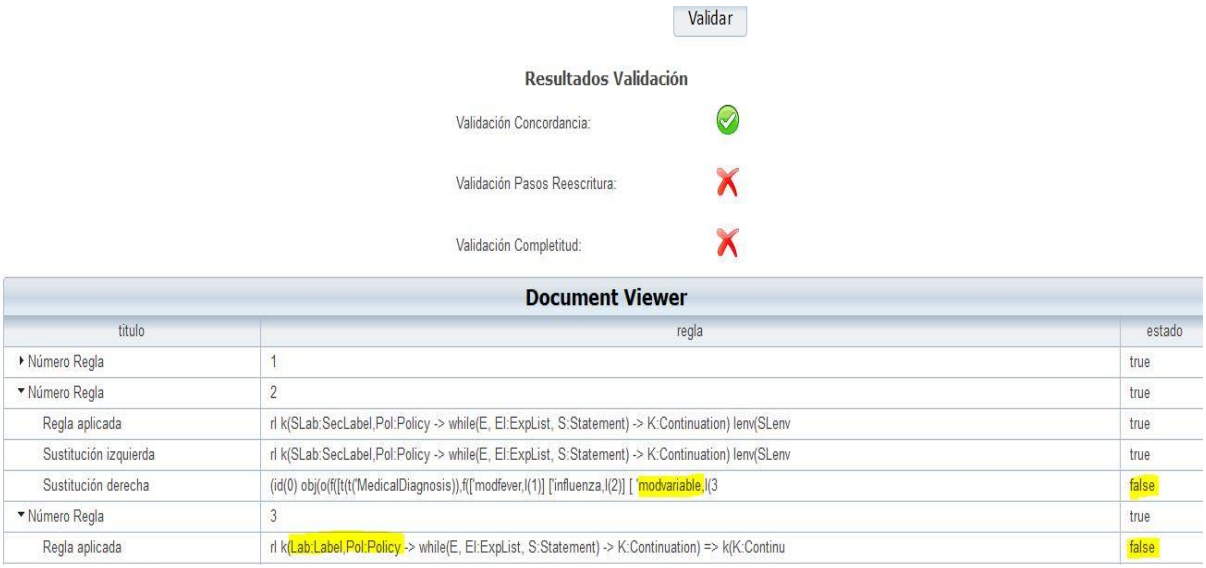


Figura 19. Resultado validación código fuente y certificado de seguridad adulterado.

En la Figura 19 se muestra el resultado de la validación, se evidencia en esta imagen que el validador detecto que la sustitución derecha de la segunda regla del certificado era incorrecta (previamente modificamos esta sustitución para probar si el validador detectaba esta inconsistencia), también vemos que detecto que la regla tres del certificado no era aplicable (esto ya que previamente modificamos la regla del certificado recibido como entrada con el fin de probar que el validador detectaba esta inconsistencia).

Al momento de validar las sustituciones de reglas, el validador fallaba, ya que en los certificados de seguridad recibidos como entrada la sustitución derecha de las reglas viene con el código fuente en términos Maude (como se muestra en la Figura 5 de la parte 1.4 del presente documento); esto porque el validador usaba el comando *rewrite* para obtener la sustitución de las reglas y este comando no genera las sustituciones con el código fuente en términos Maude, por lo tanto las sustituciones derechas generadas por el validador no se ajustaban con la sustituciones derechas de los certificados de entrada; para solucionar esta dificultad se cambió el comando *rewrite* por el comando *search* y con esto el ajuste de las sustituciones derechas fue exitoso.

Resumen de Pruebas Validación Pasos de Reescritura:

- **Pruebas sustituciones adulteradas de reglas de reescritura**

Código Fuente	Modificado Sustitución Regla	Resultado Validación
MedWebSite1p1	Si	Fallido
Safe1Erasure1p1	No	Exitoso
Safe1Erasure1p4	No	Exitoso
Safe1Noninterference44	Si	Fallido
Safe1Noninterference45	Si	Fallido
Safe1Noninterference2p1	Si	Fallido
Safe1Noninterference39	Si	Fallido
Safe1Noninterference43	Si	Fallido
Safe1Noninterference30	Si	Fallido
Safe1Noninterference40	No	Exitoso
Safe1Noninterference41	No	Exitoso
Safe1Noninterference42	No	Exitoso

Tabla 8. Resultados de pruebas validación pasos de reescritura (sustituciones de reglas).

- **Pruebas adulteración reglas de reescritura**

Código Fuente	Modificado Reglas Reescritura	Resultado Validación
MedWebSite1p1	No	Exitoso
Safe1Erasure1p1	Si	Fallido
Safe1Erasure1p4	No	Exitoso
Safe1Noninterference44	No	Exitoso
Safe1Noninterference45	No	Exitoso
Safe1Noninterference2p1	No	Exitoso
Safe1Noninterference39	No	Exitoso
Safe1Noninterference43	Si	Fallido
Safe1Noninterference30	Si	Fallido
Safe1Noninterference40	No	Exitoso
Safe1Noninterference41	No	Exitoso
Safe1Noninterference42	No	Exitoso

Tabla 9. Resultados de pruebas validación pasos de reescritura (reglas de reescritura).

