Universidad Politécnica de Valencia

Departamento de Sistemas
Informáticos y Computación

Tesis de Master

Certificación Automática de Propiedades de Seguridad de Código fuente Java en Lógica de Reescritura

Metodología e Implementación para Propiedades Basadas en Tipos, No–Interferencia y Consumo Acotado de Recursos

Candidato:
Mauricio F. Alba Castro

Director:
María Alpuente Frasnedo
Santiago Escobar

– Julio de 2008 –

Trabajo realizado con una beca del proyecto
ALFA LERNet AML/19.0902/97/0666/II-0472-FA

Correo Electrónico del autor: malba@dsic.upv.es,malba@autonoma.edu.co

Dirección del autor:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Índice general

Índice de figuras

Resumen

En la última década se ha observado un interés cada vez mayor en métodos formales diseñados para la verificación de la fiabilidad del código proveniente de fuentes no fiables. El código móvil que va acompañado de la prueba de su fiabilidad (Proof-carrying code, PCC), desarrollado por Necula [Necula, 1997; Necula and Lee, 1996], es un mecanismo para el aseguramiento del comportamiento seguro de los programas que es útil en general para el desarrollo de software pero que resulta muy adecuado para el desarrollo y distribución del código móvil.

En este trabajo se presenta una metodoloía para la certificación de propiedades de seguridad de programas Java, basada en la lógica de reescritura, un marco formal lógico y semántico muy general que está implementado eficientemente en el lenguaje de programación de alto nivel Maude. Se consideran propiedades de seguridad basadas en tipos, pero no verificables por un compilador estándar, propiedades de no interferencia que evitan el flujo ilegal de información, y propiedades de consumo acotado de recursos.

A partir de la especificación básica de semántica de Java escrita en Maude, desarrollamos extensiones de esta semántica operacional de Java, para la observación, verificación y certificación de las propiedades señaladas en programas Java. Como un subproducto de la verificación se obtiene un certificado de la propiedad verificada, que consiste de un conjunto de demostraciones de reescritura abstractas, que puede ser validado fácilmente por el consumidor del código utilizando un motor de lógica de reescritura estándar. Brevemente, las principales contribuciones de esta tesis son:

- La primera aproximación a una técnica PCC basada en interpretación abstracta para la certificación de código fuente Java que explota la automatización, expresividad y generalidad de la lógica de reescritura, con base en la semántica de Java ([Farzan et al., 2007]) escrita en Maude.

- Una extensión abstracta de la semántica de Java, para la verificación finita y la certificación de propiedades aritméticas de los resultados de métodos

de programas Java.

- Una extensión de la semántica de Java, para análisis del flujo de información también escrita en Maude, que permite observar en programas Java la no interferencia.
- Una versión abstracta de estado finito de la semántica para análisis del flujo de información, que soporta la verificación finita de programas.
- Una extensión de la semántica de Java, para la contabilización del consumo de recursos escrita en Maude, que permite observar en programas Java, el consumo de recursos de los métodos, de los hilos y del programa.
- La técnica es completamente automática, e incluye un módulo para generar la codificación Maude de la abstracción de las variables del programa a partir de anotaciones JML.

1

Introducción

Como un campo de investigación emergente, la movilidad del código está ocasionando un creciente corpus de literatura científica así como gran desarrollo industrial desde los años 90s. En la última década se ha observado un interés cada vez mayor en métodos formales diseñados para la verificación de la fiabilidad del código proveniente de fuentes no fiables. El código móvil que va acompañado de la prueba de su fiabilidad (Proof-carrying code, PCC), desarrollado por Necula [Necula, 1997; Necula and Lee, 1996], es un mecanismo para el aseguramiento del comportamiento seguro de los programas que es útil en general para el desarrollo de software pero que resulta muy adecuado para el desarrollo y distribución del código móvil. En PCC, un programa contiene el código y una demostración codificada fácil de comprobar, cuya validez supone el cumplimiento de una política de seguridad predefinida que es suministrada por el consumidor del código. La demostración es un certificado formal de la fiabilidad del código que es generado por el productor del código, y luego empaquetado junto con el código verificado para su distribución. El consumidor recibe y valida el certificado usando una infraestructura fácil de usar, para determinar la fiabilidad del código antes de su ejecución. Los aspectos claves para una realización práctica de PCC son: (i) la expresividad del lenguaje usado para expresar las políticas, (ii) el tamaño del certificado transmitido, (iii) la eficiencia en la validación, y (iv) la escalabilidad en relación con programas de diferentes tamaños y complejidades.

Las principales tecnologías aplicadas comúnmente en PCC son[1] *análisis de tipos* [Appel and Felty, 2000; Felty, 2005], *demostración de teoremas* [Besson et al., 2006; Wildmoser et al., 2004; Barthe et al., 2007], e *interpretación abstracta*

[1] El Capítulo 2 resume las principales aproximaciones de PCC

[Xia and Hook, 2004; Albert et al., 2005c; Besson et al., 2006; Chang et al., 2006; Besson et al., 2007]. La lógica de reescritura [Meseguer, 1992] es un marco formal flexible y general en el cual se puede representar de manera fiel un amplio espectro de lógicas y modelos de computación También provee una forma fácil y poco costosa de desarrollar definiciones formales de lenguajes de programación que son *directamente ejecutables* [Meseguer and Rosu, 2007] como intérpretes en un lenguaje de lógica de reescritura como Maude [Clavel et al., 2007]. La verificación de sistemas reactivos y embebidos en lógica de reescritura ofrece varias ventajas; una de las más importantes es la madurez, generalidad y sofisticación de las herramientas de análisis disponibles (véase [Clavel et al., 2007]).

En la tesis se desarrolla una técnica PCC basada en interpretación abstracta para la certificación de código fuente Java que explota la automatización, expresividad y generalidad de la lógica de reescritura. Para obtener un procedimiento de decisión se imponen modelos de estado finito de los programas mediante el uso de interpretación abstracta [Cousot and Cousot, 1977].

El trabajo se enfoca a propiedades de seguridad, es decir, propiedades de un sistema que son definidas en términos de la no ocurrencia de ciertos eventos [Manna and Pnueli, 1995], que son caracterizadas como problemas de inalcanzabilidad en lógica de reescritura: dado un sistema concurrente descrito mediante un sistema de reescritura de términos y una propiedad de seguridad que especifica los estados del sistema que no deberían ocurrir nunca, la inalcanzabilidad de todos estos estados inseguros desde los estados iniciales considerados permite inferir la propiedad de seguridad deseada. En la tesis se consideran tres propiedades de seguridad: i) propiedades aritméticas basadas en tipos no verificables por un compilador estándar, ii) propiedades de confidencialidad de datos (no interferencia), y iii) propiedades de consumo acotado de recursos como tiempo de proceso y memoria[2].

A partir de la definición de la semántica de Java en lógica de reescritura formalizada en [Farzan et al., 2007], se desarrolla una técnica de análisis para la certificación de código fuente. La idea central del análisis es probar la inalcanzabilidad de estados Java inseguros que representen la contraparte del cumplimiento de la propiedad de seguridad usando el comando estándar de búsqueda (primero-en-

[2]En este caso sólo se ha desarrollado la primera parte de la técnica.

amplitud) de Maude que explora todo el espacio de estados (finito) del programa. En caso de que la prueba sea exitosa, las correspondientes demostraciones de reescritura que demuestran que esos estados inseguros son efectivamente inalcanzables, son entregadas como el certificado esperado.

En el proceso de certificación, el desarrollador expresa la política de seguridad en JML (Java Modeling Language) [Leavens et al., 2006], como anotaciones de las variables del código fuente Java que quiere certificar. Vale la pena anotar, que en el marco formal presentado, la semántica abstracta de Java está disponible directamente al consumidor del código, por lo cual puede ser verificada solo una vez, y así considerada fiable desde ese momento.

1.1. Contribuciones de la Tesis

- A partir de la definición de la semántica de Java en lógica de reescritura formalizada en [Farzan et al., 2007], se desarrolló una semántica abstracta escrita en Maude para el análisis y la certificación de código fuente. La tesis aporta la primera aproximación a una técnica PCC basada en interpretación abstracta para la certificación de código fuente Java que explota la automatización, expresividad y generalidad de la lógica de reescritura.

- A partir de la especificación de la semántica de Java escrita en Maude [Farzan et al., 2007], se aporta una semántica abstracta de Java también escrita en Maude, para verificar y certificar propiedades aritméticas basadas en tipos de resultados de métodos.

- También partiendo de la especificación de la semántica de Java [Farzan et al., 2007], se desarrolló una extensión para análisis del flujo de información que permite observar en programas Java la no interferencia. Por lo que sabemos, no había una semántica concisa de programas Java que considere la no interferencia. La mayoría de los trabajos previos se han enfocado en imponer la no interferencia mediante compiladores y sistemas de tipos apropiados basados en flujo de información [Myers, 1999; Myers et al., 2001; Barthe et al., 2006] o comprobación de tipos del bytecode Java [Barbuti et al., 2002b; Avvenuti et al., 2003; Barthe et al., 2007].

- Se elaboró una versión abstracta de estado finito de la semántica para análisis del flujo de información, que soporta la verificación finita de programas. Gracias a la abstracción y al manejo diferente de ecuaciones y reglas en Maude, no se tiene el problema de la explosión de estados de los enfoques más tradicionales (véase [Meseguer and Rosu, 2007]).

- A partir de la especificación de la semántica de Java [Farzan et al., 2007], se desarrolló una extensión para la contabilización del consumo de recursos, que permite observar en programas Java, el consumo de recursos de los métodos, de los hilos y del programa.

- La técnica aportada es completamente automática, e incluye un módulo para generar la codificación Maude de la abstracción de las variables del programa a partir de las anotaciones JML.

1.2. Estructura de la Tesis

En el Capítulo 2 se resumen las principales aproximaciones de PCC y se presentan los principales problemas que existen en PCC. Igualmente se esquematizan las propuestas para PCC revisadas, destacando los lenguajes de programación certificados, las propiedades de seguridad consideradas, las asunciones, los formalismos y las tecnologías utilizadas. En el Capítulo 3 se presentan los preliminares, que comprenden una descripción del lenguaje JML (Java Modeling Language), la presentación de Maude y la lógica de reescritura, una descripción de la semántica Java en lógica de reescritura, y una introducción a la interpretación abstracta. En el Capítulo 4 se presentan la técnica en relación con propiedades de seguridad basadas en tipos que no son verificables por un compilador estándar, los experimentos realizados y los principales trabajos relacionados. En el Capítulo 5 se presentan la técnica en relación con propiedades de seguridad de confidencialidad de datos basadas en no interferencia, los experimentos realizados, y los principales trabajos relacionados. En el Capítulo 6 se presenta la primera parte de la la técnica en relación con propiedades de consumo acotado de recursos, con la semántica extendida de Java en lógica de reescritura para contabilizar el consumo de recursos de programas Java. La semántica abstracta está actualmente en de-

sarrollo. En el Capítulo 7 se presenta el proceso de certificación de programas Java utilizando la herramienta desarrollada. Finalmente en el Capítulo 8 se presentan las conclusiones de la tesis.

2

Estado del arte en Proof Carrying-Code PCC

El código móvil que va acompañado de la demostración de su fiabilidad (Proof-carrying code, PCC), es un mecanismo para el aseguramiento del comportamiento seguro de los programas que es útil en general para el desarrollo de software pero que resulta muy adecuado para el desarrollo y distribución de código móvil. La demostración codificada de la fiabilidad es fácil de comprobar, y su validez supone el cumplimiento de una política de seguridad predefinida que es suministrada por el consumidor del código.La demostración es un certificado formal de la fiabilidad del código que es generado por el productor del código, y luego empaquetado junto con el código verificado para su distribución. El consumidor recibe y valida el certificado usando una infraestructura fácil de usar, para determinar la fiabilidad del código antes de su ejecución.

A continuación se resumen las principales aproximaciones para resolver el problema de la seguridad del código móvil.Luego, se esquematizan las propuestas, destacando las políticas de seguridad tratadas, el lenguaje de programación cuyo código se verifica, las asunciones sobre la fiabilidad de los componentes de la infraestructura y los actores participantes (los productores y los consumidores del código), la tecnología y el formalismo empleados.

2.1. Principales Aproximaciones de PCC

PCC

En 1996 se presentó el primer resultado de la aproximación PCC a la solución del problema de seguridad del código móvil en lenguaje ensamblador para la verificación de extensiones del sistema operativo, seguras en cuanto al manejo de memoria [Necula and Lee, 1996; Necula and Lee, 1997b]. La propiedad de seguridad se expresa en una lógica de Hoare como pre y postcondiciones del código. El certificado entregado con el código es una expresión en LF[1] que representa la inferencia o derivación de un tipo cuya validación consiste solo en la comprobación del tipo. En 1997 se aplicó esta aproximación a código en lenguaje ensamblador desarrollado como extensión a programas en lenguaje ML para certificar el manejo seguro de la memoria y de los tipos tal como lo hace el compilador de ML;el enfoque fué denominado Proof Carrying Code PCC [Necula, 1997]. En este enfoque PCC el consumidor debe hacer explícita y pública su política de seguridad para que sea conocida por los productores de código. En [Necula and Lee, 1998b] se certifica la seguridad de código ensamblador en cuanto al manejo de memoria, acceso a archivos con autorización, topes en consumos de recursos como memoria, candados, ciclos de CPU y ancho de banda. La infraestructura de PCC esta compuesta por el generador de condiciones de verificación VCGen, el demostrador y el comprobador de las demostraciones. El VCGen con base en el código anotado produce el predicado que codifica la propiedad de seguridad y los lemas correspondientes que se deben demostrar. El consumidor del código utiliza el VCGen y el comprobador de las demostraciones que debe considerar como fiables.

Compiladores Certificantes

En [Tarditi et al., 1996] se presenta el primer compilador certificante, que le adjunta al código generado una prueba o certificado de seguridad que se puede validar [Tarditi et al., 2004]. El compilador TIL no solo genera código objeto sino

[1]Logical Framework: marco formal que es una extensión del cálculo lambda con tipos simples y dependientes

una demostración de que el código generado satisface un invariante crítico que se verifica e impone usando comprobación de tipos. La demostración generada se puede validar automáticamente. El compilador TIL para lenguaje ML impone manejo seguro de tipos y extiende el manejo de vectores en SML con vectores seguros de dos dimensiones en cuanto a la validación de sus límites. En [Shao and Appel, 1995; Shao, 1997], se presenta un compilador certificante dirigido por tipos para fuentes en Standard ML 1997 (SML97), extendido con módulos de orden superior llamado FLINT/ML. A diferencia de TIL, FLINT/ML soporta de forma completa los módulos ML y hace análisis de tipos y manipulación de código en tiempo de ejecución de forma más eficiente. Usa un lenguaje intermedio fuertemente tipado llamado FLINT basado en el cálculo lambda.

En 1998 la expresión LF que constituye el certificado de manejo seguro de la memoria y de los tipos del lenguaje es generada por primera vez automáticamente por un compilador certificante para el lenguaje C con tipos [Necula and Lee, 1998a]. En [Colby et al., 2000; Necula, 2001b] se presentan compiladores certificantes para un subconjunto de Java, que generan también expresiones en LF como certificados acerca del manejo seguro de los tipos y la memoria.

Otros compiladores certificantes para manejo seguro de la memoria, de los saltos y de los tipos, son los asociados al lenguaje TAL, un lenguaje ensamblador tipado definido en [Morrisett et al., 1999b] para aplicar PCC. Estos compiladores generan anotaciones de los tipos como certificado. La validación del certificado es la comprobación de los tipos con base en las anotaciones. Hay compiladores para lenguajes de bajo nivel (TAL, TALx86) [Morrisett et al., 1999a; Grossman and Morrisett, 2001] y compiladores para lenguajes de alto nivel, como dos dialectos de C, Popcorn y Cyclone [Jim et al., 2002], y de Scheme, que usan como lenguaje objeto el lenguaje ensamblador tipado TAL. El compilador TALx86 genera anotaciones con un tamaño entre 43 % y 67 % en relación con el tamaño del código, con un tiempo de validación lineal en función del tamaño. Recientemente se lanzó el proyecto TILT para desarrollar compiladores certificantes para la familia de lenguajes ML (SML '97, Caml, KML) que certifiquen el manejo seguro de tipos [TILT, 2006].

Problemas Iniciales en PCC

Con las primeras aproximaciones surgen tres problemas. El primero y quizás más importante es sobre el supuesto de la fiabilidad del código que genera y valida las demostraciones o certificados, denominado la base de código fiable TBC. ¿Es fiable la TCB?. Como se considera que la fiabilidad de la TCB es función inversa de su tamaño [Appel and Felty, 2000; Appel, 2001; Necula and Schneck, 2002], el problema de la fiabilidad de la TCB se traduce en el problema de ¿cómo reducir el tamaño de la TCB fiable? . El tamaño de la TCB del enfoque PCC del compilador certificante en C con tipos de [Necula and Lee, 1998a] es de 26 mil líneas de código aproximadamente [Wu et al., 2003]. El segundo problema es la escalabilidad y flexibilidad del enfoque para aplicarlo a diferentes lenguajes de alto y bajo nivel, y a políticas de seguridad más generales y complejas [Appel and Felty, 2000; Appel, 2001; Necula, 2001a; Necula and Schneck, 2002]. El tercer problema es el tamaño del certificado [Necula and Lee, 1997a; Necula and Rahul, 2001; Wu et al., 2003]. ¿Cómo reducirlo?, ¿Cómo reducir el tiempo de validación del certificado reducido?. En [Necula and Lee, 1997a] se utiliza la redundancia de la expresión LF del certificado para reducir su tamaño, con reducciones de hasta 15 veces en el tamaño y de 7 veces en el tiempo de validación. La nueva representación fue denominada LFi.

Foundational Proof Carrying Code: FPCC

En [Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001; Felty, 2005] se propone una alternativa para el enfoque PCC que pretende resolver los dos primeros problemas mencionados al, i) ser universal en cuanto a los tipos –sin incluir inicialmente tipos recursivos– y permitir seleccionar el lenguaje de programación de bajo nivel, y ii), reducir el tamaño de la TCB al a) suponer las reglas de tipado no fiables y b) manejar la semántica del lenguaje de máquina desde el mismo componente que maneja la política de seguridad. Las reglas de tipado están separadas de la política para que sean demostradas primero antes de demostrar la política de seguridad de un programa específico. Se modelan los tipos desde los conceptos más básicos de las matemáticas (conjuntos, funciones y relaciones). El enfoque se denomina Foundational Proof Carrying Code FPCC pues

emplea el mínimo de axiomas, usa una lógica mínima o fundamental y dispone de un sistema de ejecución más pequeño por parte del consumidor. FPCC efectivamente reduce el tamaño de la TCB pero aumenta el coste y tamaño de las demostraciones y validaciones [Appel, 2001; Hamid et al., 2003; Hamid et al., 2002; Necula and Schneck, 2003b] salvo en [Appel and Mcallester, 2001] donde se usa la semántica para simplificar las pruebas y además poder considerar tipos recursivos. Para generar la demostración usaron TWELF en [Appel and Mcallester, 2001], el demostrador de teoremas Coq en [Felty, 2005] y Lambda-Prolog. Para validar la demostración aplican la comprobación de tipos de TWELF y Coq. Coq es un demostrador de teoremas basado en el cálculo de Construcciones inductivas CiC, un cálculo lambda con tipos, que permite definir tipos y predicados inductivamente. Las propiedades de seguridad demostradas y validadas comprenden el manejo seguro de la memoria en operaciones de lectura [Appel, 2001]. Expresan otras propiedades de seguridad en LF, sin demostrarlas ni validarlas en casos específicos. Tales propiedades son: 1) salidas periódicas de corrutinas para asegurar equidad y no bloqueos, 2) asignación y desasignación de memoria, 3) secuencias seguras de invocaciones de APIs, 4) límites en el número de candados (locks), 5) registro de todas las salidas (logged), y 6) el reenvío de paquetes sin modificación [Appel and Felten, 2001].

Oracle Proof Carrying Code: OPCC

Desde el enfoque PCC, se propone una variante en [Necula, 2001a; Necula and Rahul, 2001] denominada OPCC que usa testigos u oráculos de las demostraciones para reducir el tamaño de los certificados y de los tiempos de validación de certificados acerca del manejo seguro de los tipos de programas en lenguaje ensamblador compilados de Java. Para nueve ejemplos de programas Java, en promedio se logró reducir a un 12 % la proporción entre el tamaño del certificado y el tamaño del código, lo que significó una mejora de dicha proporción de 30 veces en relación con representaciones previas como LFi. Sin embargo el tiempo de validación fue 3 veces mayor que cuando se usa comprobación de tipos de expresiones LFi [Necula and Rahul, 2001].

PCC con Reglas de Inferencia no Fiables

También desde la perspectiva de PCC, en [Necula and Schneck, 2002; Necula and Schneck, 2003a] también se reduce el tamaño de la TCB al considerar las reglas de inferencia de los tipos, que se aplican en la generación y validación del certificado, como no fiables y que deben entonces ser también verificadas. En [Necula and Schneck, 2002; Necula and Schneck, 2003a] describen un compilador certificante para un subconjunto de Java que certifica el manejo seguro de la memoria y de los tipos y genera como certificado un término del cálculo lambda con tipos que es una derivación de un tipo. La demostración se hace interactivamente con el demostrador de teoremas Coq y la validación de la demostración también se reduce a la comprobación del tipo del término, utilizando la comprobación de tipos de Coq.

Abstracción y Comprobación de Modelos BLAST

En [Henzinger et al., 2002] se presenta una metodología basada en comprobación de modelos y abstracción perezosa, y una herramienta (BLAST) para verificar y certificar código en lenguaje C basada en interpretación abstracta perezosa, que permite construir de forma automática certificados de tamaño reducido del lado del productor del código. Las propiedades de seguridad se expresan como predicados sobre los estados de un autómata y pueden ser muy generales. Aunque, no desarrollan ninguna infraestructura para la validación de los certificados del lado del consumidor del código, como el certificado es una expresión LF, podrían emplearse validadores de demostraciones/certificados que hagan comprobación de tipos LF para hacer el proceso de validación; por ejemplo, los empleados para PCC en [Necula and Lee, 1996; Necula, 1997; Necula and Lee, 1998b] y para FPCC en [Appel and Felty, 2000; Appel, 2001].

FPCC con Reglas de Inferencia no Fiables y Testigos de Demostraciones

En [Appel et al., 2003; Wu et al., 2003], desde la perspectiva de FPCC, consideran el problema de reducir el tamaño del denominado testigo de la demostración y el tamaño del código con las reglas de inferencia de tipos no fiables, de forma

simultánea. El testigo de la demostración reemplaza a la demostración y es de menor tamaño que ella y facilita la validación de la demostración por parte del validador en el sitio del consumidor permitiendo emplear validadores pequeños. El generador de la demostración puede representarse como un programa lógico en el que las cláusulas representan las reglas de inferencia; entonces el testigo de la demostración es una traza de los objetivos y subobjetivos exitosos ejecutados por el programa lógico. Logran reducir el tamaño de la TCB a 2,668 líneas de código. El validador de las pruebas LF es el intérprete de programación lógica simplificado FLIT que maneja cláusulas dinámicas pero sin retroceso (backtracking) y tiene restringidos la forma de los objetivos y programas, de manera que reducen el tamaño del validador y lo hacen eficiente. El tiempo de la validación con FLIT es menor en un orden de magnitud en relación con TWELF [Appel et al., 2003]. Sin embargo, no se logra reducir el tamaño del certificado pues en algunos casos llega a ser 1000 veces el tamaño del código. En [Appel et al., 2003] consideran una propiedad de seguridad aplicable en lenguajes de alto nivel para un programa funcional con dos declaraciones y dos expresiones aritméticas con números enteros en un lenguaje similar a ML. La propiedad establece que el resultado de la última expresión debe ser un número par.

Model Carrying Code: MCC

En [Sekar et al., 2001] se presenta MCC (Model Carrying Code), como un nuevo paradigma para la seguridad de códsigo movil, en el que el código tiene asociado un modelo de alto nivel de su comportamiento desde el punto de vista de una propiedades de seguridad generales. Este modelo le permite al consumidor conocer el código, y verificar automáticamente la conformidad del código con la política general antes de ejecutarlo. En caso de que el código no sea conforme con la política antes de ejecutarlo, el consumidor puede refinar y ajustar la política. Además, el consumidor puede verificar el modelo y la política durante la ejecución del código, con un monitor de su ejecución. Las aplicaciones son clasificadas [Sekar et al., 2003] en *file only* y *communications only*: las aplicaciones *file only* no se pueden comunicar por la red, y pueden leer todos los archivos y escribir en ellos de forma restringida; las aplicaciones *communications only* no pueden hacer acceso a ningún archivo pero si pueden comunicarse mediante la red. Una refinación

de una política de una aplicación *file only* puede ser *no permitir acceso a la red despues de haber leído archivos sensibles*. En las políticas refinadas, los archivos críticos desde el punto de vista de la seguridad son dependientes de los sitios, por ejemplo los archivos con los registros y permisos de acceso del servidor del sitio. Los modelos basados en autómatas de estados finitos extendidos se generan en un análisis estático del código, que detecta las invocaciones a funciones del sistema operativo. Este análisis podría incluir las invocaciones de bibliotecas. La limitante es que no se puede aplicar a código cargado dinámicamente, aunque se propone una solución parcial basada en aprendizaje de máquina.

PCC Configurable: CPCC

En [Necula and Schneck, 2003b] se presenta CPCC, Configurable Proof Carrying Code, un marco formal que resulta más flexible y fiable que un sistema estándar de PCC, dado que la parte del verificador en que se debe fiar es más pequeña, mientras el resto del verificador puede ser configurado para ajustarlo a la política de seguridad y a la estructura del código móvil. Por otro lado CPCC requiere menos esfuerzo que el FPCC (Foundational Proof Carrying Code). CPCC es presentado como un punto medio entre PCC y FPCC, que combina la eficiencia, escalabilidad y bajo coste de TouchStone, el compilador certificante de Java, con la flexibilidad y mayor fiabilidad de FPCC. Es aplicable a lenguaje ensamblador, lenguajes de máquinas virtuales específicas como JVM o .Net y de máquinas virtuales genéricas. La política de seguridad obligatoria y general para todos los casos incluye el manejo seguro de la memoria, operaciones aritméticas y la ausencia de efectos secundarios, pero puede especializarse en cada caso. CPCC fue diseñado para llenar las necesidades del sistema Touchstone, aunque aún no hay información de los resultados de su implementación.