3.3 Validación Completitud Certificados

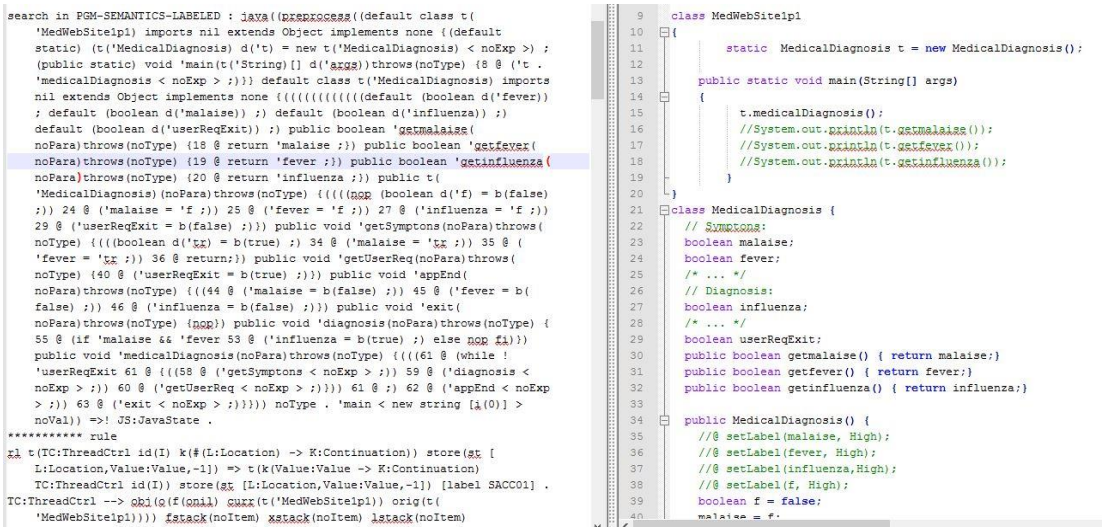
En esta parte del proceso de validación, se evalúa que el certificado de seguridad recibido como entrada tenga todos los caminos de ejecución o reglas aplicables al código fuente; esto se logra por medio de un algoritmo implementado en el lenguaje de programación Java, a continuación se describen los pasos que ejecuta el algoritmo:

- Por medio del comando *search* del lenguaje Maude, se obtienen todos los caminos de ejecución o reglas aplicables al código fuente, para ello usamos también el contenido de la traza de ejecución del código fuente en Maude de la parte 3.1 del validador; el comando quedaría asi*: search in PGM-SEMANTICS-LABELED : java((preprocess([TrazaEjecucion])) =>! JS:JavaState* ; estos caminos de ejecución que se obtienen como resultado de la ejecución del comando *search*, se almacenan en una estructura de datos (una lista de objetos).

- Después de tener todos los posibles caminos de ejecución del código fuente en una estructura de datos, se procede a evaluar por medio de *matching* uno a uno si el certificado de seguridad recibido como entrada los contiene, en caso de detectar que hay reglas que pudieron aplicarse y que no están en el certificado de seguridad de entrada (en otras palabras, que la estructura de datos contenga reglas aplicables al código fuente que el certificado de entrada no contenga), la validación de completitud fallaría; las reglas o caminos de ejecución que se detecten como “faltantes” se almacenan en una estructura de datos que es presentada al usuario final cuando termine la validación de completitud.

Pruebas tercera funcionalidad del validador:

En primer lugar, tomamos un certificado de seguridad y su código fuente, los cargamos en el validador y ejecutamos la validación:



```
search in PGM-SEMANTICS-LABELED : java((preprocess((default class t(
    'MedWebSite1p1) imports nil extends Object implements none {(default
    static) (t('MedicalDiagnosis) d('t) = new t('MedicalDiagnosis) < noExp >) ;
    (public static) void 'main(t('String)[] d('args))throws(noType) {8 @ ('t .
    'medicalDiagnosis < noExp > ;)}} default class t('MedicalDiagnosis) imports
    nil extends Object implements none {((((((((((((default (boolean d('fever))
    ; default (boolean d('malaise)) ;) default (boolean d('influenza)) ;)
    default (boolean d('userReqExit)) ;) public boolean 'getmalaise(
    noPara)throws(noType) {18 @ return 'malaise ;}) public boolean 'getfever(
    noPara)throws(noType) {19 @ return 'fever ;}) public boolean 'getinfluenza(
    noPara)throws(noType) {20 @ return 'influenza ;}) public t(
    'MedicalDiagnosis)(noPara)throws(noType) {{{{{nop (boolean d('f) = b(false)
    ;)) 24 @ ('malaise = 'f ;)) 25 @ ('fever = 'f ;)) 27 @ ('influenza = 'f ;))
    29 @ ('userReqExit = b(false) ;)}) public void 'getSymptons(noPara)throws(
    noType) {{{{(boolean d('tr) = b(true) ;) 34 @ ('malaise = 'tr ;)) 35 @ (
    'fever = 'tr ;)) 36 @ return;}) public void 'getUserReq(noPara)throws(
    noType) {40 @ ('userReqExit = b(true) ;)}) public void 'appEnd(
    noPara)throws(noType) {((44 @ ('malaise = b(false) ;)) 45 @ ('fever = b(
    false) ;)) 46 @ ('influenza = b(false) ;)}) public void 'exit(
    noPara)throws(noType) {nop}) public void 'diagnosis(noPara)throws(noType) {
    55 @ (if 'malaise && 'fever 53 @ ('influenza = b(true) ;) else nop fi)))
    public void 'medicalDiagnosis(noPara)throws(noType) {{{{61 @ (while !
    'userReqExit 61 @ {{{58 @ ('getSymptons < noExp > ;)) 59 @ ('diagnosis <
    noExp > ;)) 60 @ ('getUserReq < noExp > ;)}}}) 61 @ ;) 62 @ ('appEnd < noExp
    > ;)) 63 @ ('exit < noExp > ;)}}}}) noType . 'main < new string [i(0)] >
    noVal)) =>! JS:JavaState .
********** rule
rl t(TC:ThreadCtrl id(I) k(#(L:Location) -> K:Continuation)) store(st [
    L:Location,Value:Value,-1]) => t(k(Value:Value -> K:Continuation)
    TC:ThreadCtrl id(I)) store(st [L:Location,Value:Value,-1]) [label SACC01] .
TC:ThreadCtrl --> obj(o(f(onil) curr(t('MedWebSite1p1)) orig(t(
    'MedWebSite1p1)))) fstack(noItem) xstack(noItem) lstack(noItem)
```

```
class MedWebSite1p1
{
    static  MedicalDiagnosis t = new MedicalDiagnosis();

    public static void main(String[] args)
    {
        t.medicalDiagnosis();
        //System.out.println(t.getmalaise());
        //System.out.println(t.getfever());
        //System.out.println(t.getinfluenza());
    }
}
class MedicalDiagnosis {
  // Symptons:
  boolean malaise;
  boolean fever;
  /* ... */
  // Diagnosis:
  boolean influenza;
  /* ... */
  boolean userReqExit;
  public boolean getmalaise() { return malaise;}
  public boolean getfever() { return fever;}
  public boolean getinfluenza() { return influenza;}

  public MedicalDiagnosis() {
    //@ setLabel(malaise, High);
    //@ setLabel(fever, High);
    //@ setLabel(influenza,High);
    //@ setLabel(f, High);
    boolean f = false;
    malaise = f;
```