Sintactical Foundational Proof Carrying Code: SFPCC

En [Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005] se presenta el enfoque Syntactical Foundational Proof Carrying Code SFPCC como una variante al enfoque FPCC para reducir la complejidad y el tamaño de las demostraciones en FPCC, y para también contemplar tipos recursivos.

En esta variante se evitan los modelos semánticos y las largas demostraciones semánticas, de manera que la demostración de la seguridad la hace el comprobador de tipos al inferir los tipos de forma sintáctica. El certificado incluye además la demostración de la seguridad del sistema de tipos subyacente del cual no se supone su fiabilidad. Certifican programas en un lenguaje ensamblador simplificado con tipos denominado FTAL (Featherweigth Typed Assembly Language), que es una versión de TAL, y también puede ser empleado como lenguaje objeto de compiladores de Java, ML y un C con tipos seguros. FTAL soporta tipos recursivos, tipos mutables y punteros como valores de primera clase, además del tipo entero y del tipo registro, pero no soporta tipos polimórficos, existenciales ni de orden superior. Las propiedades de seguridad consideradas incluyen la no utilización de instrucciones ilegales que detengan la ejecución de la máquina, la comprobación de tipos de la asignación de memoria y el mantenimiento de invariantes acerca del estado de la asignación de memoria realizada. Pueden incluir además controles de acceso a determinadas regiones de la memoria. Utilizan Coq para generar las demostraciones y la comprobación de tipos de Coq para validarlas.

Con Coq también certifican programas en lenguaje XTAL [Hamid and Shao, 2004], un lenguaje ensamblador extendido con tipos, pero con la nueva posibilidad de que dichos programas puedan tener interfaz con otros programas certificados que usen otros sistemas de tipos, en lenguajes incluso de alto nivel, contribuyendo a la flexibilidad y escalabilidad del enfoque sintáctico para FPCC. En [Hamid, 2005] se presenta un lenguaje ensamblador con tipos y basado en regiones y capacidades (capabilities) llamado RgnTAL, para la administración certificada del manejo de la memoria, que había sido utilizado antes. La novedad es que ahora se incluye como parte de la base fiable del código. En [Ni and Shao, 2006] extienden el enfoque SFPCC para resolver el problema del manejo de punteros de código embebidos que hay en código binario de bajo nivel y en programas funcionales, extendiendo la sintaxis del lenguaje de aserciones para combinar relaciones de consecuencia semánticas con técnicas de demostración sintácticas. Muestran cómo manejar el polimorfismo y la mecanización de la técnica aunque no está implementada la propuesta.

TINMAN

En [Mok and Yu, 2002a; Mok and Yu, 2002b] se propone la arquitectura TINMAN para salvaguardar la seguridad de los recursos por parte de código escrito en lenguaje C, con comprobaciones estáticas y con monitorización durante la ejecución del código. El certificado producido en la comprobación estática especifica un pronóstico del consumo de recursos del código tanto en tiempo de ejecución como de memoria pedida explícitamente. Como el consumo exacto de recursos no puede de forma general predecirse antes de ejecutar el código TINMAN aplica el principio de *comprobar lo verificable y monitorizar lo no verificable*, con un conjunto de herramientas para predicción de uso de recursos, generación y validación de certificados e imposición de eventos en tiempo de ejecución. La política de seguridad está especificada con tres partes: 1) el uso de recursos por parte de cada servicio que puede ser invocado por el código y que provee el sistema huésped, 2) el uso de recursos del código y 3) el sistema de prueba para la interpretación de la especificación compuesto por axiomas y reglas. El comprobador estático o fuera de línea hace un proceso de cuatro etapas en las que: 1) hace primero un análisis sintáctico del código en C para producir el mismo código C con comentarios acerca del flujo de información; 2) se realiza el análisis del tiempo de ejecución y de la memoria demandada cuyo resultado es un esqueleto de uso de recursos que está parametrizado de forma que el valor exacto de las cotas se pueda luego determinar en el sitio del consumidor; este esqueleto está formado por un conjunto de anotaciones del código y aserciones; 3) posteriormente con base en el esqueleto de uso de recursos y la política de seguridad, se genera automáticamente una especificación del uso de recursos en la forma de un conjunto de predicados; 4) después los predicados son verificados por un modulo de generación de demostraciones para finalmente producir el certificado de uso de recursos. El análisis de tiempo determina el peor caso de uso de tiempo de ejecución que corresponde al mayor tiempo de uso, que depende de los bloques básicos y las estructuras de control del programa. Hay casos de bucles en los que no se puede inferir el tiempo de ejecución y se debe recurrir al programador para que suministre una cota superior. Para las invocaciones a servicios del sistema la política se establece con pre y postcondiciones parametrizadas de forma que la cota exacta sea determinada en el sitio del consumidor. Las políticas son flexibles y configurables para cada

plataforma específica y entorno de ejecución específico. Para la generación de la prueba del cumplimiento de la política de uso de recursos se utiliza el sistema PVS (Prototype Verification System), un sistema demostrador de teoremas para exploración y construcción de demostraciones que funciona de forma *top-down* dirigida por objetivos y que permite una amplia automatización del proceso de demostración. PVS se basa en lógica de orden superior con tipado fuerte e incluye un sistema de tipos rico y un comprobador de tipos. Como la demostración en la lógica de especificación que tiene PVS puede llegar a tener un tamaño considerable, solo se incluye en el certificado las estrategias usadas en la demostración y sus correspondientes parámetros, al que se le denomina el certificado esqueleto. En [Yu and Mok, 2004] detallan la formalización del marco empleado en TINMAN. Aunque los experimentos de la validación de la arquitectura solo consideran código escrito en C, el marco formal puede ser aplicado a otros lenguajes de alto nivel.

Prototype Proof Carrying Code: ProtoPCC

En [Wildmoser et al., 2004] se presenta ProtoPCC (Prototype Proof Carrying Code) un marco formal, que según sus autores puede ser considerado complementario del trabajo [Necula and Schneck, 2002], dado que una vez probaba la corrección del núcleo del generador de condiciones de verificación VCGen, se le puede aplicar la técnica de usar extensiones seguras y optimizadas. El marco es instanciado para un lenguaje de ensamble simple (SAL), con una política que prohibe los errores de tipos y los desbordes aritméticos. Se muestra un programa con anotaciones y la condición de verificación resultante. El productor de código puede escribir los programas anotados en Isabelle, un demostrador de teoremas inductivo basado en lógica de orden superior HOL; para obtener la condición de verificación se puede ya sea generar y ejecutar código ML para el VCGen usando resultados de otro trabajo, o utilizar el simplificador para evaluar el VCGen aplicado al programa. El productor puede probar la condición de verificación usando la herramienta de demostración, junto con una colección de teoremas HOL; en el caso del ejemplo son suficientes el simplificador y un procedimiento para aritmética de Presburger. Para el lado del consumidor, Isabelle provee términos de prueba y un validador de pruebas. Las demostraciones son codificadas como

términos Lambda que tienen un tipo que corresponde al teorema que se demuestra, de modo que la comprobación de la demostración es una comprobación del tipo. En un segundo trabajo se usó este marco para certificar programas en código bytecode, de forma que tampoco causen desborde aritmético ni desborden vectores [Wildmoser et al., 2005]. El código bytecode analizado es producido por un compilador del lenguaje Jinja, un subconjunto de Java con creación de objetos, herencia, invocación dinámica de métodos y manejo de excepciones, pero solo certifican programas con operaciones de aritmética básica e instrucciones de movimiento de datos sin creación de objetos ni invocación a métodos. Las anotaciones del código que incluyen anotaciones de los intervalos de los índices de los vectores, usan un lenguaje de primer orden específicamente diseñado para bytecode de la máquina virtual Java JVM y son generadas automáticamente en [Wildmoser et al., 2005] a diferencia de los trabajos previos [Wildmoser et al., 2004] donde son generadas manualmente.

Proyecto Mobile Resources Guarantees MRG

En 2002-2005 la Unión Europea lleva a cabo el proyecto MRG (Mobile Resources Guarantees) con el propósito de desarrollar la infraestructura necesaria para acompañar el código móvil con certificados que describan su comportamiento en relación con los recursos, de manera que tales certificados puedan ser verificados de forma independiente del productor del código [Sannella et al., 2005; Gilmore and Prowse, 2005]. En el marco del proyecto MRG se desarrollan dos lenguajes funcionales, Camelot y Grail, para certificar el consumo de recursos de memoria del *heap* por parte del código de funciones Camelot, como función lineal del tamaño de la entrada de la función. Camelot es un lenguaje de programación funcional de alto nivel, de primer orden e impuro (de la familia de ML), que tiene objetos y clases, diseñado para uso seguro de recursos. El compilador de Camelot produce dos resultados: un código objeto en el lenguaje de bytecodes Grail (Guaranteed Resource Allocation Intermediate Language), y una demostración del uso de recursos que establece el consumo de recursos por parte del programa Grail. El compilador Grail genera código para la máquina JVM y soporta los formatos de bytecode de Java, Javacard (para dispositivos *smart* basados en Java) y del bytecode CIL de Microsoft del *framework* .NET [Beringer et al., 2003; MacKenzie

and Wolverson, 2004]. El productor del código usa los compiladores de Camelot y Grial, para generar con el código JVM dos archivos Isabelle con los tipos LDF [Hofmann and Jost, 2003] del código (el certificado) y la sintaxis abstracta de Grail. El consumidor utiliza el demostrador de teoremas Isabelle para validar el certificado. En [Aspinall et al., 2004b] se presenta el marco formal de MRG, para dispositivos *smart*, y otros dispositivos móviles y pequeños en su tamaño, en los que por esto mismo hay limitaciones en su capacidad computacional, especialmente por la memoria reducida que tienen, en comparación con los ordenadores personales. En [Aspinall et al., 2004b] se mencionan dos problemas y cómo los resolverían: i) un problema del lado del consumidor es el gran tamaño de Isabelle, aún para computadores personales, por lo cual proponen el desarrollo de una herramienta como Isabelle, pero especializada en la validación del certificado, y así, de menor tamaño y menor complejidad que Isabelle. El otro problema, ii) del lado del productor, es la instanciación de muchos cuantificadores que no son instanciados de forma automática por los resolvedores estándar de Isabelle, para el cual proponen usar aserciones derivadas, suficientes pero incompletas.

Abstract Carrying Code: ACC

En [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004] se propone "Abstraction-carrying code" (ACC) como mecanismo para certificar propiedades temporales (vivacidad) de código móvil escrito en lenguaje C. En ACC el tamaño del certificado de propiedades temporales es por lo general significativamente menor que el tamaño del programa. La propiedad temporal se expresa como un predicado el lógica lineal temporal LTL sin operador "next". El certificado es un programa booleano resultado de la abstracción del programa con base en el predicado LTL. El programa booleano está codificado como anotaciones de tipos en el lenguaje ensamblador con tipos catalogados ("indexed") SDTAL ("Simplified Dependent Typed Intermediate Language"). El consumidor del código comprueba el tipo del certificado para validar la abstracción y luego ejecuta un comprobador de modelos abstractos para validar la propiedad temporal. La abstracción que produce la función booleana se hace con base en el programa fuente C. La compilación recibe además del fuente C la función booleana que es usada para anotar el programa producido en el lenguaje intermedio. Así se

garantiza que el modelo abstracto del programa fuente (la función booleana) es el mismo modelo abstracto del programa compilado en lenguaje intermedio. Las desventajas del enfoque son el gran tamaño de la base de código fiable que incluye el comprobador de tipos de SDTAL y el comprobador de modelos, y la menor eficiencia durante la validación que hace el consumidor al comprobar el modelo. La implementación ACCEPT/C usa la herramienta BLAST ([Henzinger et al., 2002]) para la generación del grafo de flujo de control del programa C.

En [Albert et al., 2005a] se presenta Abstract Carrying Code ACC que es la Aplicación de Interpretación Abstracta a PCC, como una forma de hacer MCC Model Carrying Code en el marco de lenguajes multiparadigma utilizando Constraint logic Programming CLP. En este caso el modelo usado es la interpretación abstracta del programa mediante una aproximación de punto fijo. Se usa el lenguaje de aserciones de alto nivel de Ciaopp, que usa pre y postcondiciones, para expresar las políticas de seguridad de forma parametrizada respecto a los dominios abstractos, con lo cual se tiene expresividad y generalidad. Con este enfoque se está a más alto nivel que con otros enfoques de PCC (incluso de MCC). Se tiene más flexibilidad en las políticas que con MCC en el que el comportamiento seguro del código está predefinido y que la mayoría de enfoques de PCC en los que las políticas se refieren a manejo seguro de la memoria y de los tipos, aunque las aserciones de la política de seguridad requerida deben suministrarse manualmente. En los ejemplos mencionados en [Albert et al., 2005a] se usan dominios de tipos regulares (describibles como términos regulares); el usuario puede definir sus términos regulares y emplear los preexistentes en Ciao, numeros, cadenas, listas, constantes, así como los que permiten manejar archivos; de ésta forma se pueden expresar políticas como el acceso seguro a archivos, a través de la definición de nombres de archivos seguros, que permiten definir las precondiciones para el procedimiento que abre un archivo. Otras propiedades verificadas [Albert et al., 2004a] son el determinismo, la terminación, el no fallo, y el no tropiezo. En trabajos posteriores [Albert et al., 2004b; Albert et al., 2005d] se incluyen políticas de eficiencia en el manejo de los recursos (resource awareness) de los sitios receptores del código móvil, específicamente de que el coste en espacio y tiempo de las funciones que se ejecutarían en cada sitio está acotado y que las funciones no tengan efectos secundarios. En [Albert et al., 2006] se presenta una forma de generar

certificados en ACC con tamaño reducido, que se pueden validar y reconstruir en un solo paso. En los experimentos obtuvieron reducciones de en promedio hasta tres veces en el tamaño sin afectar el tiempo de validación/comprobación más allá del 6 %. En ACC hay limitaciones en el enfoque como se indica en [Albert et al., 2004b] por la imprecisión de la aproximación de la interpretación abstracta o por la indecibilidad de algunas propiedades.

Interactive PCC: iPCC

En [Tsukada, 2000; Tsukada, 2005] se propone una extensión al mecanismo de PCC para que sea interactivo y probabilístico, con el fin de atacar el problema en PCC del crecimiento del tamaño de las demostraciones en relación con el tamaño y complejidad del código, que restringe su aplicación a casos con tamaños razonables de demostraciones que el consumidor pueda validar en un tiempo razonable. El tiempo de validación de una demostración del cumplimiento de una propiedad de un código con m líneas de código y n condicionales es $2^k * poly(m)$ con PCC, mientras que es $poly(m)$ con iPCC. La probabilidad de que la demostración con iPCC sea correcta es $\leq 1/2^m$. Se establece además, que la clase de propiedades demostrable eficientemente con iPCC (PSPACE), es más grande que la clase de propiedades (NP) que son eficientemente demostrables con PCC. La propiedad de seguridad se expresa de forma negativa en fórmulas booleanas cuantificadas existencialmente, de modo que el programa es seguro si y solo si la fórmula es falsa. La fórmula booleana es luego expresada de forma aritmética mediante una sumatoria para valores de cero y uno sobre todas las variables, en que se reemplazan las conjunciones por multiplicaciones, las disyunciones por sumas, las variables sin negar se dejan tal cual, y las variables negadas ($\neg X$) se reemplazan por la el complemento ($1 - X$). La fórmula booleana cuantificada es falsa, sí y solo si su versión aritmetizada es igual a cero. La fórmula aritmetizada es un polinomio de grado uno cuyos coeficientes envía el productor del código al consumidor de forma interactiva uno por uno. Tales coeficientes son la demostración de la propiedad que satisface el código. La propuesta no está implementada aún.

Proyecto LANDE

En [Besson et al., 2005; Besson et al., 2006] se presenta otro enfoque para FPCC pero también basado en interpretación abstracta, para certificar código de un subconjunto de Java y de la máquina virtual Java JVM (bytecode). El validador es entregado al consumidor junto con una demostración de su corrección en Coq. El consumidor valida esta demostración usando el comprobador de tipos de Coq. La interpretación abstracta realizada en Coq, produce un término lambda como certificado, que es validado usando también el demostrador de teoremas Coq. Las propiedades de seguridad certificadas se refieren en general a valores de variables que están en rangos especificados. Se certificó el manejo seguro de vectores en seis ejemplos de código (tres de ordenamiento, dos productos: convolución y de polinomios, y Floyd-Wharshall). Para hacer más eficiente la generación y validación de los certificados con Coq se utilizó el mecanismo de extracción de programas de Coq, que genera programas en lenguaje Caml. El certificado toma la forma de estrategias para reconstruir el punto fijo y está comprimido mediante una técnica de compresión del punto fijo. En la validación, se reconstruye el punto fijo usando las estrategias del certificado. Los certificados son de un tamaño menor en un grado de magnitud que el código correspondiente, y por eso el tiempo de validación es despreciable (10-60ms).

Java Resource Bounds Verifier: JVer

Otra propuesta [Chander et al., 2005a; Chander et al., 2007] relacionada con políticas de topes en el consumo de recursos físicos (CPU, memoria, disco, ancho de banda) y virtuales (ficheros, conexiones a bases de datos, procesos (hilos) establece una verificación estática de validaciones dinámicas expresadas mediante anotaciones del código que determinan la adquisición de los recursos de forma previa a su consumo. La propuesta es de aplicación general y comprende un generador de condiciones de verificación y un demostrador que aplica satisfacibilidad. En la actualidad hay dos implementaciones: una está aplicada a código fuente Java con anotaciones y usa el generador de condiciones de verificación de ESC/Java y el demostrador de teoremas Simplify [Chander et al., 2005a], pero no genera las demostraciones para que sean validadas de forma independiente; la otra

implementación incluye la herramienta JVer como generador de condiciones de verificación para Java Bytecode anotado, y usa el demostrador de teoremas Kettle como generador de demostraciones validables de forma independiente [Chander et al., 2005b].

2.2. Aproximaciones Alternativas o Complementarias a PCC

A continuación se presentan las aproximaciones relacionadas con PCC pero que son extensiones o complementos que se salen del paradigma de PCC al obviar algunos de sus problemas o asunciones. La siguiente aproximación recurre a una autoridad certificadora para aquellas propiedades no certificables con la tecnología actual de PCC.

PCC con Autorización

Saliéndose del paradigma de PCC, en [Whitehead et al., 2004] se propone un sistema de autorización para PCC que complementa un sistema PCC con un sistema de autorización basado en el lenguaje BINDER que permitiría mediante una autoridad certificadora fiable certificar con firmas digitales aquellas propiedades de seguridad más complejas y por fuera del alcance del sistema de PCC, o también la TCB, la base de código fiable. BINDER usa una lógica de autorización. Los consumidores de código expresan sus políticas en esa lógica que describe relaciones de confianza con otros agentes, definen predicados declarativamente y pueden incluir axiomas para el razonamiento. El sistema se basa en razonamiento y confianza y permite expresar e imponer políticas de seguridad. Se ilustra el uso del marco con un ejemplo de políticas de manejo seguro de la memoria y manejo acotado de comunicaciones (número limitado de paquetes por segundo).

La siguiente aproximación permite obviar la asunción acerca de la fiabilidad del validador.

Validador Certificado para FPCC

En [Barthe et al., 2007] se define el primer sistema de tipos basado en flujo de información para un lenguaje tipo bytecode de JVM, con clases, objetos, arreglos, excepciones e invocaciones de métodos que garantiza la no interferencia de los programas con tipos comprobados. La política de seguridad está basada en un retículo de niveles de seguridad (confidencialidad). El atacante, los campos de las clases, las variables locales y los parámetros de los métodos tienen niveles de seguridad, así como las excepciones que puede lanzar cada método. El nivel de seguridad del atacante determina sus capacidades de observación. La corrección fué demostrada con el demostrador de teoremas Coq y de tal demostración se extrajo un verificador liviano certificado de flujos de información de bytecode. El sistema aplica los principios de la verificación de tipos estándar de bytecode y puede ser empleado en la arquitectura estándar de seguridad en la JVM. El verificador certificado puede ser también usado como un validador de demostraciones para FPCC por parte de los consumidores de código móvil, pero con certificados compactos.

La siguiente aproximación simplifica el problema de la fiabilidad del código al certificar el programa con base en el certificado del resultado del análisis del programa en lugar del certificado del análisis en sí.

Certificación de Resultados de Análisis Relacional de Programas

La propuesta [Besson et al., 2007] define un sistema para certificación de propiedades de consumo de recursos y de memoria segura (como el acceso a vectores dentro de sus límites) en bytecode imperativo con procedimientos, vectores y variables globales. El sistema infiere de forma automática invariantes de bucles, y pre y postcondiciones de métodos, aplicando interpretación abstracta poliédrica y ejecución simbólica. El punto fijo obtenido, que representa un invariante del programa y es un sistema de restricciones lineales, es abreviado de acuerdo a la política de seguridad para reducir su tamaño y abaratar su verificación. El certificado está compuesto por las anotaciones con los invariantes y una inclusión poliédrica (sistema de restricciones lineales) producida con el método Sim-

plex. La inclusión poliédrica está representada como un vector de coeficientes de combinación lineal de restricciones validable en tiempo cuadrático. El analizador está implementado en Caml, y el comprobador del resultado en el demostrador Coq.

La siguiente aproximación elude el problema del tamaño y la representación del certificado ya que el productor no entrega un certificado que demuestre la fiabilidad del código móvil sino una extensión ejecutable del verificador específica para el código fuente y el compilador usado.

Una instancia no estándar de FPCC: Open Verifier

En [Chang et al., 2005] se presenta Open Verifier, un marco formal para el desarrollo de verificadores fundacionales, como una instancia de FPCC en la que se verifica código no fiable usando verificadores particularizados ("customized") al programa. La propuesta se sale del marco general de PCC y tiene algunas restricciones en relación con FPCC pero es más fácil de llevar a la práctica. El productor del código no envía la demostración sino la extensión ejecutable del verificador específica para un lenguaje fuente y una estrategia de compilación dada. Esto evita el principal problema de FPCC acerca de la representación de las demostraciones y su gran tamaño [Necula and Rahul, 2001]. No usa un sistema de tipos general aplicable a cualquier lenguaje de programación y es aplicable a código ensamblador en lugar de código de máquina. El consumidor genera la demostración usando el Open Verifier y la extensión correspondiente al lenguaje del código. El productor envía con el código metadatos que son usados por la extensión durante el proceso de generar la demostración. Estos metadatos pueden ser solo los tipos de las entradas y salidas de todas las funciones del programa pero pueden además incluir demostraciones completas de la seguridad de instrucciones particulares. La extensión no fiable es un generador de predicados que son postcondiciones débiles acerca del programa. Open Verifier incluye un módulo fiable para generar las postcondiciones más fuertes expresado como código ejecutable en lugar de usar axiomas lógicos para describir las transiciones de la máquina. De este modo no todas las verificaciones son fundacionales. Los otros componentes fiables del marco son el Validador y el aproximador de punto fijo. El Validador, el aproximador y el generador de postcondiciones iteran sobre los

predicados débiles producidos por la extensión no fiable. Las demostraciones se construyen por niveles. Todo el razonamiento específico al programa verificado es manejado usando cláusulas de Horn y las extensiones pueden usar un intérprete Prolog en la generación de las pruebas. Sin embargo se usa una lógica más rica y el demostrador de teoremas Coq para los constructores de fórmulas de las extensiones. Hay una implementación del marco Open Verifier con varias extensiones que soportan diferentes estrategias de verificación. Se destacan las extensiones para PCC tradicional (PCCExt), para TALx86 [Morrisett et al., 1999a](TALExt) y la del lenguaje Cool, un lenguaje orientado por objetos realista (similar a un subconjunto de Java o C#) llamada CoolAid.

La siguiente aproximación también elude el problema del tamaño y la representación del certificado. El productor no entrega un certificado que demuestre la fiabilidad del código móvil sino un verificador desarrollado usando el marco formal propuesto.

Análisis Certificado de Programas para FPCC

En [Chang et al., 2006] se propone un marco formal basado en interpretación abstracta para certificar el análisis de programas que puede ser aplicado a una infraestructura para FPCC. Este trabajo se sale del marco tradicional de PCC, pues el productor del código no suministra una demostración del cumplimiento de una política del consumidor, sino un verificador desarrollado y certificado usando el marco formal que se propone en el trabajo,. El marco se considera la primera evidencia publicada de que la certificación fundacional de código puede ser escalada. El marco distingue dos actividades: la instalación que ocurre una vez por analizador, y la verificación que ocurre una vez por cada programa que se analiza. En la instalación, con base en el código fuente del certificador y su demostración de corrección, se valida esta demostración empleando tres componentes fiables: el compilador, el extractor de modelos/especificaciones y el validador en sí. En la verificación se aplican el certificador (resultado de la compilación del fuente) y el analizador. El marco traduce de forma automática las implementaciones de los análisis escritas en ML en modelos y especificaciones para el asistente de demostraciones Coq, incluyendo efectos secundarios y funciones recursivas no primitivas. El productor del código no fiable tiene total libertad en la selección de los

mecanismos de seguridad y las estrategias de compilación. El único requisito es que pueda suministrar un verificador de código en la forma de un análisis certificado, cuya demostración de corrección atestigua que el análisis impone la política de seguridad deseada por el receptor-consumidor del código. Se implementó un prototipo del marco para analizar código de lenguaje ensamblador x86 producido por TALx86 [Morrisett et al., 1999a] excluyendo solo las características de modularidad. También se hizo una implementación parcial del verificador de código Bytecode JBV que excluía excepciones, inicialización de objetos y subrutinas. El certificador ha sido implementado en ML. El extractor de especificaciones fué implementado en Ocaml y soporta un subconjunto grande de ML. La base de código fiable incluye el compilador OCaml y el validador de demostraciones de Coq.

En la siguiente aproximación, se certifica código ensamblador compilado de lenguaje C con herramientas que imponen propiedades de tipos y punteros seguros, mediante anotaciones e instrumentación del código. Las anotaciones son usadas en la comprobación estática de los tipos, y la instrumentación comprende instrucciones para hacer comprobaciones dinámicas. El marco formal no incluye un validador del certificado.

Certificación de Código Ensamblador

En [Harren and Necula, 2005; Harren, 2007] se presenta una aproximación basada en tipos dependientes estáticos y dinámicos para certificar código ensamblador producido por herramientas que imponen propiedades de seguridad a nivel de código fuente. La propuesta ha sido implementada para certificar las propiedades impuestas por las herramientas CCured [Condit et al., 2005] y Cqual. CCured impone tipos seguros en código C mediante la clasificación de los punteros del código de acuerdo al uso que tienen. La seguridad comprende punteros, vectores y memoria segura. CCured anota e instrumenta el código C que luego puede ser compilado con un compilador optimizador a código ensamblador x86. La instrumentación es código ensamblador embedido en el código C que hace las comprobaciones dinámicas. El verificador-certificador está parametrizado por una política de tipos que describe los invariantes que impone la herramienta de seguridad. El verificador usa interpretación abstracta sobre el dominio de las expresiones simbólicas para inferir los tipos de los registros y asegurar que ca-

da instrucción preserva la memoria segura. En [Condit et al., 2007] proponen un sistema también basado en tipos dependientes para lenguajes imperativos de bajo nivel que implementan para C en el compilador anotador y comprobador fuente a fuente Deputy. Con Deputy se pueden comprobar punteros, y punteros a posiciones de arreglos seguros, aritmética de punteros segura y manejo seguro de uniones. La certificación implementada para CCure y CQual puede ser extendida para certificar las que impone Deputy.

2.3. Aspectos Tecnológicos de PCC

2.3.1. Asunciones

Las tres principales asunciones se refieren a [Necula and Lee, 1996; Lee and Necula, 1997; Necula, 1997; Necula and Lee, 1998a; Necula and Lee, 1998b; Appel and Felty, 2000; Necula, 2001b; Necula, 2001a; Necula and Rahul, 2001; Sekar et al., 2001; Appel and Mcallester, 2001; Hamid et al., 2003; Hamid et al., 2002; Necula and Schneck, 2003b; Sekar et al., 2003; Hamid and Shao, 2004; Albert et al., 2005a; Felty, 2005; Hamid, 2005; Tsukada, 2005; Besson et al., 2006]:

- la no fiabilidad del productor, distribuidor y certificador del código,
- la fiabilidad de todo o parte del código que usa el consumidor del código para validar el certificado,
- no hay un tercero fiable como autoridad certificadora.

Sólo un trabajo [Whitehead et al., 2004] supone la participación de un tercero fiable como autoridad certificadora para aquellas propiedades no certificables via PCC.

Recientemente se están desarrollando propuestas con variantes en el uso de componentes fiables en la infraestructura que usa el consumidor para asegurar la fiabilidad del código. Una, es el uso de componentes cuya fiabilidad es certificable formalmente cada vez que se instalan [Chang et al., 2006]. Otra, el uso de validadores cuya fiabilidad está certificada formalmente [Barthe et al., 2007]. Por consiguiente, en la infraestructura que usa el consumidor pueden coexistir,

componentes no fiables (pero cuya fiabilidad es verificable), componentes fiables y componentes certificados.

2.3.2. El contenido de la base de código fiable TCB

En los primeros trabajos [Necula and Lee, 1996; Necula and Lee, 1997b; Necula, 1997] la base de código fiable se compone solo del validador, un programa en C de 5 páginas de longitud que implementa el algoritmo de comprobación de tipos del marco lógico LF de forma optimizada.

En los siguientes trabajos [Necula and Lee, 1998a; Necula and Lee, 1998b; Colby et al., 2000; Necula, 2001b] el código fiable que usa el consumidor para validar los certificados contiene: el generador de las condiciones de verificación VCs, y el comprobador de tipos que valida el certificado. El generador de las condiciones de verificación incluye 1) la máquina abstracta que modela la semántica del lenguaje, y 2) el generador en sí, con los axiomas y las reglas de inferencia. En el compilador certificante para un subconjunto de Java, el Generador de las VCs tiene aproximadamente 15Kbytes de tamaño y el Validador 4Kb.

En [Necula and Schneck, 2002; Necula and Schneck, 2003a; Necula and Schneck, 2003b; Wildmoser et al., 2004; Wildmoser et al., 2005; Wildmoser and Nipkow, 2005] para reducirle el tamaño, excluyen de la TCB las reglas de inferencia que se usan en la demostración de la política de seguridad.

En [Appel and Felty, 2000; Appel, 2001] la semántica de las instrucciones de máquina no la maneja el generador de VCs sino la política de seguridad, con el fin de reducir el tamaño del generador de VCs y reducir así la TCB, pero con el coste de complicar las pruebas.

En [Necula and Schneck, 2003b] donde el verificador es configurable y se ha descompuesto en dos partes, una general aplicable en todos los casos, y otra configurable para ajustarla a la política de seguridad y a la estructura del código móvil. La parte general del verificador se supone fiable, mientras que la parte configurada no.

En [Wu et al., 2003], consideran el problema de reducir el tamaño del denominado testigo de la prueba y de las reglas de demostración no fiables de forma simultánea. Implementan el validador de las pruebas LF como un intérprete de programación lógica simplificado, que maneja cláusulas dinámicas pero no hace

retroceso (backtracking) con la forma de los objetivos y programas restringidos, de manera que reducen el tamaño del validador y lo hacen eficiente. El intérprete es denominado FLIT y se describe en [Appel et al., 2003].

En [Chang et al., 2005], los componentes fiables del Open Verifier incluyen el módulo para generar las postcondiciones más fuertes, el Validador y el aproximador de punto fijo. Se considera no fiable la extensión suministrada por el productor del código, especializada en un lenguaje de programación.

En varios trabajos [Hamid and Shao, 2004; Hamid, 2005; Besson et al., 2005; Besson et al., 2006; Besson et al., 2007] el contenido de la TCB se reduce al comprobador de tipos del demostrador de teoremas Coq, cuyo tamaño es relativamente reducido.

2.3.3. El Tamaño del Certificado

El problema es el gran tamaño de los términos producidos; por eso para PCC en [Necula and Lee, 1997a] modifican el comprobador de tipos de LF para omitir partes importantes y redundantes de los términos que producen las derivaciones, usando una representación implícita para LF (llamada LFi). Esta reducción puede ser de un factor de 3 para términos con 140 tokens o de 1000 para algunas de las pruebas más largas [Necula and Rahul, 2001].

En [Necula and Rahul, 2001; Necula, 2001a], el certificado es un oráculo representado como un flujo de bits que resuelven las opciones de interpretación no determinísticas. El oráculo es un testigo de la prueba que sintetiza las decisiones de la derivación del tipo que es la prueba. Se mencionan reducciones en el tamaño de la prueba en un factor de 30, con pruebas cuyo tamaño es en promedio el 12 % del tamaño del código Los experimentos referidos en [Necula and Rahul, 2001] incluyeron 10 programas cuyo tamaño en Java era entre 1588 y 229853 líneas de código, para los cuales los tamaños de las pruebas LFi estaban entre 49804 y 10813894 bytes, y los tamaños de los oráculos entre 1936 y 354034 bytes.

En [Grossman and Morrisett, 2001; Morrisett et al., 1999a] se presenta un compilador certificante para una versión de TAL para computadoras de arquitectura x86, llamado TALx86, en el que emplean técnicas para reducir el tamaño de las anotaciones y reducir el tiempo de la verificación, es decir de la comprobación de los tipos, que son experimentadas en una aplicación de tamaño considerable

como es el compilador mismo. El experimento con las bibliotecas de funciones y otros elementos del compilador mostró que, el tamaño de las anotaciones no superó el tamaño del código, oscilando entre el 43 y el 67 %, y que el tiempo de verificación resultó lineal en relación con el tamaño del código verificado.

En [Wu et al., 2003], el testigo de la prueba reemplaza a la prueba y es de menor tamaño que ella y facilita la validación de la prueba por parte del validador en el sitio del consumidor permitiendo emplear validadores pequeños. El generador de la prueba puede representarse como un programa lógico en el que las clá usulas representan las reglas de inferencia; entonces "el testigo de la prueba es una traza de los objetivos y subobjetivos exitosos ejecutados por el programa lógico", es decir "es un árbol en el que los nodos son las identificaciones de las cláusulas y los subárboles son los subobjetivos de dichas cláusulas" [Wu et al., 2003].

En TINMAN [Mok and Yu, 2002a; Mok and Yu, 2002b; Yu and Mok, 2004], como usan PVS para generar la demostración, y en la lógica de especificación que tiene PVS las demostraciones pueden llegar a tener un tamaño considerable, solo incluyen en el certificado las estrategias usadas en la demostración y sus correspondientes parámetros, al que se le denomina el certificado esqueleto.

En [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004], usan interpretación abstracta para generar certificados de forma que el tamaño del certificado de propiedades temporales es por lo general significativamente menor que el tamaño del programa.

En [Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005] se presenta una variante a FPCC que evita los modelos semánticos sofisticados y las largas pruebas semánticas con un enfoque sintáctico. Las pruebas se codifican en CiC, utilizando el demostrador Coq.

En [Tsukada, 2000; Tsukada, 2005] presentan la propuesta "interactive Proof Carrying Code"iPCC con la noción de demostración interactiva como una alternativa al problema de la explosión del tamaño y complejidad de las demostraciónes en PCC, que ocurre cuando se aplica a programas complejos escritos en lenguajes de bajo nivel, y con políticas de seguridad más exigentes. En PCC la validación de la demostración de un programa con condicionales es exponencial en relación con el número de condicionales, mientras que en iPCC es polinomial. En este en-

foque la demostración es probabilística y la probabilidad de error es inversamente exponencial en relación con el tamaño del código. La demostración está formada por los coeficientes de un polinomio de grado bajo.

En [Besson et al., 2005; Besson et al., 2006] el certificado está comprimido mediante una técnica de compresión del punto fijo. Los certificados son de un tamaño menor en un grado de magnitud que el código correspondiente, y por eso el tiempo de validación es despreciable (10-60ms).

En [Chang et al., 2005], para evitar el problema principal de FPCC acerca de la representación de las demostraciones y su gran tamaño, el productor del código no envía la demostración sino la extensión ejecutable del verificador específica para un lenguaje fuente y una estrategia de compilación dada. De este modo no hay certificado.

En [Albert et al., 2006] presentan la noción de certificado reducido ACC que es el subconjunto de la abstracción que el validador necesita para reconstruir y validar el certificado en una sola pasada. La ganancia en la reducción depende del número de iteraciones realizadas en el análisis y de su eficiencia. Implementaron un generador y validador de los certificados reducidos llegando a reducciones en tamaño en un factor de 3 con variaciones pequeñas en el tiempo de validación.

En [Chang et al., 2006] también resuelven el problema del formato y tamaño del certificado como en [Chang et al., 2005] eliminándolo. En su lugar el productor envía una analizador desarrollado y certificado usando el marco formal propuesto.

2.3.4. Los Problemas Clave de PCC

Los diferentes marcos formales para PCC abordan cuatro aspectos que surgen en PCC [Necula, 1997; Necula and Lee, 1997b; Lee and Necula, 1997; Appel, 2001; Albert et al., 2005a; Chang et al., 2005]:

- La semántica del lenguaje de programación específico cuyo código se quiere certificar. Este lenguaje puede ser de bajo nivel, de alto nivel o la combinación de lenguajes de alto-bajo nivel que son los lenguajes fuente y objeto de un compilador,

- La representación concreta de las políticas de seguridad, mediante la selección/diseño de un lenguaje para expresar las propiedades de seguridad,

- La representación concreta de las demostraciones mediante la selección (o desarrollo) del formalismo y de las tecnologías correspondientes, para la automatización de la generación de las demostraciones que constituyen los certificados del lado del productor o certificador del código.
- La validación de las demostraciones, los certificados, del lado del consumidor, de una manera automática, eficiente, y simple.

2.3.5. Lenguajes de Programación y Propiedades de seguridad certificadas

La mayoría de trabajos tratan propiedades de seguridad de bajo nivel (manejo seguro de la memoria y de los tipos, manejo seguro de vectores sin desborde de sus índices) de código de lenguajes de programación también de bajo nivel: ensambladores de máquinas específicas, ensambladores con tipos como TAL, FTAL, y XTAL, código para máquinas virtuales como JVM (bytecode) o .Net.

Aunque hay trabajos sobre código de lenguajes de bajo nivel y con propiedades de alto nivel como i) el control de acceso a archivos con autorización y el manejo de topes en recursos (ciclos de procesador y ancho de banda), por parte de código de lenguaje ensamblador [Necula and Lee, 1998b], ii) consumo acotado de memoria del "heap"por parte de código Grail para la máquina virtual Java JVM ("bytecode") [Beringer et al., 2003; MacKenzie and Wolverson, 2004], iii) topes en el consumo de recursos físicos (CPU, memoria, disco, ancho de banda) y virtuales (ficheros, conexiones a bases de datos, procesos (hilos) para Java Bytecode anotado [Chander et al., 2005b], y iv) consumo de recursos y de memoria segura (como el acceso a arreglos dentro de sus límites) para un subconjunto de JVM (bytecode imperativo con procedimientos, arreglos y variables globales) [Besson et al., 2007].

Los trabajos acerca de los compiladores certificantes tratan propiedades de bajo nivel como el manejo seguro de la memoria, de los vectores, y de los tipos, pero para código en lenguajes fuentes de alto nivel como subconjuntos de Java (Simple J, TouchStone), los dialectos de C con tipos y punteros seguros (Safe-C, Popcorn, Cyclone, C-Minor), y los lenguajes funcionales ML (FLINT/ML, TIL) y Camelot.

Las propiedades de seguridad de alto nivel para lenguajes de alto nivel considerados en los trabajos presentados en el estado del arte, se señalan en la Figura 2.1, a continuación.

Políticas	Lenguajes
El resultado par de una expresión o función [Wu et al., 2003]	Funcional tipo ML
Consumo acotado de memoria (espacio) en el uso del "heap"(la cantidad de memoria que consume una función depende de forma lineal del tamaño de la entrada de la función) [Hofmann and Jost, 2003; Beringer et al., 2003; MacKenzie and Wolverson, 2004; Sannella et al., 2005; Gilmore and Prowse, 2005; Aspinall et al., 2004b]	Funcional Camelot
La apertura segura de archivos en directorio temporal [Albert et al., 2005a] La terminación, el determinismo, el no tropiezo y no fallo [Albert et al., 2004a] El tiempo de proceso y el consumo de memoria para estructuras de datos acotados y ausencia de efectos secundarios [Albert et al., 2004b; Albert et al., 2005d]	Lógico con restricciones Ciao
Consumo de recursos como la memoria para estructuras de datos, la memoria del "heap"demandada explícitamente, el ancho de banda, y el tiempo de proceso acotados [Mok and Yu, 2002a; Yu and Mok, 2004]	Imperativo C
Propiedades temporales expresadas como predicados en lógica LTL sin operador "next"como vivacidad [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004]	Imperativo C
Consumo con topes de recursos físicos (CPU, memoria, disco, ancho de banda) y virtuales (ficheros, conexiones a bases de datos, procesos (hilos) [Chander et al., 2005a; Chander et al., 2007]	Imperativo orientado a objetos Java

Figura 2.1: Propiedades y lenguajes de alto nivel

Otras propiedades de seguridad no certificadas en ningún trabajo implementado de PCC, pero que son mencionadas como tales en algunas propuestas teóricas son: 1) la no interferencia (filtración de información de información confidencial) [Smith and Volpano, 1998] en programas Java mencionada en el trabajo [Barthe et al., 2006] del proyecto MOBIUS ("Mobility, Ubiquity and Security") [Mobius, 2005; Barthe, 2005] y en [Barthe et al., 2007] donde se propone el primer sistema de tipos basado en flujo de información para un lenguaje tipo bytecode de JVM, 2) propiedades temporales diferentes a vivacidad por ejemplo, aquellas no expresables en JML pero sí usando extensiones del lenguaje de especificación JML, como no solicitar el PIN después de haber iniciado una transacción y antes de terminarla [Pavlova et al., 2004].

2.3.6. La determinación de las Propiedades de seguridad

Hay diferentes enfoques:

- Inicialmente en PCC, en FPCC, y en SFPCC las propiedades de seguridad son especificadas y publicadas por los consumidores.
- Los compiladores certificantes (TIL, TAL, TILT, FLINT), manejan propiedades de seguridad preestablecidas y fijas, usualmente de bajo nivel (memoria, arreglos, tipos).
- En MCC son genéricas y generales y están definidas por el productor, pero pueden ser especializadas por el consumidor.
- En ACC pueden ser especificadas por el consumidor de forma explícita, aunque se pueden derivar del código de forma automática.
- En el proyecto MRG se establece de antemano la política de consumo de memoria para el "heap"de forma general para todo el código que se compile en Camelot.

2.3.7. Formalismos y Tecnologías

En los primeros trabajos de PCC se usa la lógica de predicados de primer orden extendida con funciones para expresar las políticas de seguridad, y las

precondiciones sobre el uso del código, como reglas de tipado y para representar la máquina abstracta y expresar las reglas de derivación de los tipos [Necula and Lee, 1996; Necula and Lee, 1997b; Necula, 1997; Necula and Lee, 1998a; Necula and Lee, 1998b], aunque se usan tecnologías basadas en la lógica de orden superior para la generación y representación de las demostraciones como ELF.

En [Appel, 2001; Appel and Mcallester, 2001; Felty, 2005] FPCC usa la lógica de orden superior con algunos axiomas de la aritmética para expresar la política de seguridad del consumidor, la semántica operacional de las instrucciones de la máquina, así como para la definición de todos los conceptos que se requieran desde los fundamentos de las matemáticas y para las respectivas demostraciones de las propiedades necesarias de tales conceptos. En ProtoPCC [Wildmoser et al., 2004; Wildmoser et al., 2005] usan la lógica de orden superior para expresar las políticas y las demostraciones, con el soporte del demostrador de teoremas Isabelle. Para expresar las propiedades de seguridad se usa por lo general la lógica de orden superior soportada por las herramientas utilizadas, aunque varias de las propiedades consideradas solo requieren de la lógica de predicados con funciones (FPCC, SFPCC, MRG, ProtoPCC).

En SFPCC [Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005] y en [Besson et al., 2005; Besson et al., 2006] se emplea CiC (cálculo de Construcciones Inductivas) soportado por el demostrador de teoremas Coq para expresar las políticas de seguridad, la semántica del lenguaje y las demostraciones. CiC es una extensión del CC (cálculo de Construcciones), el cual es un cálculo lambda de orden superior. En [Chang et al., 2005] se usa Coq para la generación de fórmulas invariantes por parte de las extensiones que provee el productor del código. En [Besson et al., 2007] se usa Coq para validar las demostraciones generadas con un programa en Caml producido por el generador de programas de Coq.

En ACC se utiliza el lenguaje de aserciones de Ciao con pre y postcondiciones que permite expresar políticas de seguridad de alto nivel como la apertura segura de archivos en directorio temporal, la terminación, el determinismo, el no tropiezo y no fallo, el tiempo de proceso y de memoria para estructuras de datos acotados, y la ausencia de efectos secundarios en código. Son numerosos los trabajos que aplican sistemas de tipos en los que la especificación de los tipos incluye las

propiedades de seguridad requeridas: PCC, FPCC, SFPCC, TAL, TILT, MRG, y MOBIUS. En estos trabajos, la derivación y comprobación de los tipos constituye la verificación del cumplimiento de las propiedades. Entre los trabajos están los que tratan código de diferentes lenguajes ensambladores, código para JVM ("bytecode"), y los que compilan código de alto nivel con los compiladores certificantes (C, Java, ML, Camelot). Se utilizan los algoritmos de derivación y comprobación de tipos de herramientas como, i) los marcos lógicos formales (ELF, TWELF), y ii) los demostradores de teoremas (Isabelle, Coq, PVS). Los compiladores certificantes además usan tipos con anotaciones y otras técnicas de compiladores como lenguajes intermedios tipados, y lenguajes objeto también tipados. En [Harren and Necula, 2005; Harren, 2007] se usan tipos dependientes estáticos y dinámicos para certificar código ensamblador.

Cinco propuestas utilizan la interpretación abstracta para generar el certificado, i) ACCEPT/C que usa BLAST para C [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004], ii) ACC con CiaoPP, un preprocesador para Ciao, iii) ProtoPCC, iv) [Besson et al., 2005; Besson et al., 2006] con el demostrador de teoremas Coq, y v) [Besson et al., 2007] que aplica interpretación abstracta poliédrica.

Otras propuestas usan interpretación abstracta para partes de la verificación como [Harren and Necula, 2005; Harren, 2007] en la que el verificador usa interpretación abstracta sobre el dominio de las expresiones simbólicas para inferir los tipos de los registros para memoria segura. [Chang et al., 2006] usan la interpretación abstracta para obtener los teoremas que deben ser demostrados para cada analizador que sea certificado.

Para representar la demostración/certificado de las propiedades de seguridad, se utiliza la sintaxis de la herramienta utilizada. Por ejemplo, los trabajos que usan implementaciones de LF ("Logical Framework") como ELF y TWELF, codifican los certificados como expresiones LF. ELF, TWELF, Isabelle y PVS utilizan la lógica de orden superior HOL. Los trabajos que usan Coq que se basa en el cálculo de construcciones inductivas CiC, codifican el certificado como un término del cálculo lambda con tipos lambda. En el caso de PVS que funciona "top-down"y se basa en un sistema de tipos fuerte y rico, el certificado son las tácticas y sus parámetros.

Para llevar a cabo la demostración de las propiedades que satisface el código en cada una de sus instrucciones, en la mayoría de trabajos en los que certifican código imperativo de bajo y alto nivel, se utiliza la lógica de Hoare (pre, postcondiciones, invariantes) para generar las condiciones de verificación del código, instrucción por instrucción así como también la postcondición que satisface el código y el predicado de seguridad, cuya satisfacción se demuestra. Desde los trabajos iniciales en PCC [Necula and Lee, 1996; Necula and Lee, 1997b; Necula, 1997] así como en trabajos posteriores como algunos de los compiladores certificantes [Necula and Lee, 1998a; Necula and Lee, 1998b; Colby et al., 2000; Necula, 2001b], el lenguaje de ensamble con tipos TAL y los compiladores certificantes asociados [Morrisett et al., 1999b; Morrisett et al., 1999a; Grossman and Morrisett, 2001; Jim et al., 2002], los que suponen las reglas de inferencia como no fiables [Necula and Schneck, 2002; Necula and Schneck, 2003a], los trabajos de OPCC [Necula and Rahul, 2001; Necula, 2001a], el marco formal TINMAN [Mok and Yu, 2002a; Yu and Mok, 2004], el proyecto MRG [Hofmann and Jost, 2003; MacKenzie and Wolverson, 2004; Beringer et al., 2003; Aspinall et al., 2004b] todos utilizan la lógica de verificación de Hoare, el JVer [Chander et al., 2005b; Chander et al., 2005a; Chander et al., 2007]. TINMAN usa una versión extendida de dicha lógica para considerar el consumo de recursos del código en lenguaje C.

2.3.8. Requerimientos de PCC

En este contexto, según los trabajos revisados, una infraestructura de PCC adecuada debería caracterizarse por:

- La flexibilidad y generalidad en cuanto a las propiedades de seguridad [Albert et al., 2005a; Wildmoser et al., 2004] que se pueden certificar y los lenguajes de programación en que se puede certificar código [Appel, 2001; Appel and Felten, 2001; Mok and Yu, 2002a; Necula and Schneck, 2003b; Hamid, 2005],

- La escalabilidad del marco desde el punto de vista del tamaño y complejidad del código que se certifica [Necula and Rahul, 2001; Necula, 2001a; Necula and Schneck, 2002; Necula and Schneck, 2003b],

- El contenido y tamaño reducido de la TCB [Appel, 2001; Necula and Schneck, 2003a; Necula and Schneck, 2003b; Appel et al., 2003; Wildmoser et al., 2004],
- El tamaño y tiempo de validación del certificado, mínimos [Necula and Lee, 1997a; Necula and Rahul, 2001; Wu et al., 2003; Albert et al., 2006].

2.4. Panorámica de Trabajos sobre código Móvil

A continuación se resume de forma esquemática las propuestas revisadas, destacando las políticas de seguridad, el lenguaje de programación cuyo código se verifica, las asunciones sobre componentes de la infraestructura y actores fiables, el formalismo empleado, el contenido y forma de la prueba, la tecnología empleada y los trabajos correspondientes.

Las propuestas están agrupadas por paradigma del lenguaje, imperativos de bajo nivel, imperativo de alto nivel C, imperativo orientado a objetos Java, funcionales, y por último los lógicos.

Cuadro 2.1: Lenguajes Imperativos de bajo nivel.