Figura 20. Código fuente y Certificado Prueba funcionalidad completitud certificado.

Figura 21. Validación inicial completitud certificado.

Se evidencia que el validador detecto que el certificado de seguridad recibido como entrada tenía aplicadas todas las reglas posibles al código fuente, en la siguiente prueba se van a quitar reglas del certificado de entrada para verificar que el validador detecte la falta de la aplicación de esta regla:

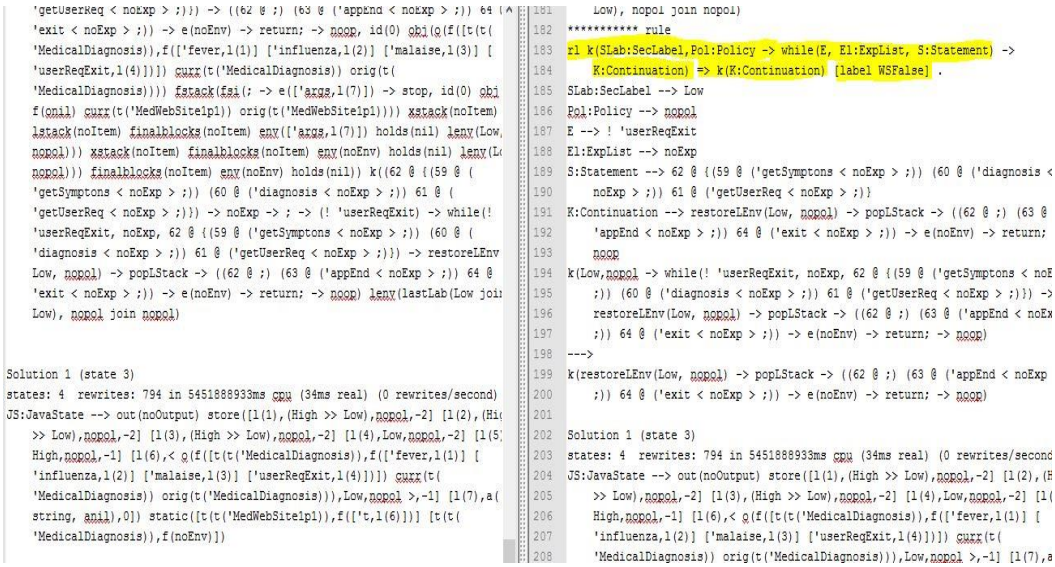


Figura 22. Paralelo entre certificado seguridad adulterado y certificado original prueba completitud.

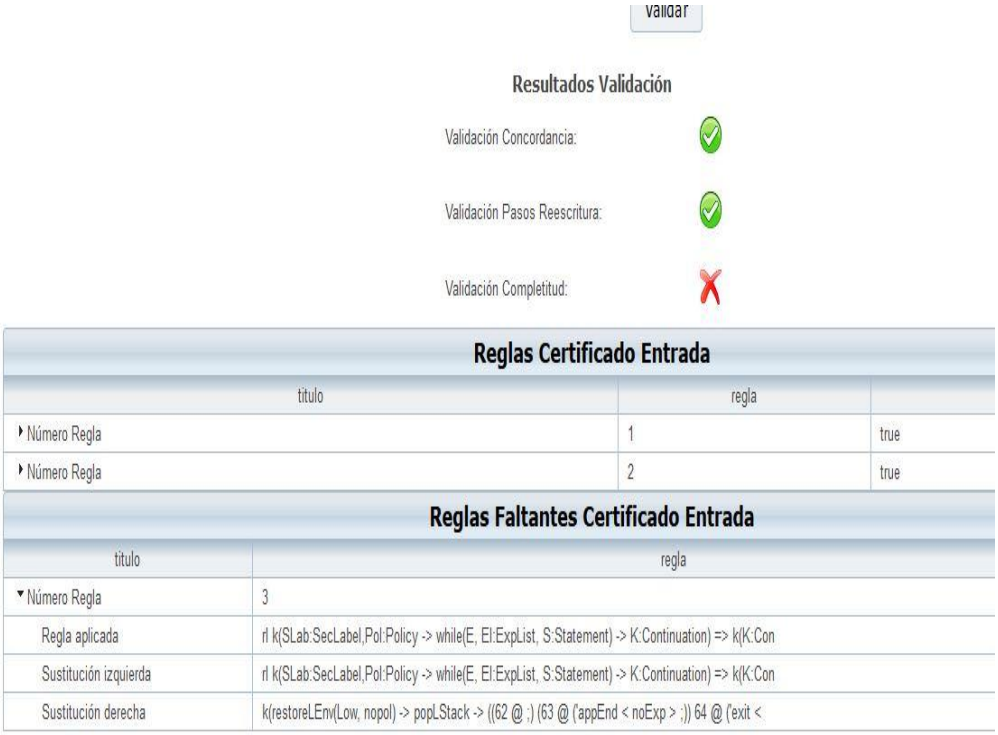


Figura 22. Paralelo entre certificado seguridad adulterado y certificado original prueba completitud.

El resultado final muestra que el validador detectó la falta de una regla en el certificado de entrada, y como se ve en la última imagen es la regla

que quitamos con el fin de probar que esta funcionalidad estuviera correcta.

Listado de Pruebas Funcionalidad Completitud de reglas:

Código Fuente	Eliminada Regla de reescritura	Resultado Validación
MedWebSite1p1	Si	Fallido
Safe1Erasure1p1	Si	Fallido
Safe1Erasure1p4	No	Exitoso
Safe1Noninterference44	No	Exitoso
Safe1Noninterference45	No	Exitoso
Safe1Noninterference2p1	Si	Fallido
Safe1Noninterference39	Si	Fallido
Safe1Noninterference43	Si	Fallido
Safe1Noninterference30	Si	Fallido
Safe1Noninterference40	No	Exitoso

Tabla 10. Resultados de pruebas validación completitud de reglas.

3.4 Registros Metodología de desarrollo

fecha	inicio	fin	interrupción(min)	tiempo delta(min)	fase	comentarios
21/10/2013	9:00 AM	2:00 PM	90	210	Planeación	Se realiza la estimación y planeación del proyecto.
23/10/2013	9:00 AM	10:00 AM	0	120	Diseño	Se inicia el diseño de la funcionalidad que validará que el certificado de seguridad y el código fuente sean coherentes.
24/10/2013	2:00 PM	6:00 PM	30	210	Diseño	Basado en los requerimientos del validador, se define como debe de realizar la validación de concordancia
27/10/2013	8:00 AM	11:00 AM	20	160	Diseño	Se inicia el diseño de la funcionalidad que validará que las reglas de reescritura del certificado de entrada sean correctas.
28/10/2013	8:00 AM	11:00 AM	40	140	Diseño	Se avanza en el diseño de la funcionalidad de validación de reglas, se define como validar que las reglas del certificado de entrada estén incluidas en la semántica abstracta de java.
29/10/2013	9:00 AM	12:00 PM	20	160	Diseño	Se avanza en el diseño de la funcionalidad de validación de reglas, se define como validar que las sustituciones izquierda y derecha de cada regla aplicada en el certificado de entrada sean correctas.
01/11/2013	9:00 AM	10:00 AM	0	120	Diseño	Se inicia el diseño de la funcionalidad que validará que el certificado de seguridad cuente con todas las reglas aplicables al código fuente.
03/11/2013	9:00 AM	12:00 PM	0	180	Diseño	Se avanza en el diseño de la funcionalidad de completitud de reglas, se define como encontrar las reglas y cambios de estado faltantes en el certificado de seguridad.

05/11/2013	8:00 AM	4:00 PM	60	420	Codificación	Se inicia la codificación del módulo de concordancia, se comienza a desarrollar la funcionalidad que va a convertir el código fuente recibido como entrada a términos maude por medio de la semántica abstracta de java.
08/11/2013	8:00 AM	4:00 PM	30	450	Codificación	Se crean los métodos en c++ encargados de tomar el código fuente recibido como entrada y convertirlo a términos maude.
11/11/2013	9:00 AM	3:00 PM	30	450	Codificación	Se crea el método en c++ encargado de tomar el código fuente en términos maude y ejecutarlo en el intérprete maude.
14/11/2013	8:00 AM	6:00 PM	60	540	Codificación	Se crea un método encargado de tomar estado inicial del certificado de seguridad recibido como entrada.
16/11/2013	8:00 AM	6:00 PM	60	540	Codificación	Se inicia la creación de los métodos necesarios para validar que el código fuente en términos maude se ajusta al estado inicial del certificado de seguridad.
17/11/2013	9:00 AM	5:00 PM	30	450	Codificación	Se finaliza la creación de los métodos que validan la concordancia entre código fuente en términos maude y el estado inicial del certificado de seguridad.
22/11/2013	9:00 AM	5:00 PM	30	450	Codificación	Se finaliza la creación de los métodos que validan la concordancia entre código fuente en términos maude y el estado inicial del certificado de seguridad.
23/11/2013	8:00 AM	6:00 PM	60	540	Codificación	Se inicia la creación de los métodos encargados de tomar las reglas del certificado de seguridad y validar si estas existen en la semántica abstracta de java.
25/11/2013	9:00 AM	5:00 PM	30	430	Codificación	Se crea un método que de cualquier certificado de seguridad identifica las reglas y las toma para su posterior validación.
27/11/2013	8:00 AM	6:00 PM	60	540	Codificación	Se crea un método que busca en la semántica abstracta de java las reglas de reescritura.
29/11/2013	10:00 AM	5:00 PM	60	360	Codificación	Se integra el método que toma las reglas del certificado de seguridad y el método que busca reglas de reescritura en la semántica con el fin de poder validar que las reglas del certificado existan en la semántica.
10/03/2014	8:00 AM	6:00 PM	60	540	Codificación	Se inicia la creación de los métodos necesarios para validar que la aplicación de la regla de reescritura (cambio de estados) sea correcto.
11/03/2014	8:00 AM	6:00 PM	90	510	Codificación	Se avanza en la creación de un método que valida la parte izquierda de las reglas de reescritura aplicadas en el certificado seguridad con la parte izquierda del cambio de estado generado por la aplicación de la regla.
15/03/2014	7:00 AM	4:00 PM	60	480	Codificación	Se finaliza el método que valida que la parte izquierda de las reglas de reescritura aplicadas en el certificado seguridad es coherente con la parte izquierda del cambio de estado generado por la aplicación de la regla.
16/03/2014	9:00 AM	5:00 PM	60	420	Codificación	Se avanza en la creación de un método que valida la parte derecha de las reglas de reescritura aplicadas en el certificado seguridad con la parte derecha del cambio de estado generado por la aplicación de la regla.