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador	Manejo seguro de la Memoria	No fiables: El Código Fiables: VCGen Validador Reglas de inferencia	Lógica de Hoare Lógica de Orden Superior Comprobación de Tipos	Una expresión en LF (la derivación de un tipo)	ELF, Generador de VCs	[Necula and Lee, 1996; Necula and Lee, 1997b]
Ensamblador como extensión de lenguaje ML	Manejo seguro de la memoria y de los Tipos ML				ELF, Generador de VCs, Comprobador de Tipos LF	[Necula, 1997]
Ensamblador como compilación de lenguaje C	Manejo seguro de la memoria y de los Tipos C				Compilador Certificante (Generador de VCs y Certificador), Comprobador de Tipos LF	[Necula and Lee, 1998a; Necula and Lee, 1998b]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador	Acceso seguro de la memoria Petición y liberación de memoria y candados con topes en la disponibilidad	No fiable: Código Fiables: VCGen Validador Reglas de inferencia	Lógica de Hoare Lógica de Orden Superior Comprobación de Tipos	Una expresión en LF (la derivación de un tipo)	ELF, Generador de VCs, Comprobador de Tipos LF	[Lee and Necula, 1997]
Ensamblador	Manejo seguro de la memoria y acceso a archivos con autorización Topes en uso de recursos como ciclos de CPU y ancho de banda				ELF Generador de VCs Comprobador de Tipos LF	[Necula and Lee, 1998b]
Ensamblador resultado de compilación de fuentes en bytecode para un subconjunto de Java Simple J, y Compilador TouchStone	Manejo seguro de la memoria y de los Tipos en Java				Compilador Certificante (incluye Generador de VCs y Certificador), Comprobador de Tipos LF	[Colby et al., 2000; Necula, 2001b]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador resultado del compilador Simple J, un subconjunto de Java	Manejo seguro de la memoria y de los Tipos en Java	No fiables: Código Reglas de inferencia Fiable: Validador	Lógica de Hoare Lógica de Orden Superior Comprobación de Tipos	Término del cálculo lambda con tipos (la derivación de un tipo)	Compilador Certificante (incluye Generador de VCs y Certificador), Demostrador Coq	[Necula and Schneck, 2002; Necula and Schneck, 2003a]
Ensamblador tipado TAL como resultado de compilación de Popcorn, Cyclone o Scheme	Manejo seguro de la memoria y de los Tipos	No fiables: Código Compilador Fiable: Validador	Lógica de Hoare Comprobación de Tipos	Anotaciones de Tipado en las etiquetas del Código	Compilador Certificante (incluye Generador de VCs y Certificador), Comprobador de tipos	[Morrisett et al., 1999b; Morrisett et al., 1999a; Grossman and Morrisett, 2001; Jim et al., 2002]
Ensamblador e intermedio FLINT, producto de compilación en ML	Manejo seguro de tipos	No fiable: Código Fiable: Compilador	Cálculo lambda tipado Sistemas de tipos		Compilador Certificante FLINT/ML	[Shao and Appel, 1995; Shao, 1997]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador de la máquina DEC / ALPHA resultante de compilación de ML	Manejo seguro de los Tipos y de vectores mono y bidimensionales	No fiable: Código Fiable: Compilador	Comprobación de Tipos	Anotaciones de Tipado en las variables del Código	Compilador Certificante TIL	[Tarditi et al., 1996]
Ensamblador de una máquina teórica Ensambladores como Sparc y Pentium	Manejo seguro de la memoria. Salidas periódicas de corrutinas Asignación / desasignación de memoria Secuencias seguras de invocaciones de APIs Límites en el número de candados Registro de Salidas Reenvío de paquetes sin modificación	No fiables: Código Prueba del código y prueba de la prueba Fiables: Validador fundamental Semántica de la máquina Política de seguridad	Lógica de Orden superior Lógica de verificación de Hoare Comprobación de Tipos Unificación	Una expresión en LF que representa la derivación de un tipo	TWELF (una implementación de LF) Lambda prolog para generar la prueba FLIT para validarla usando unificación sin retroceso	[Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001; Felty, 2005]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador como producto de bytecodes de JVM	Tipos seguros de Java	No fiable: Código Fiables: Reglas de inferencia Validador	Lógica de Hoare Unificación en lógica de orden superior Sistemas de Tipos	El oráculo que es un testigo de la prueba (un árbol de la derivación del tipo)	Validador implementado como intérprete de programación lógica	[Necula and Rahul, 2001; Necula, 2001a]
FTAL lenguaje ensamblador con tipos XTAL lenguaje ensamblador con tipos extendido Lenguaje ensamblador certificado CAP	Manejo seguro de la memoria y de los Tipos	No fiables: Código Prueba del código y prueba de la prueba Fiables: Validador fundamental Semántica de la máquina Política de seguridad	Cálculo de Construcciones inductivas CiC (extensión de CC) Lógica de Hoare	la derivación de los tipos del programa	Compilador de FTAL Demostrador de Teoremas Coq	[Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005; Ni and Shao, 2006]
Ensamblador	Generales, de bajo o alto nivel, especializables por el consumidor	No fiable: Código Fiables: Analizador estático Model checker	Autómatas extendidos EFSA Resolución tabular	Modelo del comportamiento en EFSA	Analizador estático, resolvedor de consistencia (XMC) y monitor de ejecución	[Sekar et al., 2001; Sekar et al., 2003]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Aplicable a lenguaje ensamblador y bytecode	Manejo seguro de memoria e instrucciones válidas	No fiables: Código Parte customized del VCGen Fiables: Validador Núcleo y decodificador del VCGen	Lógica con tipos simples			[Necula and Schneck, 2003b]
Bytecode Java	Topes en el consumo de recursos: Recursos físicos: CPU, memoria, disco, ancho de banda Recursos virtuales: hilos, "handles" Generales	No fiables: Código Verificador Fiables: Validador	Lógica de Hoare Demostración de teoremas	Una expresión en LF que representa la derivación de un tipo	Verificador JVer Demostrador de teoremas Kettle	[Chander et al., 2005b]
General de bajo nivel. Ejemplo en ensamblador para Alpha	Generales. Ejemplo con tipos seguros y arreglos seguros	No fiable: Código Fiables: Validador Probabilístico Interactivo Demostrador	Lógica de Hoare Falsación y Aritmetización de Fórmulas Booleanas Teoría de Números	Coeficientes de un polinomio de grado bajo	Demostrador Probabilístico Interactivo Validador Probabilístico Interactivo	[Tsukada, 2000; Tsukada, 2005]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
General de bajo nivel instanciable; ejemplo en ensamblador simple SAL y aplicación a bytecode generado por compilador de Jinja (subconjunto de Java)	Genéricas. Ejemplo con tipos seguros y operaciones aritméticas seguras (desborde)	No fiables: Código Reglas de inferencia Fiable: Generador de VCs	Lógica de Orden superior (HOL) Grafos dirigidos	Término Lambda tipado	Theorem Prover Isabelle/ HOL, Generador de VCs	[Wildmoser et al., 2004; Wildmoser et al., 2005; Wildmoser and Nipkow, 2005]
Cualquier lenguaje imperativo, implementado para bytecode	General, Ejemplos con manejo seguro de arreglos Variables enteras dentro de rangos	No fiables: Código Resolvedor de punto fijo Compresor de punto fijo Fiables: Comprobador de tipos Coq Generador de programas de Coq Compilador Caml Certificado: Validador	Interpretación abstracta Semántica de Punto fijo Comprobación de tipos	Término Lambda tipado	Demostrador de Teoremas Coq con el extractor de programas que genera programas Caml Compilador Caml	[Besson et al., 2005; Besson et al., 2006]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Aplicable a cualquier Ensamblador y Bytecode Ensamblador TALx86 Bytecode producido por compiladores Cool	Generales ejemplos con Tipos seguros y Memoria segura	No fiables: Código Extensiones del verificador para lenguajes específicos Fiables: Generador de la post condición más fuerte Validador Buscador de Punto fijo	Interpretación abstracta Semántica de Punto fijo Comprobación de tipos Lógica de Horn Cálculo de Construcciones inductivas CiC (extensión de CC)		Open Verifier: Post, Validador, Fix Point Extensiones (verificadores): PCCExt, TALExt, CoolAid	[Chang et al., 2005]
General: Ensamblador x86 producido por TALx86 Bytecode	Tipos seguros y memoria segura	No fiables: Código Analizador Certificado: Certificador Fiables: Extractor de especificaciones Compilador Ocaml Validador de demostraciones de Coq	Interpretación abstracta Semántica de punto fijo Demostración de teoremas		Analizador Certificador Extractor de especificaciones Compilador Ocaml Validador de demostraciones de Coq	[Chang et al., 2006]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ensamblador en general Ensamblador x86 resultante de compilación de código de un subconjunto de lenguaje C anotado por CCured o Cqual	Política de Tipos seguros de CCured: apuntadores y memoria segura Políticas de Tipos seguros generales	No fiables: Código Compilador C Herramienta de seguridad (CCured, Cqual) Fiable: Verificador	Sistema de Tipos dependientes Interpretación abstracta para inferencia de tipos	Anotaciones en el código	Herramienta CCured Herramienta Cqual Prototipo Comprobador de tipos	[Harren and Necula, 2005; Harren, 2007]
Byte code imperativo con arreglos procedimientos y variables globales	Basadas en invariantes de bucles y pre-/post condiciones de los métodos: Consumo de recursos Memoria segura Arreglos seguros	No fiables: Código Validador Fiable: Comprobador de tipos de Coq	Lógica de Hoare Interpretación abstracta relacional Semántica de punto fijo Ejecución simbólica Sistemas de restricciones lineales	Anotaciones con invariantes, pre y post condiciones, e inclusión polihédrica vector de coeficientes de combinación lineal de restricciones	Analizador relacional de bytecode implementado en Caml Validador de resultados Comprobador de tipos de Coq	[Besson et al., 2007]

Continúa en la página siguiente

Lenguajes Imperativos de bajo nivel

Continúa de la página anterior						
Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Bytecode con clases, objetos, arreglos y excepciones	No interferencia (Privacidad) basada en un retículo de niveles de confidencialidad de los datos Tienen niveles de seguridad: El atacante Los campos de las clases Los parámetros, variables locales y resultados de los métodos Las excepciones lanzables por los métodos	No fiable: Código Certificado: Validador Fiable: Comprobador de tipos de Coq	Sistemas de Tipos Flujos de Información	Anotaciones en el código Táctica para el demostrador	Analizadores (PA, CDR, IF) Validadores (PA, CDR, IF) Demostrador Coq	[Barthe et al., 2007]

Cuadro 2.2: Lenguaje Imperativo C

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
C	Corrección	No fiable: Código Fiables: Comprobador de modelos y demostrador de teoremas BLAST	Interpretación abstracta "on the fly" grafos CFA	Una expresión en LF	Comprobador de modelos y demostrador de teoremas BLAST No incluye Validador pero podría usarse el mismo de PCC	[Henzinger et al., 2002]
C	Consumo de recursos: de tiempo de ejecución, de memoria del "heap" demandada explícitamente y ancho de banda	No fiable: Código Fiables: Comprobadores fuera de línea y en línea Analizador de tráfico de la red	Lógica de Hoare extendida Interpretación abstracta Demostración táctica de teoremas	Tácticas de la demostración con PVS y parámetros de esas tácticas	Demostrador de teoremas táctico PVS Analizador estático TINMAN Analizador dinámico TINMAN	[Mok and Yu, 2002a; Mok and Yu, 2002b; Yu and Mok, 2004]

Continúa en la página siguiente

Lenguaje Imperativo C

Continúa de la página anterior

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
C	Propiedades temporales: Vivacidad (“liveness”)	No fiable: Código Fiables: Comprobador de tipos Comprobador de modelos abstractos	Sistema de Tipos catalogados (“indexed”) Interpretación abstracta Autómatas de control de flujo Comprobación de modelos LTL sin operador “next”	Programa booleano codificado como tipos catalogados	Compilador C a SDTAL ACCEPT/C Comprobador de tipos catalogados Comprobador de modelos abstractos	[Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004]

Cuadro 2.3: Lenguaje Imperativo Orientado a Objetos Java

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Java Anotado	Topes en el consumo de recursos: Recursos físicos: CPU, memoria, disco, ancho de banda Recursos virtuales: ficheros, conexiones a bases de datos, procesos (hilos)	No fiables: Código Demostrador (Prover) Fiable: Validador del Demostrador de Teoremas	Lógica de Hoare Cálculo precondición más débil de Dijkstra Inecuaciones lineales Demostración de teoremas Satisfacibilidad	Una demostración de satisfacibilidad	ESC/Java Demostrador de teoremas Simplify	[Chander et al., 2005a; Chander et al., 2007]

Cuadro 2.4: Lenguajes Funcionales

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Bytecode compilación del lenguaje funcional de bajo nivel Grail Grail compilación de lenguaje funcional Camelot	Consumo del "heap" para las funciones acotado de forma lineal en relación con el tamaño de la entrada	No fiable: Código Fiables: Funcionalizador de bytecode Demostrador Isabelle	Lógica de Hoare Lógica de orden superior Teoría de Tipos Extendidos LFD	Tipos LFD	Compilador certificante Camelot Compilador Grail Funcionalizador de bytecode Demostrador Isabelle	[Hofmann and Jost, 2003; Beringer et al., 2003; MacKenzie and Wolverson, 2004; Aspinall et al., 2004b; Gilmore and Prowse, 2005; MRG, 2005]

Cuadro 2.5: Lenguajes Lógicos

Lenguaje	Políticas de Seguridad	Asunciones	Formalismo	Prueba	Tecnología	Trabajos
Ciao (CLP)	Cualquiera expresadas en CLP, como propiedades de las variables o los procedimientos, como determinismo, terminación, no fallo, y no tropiezo: se presenta ejemplo con acceso seguro a archivos	No fiables: Código Certificado (prueba) Fiable: Validador	Interpretación abstracta Semántica de Punto fijo CLP	Resultado de la interpretación abstracta del programa	Preprocesador de Ciao CiaoPP Herramientas en Ciao para análisis de determinismo, terminación, no tropiezo y no fallo	[Albert et al., 2005a; Albert et al., 2004a]
	Uso de memoria para datos y funciones acotado, coste tiempo de funciones acotado, funciones sin efectos secundarios				Preprocesador de Ciao, CiaoPP	[Albert et al., 2004b; Albert et al., 2005b; Albert et al., 2006]

3

Preliminares

En este capítulo se introducen el lenguaje JML , la semántica de Java especificada en lógica de reescritura, y la interpretación abstracta, que son usadas en el presente trabajo.

Las políticas de seguridad son expresadas mediante un conjunto de requerimientos abstractos sobre las variables y los valores de las variables del programa en código fuente Java , y están escritas usando la sintaxis del lenguaje JML (Java Modeling Language).

3.1. El lenguaje de especificación JML

El lenguaje JML (Java Modeling Language) es un lenguaje de especificación de interfaces y comportamiento de módulos Java (clases e interfaces) [Leavens et al., 2006], que acepta los operadores Java de forma que evita a los programadores Java la complicación de aprender un lenguaje general de especificación de software como OCL ("Object Constraint Language") [Burdy et al., 2005]. Los desarrolladores Java pueden especificar con JML las propiedades funcionales de sus programas en una generalización de la lógica de Hoare adaptada al lenguaje de programación. JML fué diseñado como un lenguaje acsequible que combina el diseño por contrato y el enfoque basado en modelos para la especificación. Como lenguaje de especificación de interfaces, JML puede describir los nombres y toda la información estática de las declaraciones de los módulos Java con precondiciones (cláusulas `requires`), postcondiciones normales (cláusulas `ensures`), invariantes (cláusulas `invariant`) y postcondiciones excepcionales (cláusulas `signals`) que son expresiones lógicas de primer orden. Como lenguaje de especificación del

comportamiento puede además describir cómo se comportan los módulos Java mediante aserciones (cláusulas `assert`) mezcladas con el código de los métodos. La notación JML incluye cuantificadores existenciales (`\exists`) y universales (`\forall`) así como campos, métodos y clases propios de la especificación que permiten especificaciones más precisas y completas. JML viene con una biblioteca de tipos Java que pueden usarse para describir comportamientos matemáticamente usando conjuntos, relaciones y secuencias, entre otros. En JML se pueden escribir especificaciones detalladas y completas usando una palabra clave de comportamiento (como `normal_ behavior`), o especificaciones parciales y no detalladas sin usar palabras clave de comportamiento. Las especificaciones completas son llamadas especificaciones pesadas y las parciales livianas. JML es un lenguaje producto de la investigación, diseñado para ser usado por diferentes herramientas, y también es usado en la investigación, por lo cual evoluciona permanentemente. Tal evolución significa que algunas características no están documentadas completamente ni están implementadas en todas las herramientas. Todo esto potencialmente puede dificultar la portabilidad y hacer que JML sea difícil de aprender y usar. Los grupos de investigación que trabajan en JML están comprometidos a hacer que estos problemas sean transparentes para los no-investigadores como sea posible, por lo cual se han definido diferentes niveles del lenguaje. Hay seis niveles, desde el 0 que sorportan todas las herramientas JML hasta el nivel X que incluye características experimentales. Las especificaciones JML pueden escribirse como anotaciones del código de los programas Java o en ficheros aparte. Las anotaciones JML del código son manejadas como los comentarios Java que son ignorados por el compilador, pero que son automáticamente manejadas por nuestra técnica de certificación. El texto de una anotación puede estar en una única línea después de la marca `//@`, o en varias líneas delimitadas entre las marcas `/*@` y `@*/`.

```
/*@         requires       <pre-condition>;
  @         ensures       <normal post-condition - si no ocurre una exception>;
  @         signals(E)    <post-condition cuando la  exception E es lanzada>;
  @         assert          <predicado >;
  @         assignable    <variables y campos modificables>;
  @         invariant        <asertion sobre el estado de un objeto>;    @*/
```

En este trabajo se consideran solamente especificaciones parciales o livianas, y se usan las cláusulas más simples del nivel 0 del lenguaje JML para especificar

los métodos y tipos Java . Se usan las cláusulas de especificación de interfaz de métodos, `requires` y `ensures`, para especificar las pre y postcondiciones normales que deben cumplir las entradas y salidas de los métodos, y la cláusula de especificación de comportamiento `assert` para establecer condiciones sobre las variables locales. Las condiciones especificadas en dichas cláusulas JML deben estar de acuerdo con la propiedad de seguridad que se quiere certificar en el código. Las especificaciones estándar JML básicamente establecen condiciones sobre los valores de las variables. En el presente trabajo se usan las especificaciones JML no solo para establecer condiciones sobre los valores de las variables, sino también para establecer condiciones sobre las variables en sí. En éste sentido se usa la sintaxis estándar de JML nivel 0 pero no su semántica.

3.2. Maude y la Lógica de Reescritura

Maude es un lenguaje de programación declarativo de alto nivel que permite escribir especificaciones ejecutables en lógica de reescritura. Maude implementa la computación en lógica ecuacional y lógica de reescritura ya que la lógica de reescritura incluye como sublógica la lógica ecuacional [Clavel et al., 2002; Clavel et al., 2005; Clavel et al., 2007]. Un programa Maude es una teoría en lógica de reescritura que posiblemente incluye una subteoría ecuacional. Una computación Maude es una deducción lógica que usa los axiomas especificados en la teoría, es decir, las reglas de reescritura de la teoría de reescritura y las ecuaciones de la subteoría ecuacional.

Maude tiene módulos funcionales y de sistema. Los módulos funcionales son teorías en la lógica ecuacional de pertenencia, una lógica de Horn cuyas expresiones atómicas son igualdades $t = t'$ y aserciones de pertenencia de la forma $t : s$, que establecen que un término t es de un género ("sort") s. Esta lógica extiende la lógica ecuacional de géneros ordenados y soporta géneros, subgéneros, relaciones de géneros, sobrecarga polimórfica subgenérica de operadores y la definición de funciones parciales con dominios definidos ecuacionalmente. Los módulos funcionales son teorías ecuacionales $(\Sigma, E \cup A)$, donde Σ es una signatura (que especifica géneros, subgéneros, especies -"kinds"- y operadores) y E es un conjunto de ecuaciones con sus atributos ecuacionales A. Los atributos pueden

ser cualquier combinación de asociatividad, conmutatividad e idempotencia. Los módulos funcionales Maude se suponen confluentes, terminantes y decrecientes en los géneros. La computación en los módulos funcionales es la deducción ecuacional realizada mediante la reescritura, usando las ecuaciones como reglas de reescritura hasta obtener una forma canónica. Las ecuaciones son orientadas de su parte izquierda hacia su parte derecha. Esta reescritura ecuacional es realizada módulo los axiomas de asociatividad, conmutatividad e identidad que corresponden a los atributos de las ecuaciones. Las ecuaciones y aserciones de pertenencia de Maude pueden ser normales o pueden tener condiciones constituidas por conjunciones de ecuaciones individuales y aserciones de pertenencia. Las ecuaciones de las condiciones tienen dos variantes, ecuaciones normales $t = t'$ y ecuaciones de ajuste ("matching") $t := t'$ [Clavel et al., 2002; Clavel et al., 2007].

Los módulos de sistema de Maude son teorías en la lógica de reescritura (T, R) que incluyen una teoría ecuacional T y una colección de reglas de reescritura posiblemente etiquetadas y condicionales R. Las reglas de reescritura condicionales pueden tener ecuaciones, aserciones de pertenencia, reglas de reescritura o expresiones booleanas en sus condiciones [Clavel et al., 2003; Clavel et al., 2007]. Las reglas de reescritura $r : t \rightarrow t'$ no son ecuaciones. Las reglas de reescritura son interpretadas lógicamente como reglas de inferencia de un sistema lógico, y son interpretadas computacionalmente como reglas de transiciones locales de un sistema posiblemente concurrente. La reescritura en (T, R) ocurre módulo los axiomas ecuacionales de T con todas las combinaciones de asociatividad, conmutatividad e identidad que corresponden a los atributos de las ecuaciones. Las reglas en R no tienen que ser confluentes ni terminantes, pero sí deben ser coherentes con respecto de las ecuaciones. Así, dado un conjunto de reglas, varios caminos de reescritura diferentes son posibles desde un término.

En la siguiente ecuación el operador de unión multiconjunto `__` (denota la cadena vacía como símbolo del operador) tiene los atributos de asociatividad y conmutatividad:

```
op __ : State State -> State [comm assoc]
```

Estos atributos significan que el operador de unión multiconjunto satisface las siguientes ecuaciones de asociatividad y conmutatividad sin la necesidad de especificarlas explícitamente [Clavel et al., 2005]:

```
X ( Y Z ) = ( X Y ) Z
X Y = Y X
```

En este caso Maude utiliza un algoritmo de ajuste ("matching") multiconjunto en el que la unión multiconjunto es ajustada ("matched") módulo la asociatividad y conmutatividad. La reescritura de términos del multiconjunto es denominada reescritura multiconjunto. En los módulos funcionales Maude se pueden evaluar expresiones con el comando `reduce` (`red`), que simplifica la expresión, un término, a su forma canónica usando las ecuaciones y los axiomas de pertenencia del módulo. En los módulos de sistema Maude se pueden ejecutar teorías de reescritura con el comando `rewrite` (`rew`), que aplica las reglas a un término hasta la terminación, de una forma descendente ("top-down") y dándole opción a todas las reglas. Sin embargo, antes de aplicar una regla al término, este es reducido a su forma canónica usando las ecuaciones de la subteoría. Debido a que puede haber computaciones no terminantes, el comando `rewrite` puede tener un parámetro numérico que indica el máximo número de pasos de reescritura. Como las reglas de reescritura de la teoría podrían corresponder a un sistema de transición indeterminista, el comando `rewrite` sólo produce uno de los muchos posibles comportamientos.

Para obtener todos los posibles comportamientos a partir de un estado inicial se puede usar el comando `search`, que es la facilidad que ofrece Maude para hacer búsquedas del primero-en-amplitud con detección de ciclos. El comando `search` busca entre todas las posibles reescrituras de un estado dado, aquellas que cumplen con un patrón dado y que satisfacen una condición opcional. Cuando el comando `search` termina, Maude tiene almacenado en la memoria el grafo de estados con todas las transiciones realizadas, de manera que es posible interrogarlo por el camino desde el estado inicial (un término) hasta cualquier estado alcanzable [Clavel et al., 2003; Clavel et al., 2007]. Este camino incluye los estados recorridos y las reglas empleadas por las transiciones, y puede ser usado como testigo o certificado de la deducción en lógica de reescritura.

Otra característica mejorada de Maude aún no usada en el presente trabajo pero con mucho potencial en la generación y validación de certificados, es el metanivel, que se fundamenta en el hecho de que la lógica de reescritura es reflectiva. Una lógica reflectiva es una lógica que puede ser representada con exactitud

en ella misma.

3.3. Semántica de Java en Lógica de Reescritura

A continuación se describe la semántica operacional de Java en lógica de reescritura escrita en el lenguaje Maude, dada en [Farzan et al., 2007] y empleada por la herramienta de verificación JavaFAN [Farzan et al., 2004a; Farzan et al., 2004b]. Su novedad e interés se basa en las siguientes cuatro ventajas:

1. Las especificaciones formales dan una definición semántica rigurosa de un lenguaje que puede ser revisada matemáticamente.

2. Tales especificaciones semánticas formales pueden ser desarrolladas con, relativamente, poco esfuerzo aún para lenguajes complejos como Java [Farzan et al., 2004a] y el lenguaje de la máquina virtual de Java JVM (bytecode) [Farzan et al., 2004b].

3. Las herramientas de análisis formal del lenguaje de programación Maude (tales como la búsqueda en amplitud en el espacio de estados y la comprobación de modelos con lógica lineal temporal LTL) están disponibles sin coste para cada lenguaje de programación que sea especificado en Maude.

4. A pesar de su generalidad esos análisis formales pueden llevarse a cabo con eficiencia competitiva (ver [Farzan et al., 2004a]).

La especificación de la semántica operacional de Java es una teoría de reescritura, es decir, una tripleta $\mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}})$, con Σ_{Java} una signatura de géneros ordenados, $E_{\text{Java}} = \Delta_{\text{Java}} \uplus B_{\text{Java}}$ un conjunto de axiomas ecuacionales de Σ_{Java}, donde B_{Java} son axiomas tales como asociatividad, conmutatividad e identidad, y Δ_{Java} es el conjunto de ecuaciones terminante y confluente (módulo B_{Java}) de Σ_{Java}, y R_{Java} el conjunto de reglas de reescritura de Σ_{Java}. De manera intuitiva los géneros y símbolos de función de Σ_{Java} describen la estructura estática del espacio de estados del programa Java como un tipo algebraico de datos, las ecuaciones de Δ_{Java} describen las operaciones semánticas de sus características deterministas, y las reglas de R_{Java} describen sus características concurrentes. Siguiendo el marco formal de la lógica de reescritura [TeReSe, 2003; Meseguer,

1992], se denota con $u \rightarrow^{r}_{\text{Java}} v$ qué términos concretos u, v, que denotan estados del programa Java, son reescritos (en la posición "top", ver [Farzan et al., 2007]) usando r, que es o una regla de R_{Java} o una ecuación de Δ_{Java} aplicadas ambas módulo B_{Java}. Se escribe simplemente $u \rightarrow_{\text{Java}} v$ cuando no haya confusión. Se denota con $\rightarrow^{*}_{\text{Java}}$ la extensión de $\rightarrow_{\text{Java}}$ a varios pasos de reescritura, es decir, $u \rightarrow^{*}_{\text{Java}} v$ si existe $u_1, \ldots, u_k$ tal que $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$. Los axiomas de asociatividad, conmutatividad y unidad (escritos ACU) de las operaciones binarias de B_{Java} permiten definir (e implementar implícitamente) de forma efectiva y elegante la infraestructura del lenguaje de programación Java, incluyendo los ambientes, los hilos, la memoria, la entrada y salida, los candados para sincronización, la actualización y el acceso de variables asi como sus características orientadas a objetos; todo esto implementado con operaciones de unión de multiconjuntos que construyen una "sopa"de elementos; véase [Farzan et al., 2007]. La teoría de reescritura $\mathcal{R}_{\text{Java}}$ está definida como términos del género concreto JavaState, con los principales atributos del estado del programa, que son los constructores del tipo algebraico JavaState, como `store`, `in` o `out`, para la memoria, la entrada y la salida respectivamente. Intuitivamente las ecuaciones de Δ_{Java} y las reglas de R_{Java} son usadas para especificar los cambios de estado del programa, o sea, los cambios en la memoria, los hilos, la entrada/salida, etc.

La especificación formal de Java usa un "front-end"de dominio específico para la lógica de reescritura que consiste de la notación y la técnica `K` [Rosu, 2005]. `K` es un marco formal algebraico para la definición de lenguajes de programación que hace uso intensivo del ajuste ("matching") módulo asociatividad, conmutatividad e identidad. Las definiciones `K` son reglas contextuales que pueden ser consideradas como reglas de reescritura de la forma $C[t_1, ..., t_n] \rightarrow C[t'_1, ..., t'_n]$ donde C es el contexto, y $t_1, ..., t_n$ son los subtérminos que pueden ser reemplazados en paralelo con los subtérminos $t'_1, ..., t'_n$. `K` se basa en una representación de primer orden de las continuaciones. Las continuaciones hacen explícito el control de ejecución de un programa concurrente manteniendo el contexto de control de cada hilo, al especificar explícitamente los siguientes pasos a ejecutar por los hilos. La técnica de las continuaciones es típica para transformar un contexto de control incontrolable en un contexto de datos controlable, mediante la apilación de las acciones que quedan pendientes de ejecutar. La técnica `K` representa las continuaciones

como listas asociativas de tareas separadas por una flecha de izquierda a derecha ($\curvearrowright$) y con un elemento identidad de las listas y conjuntos. Una vez se evalúa la expresión e en el tope de la continuación ($e \curvearrowright k$), su resultado es pasado al resto de la continuación k. Las continuaciones facilitan de forma significativa la definición de las instrucciones de flujo de control tales como las invocaciones de métodos, `break`, `continue`, `return` y el manejo de las excepciones. `K` permite definiciones de las características de los lenguajes intuitivas y modulares, que son además fáciles de entender, leer y modificar. `K` ha sido usada para definir lenguajes funcionales, dos lenguajes orientados a objetos, KOOL y Java [Chen et al., 2006], y el lenguaje de la máquina virtual de Java (JVM bytecode) [Farzan et al., 2004b].

En [Farzan et al., 2007] está especificado en Maude un subconjunto del lenguaje Java versión 1.4, lo suficientemente grande para incluir multihilos, herencia, polimorfismo, referencias a objetos y asignación dinámica de objetos. Sin embargo, los métodos nativos de Java y la mayoría de su biblioteca de clases no está soportada. Algunos métodos para entrada-salida de las clases públicas del sistema, como los métodos `system.out.println` y `system.out.print` sí están definidos directamente en la especificación Java. La semántica de Java está definida modularmente de manera que diferentes características del lenguaje están especificadas en módulos separados, para así facilitar las extensiones y el mantenimiento. Para más detalles véase [Farzan et al., 2007].

Ejemplo 1. Para poder interpretar un programa Java hay que transformarlo previamente en una forma adecuada para la técnica empleada. Para ello se aplica el programa "JavaWrapper"(también disponible en [FSL, 2006]) que hace un envoltorio con el programa Java y lo convierte en el término Maude de un comando `rew` que puede ser almacenado en un fichero. El fichero contiene también la operación modular para incluir los ficheros con la especificación de la semántica. Para suministrar los datos de entrada del programa Java estos se deben introducir como parámetros adicionales al "JavaWrapper".

A continuación se muestran un programa Java:

```
class Safe1Even2       {
        public static void main(String[] args)   {
                  System.out.println(evenOdd(0))
        }
```

```
        static int evenOdd(int j)       {
                int u = 3;
                int v =4,z = 4;
                z += 30;
                v = u*8 + j;
                return z - v;
        }
}
```

y el contenido del fichero Maude con el programa Java en su envoltura:

```
in java-rl-spec/java-rl-infint.maude
rew java(___(preprocess(__(noClass,_class_imports_extends_implements
__(default,t('Safe1Even2), nil,Object,none, '{_'}(__(__(noMember,___'(_')__(__(public,static)
,void,'main,__(_'['](t('String)),d('args)), throws(noType),'{_'}(_@_(4,_;(_._(_._('System,'out),
_<_>('println,_<_>('evenodd,i(0))))))))) ,___'(_')__(__(static,default), int,'evenOdd,__(int,
d('j)),throws(noType),'{_'}(__(__(__(__(_;(__(int,_=_(d('u),i(3)))),_;(__(int,_',_(_=_(d('v),
i(4)) ,_=_(d('z),i(4))))))),_@_(12,_;(_+=_('z,i(30))))),
_@_(13,_;(_=_('v,_+_(_*_('u,i(8)),'j))))),_@_(14,return_;(_-_('z,'v))))))))))),_._(noType,
_<_>('main, new__(string, [ i(0) ]))), noVal) ) .
```

Tal como cualquier definición K, la especificación Java inicia con los operadores estructurales (constructores) que definen el estado de un programa Java mediante una subsignatura usada para construir las partes del estado (los atributos del estado). Cada atributo del estado tiene un género ("sort") y unas operaciones no interpretadas (constructoras) y el estado mismo es un multiconjunto asociativo y conmutativo de sus atributos. La infraestructura del estado es una estructura de dos niveles, como se muestra en la figura 3.1, que diferencia entre el estado global del programa (en la parte izquierda, arriba) y el estado local de los hilos (en la parte izquierda, abajo).

El estado global tiene nueve atributos empaquetados mediante operaciones constructoras (en la segunda columna, de izquierda a derecha): 1) `out` para la lista de valores de salida, 2) `in` para la lista de valores de entrada, 3) `t` para los estados locales de los hilos, 4) `store` para las localizaciones de memoria usadas y los valores almacenados allí, 5) `code` con el código del programa envuelto, 6) `static` para almacenar el ambiente de los objetos estáticos, 7) `busy` para los candados usados por todos los hilos, 7) `nextLoc` para el identificador de la última posición de memoria asignada, y 8) `nextTid` para el siguiente identificador de hilo disponible. Los géneros de estos atributos están en la columna de la derecha. Esta infraestructura permite identificar un componente del estado mediante la

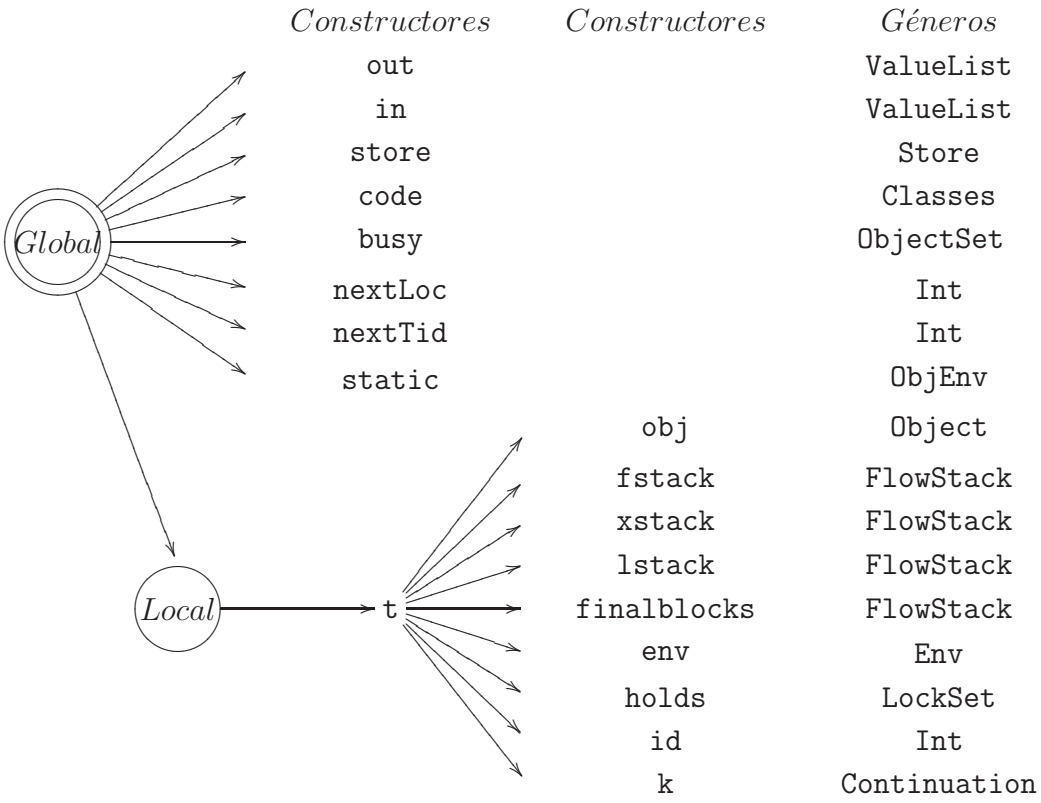

Figura 3.1: Estado del programa

mención del nombre del correspondiente atributo, lo cual contribuye a que la definición del lenguaje Java sea modular.

El código Maude de la Figura 3.2 muestra la especificación del estado global.

El atributo para control de los hilos del estado local es también un multiconjunto asociativo y conmutativo y tiene nueve partes o subatributos para cada hilo activo, cuyas operaciones constructoras (la tercera columna de la figura 3.1 de izquierda a derecha, en la parte inferior) son: 1) `id` con el único identificador del hilo, 2) `k` con las continuaciones que permiten controlar la interpretación del código, 3) `obj` con la descripción del objeto activo del hilo, 4) `fstack` con la pila para las funciones donde, cuando un método es invocado, se apila el atributo del hilo en ese momento de la invocación -exceptuando su subatributo `fstack`, 5) `xstack` con la pila para las excepciones que se usa cuando se lanza una excepción, 6) `lstack` con la pila para los bucles donde, cuando se entra a un bucle, se apila el atributo del hilo en ese momento de la invocación -exceptuando su subatributo

```
fmod STATE is
  ...
  sort JavaState .
  op out : Output -> JavaState . --- lista de valores de salida
  op in : ValueList -> JavaState . --- input values list
  op t : ThreadCtrl -> JavaState .  --- thread control local substate
  op store : Store -> JavaState . --- memory
  op code : Classes -> JavaState . --- wrapped Java source code
  op static : ObjEnv -> JavaState . --- the static fields of classes
  op busy : ObjectSet -> JavaState . --- the locks are in use
  op nextLoc : Int -> JavaState . --- the last allocated location
  op nextTid : Int -> JavaState . --- the next thread id
  op __ : JavaState JavaState -> JavaState [comm assoc] . --- the state AC multiset
endfm
```

Figura 3.2: Definición del multiconjunto AC del estado global `JavaState`

```
fmod THREADCTRL is
  ...
  sorts ThreadCtrl FlowStackItem FlowStack .
  op id : Int -> ThreadCtrl . --- the thread id
  op k : Continuation -> ThreadCtrl .
  op obj : Object -> ThreadCtrl .
  op fstack : FlowStack -> ThreadCtrl . --- function stack
  op xstack : FlowStack -> ThreadCtrl . --- exception stack
  op lstack : FlowStack -> ThreadCtrl . --- loop stack
  op finalblocks : FlowStack -> ThreadCtrl . --- final blocks
  op env : Env -> ThreadCtrl . ---  environment
  op holds : LockSet -> ThreadCtrl . --- the locks held by the thread
  op __ : ThreadCtrl ThreadCtrl -> ThreadCtrl [comm assoc] . --- Multiset AC
endfm
```

Figura 3.3: Definición del multiconjunto AC del estado local `ThreadCtrl`

`xstack`, 7) `finalblocks` con la pila de los bloques finales, 8) `env` con el ambiente del hilo que mapea las variables no estáticas en sus posiciones de memoria, y 9) `holds` para los seguros mantenidos por el hilo. Este estado local para el control de los hilos tiene las tres pilas mencionadas (`fstack`,`xstack`, `lstack`) para modularizar y hacer eficiente los cambios del flujo de control con las funciones, las excepciones y los bucles. La especificación Maude que describe el estado local está en la Figura 3.3.

El estado inicial (global y local) del programa Java del ejemplo 1 es el mostrado abajo en la figura 3.4:

Nótese en el estado global inicial del programa que: 1) el atributo `out(noVal)` indica que la lista con los valores de salida esta vacía, 2) al igual que la lista de valores de entrada `in(noVal)` como corresponde al programa del ejemplo 1, 3) el

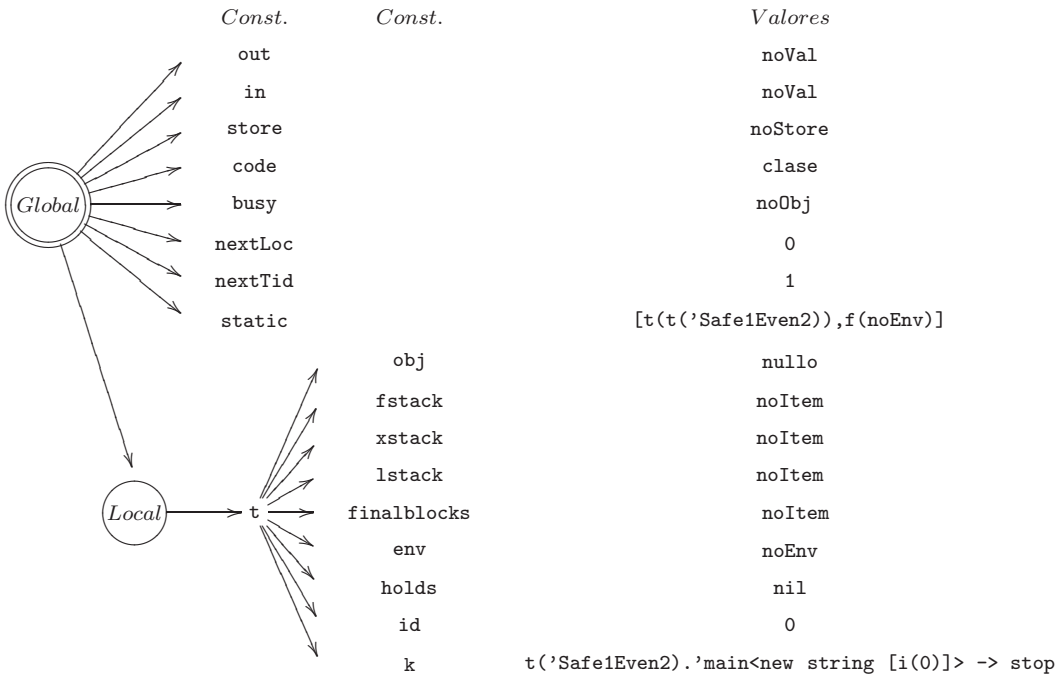

Figura 3.4: Estado inicial del programa del Ejemplo 1

atributo de la memoria `store(noStore)` significa que está vacía pues aún no ha sido asignada ninguna posición para almacenar algún valor, 4) el atributo con el código del programa `code` tiene el código envuelto y preprocesado que corresponde al ejemplo 1 [1], 5) el atributo estático `static([t(t('Safe1Even2)),f(noEnv)])` significa que está creado el objeto estático que corresponde a la clase principal (`Safe1Even2`) del método `main`, 6) el atributo con los candados `busy(noObj)` indica que aún no ha sido creado ninguno, 7) el atributo `nextLoc(0)` señala que ninguna posición de memoria ha sido asignada, lo cual es consistente con el estado vacío de la memoria, y 8) el atributo con el identificador de hilo disponible `nextTid(1)` indica que el siguiente hilo que sea creado tendrá como identificador el entero `1`. Esto último también significa que solo ha sido creado el hilo principal identificado con el número `0`, lo cual es consistente con el valor del atributo del estado local `id`.

El atributo estado local inicial del programa del ejemplo 1 (el estado del hilo básico), tiene los siguientes subatributos iniciales (mostrados en la parte inferior de la figura 3.4): 1) el subatributo `obj(nullo)` tiene como valor el que corre-

[1] el valor se muestra en la Figura 3.5

```
out(noOutput)                  in(noVal)                        store(noStore)
code(default class t(
    'Safe1Even2) extends Object implements none {(default static) int 'evenOdd(
    int d('j))throws(noType) {(int d('u) = i(3) ;) (int d('v) = i(4),d('z) = i(
    4) ;) (12 @ ('z += i(30) ;)) (13 @ ('v = 'u * i(8) + 'j ;)) 14 @ return 'z
    - 'v ;} (public static) void 'main(string[] d('args))throws(noType) {4 @ (
    'System . 'out . 'println < 'evenOdd < i(0) > > ;)} public t('Safe1Even2)(
    noPara)throws(noType) super(noExp){nop}})
busy(noObj)                     nextLoc(0)                    nextTid(1))
static(onil [t(t('Safe1Even2)),f(noEnv)]))
t((  obj(nullo)                     fstack(noItem)              xstack(noItem)
     lstack(noItem)               finalblocks( noItem)      env(noEnv)
     holds(nil))                    id(0)
     k(t('Safe1Even2) . 'main < new string [i(0)] > -> stop))
```

Figura 3.5: Término Maude con el estado inicial del programa del Ejemplo 1

sponde a un objeto nulo lo cual indica que no ha sido creado ninguna instancia (objeto) de ninguna clase que se esté ejecutando en el hilo, 2) todos los subatributos con las pilas de funciones, bucles y excepciones tienen el valor `noItem` que significa que están inicialmente vacíos, al igual que el subatributo `finalblocks`, 3) también están vacíos los subatributos del ambiente `env(noEnv)` y los candados `holds(nil)`, pues no ha sido instanciada variable ni candado alguno, 4) el subatributo `id(0)` significa que éste estado local corresponde al hilo inicial del programa, y finalmente, 5) la continuación `k(t('Safe1Even2).'main<new string [i(0)]> -> stop)` tiene solo dos tareas que se deben ejecutar en el orden señalado por la flecha, es decir, primero la invocación al método `main`, para luego detener el programa.

El término de la semántica Java correspondiente al estado inicial del programa del ejemplo 1 se muestra en la figura 3.5. Nótese en el atributo código (`code`) la representación interna de las constantes enteras `i(3)`, `i(4)`, `i(30)` y `i(8)` que usan el constructor `i`.

Ahora se describen los géneros de los principales atributos del estado así como las principales ecuaciones y reglas que especifican su comportamiento, empezando por la memoria y las variables del programa[2]

[2]La sintaxis Maude casi que se explica por sí misma. El punto central es que cada item, un género ("sort"), un subgénero ("subsort"), una operación, una ecuación ("equation"), una regla ("rule"), etc., está declarado con una palabra clave obvia: `sort`, `subsort`, `op`, `eq`, `rl`, etc., donde cada declaración está terminada con un espacio en blanco y un punto. Se usan letras mayúsculas para denotar las variables Maude y letras minúsculas para los símbolos constructores Maude.

```
fmod STORE is
  pr LOCATION .
  pr VALUE .
  sort Store .
  op noStore : -> Store . --- Empty store
  op [_,_,_] : Location Value Int -> Store .
--- the last bit is used to store the thread id,
  op __ : Store Store -> Store [assoc comm id: noStore] . --- multiconjunto ACU
endfm
```

Figura 3.6: Definición del multiconjunto ACU `Store` de la memoria

La memoria (`store`) para almacenar los valores de las variables es un multiconjunto ACU cuyos elementos son tripletas con las posiciones, los valores y los identificadores de los hilos a los que corresponden tales valores. Una tripleta `[L,V,Tid]` indica que en la posición `L` está almacenado el valor `V` del hilo cuyo identificador es `Tid`. La posición representa el lugar donde el valor es almacenado. Cada posición de memoria está identificada con la función `l` que asocia a cada posición un número natural positivo. El identificador de hilo `Tid` significa que ese hilo es el propietario del valor en la memoria. Cuando un nuevo valor puede ser alcanzado desde otro valor preexistente, la propiedad del valor preexistente tiene que ser propagada al nuevo. Hay dos identificadores de hilo especiales, i) `-1` para decir que la posición de memoria está compartida, y ii) `-2` para indicar que la propiedad no necesita ser propagada a los valores alcanzables. El código Maude de la figura 3.6 muestra la especificación del género `Store`.

El género `Env`, que representa el ambiente, está establecido para asociar las variables con sus posiciones en la memoria, y es un multiconjunto ACU de pares de identificadores de variables y de posiciones. El par `[VarId,Loc]` indica que la variable `VarId` está almacenada en la posición `Loc`. El identificador de una variable es del género `Qid`. Parte de la especificación del género `Env` se muestra en la figura 3.7:

El símbolo constructor `buildEnv` usado en la continuación (`k`) de los hilos, se muestra en la Figura 3.8. Este símbolo asigna una posición de memoria del almacenamiento (`store`) a cada nueva variable, construyendo el ambiente correspondiente. Las ecuaciones correspondientes contienen cinco elementos del estado de un programa Java con sus correspondientes símbolos constructores, el hilo al

Para más detalles véase [Clavel et al., 2007].

```
fmod ENVIRONMENT is
  ...
  sort Env .
  op noEnv : -> Env . --- Env  empty
  op [_,_] : Qid Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] . ---multiconjunto ACU
 ...
endfm
```

Figura 3.7: Definición del multiconjunto ACU para el ambiente `Env`

que corresponde al ambiente (`t`), la continuación del hilo (`k`), el ambiente (`env`), la memoria común a todos los hilos (`store`), y el contador que indica la última posición de memoria utilizada (`nextLoc`). Además se usa el símbolo constructor `->` para concatenar las continuaciones.

Para manejar los valores la semántica define una estructura algebraica que es paramétrica en el género genérico Value que define todos los posibles valores almacenados en la memoria, o que son operandos de todos los operadores Java, etc. Por ejemplo, los constructores `int` y `bool` describen los valores Java enteros y booleanos que son definidos en Maude como "`op int : Int -> Value .`" y "`op bool : Bool -> Value .`", donde `Int` y `Bool` son los géneros internos predefinidos ("built–in") de Maude para los enteros y booleanos.

```
--- No new variable, end buildEnv continuation
eq k(buildEnv(noParameters, noValues) -> K) = k(K) .
--- First look if new variable X and value V could be assigned to
--- an used but released Location L
eq t(k(buildEnv(((T d(X)), Pl), (V,Vl)) -> K) env(Env) id(I) TC) store([L, V', -3] Store)
    nextLoc(I')
= t(k(buildEnv(Pl, Vl) -> K) env([X, L] Env) id(I) TC) store([L,setTid(V, I), I] Store)
    nextLoc(I') .
---New variable is assigned to Location I' + 1
eq t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) TC) store(Store) nextLoc(I')
= t(k(buildEnv(Pl, Vl) -> K) env([X, l(I' + 1)] Env) id(I) TC)
   store([l(I' + 1), setTid(V, I), I] Store) nextLoc(I' + 1) [owise] .
```

Figura 3.8: Ecuaciones, basadas en continuaciones, para construir el ambiente

En la Figura 3.9 se muestra cómo se recupera el valor de una variable Java de la memoria, usando el ambiente que asocia los nombres de las variables con las posiciones de memoria.

El operador de asignación de variables Java se especifica a continuación en la Figura 3.10. Nótese que el orden relativo entre las operaciones de asignación y

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) obj(Obj) = k(#(Loc) -> K) env([Var, Loc] Env) obj(Obj) .
---Then obtain value stored in such location
rl t(k(#(Loc) -> K) id(I) TC) store(Loc, Value, -1] Store)
=> t(k(Value -> K) id(I) TC) store([Loc,Value,-1] Store) .
```

Figura 3.9: Ecuaciones, basadas en continuaciones, para recuperar los valores de las variables

```
---First obtain location in store of the variable while keeping expression in the continuation
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E) -> K)) .
---Once the location is obtained, evaluate expression while keeping location in the continuation
eq k(Loc -> (=(E) -> K)) = k(E -> (=(Loc) -> K)) .
---Once the expression is computed, assign to location
eq k(Value -> (=(L) -> K)) = k([Value -> L] -> (V -> K)) .
---General procedure to update the shared memory
rl t(k([Value -> Loc] -> K) id(I) TC) store([Loc,Value',-1]  ST)
=> t(k(K) id(I) TC) store([Loc, shared(Value), -1] ST) .
```

Figura 3.10: Ecuaciones y reglas, basadas en continuaciones, para el operador de asignación Java

recuperación es relevante ya que varios hilos pueden intentar asignar un valor a una misma variable concurrentemente, o recuperar su valor de la memoria; por esto se usa una regla en lugar de una ecuación para especificar tanto la asignación como la recuperación física de la memoria.

En otras palabras, el operador de asignación y la recuperación del valor de una variable son no determinísticos debido a la presencia de diferentes hilos y son entonces especificados con reglas en lugar de ecuaciones Maude.

Ahora se describen, usando continuaciones, dos operadores Java de números enteros, el operador suma en la Figura 3.11 y el operador relacional menor-o-igual en la Figura 3.12. Ambos operadores son binarios, es decir, que tienen aridad 2 [3]. Dentro del constructor usado para los números enteros `int` está el símbolo Maude de la suma + mientras que afuera del constructor `int` está el símbolo Java de la suma + con aridad 2. Por otra parte, + con aridad 0 es un símbolo de la continuación, para recordar que la acción Java del operador suma está pendiente de que termine la evaluación de los operandos.

La operación booleana Java menor-o-igual de enteros Java está especificada de una manera similar en la Figura 3.12.

[3]La sintaxis Maude permite la sobrecarga de operadores con diferentes aridades.

```
--- First evaluate arguments
eq k((E + E') -> K) = k((E, E') -> (+ -> K)) .
--- Once arguments are evaluated to integers, compute addition
eq k((int(I), int(I')) -> (+ -> K)) = k(int(I + I') -> K) .
```

Figura 3.11: Ecuaciones, basadas en continuaciones, para la suma de enteros Java

```
--- First evaluate arguments
eq k((E <= E') -> K) = k((E, E') -> (<= -> K)) .
--- Once arguments are evaluated to integers, compute boolean
eq k((int(I), int(I')) -> (<= -> K)) = k(bool(I <= I') -> K) .
```

Figura 3.12: Ecuaciones, basadas en continuaciones, para la comparación (menor-o-igual) de enteros Java

Unos aspectos de la semántica Java que son relevantes para la no interferencia, son la instrucción condicional ("if-then-else") y la iteración indefinida ("`while`"), que se muestran en las Figuras 3.13 y 3.14, respectivamente.

La instrucción condicional está especificada de dos maneras, como instrucción del programa Java y como elemento de la continuación. Como instrucción Java puede tener aridad 3, cuando el primer operando es la expresión booleana, y los otros dos, son las subinstrucciones que corresponden al caso verdadero ("then") y al caso falso ("else"). Como instrucción Java también puede tener aridad 2, un operando es la expresión booleana, y el otro la subinstrucción del caso verdadero ("then"). Como elemento de la continuación es un operador con aridad 2 para recordar que se tiene pendiente el flujo de control correspondiente mientras se evalúa la expresión booleana.