20/03/2014	8:00 AM	6:00 PM	30	570	Codificación	Se finaliza el método que valida que la parte derecha de las reglas de reescritura aplicadas en el certificado seguridad es coherente con la parte derecha del cambio de estado generado por la aplicación de la regla.
21/03/2014	10:00 AM	5:00 PM	60	360	Codificación	Se realiza la integración entre los métodos que validan que la parte izquierda y parte derecha del cambio de estado generado por la aplicación de una regla concuerda con la parte izquierda y parte derecha de la regla aplicada.
23/03/2014	8:00 AM	6:00 PM	60	540	Codificación	Se inicia la creación de los métodos necesarios para validar que el certificado de seguridad tenga todas las reglas y cambios de estado aplicables al código fuente.
24/03/2014	8:00 AM	6:00 PM	60	540	Codificación	Se avanza en la creación de un método encargado de ejecutar el comando search sobre el código fuente en términos maude para obtener todos posibles caminos de ejecución(cambios de estado consecuentes a la aplicación de reglas).
26/03/2014	8:00 AM	6:00 PM	60	540	Codificación	Se finaliza la implementación del método que ejecuta el comando search sobre el código fuente en términos maude.
02/05/2014	9:00 AM	5:00 PM	60	420	Codificación	Se crea un método que toma todas las reglas y cambios de estado del certificado de seguridad.
04/05/2014	7:00 AM	6:00 PM	60	600	Codificación	Se crea un método que busca que el certificado de seguridad contenga todos los posibles caminos de ejecución aplicables al código fuente.
07/05/2014	8:00 AM	5:00 PM	60	480	Codificación	Se inicia la codificación del front-end(interfaz gráfica) web para que los usuarios finales puedan realizar las validaciones de certificados.
09/05/2014	8:00 AM	5:00 PM	60	480	Codificación	Se avanza en la implementación de la interfaz gráfica para el validador de certificados.
11/05/2014	9:00 AM	4:00 PM	30	390	Codificación	Se finaliza la implementación de la interfaz gráfica para el validador de certificados.
15/05/2014	8:00 AM	6:00 PM	60	540	Compilación	Se realiza la compilación del módulo de concordancia del validador de certificados.
17/05/2014	8:00 AM	6:00 PM	60	540	Compilación	Se inicia categorización y corrección de errores de compilación del módulo de concordancia del validador de certificados.
18/05/2014	9:00 AM	5:00 PM	30	450	Compilación	Se finaliza categorización y corrección de errores de compilación del módulo de concordancia del validador de certificados.
22/05/2014	8:00 AM	6:00 PM	60	540	Compilación	Se realiza la compilación del módulo "Validación de Reglas" del validador de certificados.
23/05/2014	8:00 AM	6:00 PM	60	540	Compilación	Se inicia categorización y corrección de errores de compilación del módulo "Validación de Reglas" del validador de certificados.
24/05/2014	9:00 AM	5:00 PM	30	450	Compilación	Se finaliza categorización y corrección de errores de compilación del módulo "Validación de Reglas" del validador de certificados.
01/07/2014	8:00 AM	6:00 PM	60	540	Compilación	Se realiza la compilación del módulo "Completitud de Reglas" del validador de certificados.
03/07/2014	8:00 AM	6:00 PM	60	540	Compilación	Se inicia categorización y corrección de errores de compilación del módulo "Completitud de Reglas" del validador de certificados.

04/07/2014	9:00 AM	5:00 PM	30	450	Compilación	Se finaliza categorización y corrección de errores de compilación del módulo "Completitud de Reglas" del validador de certificados.
07/09/2014	8:00 AM	6:00 PM	60	540	Pruebas	Se inicia la ejecución del plan de pruebas sobre el módulo de concordancia.
08/09/2014	8:00 AM	6:00 PM	60	540	Pruebas	Se finaliza la ejecución del plan de pruebas sobre el módulo de concordancia.
10/10/2014	9:00 AM	6:00 PM	60	480	Pruebas	Se inicia la categorización y corrección de errores de las pruebas sobre el módulo de concordancia.
14/10/2014	9:00 AM	6:00 PM	60	480	Pruebas	Se continúa con la categorización y corrección de errores de las pruebas sobre el módulo de concordancia.
15/10/2014	9:00 AM	6:00 PM	60	480	Pruebas	Se finaliza con la categorización y corrección de errores de las pruebas sobre el módulo de concordancia.
19/11/2014	8:00 AM	6:00 PM	60	540	Pruebas	Se inicia la ejecución del plan de pruebas sobre el módulo de Validación de reglas.
21/11/2014	8:00 AM	6:00 PM	60	540	Pruebas	Se finaliza la ejecución del plan de pruebas sobre el módulo de Validación de reglas.
04/01/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se inicia la categorización y corrección de errores de las pruebas sobre el módulo de Validación de reglas.
10/01/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se continúa con la categorización y corrección de errores de las pruebas sobre el módulo de Validación de reglas.
12/01/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se finaliza con la categorización y corrección de errores de las pruebas sobre el módulo de Validación de reglas.
10/02/2015	8:00 AM	6:00 PM	60	540	Pruebas	Se inicia la ejecución del plan de pruebas sobre el módulo de Completitud de reglas.
13/02/2015	8:00 AM	6:00 PM	60	540	Pruebas	Se finaliza la ejecución del plan de pruebas sobre el módulo de Completitud de reglas.
16/02/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se inicia la categorización y corrección de errores de las pruebas sobre el módulo de Completitud de reglas.
19/02/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se continúa con la categorización y corrección de errores de las pruebas sobre el módulo de Completitud de reglas.
21/02/2015	9:00 AM	6:00 PM	60	480	Pruebas	Se finaliza con la categorización y corrección de errores de las pruebas sobre el módulo de Validación de reglas.