```
--- First evaluates the boolean expression while keeping the then and else expressions
eq k((if E S else S' fi) -> K) = k(E -> (if(S, S') -> K)) .
--- If the result value is true continues with the then statement
eq k(bool(true) -> (if(S, S') -> K)) = k(S -> K) .
--- If the result value is false continues with the else part
eq k(bool(false) -> (if(S, S') -> K)) = k(S' -> K) .
```

Figura 3.13: Ecuaciones, basadas en continuaciones, para la instrucción condicional "`if-then-else`"

Las especificaciones de las instrucciones para el escape "`break`"de la instrucción "`while`"y el retorno de valores de métodos que son funciones, también son relevantes para la no interferencia por que son instrucciones de flujo de control. En las Figuras 3.15 y 3.16 se muestran sus definiciones. Nótese el manejo de las pilas de los bucles y de las funciones, denotados con los símbolos constructores

```
---First, the thread state is stacked in the loop stack and the boolean expression is evaluated
eq t(k((while E S) -> K) lstack(Lstack) TC)
    = t(k(E -> while(E, noExp, S) -> popLStack -> K)
       lstack(fsi((while(E, noExp, S) -> K), TC) Lstack)  TC) .
--- if true then continues with the statement within the while body
eq k(bool(true) -> (while(E, El, S) -> K)) = k(S -> (El -> ; -> (E -> (while(E, El, S) -> K)))) .
--- if false, the while loop is exited
eq k(bool(false) -> (while(E, El, S) -> K)) = k(K) .
--- the stacked state is removed from the loop stack
eq k(popLStack -> K) lstack(LItem Lstack) = k(K) lstack(Lstack) .
```

Figura 3.14: Ecuaciones, basadas en continuaciones, para la instrucción iterativa "`while`"

`lstack` y `fstack`, respectivamente. La pila `lstack` mantiene apilados los bucles actualmente en ejecución precisamente para controlar adecuadamente el escape del bucle "`break`"(Figuras 3.14 y 3.15).

```
---The thread state is restored from the loop stack
eq t(k(break; -> K) lstack(fsi((while(E,El,S) -> K'), TC) Lstack) TC')
   = t(k(K') lstack(Lstack) TC) .
```

Figura 3.15: Ecuaciones, basadas en continuaciones, para el escape "`break`"de la instrucción "`while`"

La instrucción "`return`"restaura el ambiente previo a la invocación del método, así como también los candados y el estado local del hilo, que están apilados junto con la continuación correspondiente en la pila `fstack`. Posteriormente apila en la continuación restaurada, la acción para liberar el ambiente local del método y sus candados.

```
eq t(k(V -> return -> K) holds(Ll') env(Env')
    fstack( fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
 = t(k(releaseEnv(Env') -> release(Ll, Ll') -> (V -> K')) holds(Ll) env(Env)
    fstack(Fstack) TC) .
```

Figura 3.16: Ecuaciones, basadas en continuaciones, para la instrucción "`return`"

Como las ecuaciones se evalúan de forma determinista, el espacio de estados asociado a una teoría de reescritura está determinado en Maude por las reglas únicamente. Esto significa que aunque las reglas y ecuaciones son aplicadas en Maude de la misma forma, Maude solo mantiene el registro de las reglas aplicadas y omite la información acerca de las ecuaciones aplicadas. Así pues, el número de ecuaciones y reglas de una especificación Maude es relevante y mientras más

pequeño sea el número de reglas más eficiente es la verificación del análisis ya que el espacio de estados para la búsqueda es más pequeño.

Según [Farzan et al., 2007], la semántica operacional de Java contiene 424 ecuaciones y solo 7 reglas, lo cual ahorra memoria y tiempo de ejecución. El siguiente ejemplo ilustra la mecanización de la semántica de Java.

Ejemplo 2. Consideremos el programa Java del Ejemplo 1. El comando `search` es la búsqueda primero-en-amplitud predefinida de Maude, que provee todas las secuencias de reglas aplicadas desde un término inicial (sin variables) hasta un término final (con variables posiblemente) [Clavel et al., 2007]. El término inicial describe un estado Java inicial concreto, y el término final (con variables posiblemente) un conjunto de estados Java finales, posiblemente infinito. En el comando `search` abajo se pregunta por el conjunto de todos los posibles estados correspondientes a todos los posibles valores retornados por la función Java `main` del Ejemplo 1, y por eso, una variable término denota el estado objetivo a ser alcanzado. Nótese que el código de las dos funciones Java `evenOdd` and `main` está embebido en el comando de búsqueda, así como también la invocación a `main` (para los detalles de cómo construir un estado Java inicial puede verse [Farzan et al., 2007]).

```
search in PGM-SEMANTICS : java((preprocess(default class t('Safe1Even2) imports
    nil extends Object implements none {(default static) int 'evenOdd(int d(
    'j))throws(noType) {{{{(int d('u) = i(3) ;) (int d('v) = i(4),d('z) = i(4)
    ;)) 12 @ ('z += i(30) ;)) 13 @ ('v = 'u * i(8) + 'j ;)) 14 @ return 'z - 'v
    ;} (public static) void 'main(t('String)[] d('args))throws(noType) {4 @ (
    'System . 'out . 'println < 'evenOdd < i(0) > > ;)}}) noType . 'main < new
    string [i(0)] > noVal)) =>! X:Output .

Solution 1 (state 1)
states: 2  rewrites: 287 in 76ms cpu (4ms real) (0 rewrites/second)
X:Output --> pl(int(10))

No more solutions.
states: 2  rewrites: 287 in 76ms cpu (7ms real) (0 rewrites/second)
```

El comando de búsqueda retorna que solo una única posible ejecución del programa Java es posible como resultado de la instrucción Java "`System.out.println(evenOdd(0));`", que produce el valor Java 10. La secuencia de reescritura completa que lleva a este valor Java es también entregada por Maude con el comando `search path 1`.

3.4. Interpretación Abstracta

Una *interpretación abstracta* (o abstracción) [Cousot and Cousot, 1977] de la semántica de un programa está dada por un *operador de cierre superior* $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$, el cual es *monótono* (para todo $SSt_1, SSt_2 \in \wp(\mathsf{State})$, $SSt_1 \subseteq SSt_2$ implica $\alpha(SSt_1) \subseteq \alpha(SSt_2)$), *idempotente* (para todo $SSt \in \wp(\mathsf{State})$, $\alpha(SSt) \subseteq \alpha(\alpha(SSt))$), y *extensivo* (para todo $SSt \in \wp(\mathsf{State})$, $SSt \subseteq \alpha(SSt)$).

La intuición de esta definición es que cada estado del programa Java, $St \in \mathsf{State}$ es abstraido por su cierre $\alpha(\{St\})$.

Los operadores de cierre tienen varias propiedades interesantes. Por ejemplo, cuando el dominio abstracto considerado es un retículo completo, es decir, $\langle \alpha(\mathsf{State}), \subseteq \rangle$, cada operador de cierre está determinado únicamente por el conjunto de sus puntos fijos. En el contexto de la interpretación abstracta, los operadores de cierre son importantes porque los dominios abstractos pueden definirse de forma equivalente usando los operadores, o mediante inserciones de Galois, tal como se presenta en [Cousot and Cousot, 1979]. Sea $\iota : \alpha(\wp(\mathsf{State})) \to A$ un isomorfismo. Entonces, dado un operador de cierre superior $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$, la estructura $(\wp(\mathsf{State}), \alpha \circ \iota, \iota^{-1}, A)$ es una inserción de Galois, donde $\alpha \circ \iota$ y ι^{-1} son las funciones de abstracción y de concretización respectivamente (véase [Cousot and Cousot, 1979] para mayores detalles).

En el enfoque propuesto, un consumidor de código puede asignar dominios abstractos diferentes a cada variable del código Java para obtener un modelo de estado finito del programa. Esta asignación puede hacerla cualquier usuario potencial de la herramienta (el desarrollador del código, un certificador, el consumidor o su representante), ayudado por la interfaz gráfica y las ayudas provistas. El usuario simplemente anota el código fuente Java con las cláusulas JML que codifican la política de seguridad, de forma que tanto las variables críticas como sus correspondientes dominios abstractos se pueden inferir automáticamente.

En el proceso de asignar un dominio abstracto a una variable del código fuente, se tienen por un lado los aspectos teóricos, pero también por otro, los aspectos prácticos. Los dominios abstractos específicos considerados así como los aspectos prácticos se considerarán adelante en los capítulos donde se presentan las semánticas abstractas de Java en Maude para la certificación de propiedades aritméticas

basadas en tipos (Capítulo 4), y la no interferencia (Capítulo 5).

A nivel teórico se define una función de abstracción para cada identificador de variable $\mathtt{x}$, i. e. $\alpha_{\mathtt{x}} : \wp(\mathsf{Int}) \to \wp(\mathsf{Int})$, y homomórficamente se extienden estas funciones de abstracción a todo el estado del programa mediante la función de abstracción $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$. Así, α abstrae para cada variable $\mathtt{x}$ los valores almacenados en la memoria Java para $\mathtt{x}$ usando $\alpha_{\mathtt{x}}$, que es la función identidad si no se ha seleccionado dominio abstracto.

En los Capítulos sobre propiedades aritméticas basadas en tipos (Capítulo 4) y propiedades de no interferencia (Capítulo 5), se presentan las definiciones formales de la interpretación abstracta para cada caso así como el teorema de corrección y completitud correspondiente.

4

Propiedades Aritméticas basadas en Tipos

Las propiedades basadas en tipos son propiedades de los resultados numéricos de métodos Java que no son verificables por el compilador de Java y que se pueden definir con base en propiedades aritméticas de los números enteros. Nuestro trabajo en ésta línea comenzó en [Alba-Castro et al., 2008]. En este trabajo se consideró inicialmente la propiedad de que el resultado entero de un método sea un número par o impar [Wu et al., 2003], lo cual llevó posteriormente a considerar también las propiedades del número entero modulo 4, y de que fuera menor o igual (o mayor) que otro número entero .

Para motivar nuestro trabajo consideremos la certificación de programas Java simples, comenzando por uno tomado prestado de la literatura relacionada.

Ejemplo 3. Se considera un programa Java muy simple tomado de [Wu et al., 2003], con el requerimiento de producir un número entero par como resultado. Primero se expresa este requerimiento como una política de seguridad usando el lenguaje de especificación para Java JML mediante una cláusula `ensures` y el operador `\result`. Esto significa que se requiere que el resultado Java no sea un número impar cuando la ejecución del método se termine. La función `AbsValue` se usa para referirse al valor abstracto de una variable, en el ejemplo, al valor retornado por el método `even16`. El `#even` denota un valor abstracto que representa en este caso que el valor concreto retornado debe ser un número par.

Estamos usando JML respetando su sintaxis con la única excepción del uso de la constante `#even` que no existe como tal en Java y que como identificador Java tampoco es permitida por contener el símbolo `#`.

```
int even16()    {
   /*@  Safety Specification:
     @           requires true;
     @           ensures AbsValue(\result) == #even;                                    @*/
         int x = 4;
         int y = x + 8;
         /*@      assert AbsDomain(x) == EvenOdd && AbsDomain(y) == EvenOdd;        @*/
         return x+y;
}
```

El espacio de estados de este programa es finito y podría ser explorado por herramientas para Java que apliquen computación simbólica o comprobación de modelos con estados explícitos (concretos) como JavaFAN [Farzan et al., 2004a], aunque no pueden generan un certificado formal (una demostración en alguna lógica o sistema formal).

Ejemplo 4. Consideremos ahora un ejemplo algo más elaborado con una política similar de resultado par, pero que en este caso debe tomar en cuenta que el resultado del método es par si la entrada es también par. La especificación de la política usa entonces una cláusula `requires` para establecer la condición sobre la entrada, con la función `AbsValue` y el valor abstracto `#even`.

```
int evenOdd(int j)   {
     /*@  Safety Specification:
       @            requires  AbsValue(j)  == #even ;
       @            ensures  AbsValue(\result) == #even ;              @*/
          int u = 3; int v,z = 4;
          z += 30;
          v = u*8 + j;
          return z - v;
}
```

En este programa tenemos un número inicial de estados infinito, aunque el espacio de estados de búsqueda es finito para cada estado inicial. Una herramienta como JavaFAN solo podría usarse como un procedimiento de semi-decisión para buscar violaciones de la política para estados iniciales específicos. Otra herramienta para Java como Bogor [Robby et al., 2006] que usa JML para especificar las propiedades a verificar y aplica comprobación de modelos en estados basados en abstracciones, podría verificar la política (reescribiéndola para respetar la sintaxis JML [1]) pero no pueden entregar una demostración como certificado.

[1]Por ejemplo usando el residuo de la división entera con el operador Java %

El siguiente ejemplo es más realista, con bucles y condicionales: un método con un parámetro `n` que computa la suma de los primeros `n` números enteros no negativos.

Ejemplo 5. Consideremos ahora un método Java más realista, que requiere una condición más compleja sobre la entrada para asegurar el cumplimiento de la política de seguridad requerida, así como otras condiciones adicionales, más complicadas aún, sobre las dos variables locales del método. El resultado debe ser también par pero con una política "módulo 4"en el parámetro de entrada `n` y en la variable local `i`, con la condición adicional de que `i` sea menor o igual que `n` antes de entrar al bucle. Además la variable local `sum` debe ser abstraída según la paridad (par o impar).

```
int summation(int n)  {
     /*@  Safety Specification:
       @            requires  AbsValue(n)  == #0 ||  AbsValue(n)  == #3 ;
       @            ensures  AbsValue(\result)  == #even ;                              @*/
      int sum ;
      //@  assert AbsDomain(sum) == EvenOdd;
      int i = 0;
      //@  assert AbsDomain(i) == (Mod4, (<=,n));
      while (i<=n) {
             sum += i;
             i++;
      }
      return sum;
}
```

El último ejemplo también tiene un bucle pero la política de seguridad "par"es más simple de especificar.

Ejemplo 6. Este ejemplo que se presenta realiza el mismo cómputo del Ejemplo 5 y tiene el mismo parámetro entero, pero usa menos variables. Solo usa una variable local con la correspondiente aserción, pero más simple.

```
int summationMod4(int n) {
    /*@
      @           requires AbsValue(n) == #0 || AbsValue(n) == #3 ;
      @           ensures AbsValue(\result)  == #even ;
      @*/
      int sum = 0 ;
      /*@         assert AbsDomain(sum) == EvenOdd ;                      @*/
      while (0!=n)   {
```

```
            sum += n;
            n--;
        }
    return sum;
}
```

4.1. La Semántica Abstracta de Java en Lógica de Reescritura

En esta parte se desarrolla la versión abstracta de la semántica Java en lógica de reescritura (dada en [Farzan et al., 2007] y presentada en el Capítulo 3.3), descrita por la teoría de reescritura $\mathcal{R}_{\text{Java}\#} = (\Sigma_{\text{Java}\#}, E_{\text{Java}\#}, R_{\text{Java}\#})$, $E_{\text{Java}\#} = \Delta_{\text{Java}\#} \uplus B_{\text{Java}\#}$ y su correspondiente relación de reescritura $\rightarrow_{\text{Java}\#}$. Dado que la teoría $\mathcal{R}_{\text{Java}}$ está definida sobre un género genérico Value, el enfoque aplicado consiste en extender $\mathcal{R}_{\text{Java}}$ aprovechando su modularidad mediante la definición de dominios abstractos como subgéneros del género Value con las correspondientes y apropiadas versiones de las construcciones Java y de los operadores para los dominios abstractos.

El siguiente ejemplo muestra algunos de los dominios abstractos relevantes para el presente trabajo.

Ejemplo 7. Se considera una función de abstracción que clasifica los enteros Java en enteros pares e impares, i.e., `mod2` : $\wp(\texttt{int(Int)}) \rightarrow \wp(\texttt{int(Int)})$ donde `int(Int)` denota los términos Maude de género Value que corresponden a los enteros Java.

Esta abstracción es relevante para los Ejemplos 3, 4, 5 y 6. Se pueden seleccionar los siguientes símbolos abstractos $EvenOdd = \{\texttt{\#even}, \texttt{\#odd}, \texttt{top}, \texttt{bot}\}$ para denotar los siguientes subconjuntos `top` = `int(Int)`, `bot` = $\emptyset$, `#even` = $\{\texttt{int}(n) \mid n \ mod \ 2 = 0\}$, y `#odd` = $\{\texttt{int}(n) \mid n \ mod \ 2 = 1\}$.

Como se dijo en la sección 3.4 las asignaciones de las variables del programa son inferidas de las anotaciones JML del código fuente Java, por ejemplo, de la anotación del Ejemplo 3:

```
/*@        assert AbsDomain(x) == EvenOdd && AbsDomain(y) == EvenOdd;     @*/
```

se infiere que las variables `x` y `y` están asignadas al dominio abstracto denotado por el símbolo *EvenOdd*. Se puede refinar este dominio abstracto mediante la abstrac-

ción de los enteros Java módulo 4, así, `mod4` : $\wp(\texttt{int(Int)}) \rightarrow \wp(\texttt{int(Int)})$. Esta abstracción es relevante para los Ejemplos 5 y 6. Se pueden tener los siguientes símbolos abstractos $Mod4 = \{\texttt{top}, \texttt{bot}, \texttt{\#0}, \texttt{\#1}, \texttt{\#2}, \texttt{\#3}\}$ donde $\texttt{\#k} = \{\texttt{int}(n) \mid n \ mod\ 4 = k\}$ for $k \in \{0, 1, 2, 3\}$. El retículo inducido por la relación $\subseteq$ entre conjuntos de enteros Java se muestra en la Figura 4.1.

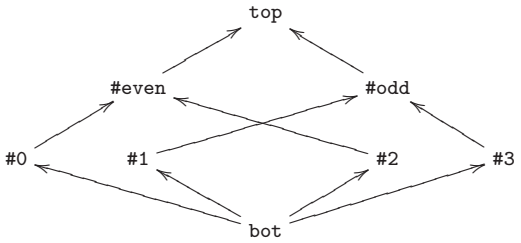

Figura 4.1: Retículo de enteros para las abstracciones $mod2$ y $mod4$

A nivel práctico, se debe complementar la semántica original Java con una nueva función Maude, denominada `inAbsDomain`, que registra el dominio abstracto asociado a cada nombre de variable y que será usada en dos momentos: cuando el ambiente de la variable es creado por primera vez en la memoria Java y siempre que su valor sea actualizado en la memoria. Por ejemplo, la Figura 4.2 muestra el código de `inAbsDomain` para las variables `x`,`y` del Ejemplo 3, según las anotaciones JML, junto con el código Maude para las funciones de abstracción `mod2` y `mod4`. En dicha Figura también están las ecuaciones de las funciones para comprobar si el dominio abstracto de una variable es *EvenOdd* (`AbsDomainEvenOdd`) o $Mod4$ (`AbsDomainMod4`), que están definidas ambas con base en la función `inAbsDomain`. Como ya se dijo, se tienen que adicionar las invocaciones a la función `inAbsDomain` en el símbolo de continuación `buildEnv` de la Figura 3.8 y al operador de asignación de Java de la Figura 3.10; todas estas modificaciones se muestran en las Figuras 4.3 y 4.4.

Se debe notar que las ecuaciones del operador de asignación diferencian tanto el valor por asignar a la variable como el dominio abstracto de ésta. En el caso de asignar un valor del género *EvenOdd* a una variable de género $Mod4$ se deben usar reglas por el indeterminismo que existe, pues un valor *even* (número par) puede ser (módulo 4) #0 o #2 y un valor *odd* (número impar) puede ser (módulo 4) #1 o #3.

```
--- Define abstract domains
  subsort AbstInt < Value .
  ops top bot : -> AbstInt .
  sorts EvenOdd Mod4  .
  subsorts Mod4 EvenOdd  < AbstInt .
  ops even odd : -> EvenOdd .
  op #_ : Int -> Mod4 .
--- Define abstraction functions
  op mod2 : Value -> EvenOdd .
  eq mod2(int(I)) = if (I rem 2 == 0) then even else odd fi .
  op mod4 : Value -> Mod4 .
  eq mod4(int(I)) = #(I rem 4) .
--- Equations for abstracting concrete values
  op inAbsDomain : Var Value -> Value .
  eq inAbsDomain('x,int(I)) = mod2(int(I)) .
  eq inAbsDomain('y,int(I)) = mod2(int(I)) .
  eq inAbsDomain(Var,Value) = Value [owise] .
```

Figura 4.2: Dominios abstractos y asociación de dominios abstractos a nombres de variables

```
--- New location
eq t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) TC) store([L, V', -3] Store)
      nextLoc(I')
  =  t(k(buildEnv(Pl, Vl) -> K) env([X, L] Env) id(I) TC)
      store([L, setTid(inAbsDomain(X,V), I), I] Store) nextLoc(I')  .
--- Used location
eq t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) TC) store( Store)
      nextLoc(I')
  =  t(k(buildEnv(Pl, Vl) -> K) env([X, l(I' + 1)] Env) id(I) TC)
      store([l(I' + 1), setTid(inAbsDomain(X,V),I), I] Store) nextLoc(I' + 1)  [owise] .
```

Figura 4.3: Ecuaciones modificadas, basadas en continuaciones, para construir el ambiente

Se tienen que especificar las versiones abstractas de todos los operadores Java de la semántica Java que tengan que ver con los valores abstractos. Entonces se debe definir la aproximación de la suma de enteros, del operador de comparación menor-o-igual, etc. para los nuevos dominios abstractos de los números enteros. También se debe considerar la operación con operandos combinados, es decir, con valores abstractos y valores concretos, y con valores de dos diferentes dominios abstractos. Por ejemplo, dada la función abstracta `mod2`, la operación suma de enteros *EvenOdd* está especificada en la Figura 4.5.

Por ejemplo, dada la función abstracta `mod4`, la operación suma de enteros *Mod*4 está especificada en la Figura 4.6.

```
--- Assignment modified equations
op = : Exp Qid -> Continuation . --- new definition
op = : Location Qid -> Continuation . --- new definition
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E,Var) -> K)) .
eq k(Loc -> (=(E,Var) -> K)) = k(E -> (=(Loc,Var) -> K)) .
--- EvenOdd Value to EvenOdd Var
ceq k(ValMod2 -> (=(L,Var) -> K)) = k([ValMod2 -> L] -> (ValMod2 -> K))
      if AbsDomainEvenOdd(Var)  .
--- Mod4 Value to Mod4 Var
ceq k(ValMod4 -> (=(L,Var) -> K)) = k([ValMod4 -> L] -> (ValMod4 -> K))
      if AbsDomainMod4(Var) .
--- Mod4 Val to EvenOdd Var
ceq k(# I -> (=(L,Var) -> K)) = k([mod2(int(I)) -> L] -> (mod2(int(I)) -> K))
      if AbsDomainMod4(Var) .
--- EvenOdd Val to Mod4 Var
crl k(even -> (=(L,Var) -> K)) => k([inAbsDomain(Var,# 0) -> L] -> (even -> K))
      if AbsDomainMod4(Var) .
crl k(even -> (=(L,Var) -> K)) => k([inAbsDomain(Var,# 2) -> L] -> (even -> K))
      if AbsDomainMod4(Var) .
crl k(odd -> (=(L,Var) -> K)) => k([inAbsDomain(Var,# 1) -> L] -> (odd -> K))
      if AbsDomainMod4(Var) .
crl k(odd -> (=(L,Var) -> K)) => k([inAbsDomain(Var,# 3) -> L] -> (odd -> K))
      if AbsDomainMod4(Var) .
--- Integer Val to any Var
eq k(Val -> (=(Loc,Var) -> K)) = k([inAbsDomain(Var,Val) -> Loc] -> (Val -> K)) [owise] .
```

Figura 4.4: Ecuaciones modificadas, basadas en continuaciones, para la asignación Java

En interpretación abstracta es usual comprimir numerosos pasos de computación en un único paso de computación abstracta, para reflejar que muchos y diferentes comportamientos son emulados en un estado abstracto. Por ejemplo el operador Java menor-o-igual `<=` de la Figura 3.12 y la función abstracta `mod2`, cuando se comparan dos valores $\#even$ un resultado inexacto y aproximado es la unión de `true` y `false`, que está denotada por el símbolo `top`. Esta idea implicaría incluir la siguiente ecuación en la semántica abstracta $\mathcal{R}_{\text{Java}^{\#}}$:

```
eq k((even, even) -> (<= -> K)) = k(top -> K) .
```

Esta instrumentalización de la semántica Java para manejar la abstracción significaría muchas modificaciones, por ejemplo en las ecuaciones de las instrucciones condicionales, debido a que se generan estados Java completamente diferentes que tendrían que abstraerse en un único estado abstracto. Se adoptó entonces un enfoque diferente. Cuando varios pasos de reescritura $\rightarrow_{\text{Java}}$ son emulados por un

```
--- Excecute addition with EvenOdd values
eq k((even, even) -> (+ -> K)) = k(even -> K) .
eq k((even, odd) -> (+ -> K)) = k(odd -> K) .
eq k((odd, even) -> (+ -> K)) = k(odd -> K) .
eq k((odd, odd) -> (+ -> K)) = k(even -> K) .
--- Standard concrete values are abstracted prior to add to EvenOdd values
var ValEO : EvenOdd .
eq k((ValEO, int(I)) -> K) = k((ValEO, mod2(int(I))) -> K)  .
eq k((int(I),ValEO) -> K) = k((mod2(int(I)) ,ValEO) -> K)  .
```

+	even	odd
even	even	odd
odd	odd	even

Figura 4.5: Definición abstracta y ecuaciones para el operador suma de Java en el dominio *EvenOdd*

```
--- Excecute addition with Mod4 values
eq k((# I, # I') -> (+ -> K)) = k(mod4(int(I + I')) -> K) .
--- Standard integer values are abstracted prior to add to Mod4 values
var ValM4  : Mod4 .
eq k((ValM4, int(I)) -> K) = k((ValM4, mod4(int(I))) -> K)  .
eq k((int(I),ValM4) -> K) = k((mod4(int(I)) ,ValM4) -> K)  .
--- Mod4 values are approximate when combined with EvenOdd values
var ValEO : EvenOdd .
eq k((ValEO,# I) -> K) = k((ValEO,mod2(int(I))) -> K) .
eq k((# I, ValEO) -> K) = k((mod2(int(I)),ValEO) -> K) .
```

+	#0	#1	#2	#3
#0	#0	#1	#2	#3
#1	#1	#2	#3	#0
#2	#2	#3	#0	#1
#3	#3	#0	#1	#2

Figura 4.6: Definición abstracta y ecuaciones para el operador suma de Java en el dominio $Mod4$

estado abstracto y esos pasos de reescritura aplican diferentes reglas y ecuaciones, se usa la concurrencia a nivel de Maude. Esto significa, que adicionamos reglas a $\rightarrow_{\text{Java}}$ para reflejar las posibles y diferentes evoluciones del sistema.

Siguiendo el enfoque adoptado, el operador Java menor-o-igual <= de la Figura 3.12 se especifica como sigue, indicando que el operador de comparación <= puede evaluarse indistintamente, a verdadero `true` o a falso `false`.

```
rl k((even, even) -> (<= -> K)) = k(bool(true) -> K) .
rl k((even, even) -> (<= -> K)) = k(bool(false) -> K) .
```

Ahora, se formaliza la relación de reescritura abstracta $\rightarrow_{\text{Java}^{\#}}$, que intuitivamente desarrolla la idea de aplicar solamente una regla o ecuación de la semántica concreta de Java a un estado abstracto mientras se exploran las diferentes alternativas de una forma no determinista. Se tiene como referencia la introducción de la formalización de la interpretación abstracta de la Sección 3.4. Abusando de la notación, se denota la abstracción de la regla $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ por $\alpha(\{l\} \rightarrow \{r\})$.

Definición 1 (Reescritura abstracta)**.** Sea $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ una abstrac-

ción. Se define la versión aproximada de la reescritura $\rightarrow_{\text{Java}\#} \subseteq \wp(\mathsf{State}) \times \wp(\mathsf{State})$ con:

$$SSt_1 \rightarrow_{\text{Java}\#} SSt_2 \text{ usando } \alpha(\{l\} \rightarrow \{r\}) \in (R_{\text{Java}\#} \cup \Delta_{\text{Java}\#})$$
$$\text{iff } \forall u \in \alpha(SSt_1), \exists v \in SSt_2 \text{ s.t. } u \rightarrow_{\text{Java}} v, \text{usando } l \rightarrow r \in R_{\text{Java}} \cup \Delta_{\text{Java}}.$$

$\rightarrow^*_{\text{Java}\#}$ denota la extensión de $\rightarrow_{\text{Java}\#}$ a muchos pasos de reescritura. El siguiente resultado se obtiene directamente de las propiedades de monotonía, idempotencia y extensibilidad del operador de clausura superior α.

Teorema 1 (Corrección & Completitud)**.** Sea $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ una abstracción. Sea $SSt_1, SSt_2 \in \wp(\mathsf{State})$. Si $SSt_1 \rightarrow^*_{\text{Java}\#} SSt_2$, entonces para todo $u \in \alpha(SSt_1)$, existe un $v \in SSt_2$ tal que $u \rightarrow^*_{\text{Java}} v$. Sea $St_1, St_2 \in \mathsf{State}$. Si $St_1 \rightarrow^*_{\text{Java}} St_2$, entonces existe $SSt_3 \subseteq \wp(\mathsf{State})$ s.t. $\alpha(St_1) \rightarrow^*_{\text{Java}\#} SSt_3$ y $St_2 \in SSt_3$.

La búsqueda primero-en-amplitud para el sistema abstracto de estado finito (es finito debido al uso de dominios abstractos finitos) permite obtener una herramienta útil para ejecución simbólica, a la vez, que resultan simples las modificaciones que se requieren de la semántica Java en Maude. La verificación se reduce a la exploración de todas las secuencias de reescritura.

Ejemplo 8. Considérese las función Java `evenOdd` de la Figura 4, la función `main` en el programa Java completo del Ejemplo 1 y la semántica abstracta de Java mostrada arriba con la función `inAbsDomain`de la Figura 4.2. Nótese que el único cambio en el comando de búsqueda del Ejemplo 2 es el reemplazo de `PGM-SEMANTICS` con `PGM-SEMANTICS-EVENODD-ABSTR`.

```
search in PGM-SEMANTICS-EVENODD-ABSTR : java((preprocess(default
    class t('Safe1Even2) imports nil extends Object implements none
    {(default static) int 'evenOdd(int d( 'j))throws(noType) {((((int d('u) = i(3) ;)
    (int d('v) = i(4),d('z) = i(4) ;)) 12 @ ('z += i(30) ;)) 13 @ ('v = 'u * i(8) + 'j ;))
    14 @ return 'z - 'v ;} (public static) void 'main(t('String)[] d('args))
    throws(noType) {4 @ ( 'System . 'out . 'println < 'evenOdd < i(0) > > ;)}})
    noType . 'main < new  string [i(0)] > noVal))

=>! X:Output .
```

Ahora, el comando de búsqueda produce el resultado siguiente, lo que significa que exactamente una traza de ejecución Java está demostrada, que retorna el valor abstracto `even` como resultado de la instrucción Java "`System.out.println(evenOdd(0));`":

```
Solution 1 (state 1)
states: 2  rewrites: 328 in 879ms cpu (11ms real) (0 rewrites/second)
X:Output --> pl(even)
```

y por consiguiente toda ejecución real del programa Java de la figura 1 también siempre retorna un valor par de acuerdo con el Teorema 1.

Sin embargo, la abstracción definida en el Ejemplo 7 no es lo suficientemente precisa para el programa Java del Ejemplo 5, como lo muestra el siguiente ejemplo.

Ejemplo 9. Considérese el código del Ejemplo 5 con la siguiente función `main`:

```
void main() { System.out.println(summation(0)); }
```

Y con la siguiente asignación de dominios abstractos a las variables del programa Java :

```
op inAbsDomain : Var Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = mod4(int(I)) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

Cuando hacemos la búsqueda de los resultados de la función `main`

```
search in PGM-SEMANTICS-EVENODD-ABSTR : java((preprocess(default class t(
    'Safe1Even3) imports nil extends Object implements none {(default static)
    int 'summation(int d('n))throws(noType) {(((int d('sum) = i(0) ;) (int d(
    'i) = i(0) ;)) 15 @ (while 'i <= 'n 15 @ {(13 @ ('sum += 'i ;)) 14 @ ('i ++
    ;)})) 16 @ return 'sum ;} (public static) void 'main(t('String)[] d(
    'args))throws(noType) {4 @ ('System . 'out . 'println < 'summation < i(0) >
    > ;)}}) noType . 'main < new string [i(0)] > noVal))
=>! X:Output .
```

Maude entrega los dos resultados siguientes:

```
Solution 1 (state 5)
states: 8  rewrites: 482 in 3887ms cpu (431ms real) (0 rewrites/second)
X:Output --> pl(even)

Solution 2 (state 9)
states: 10  rewrites: 569 in 7384ms cpu (580ms real) (0 rewrites/second)
X:Output --> pl(odd)
```

los cuales no son útiles, ya que ambos resultados `even` y `odd` son posibles. El problema es que la condición booleana `(i <= n)` retorna ambos valores `true` y `false` (de una forma no determinista) cuando se aplica a enteros bajo la acción de los operadores de abstracción `mod2` y `mod4` en demasiados casos.

Para mejorar la precisión, se define un nuevo dominio abstracto más detallado (menos aproximado) $\texttt{leq}^{\#}_{x,y}$ que está parametrizado por dos nombres de variables Java x, y (que pueden tener diferentes dominios abstractos). Este nuevo dominio puede usarse para en el ejemplo anterior abstraer la variable `i` relacionada con `n`. A nivel teórico, tenemos dos dominios abstractos $\alpha_x, \alpha_y : \wp(\mathsf{Int}) \rightarrow \wp(\mathsf{Int})$ que se usan para los valores almacenados en la memoria Java de las variables x, y, respectivamente. La extensión $\texttt{leq}^{\#}_{x,y} : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ toma esos dos dominios abstractos α_x, α_y y captura además la relación $x \leq y$ o $x > y$.

A nivel práctico, se usan los símbolos `leq#` y `gt#` definidos en Maude como sigue "`leq# : Abst Qid -> AbstLeqN`" y "`gt# : Abst Qid -> AbstLeqN`" en donde el primer argumento denota el dominio abstracto para la variable x (o sea α_x) y el segundo argumento es el nombre de la segunda variable y. Para el ejemplo se tendrá una expresión abstracta para la variable `i` tal como `leq#(#0,'n)` que denota que el valor actual de la variable `i` módulo 4 es 0, y que la variable `i` es menor o igual que la variable `n`, cualquiera que sea el valor asignado a `n` durante la ejecución.

La versión apropiada de los operadores Java relevantes para este nuevo dominio abstracto se definen en las Figuras 4.7 y 4.9, y su especificación Maude se muestra en las Figuras 4.8 y 4.10, respectivamente. En el caso de los operadores relacionales (menor-o-igual, mayor, etc.) aplicados a expresiones que son variables, se apilan en la continuación estas variables, para recordar que se están comparando dos variables, y controlar el caso en que en la abstracción alguna esté relacionada con la otra mediante `leq#` o `gt#`. Una vez obtenidos los valores abstractos de las variables comparadas, el resultado de la comparación depende no solo de tales valores sino de las variables comparadas apiladas en la continuación. Si las variables son dependientes entre sí en la abstracción, el resultado de la comparación dependerá del valor abstracto de la variable dependiente y no del valor abstracto de la otra variable. Una vez realizada la comparación, se desapila de la continuación el par de variables comparadas. Si las variables son

independientes en la abstracción, se desapila el par de variables comparadas y la comparación de los valores se ejecuta normalmente. Debe notarse que no se

<=	any variable V value
leq#(Val,V)	true
gt#(Val,V)	false

>	any variable V value
leq#(Val,V)	false
gt#(Val,V)	true

Figura 4.7: Definición de los operadores relacionales Java $\leq$ y $>$ para enteros de AbstLeqN

puede usar arriba el dominio abstracto de la segunda variable en lugar de su nombre, debido a que el valor de esa variable puede cambiar. Por ejemplo, considérese la siguiente variante del Ejemplo 5 en la que el bucle contiene la asignación `n -= 1 ;`, y entonces la variable `n` cambia en cada iteración.

Ejemplo 10. Reconsideramos ahora el Ejemplo 9. El código de la función `inAbsDomain` para el Ejemplo 5 es el que sigue, denotando que las variables `i` y `n` tienen dominios `mod4`, la variable `sum` tiene el dominio `mod2` y la relación $i \leq n$ está también representada en el dominio abstracto:

```
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = leq#(mod4(int(I),'n) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

Cuando se busca por las soluciones para la función Java `main` utilizando el siguiente comando

```
search in PGM-SEMANTICS-EVENODD-ABSTR : java((preprocess(default class t(
    'Safe1Even3) imports nil extends Object implements none {(default static)
    int 'summation(int d('n))throws(noType) {(((int d('sum) = i(0) ;) (int d(
    'i) = i(0) ;)) 15 @ (while 'i <= 'n 15 @ {(13 @ ('sum += 'i ;)) 14 @ ('i ++
    ;)})) 16 @ return 'sum ;} (public static) void 'main(t('String)[] d(
    'args))throws(noType) {4 @ ('System . 'out . 'println < 'summation < i(0) >
    > ;)}}) noType . 'main < new string [i(0)] > noVal))
=>! X:Output .
```

se obtiene el siguiente único resultado, que significa que exactamente una traza de ejecución abstracta está demostrada, la cual retorna el valor abstracto `even` como resultado de la instrucción Java "`System.out.println(summation(0))`":

```
--- The compared variables are stacked into the continuation
op leqN : Qid Qid -> Continuation . --- new definition
ceq k((Var,Var') -> ( Con -> K)) = k((Var,Var') -> (leqN(Var,Var') -> ( Con -> K)))
      if operRel(Con) .
--- Two equations for the Java less-or-equal operator on dependent <= integers
eq k((leq#(V, Var'), V') -> (leqN(Var,Var') -> ( <= -> K))) = k(bool(true) -> K) .
eq k((leq#(V, Var'), V') -> (leqN(Var,Var') -> ( > -> K))) = k(bool(false) -> K) .
--- Two equations for the Java greater-than operator on dependent > integers
eq k((gt#(V, Var'), V') -> (leqN(Var,Var') -> ( > -> K))) = k(bool(true) -> K) .
eq k((gt#(V, Var'), V') -> (leqN(Var,Var') -> ( <= -> K))) = k(bool(false) -> K) .
--- Remove stacked variables pair when comparing non related values
eq k((V,V') -> (leqN(Var,Var') -> ( Con -> K))) = k((V,V') -> ( Con -> K)) [owise] .
```

Figura 4.8: Ecuaciones, basadas en continuaciones, para los operadores Java $\leq$ y $>$ de enteros

++	
leq#(#(I),y)	leq#(mod4(I + 1),y) if y = #(I') $\wedge$ (I = I' $\vee$ I $\neq$ I')
leq#(#(I),y)	gt#(mod4(I + 1),y) if y = #(I') $\wedge$ I = I'
gt#(#(I),y)	gt#(mod4(I + 1),y)
leq#(even,y)	leq#(odd,y) if y = odd $\vee$ y = even
leq#(even,y)	gt#(odd,y) if y = even
leq#(odd,y)	leq#(even,y) if y = even $\vee$ y = odd
leq#(odd,y)	gt#(even,y) if y = odd
gt#(even,y)	gt#(odd,y)
gt#(odd,y)	gt#(even,y)

Figura 4.9: Definición del operador Java ++ (post-incremento) de enteros de AbstLeqN

```
Solution 1 (state 2)
states: 3  rewrites: 571 in 779ms cpu (591ms real) (0 rewrites/second)
X:Output --> pl(even)
```

Este resultado certifica que toda posible ejecución que comienza con un entero n tal que n *mod* 4 = 0 siempre retorna un valor par(even). Es más, se puede verificar que las invocaciones iniciales "System.out.println(summation(0))" y "System.out.println(summation(3))" siempre retornan even mientras que "System.out.println(summation(1))" y "System.out.println(summation(2))" retornan odd.

```
--- To keep in the continuation the abstract relation value
op ++' : Value Location Value -> Continuation .
--- Keep in the continuation the abstract relation value and operates internal value
eq k(gt#(Val,Var) -> ++'(L) -> K) = k(Val -> ++ -> (++'(gt#(nullv,Var),L,gt#(Val,Var)) -> K)) .
--- Once evaluated operation with internal value build new resulting abstract relation value
eq k(Val -> ++'(gt#(nullv,Var),L,Val') -> K) = k([gt#(Val,Var) -> L] -> (Val' -> K)) .
--- Keep in the continuation the abstract relation value and operates internal value
eq t(k(leq#(Val,Var) -> ++'(L) -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      = t(k(leq#(Val,Var) -> ++ -> (++'(nullv,L,leq#(Val,Var)) -> K)) env(E [Var,L2]) Tc)
         store(St [L2,Val',N']) .
--- Once evaluated operation with internal value build new resulting abstract relation value
eq k(Val -> ++'(nullv,L,Val') -> K) = k([Val -> L] -> (Val' -> K)) .
--- Operations on internal abstract relation value
eq k(even -> ++ -> K) = k(odd -> K) .
eq k(odd -> ++ -> K) = k(even -> K) .
eq k(# I -> ++ -> K) = k(mod4(int(I + 1)) -> K) .
eq k(gt#(Val,Var) -> ++ -> K) = k(Val -> ++ -> (gt#(nullv,Var)-> K)) .
eq k(Val -> gt#(nullv,Var) -> K) = k(gt#(Val,Var) -> K) .
--- In case of leq# value there are three possible results
--- Abstract relation value do not change if Val =/= Val'
ceq t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
     = t(k(Val -> ++ -> (leq#(nullv,Var)-> K)) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      if (Val =/= Val')  .
--- Rules because abstract relation value could change if Val == Val'
crl [AUE01] : t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      => t(k(Val -> ++ -> (leq#(nullv,Var)-> K)) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      if (Val == Val')  .
crl [AUE02] : t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      => t(k(Val -> ++ -> (gt#(nullv,Var)-> K)) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
      if (Val == Val')  .
```

Figura 4.10: Ecuaciones, basadas en continuaciones, para el operador Java ++ (post-incremento) de enteros de `AbstLeqN`

4.2. Experimentos

En las primeras seis columnas a la izquierda en la Tabla 5.1, se estudian experimentalmente tres aspectos clave para la practicidad de la propuesta: el tamaño relativo del código y los certificados, el tamaño comparado de los certificados reducidos y completos, y la eficiencia relativa de la generación del certificado reducido versus la generación del certificado completo. Los experimentos han sido llevados a cabo en un Dell Inspiron 6000 con procesador Pentium a 1.6 Ghz y 1Gb de RAM.

Por otra parte, en las dos últimas columnas de la derecha se hacen estimaciones de la eficiencia relativa de la validación de los certificados contra su generación,

Ejemplos de Programas	Tamaño Código Fuente (bytes)	Tamaño Cert. Completo (Kbytes)	Tamaño Cert. Reducido (Kbytes)	C/R	Tiempo Gen. Cert. Comp. (ms)	Tiempo Gen. Cert. Redu. (ms)	Tiempo Valid. Cert. Comp. (ms)	Tiempo Valid. Cert. Redu. (ms)
even16	562	117	0.93	126	∼0	∼0	∼0	∼0
even16*	767	401	3.58	112	6	4	4	2
evenOdd	671	312	1.08	288	∼0	∼0	∼0	∼0
summation	870	1802	8.707	207	1723	109	1891	22.97
summationMod4	542	1217	7.844	155	1120	1277	1274	21.56

Cuadro 4.1: Tamaños del código y certificados, y tiempos de generación y validación de los certificados

con base en experimentos comparativos sobre los tiempos de ejecución de un proceso que hace ajuste de términos ("matching") con elementos hipotéticos de los certificados completos y reducidos.

Los programas even16, evenOdd, summation y summationMod4 son los programas Java con los métodos de los Ejemplos 3, 4, 5, y 6, respectivamente. El programa even16* ejecuta computaciones aritméticas más intensivas que even16, incluyendo resta y multiplicación aunque retorna el mismo valor. La primera columna contiene el tamaño en bytes del código fuente de cada programa. Las tres columnas para los certificados completos muestran su tamaño en Kbytes, y los tiempos de generación y validación, respectivamente. De manera similar las columnas para los certificados reducidos. Los tiempos de ejecución están dados en milisegundos y son el resultado de promediar sobre un número suficiente de iteraciones. Las cifras muestran que la reducción en el tamaño del certificado es significativa en todos los casos. Esta reducción (el cociente entre el tamaño completo y el tamaño reducido) oscila entre 112 para even16* hasta 288 para evenOdd.

Cuando se comparan los tiempos de generación de los certificados completos y reducidos, contra los tiempos estimados de validación correspondientes, se tiene que el tiempo de validación, según el estimativo, se reduce hasta en un factor de 50 %. Entonces se puede concluir que al minimizar el número de ecuaciones en el certificado, se obtiene un certificado de menor tamaño y más simple que podría ser verificado más eficientemente.

4.3. Trabajos Relacionados

En cuanto a propiedades de alto nivel basadas en tipos, que no sean verificables por el compilador estándar de un lenguaje, como la política de resultados "even", en teoría se podrían emplear las herramientas de verificación estándar JML de Java pues esta política se puede escribir en JML sin usar valores abstractos, sino el predicado que establece la condición que deben cumplir los valores del programa Java. La herramienta ESC/Java2 [Chalin et al., 2005] soporta código Java 1.4 con anotaciones JML pero es incompleta e incorrecta, no tiene una axiomatización aritmética que le permita razonar acerca de programas con computaciones enteras [Burdy et al., 2005] y no genera las demostraciones de las verificaciones positivas. La herramienta LOOP está basada en una formalización de Java y de JML y es correcta y completa pero no es completamente automática y requiere del asistente interactivo de demostraciones PVS [Burdy et al., 2005]. La herramienta BOGOR [Robby et al., 2006] usa JML para la especificación de propiedades que son verificadas mediante comprobación de modelos y abstracción [Robby et al., 2006] pero tampoco genera las demostraciones de las verificaciones positivas. El trabajo [Wu et al., 2003] presenta un sistema de tipos para verificar y certificar la política de resultados "even"en el marco de FPCC, pero para código de bajo nivel. De allí se tomó prestado el Ejemplo 3.

5

No Interferencia

En este capítulo se extiende la metodología propuesta en el Capítulo 4 y publicada en [Alba-Castro et al., 2008], para la certificación de *confidencialidad de información* mediante el análisis de la *no interferencia*. La confidencialidad es la propiedad en la que la información relacionada con una entidad o parte, no está disponible al público, o no es revelada a personas, entidades o procesos no autorizados [ATIS, 2001].

La forma estándar de proteger la confidencialidad de los datos es mediante la implementación de una *política de control de acceso* [Bishop, 2004] que restringe el acceso a los objetos dependiendo de la identidad o del rol que lleva a cabo el interesado. Esto significa que se requiere de un privilegio para acceder a la información confidencial. Sin embargo, un programa autorizado para acceder datos confidenciales o secretos, puede filtrarlos de una forma inapropiada, a propósito o accidentalmente. Esta filtración no autorizada puede ocurrir aún cuando se estén usando claves criptográficas, verificación de firmas digitales, programas anti-virus o anti-espías. Para asegurar que los programas no revelen datos secretos y cumplan *políticas de confidencialidad de datos* es entonces necesario analizar y controlar cómo fluye la información dentro de un programa. Lo anterior a llevado a denominar las políticas de confidencialidad como *políticas de flujo de información* [Denning and Denning, 1977; Sabelfeld and Myers, 2003]. Usualmente se les permite a los programas acceder y modificar datos privados en la medida en que sus salidas visibles no den información acerca de tales datos. Una manera de mantener la privacidad o el secreto de los datos confidenciales de un usuario, es establecer que los datos públicos visibles a otros usuarios no puedan ser afectados por los datos confidenciales de ése usuario. Esta política es

denominada una *política de no interferencia* porque los datos privados y secretos de un usuario no pueden interferir con datos no secretos y públicos [Goguen and Meseguer, 1982; Sabelfeld and Myers, 2003].

La no interferencia de programas es una propiedad de seguridad de alto nivel que garantiza que no hay flujos de información ilícitos durante la ejecución de los programas. La no interferencia diferencia entre entradas públicas y privadas y salidas públicas y privadas, y requiere que las salidas públicas no dependan de los valores de entradas privadas [Barthe et al., 2004]. El presente enfoque sigue la evaluación estándar de la confidencialidad [Denning and Denning, 1977; Sabelfeld and Myers, 2003]: Las variables (y sus valores) están etiquetadas con niveles de confidencialidad (desde un nivel bajo, las públicas, hasta un nivel alto, las secretas o privadas), las constantes tienen una etiqueta de confidencialidad no secreta porque son datos públicos, y la etiqueta de confidencialidad de las expresiones es la menor cota superior ("join") de las etiquetas de las subexpresiones [Denning and Denning, 1977; Barbuti et al., 2002a; Jacobs et al., 2005; Hunt and Sands, 2006; Francesco and Martini, 2007]. También se considera que los valores pueden cambiar dinámicamente su etiqueta de confidencialidad como en [Barbuti et al., 2002a; Jacobs et al., 2005; Hunt and Sands, 2006], e igualmente se permiten brechas temporales a la confidencialidad, de forma que no se tienen etiquetas de valores monótonamente crecientes y se evita el problema del crecimiento gradual de las etiquetas (label creep) [Sabelfeld and Myers, 2003]. Se usa la etiqueta de confidencialidad usual para el contexto o el contador del programa [Denning and Denning, 1977; Barbuti et al., 2002a; Sabelfeld and Myers, 2003; Jacobs et al., 2005; Hunt and Sands, 2006] que permite controlar los flujos indirectos de información en las instrucciones de flujo de control, como condicionales, iteraciones, invocaciones y retornos de procedimientos y funciones.

En este capítulo se trata la certificación de la confidencialidad de datos de programas Java, en particular en el caso en que los programas certificados tienen acceso a los datos privados. Esto significa que se usa la noción de no interferencia de [Sabelfeld and Myers, 2003; Barthe and Rezk, 2005; Warnier, 2005; Dufay et al., 2005], para asegurar que los datos privados no afectan los datos públicos. No se considera el aspecto de integridad de la no interferencia señalado en [Warnier, 2005], de manera que se permite que las variables de bajo nivel de confidencialidad

(públicas) influencien las variables de alto nivel (secretas). Tampoco se considera el flujo controlado de información de variables de alto nivel hacia variables de bajo nivel, es decir la desclasificación.

El problema de la verificación y certificación de la no interferencia de programas usando análisis de flujos de información, fué considerada por primera vez en [Denning and Denning, 1977]. La política de flujos de información es representada usualmente como una relación de flujo entre clases de seguridad que especifican los flujos permitidos entre las clases. Cada objeto de almacenamiento (constante, variable escalar, arreglo o archivo) tiene asignada una clase de seguridad. Esta asignación es estática y es inferida de las declaraciones del programa. Una política de no interferencia significa que las variables tienen niveles de confidencialidad fijos y que las entradas con nivel de confidencialidad alto no influencian las salidas con nivel de confidencialidad bajo. [Sabelfeld and Myers, 2003; Barthe and Rezk, 2005; Warnier, 2005; Dufay et al., 2005]. Esto significa que los valores almacenados en las variables con nivel de confidencialidad alto no pueden fluir hacia variables con niveles de confidencialidad menores. Se asume que las constantes que aparecen en los programas fuente (Java por ejemplo), siempre tienen el nivel de confidencialidad más bajo, pues el programa tiene acceso a datos secretos, pero él mismo no contiene datos secretos. Una política de no interferencia puede representarse con la relación $\langle L, \leq \rangle$ y la función de etiquetado $Lab : Var \rightarrow L$, donde L es el conjunto finito de niveles de confidencialidad, $\leq$ representa el orden parcial entre los niveles de confidencialidad, y Var es el conjunto finito de las variables del programa. Usualmente hay dos niveles de confidencialidad, i.e., $L = \{\texttt{Low}, \texttt{High}\}$, que representan los datos no secretos y públicos (baja confidencialidad) y los secretos y privados (alta confidencialidad), de manera que $\texttt{Low} \leq \texttt{High}$. $\langle L\ , \leq \rangle$ forma un retículo, donde `Low` es la mayor de las cotas inferiores o *ínfimo* ("bottom") ($\bot$), `High` es la menor de las cotas superiores o *supremo* ($\top$), y el operador *menor cota superior* ($\sqcup$) está definido como $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$ y, para otros casos, $X \sqcup Y = \texttt{High}$. Esto significa que los valores de las variables etiquetadas `Low` pueden fluir hacia las variables etiquetadas `High`, pero también que los valores de las variables etiquetadas `High` no pueden fluir hacia las variables etiquetadas `Low`.

Para expresar las políticas de seguridad de no interferencia para asegurar

la confidencialidad, se usa el lenguaje JML presentado en la sección 3.1. Cada variable del código Java está anotada con una etiqueta que representa el nivel de confidencialidad de la variable. El valor inicial de una variable es etiquetado con la etiqueta de la variable, pero durante la ejecución del programa puede cambiar la etiqueta del valor. Para referirse a la etiqueta de confidencialidad de una variable o de un valor de una variable se usa la función `AbsValue(Variable)` ya utilizada para las políticas de seguridad de propiedades basadas en tipos en el Capítulo 4, pero que en este caso implementa la función de etiquetado de variables *Lab* .

Para motivar este capítulo se presentan varios ejemplos Java tomados de la literatura relacionada, que incluyen la especificación JML de la política de no interferencia.

El flujo de información en un programa puede ser explícito (directo) o implícito (indirecto). Un flujo explícito es causado por instrucciones de asignación. Un flujo explícito ilegal es ocasionado por la asignación de valores de expresiones con variables etiquetadas con niveles altos, a variables etiquetadas con niveles bajos [Sabelfeld and Myers, 2003; Hunt and Sands, 2006], como en el ejemplo siguiente.

Ejemplo 11. Considérese el programa simple Java con dos instrucciones de asignación, tomado de [Jacobs et al., 2005], que incluye las cláusulas JML `requires` y `ensures` y el operador `\result`. El patrón de especificación de no interferencia propuesto en [Jacobs et al., 2005] puede aplicarse sin modificar el programa, solo en el caso en que los valores de las variables etiquetadas alto `High` y bajo `Low` sean `1` y `0`, respectivamente. Como no es el caso pues el valor retornado es 2, el patrón no se puede aplicar a este ejemplo sin modificarlo. Este ejemplo tiene un flujo directo ilegal de la variable `high` con nivel de confidencialidad `High` hacia la variable `low` con etiqueta `Low` en la primera instrucción de asignación. Sin embargo, como posteriormente, en la segunda instrucción de asignación el valor asignado a la variable `low` es el de una constante con etiqueta `Low`, el valor retornado como salida es legal.