Tabla 11. Registro de tiempos PSP.

Fecha	**número**	**tipo**	**inyección**	**eliminación**	**tiemp. reparación**	**tiemp. defecto**	**descripción**
17/05/2014	1	Sintaxis	Codificación	18/05/2014	1 días	1 días	Errores de sintaxis en el módulo de concordancia.
23/05/2014	2	Sintaxis	Codificación	24/05/2014	1 días	1 días	Errores de sintaxis y en rutas de algunos archivos en el módulo de validación de reglas.

03/07/2014	3	Datos	Codificación	04/07/2014	1 días	1 días	Errores de sintaxis y de estructuras de datos en módulo de completitud de reglas.
10/10/2014	4	Diseño	Diseño-codificación	12/10/2014	2 días	2 días	Se diseñó que se debía de tomar el código fuente y transformarlo a términos maude, pero si solo se hace esto queda el código en un estado "intermedio", ese código en estado intermedio se debía de ejecutar en maude para obtenerlo de la manera como esta en el "search" del certificado de seguridad
04/01/2015	5	Diseño	Diseño-codificación	05/01/2015	2 días	2 días	En el diseño no se tuvo en cuenta que cuando se genera el código fuente en términos maude localmente las instrucciones se les antepone un carácter "@" y un número, y estos "@#" del certificado de seguridad no concuerdan con los generados localmente aunque la instrucción sea igual.
16/02/2015	6	Diseño	Diseño-codificación	20/02/2015	5 días	5 días	En el diseño no se tuvo en cuenta que las variables con etiqueta "high" fueran agregadas en la semántica abstracta de java, ya que si no se agregan el intérprete maude local deja de aplicar reglas que el maude de la herramienta de certificación si aplica a los certificados de seguridad.
19/02/2015	7	Diseño	Diseño-codificación	23/04/2014	5 días	5 días	En el diseño no se tuvo en cuenta agregar en la semántica java las etiquetas de borrado y de confidencialidad que vienen del código fuente java.

Tabla 12. Registro de defectos PSP.

Fase	Actividades	Tareas	tiemp. estimado(min)	tiemp. real(min)	inyect. estimados	inyectados	removidos estimados	removidos
plan		Estimación y planeación Desarrollo validador	180	210	2	3	2	3
desarrollo	Diseño	funcionalidad concordancia	480	330	3	5	3	5
		funcionalidad validación reglas reescritura	480	460	5	5	5	5
		funcionalidad Completitud Reglas Certificado	480	300	4	3	4	3
	codificación	Método convertir código fuente a términos maude(concordancia)	2400	870	3	3	3	3
		Procesamiento código fuente a términos maude(concordancia)	2400	450	2	1	2	1

		Método validación código fuente en términos maude vs estado inicial certificado de entrada (concordancia).	1440	1530	4	6	4	6
		Método validación existencia de reglas del certificado de seguridad en semántica abstracta java (Validación Reglas).	4800	1870	2	2	2	2
		Método validación sustitución correcta de reglas en certificado de seguridad (Validación Reglas).	7200	2880	3	3	3	3
		Método validación completitud reglas en certificado de seguridad (completitud).	9600	2640	7	4	7	4
		Integración de las funcionalidades del validador y desarrollo GUI para realización de validaciones.	7200	1350	2	3	2	3
	compilación	Compilar módulo de Concordancia código fuente y certificado de seguridad.	960	540	0	0	0	0
		Categorización y corrección de errores de compilación de módulo de concordancia.	2400	990	0	0	2	2
		Compilar módulo de validación de reglas de reescritura.	960	540	0	0	0	0
		Categorización y corrección de errores de compilación de módulo de validación de reglas de reescritura.	4800	990	0	0	3	3
		Compilar módulo completitud de reglas.	960	540	0	0	0	0
		Categorización y corrección de errores de compilación de módulo de completitud de reglas.	4800	990	0	0	4	4
	pruebas	Ejecutar plan de pruebas definido sobre el módulo de concordancia.	1440	1080	0	0	0	0
		Categorización y corrección de errores de las pruebas sobre el módulo de concordancia.	2400	1440	0	0	2	2
		Ejecutar plan de pruebas definido sobre el módulo de validación de reglas.	1440	1080	0	0	0	0
		Categorización y corrección de errores de las pruebas sobre el módulo de validación de reglas.	3360	1440	0	0	3	3
		Ejecutar plan de pruebas definido sobre el módulo de completitud de reglas.	1440	1080	0	0	0	0
		Categorización y corrección de errores de las pruebas sobre el módulo de completitud de reglas.	3360	1440	0	0	4	4

Tabla 13. Registro del plan PSP.

3.5 Análisis de Resultados

El conjunto de pruebas planteadas sobre el validador de certificados de seguridad se dividieron en tres categorías y cada una apunta a las funcionalidades del validador, las cuales son: concordancia, validez de reglas de certificación y completitud; la funcionalidad de concordancia se probó contra un grupo de diez certificados de seguridad y en promedio se realizaron cuatro pruebas por cada certificado, la funcionalidad de validez de reglas se probó contra un grupo de doce certificados de seguridad y en promedio se realizaron siete pruebas por cada certificado y la funcionalidad de completitud de reglas se probó contra un grupo de diez certificados de seguridad y en promedio se realizaron cinco pruebas por cada certificado; en las pruebas se tomaron mediciones del tiempo que tardaba el validador en finalizar cada validación, la siguiente tabla muestra una lista de certificados donde se especifica el tamaño de cada uno y el tiempo de validación:

Programa Java	**Peso Archivos(kb)**		**Concordancia(kb, seg, nano)**			**Pasos de Reescritura(kb, seg, mili)**			**Completitud(kb, seg)**		**(Seg)**
	CF	Certificado	peso	Tiempo total	tiempo	peso	Tiempo total	tiempo	peso	Tiempo	**Total**
MedWebSite1p1	3	8	1.3	26	2369	5	31	4	0.4	147632	57
Safe1Erasure1p1	1.4	1.3	0.5	27	3158	0.3	19	2	0.2	10263	46
Safe1Noninterference2p1	1	10	0.7	27	3157	8	31	5	0.3	85658	58
Safe1Noninterference30	1	14	0.7	27	1973	13	43	11	0.7	160658	70
Safe1Noninterference39	1	11	0.75	28	2378	9	31	4	0.4	256975	59
Safe1Noninterference40	1	6	0.6	27	3553	4	31	1	0.5	197764	58
Safe1Noninterference41	1	19	0.7	27	3552	16	49	4	1	116843	76
Safe1Noninterference42	2	15	0.8	28	1973	14	43	10	0.8	200133	71
Safe1Noninterference43	5	144	2.5	28	5132	90	140	25	1	823871	168
Safe1Noninterference44	2	42	1.2	27	3947	40	79	16	0.8	444871	106

Tabla 14. Pruebas tamaño certificado vs tiempo de validación.

Tiempo mínimo validación (seg)	**Tiempo máximo validación (seg)**	**Promedio**
46	**168**	**76,9**

Las pruebas realizadas muestran que el validador cumple exitosamente con cada funcionalidad planteada en la estrategia metodológica del proyecto.

4. Anexo 1. Código Fuente C++ funcionalidad “Validación de Concordancia”

```
#include <cstdlib>
#include <stdio.h>
#include <string>
#include <iostream>
#include <windows.h>
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>


using namespace std;

string exec(string cmd) {
  const char * c = cmd.c_str();
  FILE* pipe = popen(c, "r");
    if (!pipe) return "ERROR";
    char buffer[128];
    string result = "";
    while(!feof(pipe)) {
            if(fgets(buffer, 128, pipe) != NULL)
                    result += buffer;
    }
    pclose(pipe);
    return result;
}



 string exec2(string cmd, string param){
    FreeConsole();
    HWND hwnd;  //<---- manejador
    char str[100]; // <---- string
    DWORD t= sizeof(str);  //<---- obtiene el tamano de str  y se lo asigna a DWORD
    GetConsoleTitleA(str,t);// <--- obtiene el nombre de la actual ventana de consola
    hwnd=FindWindow(0,str); // <--- asigna el nombre de la ventana a hwnd
    FILE *fp = fopen(cmd.c_str(),"r");
    string result = "";

    if( fp ) {          // <---esto busca si esta instalado firefox en el ordenador

        ShellExecute(hwnd,NULL,cmd.c_str(),param.c_str(),NULL,SW_HIDE);
             char buffer[128];

          while(!feof(fp)) {
             if(fgets(buffer, 128, fp) != NULL)
                   result += buffer;
```

```
        }
        pclose(fp);

    }

  printf  ("%s \n","termino .bat");
  printf  ("%s \n",result.c_str());
  return result;
}

string substring(string texto, string param_in, string param_fin){
  size_t s = texto.find(param_in);
  size_t e = texto.find(param_fin, s);
  string sub = texto.substr(s + 1, e - s -1);
  return sub;

}


void ejecutarCmdSistema(string cmd){
    char arr[100];
    printf  ("%s \n",cmd.c_str());
    sprintf(arr,cmd.c_str());
    system(arr);
}

void ejecutarjar(string jar, string param){
   printf  ("%s \n","inicio jar");
   string cmd="java -jar "+jar+" "+param;
   ejecutarCmdSistema(cmd);
   printf  ("%s \n","termino jar");
}

string leerArchivo(string ruta){
 string texto="";
 string line;
 ifstream myfile (ruta.c_str());
 if (myfile.is_open())
 {
  while (getline (myfile,line) )
  {

    texto=texto+line;
  }
  myfile.close();

 }

 else cout << "Unable to open file";
 return texto;
```

```
}

int main(int argc, char** argv) {
   string nomCodigoFuente="codFuente";
   string nombreJar="FormatFileApp.jar";
   string nomcertificado="certificado.cert";
   string certificadoFormated="certificadoFormated.cert";
   string rutaBase”";
   string ruta_CodFuen_Maude=rutaBase+"Trace"+nomCodigoFuente;
   string ruta_certificadoFormated=rutaBase+certificadoFormated;
   exec2("ejecutarJavaWrapper.bat",nomCodigoFuente);//ejecuta el java wrapper y crea un archivo con el codigo fuente en terminos maude
   ejecutarjar(nombreJar,"modEncabezadoFile "+nomCodigoFuente);//ejecuta jar; con esta opcion, se modifica el encabezado del .maude generado previamente
   ejecutarjar(nombreJar,"generarFileMaude "+nomCodigoFuente);//ejecuta jar; con esta opcion, se genera un archivo que carga en maude "EL archivo generado" en la linea anterior
   exec2("batchMaude1.bat",nomCodigoFuente);//ejecuta .bat que ejecuta .maude y guarda la ejecucion en archivo "trace"
   ejecutarjar(nombreJar,"codigo Trace"+nomCodigoFuente);//ejecuta jar que formatea el archivo del cf en terminos maude
   ejecutarjar(nombreJar,"certificado "+nomcertificado+" "+certificadoFormated);//ejecuta jar que formatea el archivo del certificado que se recibe como entrada
   return 0;
}
```

5. CONCLUSIONES

1) Basados en la metodología JavaPCC, el consumidor de código por medio del validador puede tener la garantía de que el certificado de seguridad pertenece al código fuente que va a ejecutar y con ello evitar la ejecución de código malicioso al cual le relacionen un certificado correcto.

2) El validador de certificados de seguridad implementado es una herramienta útil para los consumidores de código que tengan un certificado de seguridad de este, ya que por medio del validador identifican si el certificado es correcto y con ello pueden tener la certeza que el código fuente cumple con políticas de confidencialidad como la “no interferencia”.

3) Las pruebas de validación muestran el tiempo mínimo (47 segundos) y máximo (254 segundos o 4.23 minutos) que tarda el validador en finalizar el proceso y podemos afirmar con estos datos que hay una relación entre el tamaño del certificado de seguridad y el tiempo de validación.

4) Basados en las pruebas realizadas, podemos concluir que el validador es eficiente en el sentido que detectó en todos los casos de prueba si el certificado de seguridad poseía anomalías o adulteraciones, también podemos concluir que el validador tiene un buen rendimiento ya que realizó validaciones de certificados con más de treinta reglas de reescritura y la duración máxima fue 4.23 minutos.

REFERENCIAS

(Sabelfeld y Myers, 2003) Sabelfeld A. & Myers A. (2003). Language-based Information-flow Security.

IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, VOL. 21, NO. 1

Obtenido en: http://www.utd.edu/~hamlen/Papers/sm-jsac03.pdf

(Bavera et al., 2009) Bavera et al. (2009).

un Survey sobre *Proof-Carrying Code*.

Obtenido en: http://dc.exa.unrc.edu.ar/staff/fbavera/papers/Survey_PCC.pdf

(Zdancewic, 2004) Zdancewic S. (2004).

Challenges for Information-flow Security

Obtenido en: http://www.cis.upenn.edu/~stevez/papers/Zda04.pdf

(Smith, 2006) Smith G. (2006)

Principles of Secure Information Flow Analysis.

Miami, Florida: School of Computing and Information Sciences.

Obtenido en :

http://users.cis.fiu.edu/~smithg/papers/sif06.pdf

(Clavel et al. , 2011)	Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N.(2011) Maude Manual Obtenido en : http://Maude.cs.uiuc.edu/Maude2-manual/
(Clavel et al., 2000)	Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N.(2000) Maude Tutorial Obtenido en : http://Maude.cs.uiuc.edu/Maude1/tutorial/
(McCombs, 2003)	McCombs T. (2003) Maude Primer Obtenido en : http://Maude.cs.uiuc.edu/primer/
(Alba, 2011)	Alba M. (2011) Abstract Certification of Java Programs in Rewriting Logic. Tesis Doctoral. Valencia: Universitat Politecnica de Valencia
(Mesenger, 1992)	Mesenger J. (1992) Conditional Rewriting Logic as a unified model of concurrency Theoretical Computer Science: Elsevier Obtenido en:

	http://www.sciencedirect.com/science/article/pii/030439759290182F
(US Department of justice, 2008)	US Department of justice. (2008) Cybercrime Against business Obtenido en : http://www.bjs.gov/content/pub/pdf/cb05.pdf
(Computer Forensics recruiter, 2007)	Computer Forensics recruiter. (2007) Cyber Crime Statistics Obtenido en: www.computer-forensics-recruiter.com/home/cyber_crime_statistics.html
(Forrester Consulting, 2012)	Forrester Consulting. (2012) The software security risk report Obtenido en: http://softwareintegrity.coverity.com/rs/coverity/images/2012-software-security-risk-report.pdf
(Alba & Otros, 2012)	Alba M. & Otros (2012). Herramienta de certificación de programas Java en lógica de reescritura, Universitat Politécnica de Valencia. Obtenido en: http://zenon.dsic.upv.es:8080/certificateX/
(Springer, 2007)	Springer (2007) All About Maude –A High-Performance Logical Framework

(Alba et al., 2008) Alba M., Alpuente M., Escobar S. (2008)

Automatic Certification of Java Source Code In Rewriting Logic.

Valencia: Universitat Politecnica de Valencia

(FSL, 2013) Formal Systems Laboratory (2013)

Maude-based rewriting specification of Java

Obtenido en:

http://fsl.cs.illinois.edu/index.php/Rewriting_Logic_Semantics_of_Java

(SEI, 2006) Software Engineering Institute- Carnegie Mellon University(2006)

Personal Software Process for Engineers

Obtenido en:

http://www.sei.cmu.edu/reports/00tr022.pdf

(Navarro & Raffinot, 2002) Navarro, G., & Raffinot, M. (2002).

Flexible Pattern Matching in Strings.

Cambridge: Cambridge University Press.