```
public int mE1(int high, int low) {
/*@      requires AbsValue(high) == High && AbsValue(low) == Low;
  @       ensures AbsValue(\result) == Low;                                    @*/
      low = high;
      low = 2;
      return low;
}
```

Otro flujo explícito o directo ocurre en las invocaciones a procedimientos y funciones cuando estas tienen parámetros, como se muestra en el siguiente ejemplo.

Ejemplo 12. Consideremos el siguiente programa Java tomado de [Warnier, 2005] cuyo método `mE522` invoca el método `decrementing` con dos parámetros. El flujo ilícito ocurre en la invocación a `decrementing`, en la que se pasa la variable `high` (etiquetada con `High`) al parámetro `i` (etiquetado con `Low`).

```
int decrementing(int high,int i)  {
      high = high - 1;
      return i ;
}
int mE522(int high,int low)  {
/*@            requires AbsValue(high) == High && AbsValue(low) == Low;;
  @            ensures  AbsValue(\result) == Low;                                    @*/
      low = decrementing(high,high);
      return low;
}
```

La causa común de los flujos implícitos o indirectos ilícitos, que pueden pasar desapercibidos [Sabelfeld and Myers, 2003; Hunt and Sands, 2006], son las instrucciones de flujo de control guardadas por expresiones booleanas con variables etiquetadas `High`, como se muestra en el siguiente ejemplo.

Ejemplo 13. El programa Java que se presenta, tomado de [Warnier, 2005], tiene una instrucción de flujo de control condicional (`if`). Si el valor pasado en el parámetro `low` no es `0` y el valor retornado es `0`, entonces se llega a saber que la variable secreta `high` tiene un valor mayor que `2`.

Nótese que la noción de una etiqueta de confidencialidad global (denominada etiqueta del contexto) que sea actualizada en cada expresión condicional es necesaria para la verificación apropiada de tales flujos implícitos [Denning and Denning, 1977; Jacobs et al., 2005; Hunt and Sands, 2006].

```
/*@            requires AbsValue(high) == High && AbsValue(low) == Low;
  @            ensures  AbsValue(\result) == Low;                          @*/
public int mE2(int high,int low) {
          if (high > 2)
                    low = 0;
          return low;
}
```

Para evitar falsos positivos, se debe dinámicamente restaurar la etiqueta de confidencialidad global después de cada construcción condicional, como se requiere

para el siguiente ejemplo [Jacobs et al., 2005; Hunt and Sands, 2006].

Ejemplo 14. Considérese un programa que resulta de una pequeña modificación del programa del Ejemplo 13 en la que el valor retornado no depende del valor de la variable `high` etiquetada `High`. Esto es, la variable `j` es afectada indirectamente por el valor de la variable `high` pero no así la variable `low` usada en la expresión de retorno.

```
/*@          requires AbsValue(high) == High && AbsValue(low) == Low;
  @          ensures  AbsValue(\result) == Low;                                    @*/
public int mE2*(int high,int low) {
          int j=0;
          low = 0;
          /*@        assert j == Low;                                             @*/
          if (high > 2)
              j = 1;
          return low;
}
```

Ahora se presenta un ejemplo más realista con un condicional y un bucle que contiene un flujo indirecto e ilícito de información.

Ejemplo 15. El programa Java tomado de [Warnier, 2005] se presenta aquí transformado en una función con dos parámetros de entrada `high` y `low` y la misma política de seguridad de los ejemplos anteriores. Este ejemplo tiene también un flujo indirecto o implícito ilegal de la variable `high`, usada en la expresión booleana que guarda el bucle, hacia la variable `low`. Al terminar el bucle, la variable `low` tiene el valor de la variable `high`.

```
public int mE523(int high,int low)  {
/*@         requires AbsValue(high) == High && AbsValue(low) == Low;
  @         ensures AbsValue(\result) == Low;                                   @*/
    while (high > 0) {
        high--;
        low++;
    }
    return low;
 }
```

Se considera ahora un programa que es una variante del anterior, en el que se ha eliminado la expresión booleana del bucle y se reemplazó por la constante `true` cuya etiqueta es `Low`, pero se ha introducido al interior del bucle una instrucción condicional `if` cuya guarda sí tiene una expresión booleana con la variable `high`

etiquetada `High` y que además incluye la instrucción de escape del bucle `break`, un caso no considerado en [Warnier, 2005] ni en [Jacobs et al., 2005].

Ejemplo 16. El ejemplo también tiene un flujo de información indirecto e ilícito de la variable secreta `high` hacia la variable pública `low` usada en la expresión de retorno. Pero en este caso el flujo ilegal e indirecto ocurre en la instrucción `if`.

```
public int mE523v3(int high,int low)   {
/*@          requires AbsValue(high) == High && AbsValue(low) == Low ;
  @          ensures AbsValue(\result) == Low;                                    @*/
          while (true) {
              high--;
              low++;
              if (low > high)
                  break;
          }
          return low;
}
```

5.1. Semántica de Java en Lógica de reescritura con Flujos de Información

En esta sección se desarrolla una semántica extendida con flujos de información del lenguaje Java en lógica de reescritura, con base en la semántica de Java en lógica de reescritura dada en [Farzan et al., 2007] y presentada en el Capítulo 3.3.

Se describe la versión extendida con análisis del flujo de información de la semántica de Java en lógica de reescritura con la teoría de reescritura $\mathcal{R}_{\text{Java}^{\text{E}}} = (\Sigma_{\text{Java}^{\text{E}}}, E_{\text{Java}^{\text{E}}}, R_{\text{Java}^{\text{E}}})$, $E_{\text{Java}^{\text{E}}} = \Delta_{\text{Java}^{\text{E}}} \uplus B_{\text{Java}^{\text{E}}}$ y su correspondiente relación de reescritura $\rightarrow_{\text{Java}^{\text{E}}}$. En la nueva semántica los datos del programa no solo consisten en sus valores concretos estándar, sino que además cada valor está decorado con su correspondiente etiqueta de nivel de confidencialidad.

El enfoque consiste en extender $\mathcal{R}_{\text{Java}}$ (aprovechando su modularidad) mediante la complementación adecuada del dominio concreto Value para considerar el dominio extendido Value $\times$ LValue.

Se introduce el género LValue para representar los valores `Low` y `High`. Se escribe `<Value,LValue>`, el par formado por valor concreto y su correspondiente etiqueta de nivel de confidencialidad. También se dan las versiones apropiadas de

las construcciones y operadores Java para el nuevo dominio extendido. Se tiene en cuenta que los símbolos `env` y `store` son los símbolos constructores usados por la semántica original de Java en lógica de reescritura para el ambiente del programa y el almacenamiento en memoria, respectivamente. El nuevo constructor `lenv` se usa para almacenar el nivel de confidencialidad global (etiqueta del contexto).

Para efectos del análisis del flujo de información para la confidencialidad, se consideran las siguientes expresiones Java como casos especiales de evaluación: los literales constantes, accesos de variables, operaciones binarias, las expresiones de asignación, los operadores unarios prefijos y postfijos, y las expresiones de retorno. Las expresiones de asignación y los operadores unarios prefijos y postfijos (cuyo efecto secundario es una asignación), se consideran de forma indirecta, pero con la misma ecuación para todos los casos (la que especifica las escrituras en memoria). Gracias a la modularidad del enfoque en lógica de reescritura para la formalización de la semántica de programas, los cambios requeridos para extender la semántica de la Sección 3.3 son incrementales y mínimos.

Los valores de las variables reciben un nivel de confidencialidad inicial, que es almacenado en la memoria cuando la variable o el parámetro es creado. Cualquier operación que escribe un valor de una variable en la memoria, almacena como etiqueta de nivel de confidencialidad del valor almacenado, la menor cota superior ("join") de la etiqueta del valor y de la etiqueta del contexto en ese momento, como se muestra en la Figura 5.1. La etiqueta de cualquier valor constante entero, mostrado en la Figura 5.2, es `Low` como es de esperar, ya que las constantes son datos públicos. La etiqueta de una variable es la etiqueta de confidencialidad de su valor en memoria, y entonces, las ecuaciones originales de la Figura 3.9 no necesitan ajuste alguno. La etiqueta del resultado de un operador "built-in"es la menor cota superior de las etiquetas de sus operandos, como se muestra en la Figura 5.3 para el operador $>$.

```
rl t(k([< Value,LValue > -> L] -> K) id(I) lenv(LEnv) TC) store([L, Value', -1] ST)
  => t(k(K) id(I) lenv(LEnv) TC) store([L, shared(< Value,LValue join LEnv >), -1] ST) .
```

Figura 5.1: Regla extendida de la escritura en memoria

Para el etiquetado dinámico del contexto, la etiqueta inicial del contexto de cualquier hilo es `Low` como es usual [Denning and Denning, 1977; Barbuti et al., 2002a; Jacobs et al., 2005; Hunt and Sands, 2006]. La invocación de métodos

```
eq k(i(I) -> K) = k(<int(I),Low> -> K) .
eq k(b(B) -> K) = k(<bool(B),Low> -> K) .
```

Figura 5.2: Ecuaciones para la evaluación extendida de constantes

```
eq k((<int(I),L1>, <int(I'),L2>) -> (> -> K)) = k(<bool(I > I'),L1 join L2> -> K) .
```

Figura 5.3: Ecuaciones del operador extendido Java >

propaga la etiqueta del contexto sin cambios tal como se propone en [Jacobs et al., 2005]. Las instrucciones de asignación y las expresiones no cambian la etiqueta del contexto. La etiqueta del contexto puede cambiar solo por causa de las instrucciones de flujo de control, para controlar los flujos indirectos o implícitos de información, como se muestra en la Figura 5.4. La etiqueta actual del contexto es almacenada en la continuación usando el nuevo operador de continuaciones `restoreLEnv`, el cual luego restaura la etiqueta previa del contexto al terminar de ejecutar la instrucción condicional; véase la última ecuación de la Figura 5.4. De acuerdo con [Denning and Denning, 1977; Barbuti et al., 2002a; Jacobs et al., 2005; Hunt and Sands, 2006], la evaluación de expresiones booleanas retorna un nivel de confidencialidad asociado al valor resultante `true` o `false` y, posiblemente, una etiqueta de contexto modificada. En este trabajo se actualiza la etiqueta del contexto para reflejar el nivel de confidencialidad retornado por la evaluación de la expresión booleana, y luego, las dos ramas de la instrucción condicional usarán la nueva etiqueta de confidencialidad del contexto en las actualizaciones de la memoria.

```
--- First evaluates the boolean expression
--- while keeping the then and else statements
eq k((if E S else S' fi) -> K) lenv(LEnv)
    = k(E -> if(S, S') -> restoreLEnv(LEnv) -> K) lenv(LEnv) .
--- If the result value is true -> the then statement
eq k(< bool(true), LValue > -> (if(S, S') -> K)) lenv(LEnv)
    = k(S -> K) lenv(LEnv join LValue) .
--- If the result value is false -> the else part
eq k(< bool(false), LValue> -> (if(S, S') -> K)) lenv(LEnv)
    = k(S' -> K) lenv(LEnv join LValue) .
---New equation to restore previous context label
eq k(restoreLEnv(LEnv) -> K) lenv(LEnv') = k(K) lenv(LEnv) .
```

Figura 5.4: Ecuaciones para la instrucción "`if-then-else`"

En la instrucción "`while`", los flujos indirectos también pueden ocurrir en la

evaluación de la expresión booleana que guarda el bucle, tal como se muestra en la Figura 5.5. Los únicos cambios en las ecuaciones de la Figura 3.14, son la inclusión del operador de continuaciones `restoreLEnv` justo antes del operador de continuaciones `popLStack` y el uso de valores booleanos extendidos. La última ecuación de la Figura 3.14 para el operador `while` y las ecuaciones de la Figura 3.15 para el símbolo `break` se mantienen sin cambios. Intuitivamente, cuando la ejecución de la instrucción "`while`"termina normalmente, se ejecuta el `restoreLEnv` y la etiqueta original del contexto es restaurada. La instrucción "`break`"(dentro de una construcción `while`) no ejecuta tal continuación `restoreLEnv` y no restaurará la etiqueta original del contexto, propagando así el contexto actual hacia afuera del bucle.

```
--- First, thread state is stacked in the loop stack and boolean expression is evaluated
eq t(k((while E S) -> K) lstack(Lstack) lenv(LEnv) TC)
    = t(k(E -> (while(E, noExp, S) -> restoreLEnv(LEnv) -> popLStack -> K))
       lstack(fsi((while(E, noExp, S) -> K), lenv(LEnv) TC) Lstack) lenv(LEnv) TC) .
---  If true enter loop body and updates context label
eq k(< bool(true),Lab > -> (while(E, El, S) -> K)) lenv(LEnv)
= k(S -> (El -> ; -> (E -> ( while(E, El, S) -> K)))) lenv(LEnv join Lab) .
---  If false do not enter while loop
eq k(<bool(false),Lab> -> (while(E, El, S) ->K)) = k(K) .
```

Figura 5.5: Ecuaciones basadas en continuaciones para la instrucción extendida "`while`"

La semántica extendida de la instrucción `return` considera no solo la etiqueta de confidencialidad del valor retornado, sino también la etiqueta de confidencialidad del contexto, tal como se muestra en la Figura 5.6.

```
eq t(k(< V,LValue > -> return -> K) holds(Ll') env(Env') lenv(LEnv)
       fstack(fsi(K', (holds(Ll) env(Env) TC))  Fstack) TC')
    =  t(k(releaseEnv(Env') -> release(Ll, Ll') -> (< V,LValue join LEnv > -> K'))
          holds(Ll) env(Env) fstack(Fstack) TC) .
```

Figura 5.6: Ecuaciones basadas en continuaciones para la instrucción "`return`"

El ejemplo siguiente ilustra la mecanización de la semántica extendida de Java.

Ejemplo 17. Se considera el programa Java del Ejemplo 11, junto con la invocación a la función `main` del Ejemplo 2. En el comando de búsqueda `search` abajo, se pregunta por los posibles valores retornados por la función Java `main` del Ejemplo 11 usando la semántica extendida con valores etiquetados. Una variable término denota el estado objetivo por alcanzar.

```
search in PGM-SEMANTICS-LABELED :
  java((preprocess(default class t('Safe1NonInterference) imports nil extends Object
  implements none {(public static) int 'mE1((int d('high)), (int d( 'low)))throws( noType)
  {((10 @ ('low = 'high ;)) 11 @ ('low = i(2) ;)) 12 @ return 'low ;} (public static) void
  'main (t('String) [] d('args))throws( noType) {5 @ ('System .'out .'println < 'mE1 <
  i(1), i(0) > > ;)}}) noType . 'main < new string [i(0)] > noVal))

=>! X:Output .

Solution 1 (state 1)
states: 2  rewrites: 248 in (7ms real) (0 rewrites/second)
X:Output --> pl(<int(2),Low>)
No more solutions.
```

El comando `search` retorna que una única posible traza de ejecución Java es posible, la cual lleva a obtener como resultado de la instrucción Java "`System.out.println(mE1(1,0));`" el valor extendido Java `<int(2),Low>`.

5.2. La Semántica Abstracta de Java para No Interferencia en Lógica de Reescritura

En esta sección se desarrolla una versión abstracta de la semántica extendida en lógica de reescritura, descrita por la teoría de reescritura $\mathcal{R}_{\text{Java}\#} = (\Sigma_{\text{Java}\#}, E_{\text{Java}\#}, R_{\text{Java}\#})$, $E_{\text{Java}\#} = \Delta_{\text{Java}\#} \uplus B_{\text{Java}\#}$ y su correspondiente relación de reescritura $\rightarrow_{\text{Java}\#}$.

Tal como se hizo en la Sección 5.1, el enfoque adoptado para la semántica abstracta de Java consiste en extender la teoría original $\mathcal{R}_{\text{Java}}$ (aprovechando su modularidad) mediante la abstraccción al dominio $\mathsf{LValue} = \{\mathtt{Low}, \mathtt{High}\}$, y la introducción de las versiones aproximadas de las construcciones y operadores Java apropiados para este dominio, teniendo como referencia la introducción de la formalización de la interpretación abstracta de la Sección 3.4..

En el enfoque presentado en la Sección 4.1 ([Alba-Castro et al., 2008]), solo se necesitaba una función abstracta para cada nombre de variable Java, $\alpha_{\mathtt{x}} : \wp(\mathsf{Value}) \rightarrow \wp(\mathsf{Value})$, que era extendida homomórficamente a la función abstracta al conjunto de estados del programa, $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$.

En esta sección, la función de abstracción $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ es una extensión homomórfica al conjunto de estados de la funcíon $2nd : \mathsf{Int} \times \mathsf{LValue} \rightarrow$

LValue, lo que significa que en la abstracción se descartan los valores concretos de los datos.

Esta abstracción puede ser extendida a otros dominios infinitos de datos como las cadenas, y los números reales.

De manera similar a la semántica abstracta de la sección 4.1 cuando muchos pasos de reescritura $\rightarrow_{\text{Java}^{\text{E}}}$ son emulados mediante un único estado de reescritura abstracta que lleva a un estado abstracto Java, y esos pasos de reescritura aplican diferentes reglas o ecuaciones, aquí se usa la concurrencia a nivel de Maude. Es decir que también se adicionan reglas a $R_{\text{Java}^{\#}}$ para reflejar las posibles evoluciones del sistema.

Ahora se define formalmente la relación de reescritura $\rightarrow_{\text{Java}^{\text{E}}}$ de manera similar que en la Sección 4.1, también denotando la abstracción de la regla $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ con $\alpha(\{l\} \rightarrow \{r\})$.

Definición 2 (Reescritura Abstracta)**.** Sea $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ una abstracción. Se define la versión aproximada de la reescritura $\rightarrow_{\text{Java}^{\#}} \subseteq \wp(\mathsf{State}^E) \times \wp(\mathsf{State}^E)$ con:

$$SSt_1 \rightarrow_{\text{Java}^{\#}} SSt_2 \quad \text{usando } \alpha(\{l\} \rightarrow \{r\}) \in (R_{\text{Java}^{\#}} \cup \Delta_{\text{Java}^{\#}})$$
$$\text{iff } \forall u \in \alpha(SSt_1), \exists v \in SSt_2 \text{ s.t.} \quad u \rightarrow_{\text{Java}^{\text{E}}} v, \text{usando } l \rightarrow r \in R_{\text{Java}^{\text{E}}} \cup \Delta_{\text{Java}^{\text{E}}}.$$

Se denota con $\rightarrow^*_{\text{Java}^{\#}}$ la extensión de $\rightarrow_{\text{Java}^{\#}}$ a muchos pasos de reescritura. El siguiente resultado se obtiene directamente de las propiedades de monotonía, idempotencia y extensibilidad del operador de clausura superior α.

Teorema 2 (Corrección & Completitud)**.** Sea $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ una abstracción. Sea $SSt_1, SSt_2 \in \wp(\mathsf{State}^E)$. Si $SSt_1 \rightarrow^*_{\text{Java}^{\#}} SSt_2$, entonces para todo $u \in \alpha(SSt_1)$, existe un $v \in SSt_2$ tal que $u \rightarrow^*_{\text{Java}^{\text{E}}} v$. Sea $St_1, St_2 \in \mathsf{State}^E$. Si $St_1 \rightarrow^*_{\text{Java}^{\text{E}}} St_2$, entonces existe un $SSt_3 \subseteq \wp(\mathsf{State})$ s.t. $\alpha(St_1) \rightarrow^*_{\text{Java}^{\#}} SSt_3$ y $St_2 \in SSt_3$.

Por consiguiente, en lo que sigue a continuación, se abstrae la semántica de la Sección 5.1 de forma que (i) cada par `<Value,LValue>` de las ecuaciones y reglas es aproximado por el segundo componente `LValue`; y (ii) aquellas ecuaciones cuya confluencia no se pueda demostrar[1] después de la transformación, son transfor-

[1]Véase el comprobador Church-Rosser para Maude disponible en `http://www.lcc.uma.es/~duran/CRC/`.

madas en reglas para reflejar las posiblemente diferentes reescrituras denotadas por un estado abstracto. Se adiciona una regla para manejar valores que son solo etiquetas de confidencialidad, que se muestra en la Figura 5.7. Como valor abstracto de un valor constante entero, se retorna, como es de esperar, `Low`, según se muestra en la Figura 5.8. Se debe notar que esto puede expresarse con una ecuación, ya que se preservan la coherencia y confluencia [Clavel et al., 2007]. La etiqueta de una variable es la etiqueta de su valor en memoria, luego se pueden mantener las ecuaciones originales de la Figura 3.9. La etiqueta de los operadores "built-in"de enteros, es la menor cota superior de los operandos (etiquetas) tal como se muestra en la Figura 5.9 para el operador >.

```
rl t(k([LValue -> L] -> K) TC) store([L, Value'] ST)  lenv(LEnv)
=> t(k(K) TC) store([L, LValue join LEnv] ST) lenv(LEnv) .
```

Figura 5.7: Regla abstracta para la escritura en memoria

```
eq k(i(I) -> K) = k(Low -> K) .
```

Figura 5.8: Ecuación abstracta para la evaluación de constantes

```
rl k((LValue1, LValue2) -> > -> K) => k(<bool(true),LValue1 join LValue2> -> K) .
rl k((LValue1, LValue2) -> > -> K) => k(<bool(false),LValue1 join LValue2> -> K) .
```

Figura 5.9: Reglas abstractas para el operador Java >

En cuanto a la semántica de los condicionales y la instrucción `while`, las ecuaciones de las Figuras 5.4 y 5.5 respectivamente, funcionan en la semántica abstracta. Debido a que los pares <Bool,LValue> son manejados por la instrucción `return`, su semántica abstracta es la misma de las ecuaciones de la Figura 5.6. Sin embargo, se necesita una ecuación adicional para retornar valores abstractos únicamente, es decir las etiquetas, la cual es mostrada en la Figura 5.10. Nótese que es casi igual a las ecuaciones de la Figura 5.6.

```
eq t(k(LValue -> return -> K) holds(Ll') env(Env') lenv(LEnv)
       fstack(fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll, Ll') -> LValue join LEnv -> K')
     holds(Ll) env(Env) lenv(LEnv) fstack(Fstack) TC) .
```

Figura 5.10: Ecuación abstracta para la instrucción `return`

El ejemplo siguiente ilustra la mecanización de la semántica abstracta de Java.

Ejemplo 18. Considérese el método Java junto con la invocación a la función `main` del Ejemplo 11. En el comando `search` abajo, se pregunta por todos los posibles valores retornados por la función Java `main` del Ejemplo 11.

```
search in PGM-SEMANTICS-ABSTRACT :
 java((preprocess(default class t('Safe1NonInterference) imports nil extends Object
 implements none {(public static) int 'mE1((int d('high)), (int d( 'low))) throws( noType)
 {((10 @ ('low = 'high ;)) 11 @ ('low = i(2) ;)) 12 @ return 'low ; } (public static) void
 'main (t('String) [] d('args))throws( noType) {5 @ ('System . 'out . 'println < 'mE1 <
 i(1), i(0) > > ;)}}) noType . 'main < new string [i(0)] > noVal))

=>! X:Output .

Solution 1 (state 1)
states: 2  rewrites: 248 in (7ms real) (0 rewrites/second)
X:Output --> pl(Low)
No more solutions.
```

El comando `search` retorna que solo una única y posible traza de ejecución abstracta Java es posible, y que produce como resultado de la instrucción Java "`System.out.println(mE1(1,0));`" el valor abstracto `Low`.

5.3. Experimentos

Ejemplos de Programas	Tamaño Código Fuente (bytes)	Tamaño Cert. Completo (Kb)	Tamaño Cert. Reducido (Kb)	Relación de Tamaño C/R	Tiempo Generación Cert.Comp. (ms)	Tiempo Generación Cert.Redu. (ms)
mE1	963	443	2.62	2.29	16	3.5
mE2*	935	561	4.65	4.97	213.5	28.5
mE2v1E1	897	615	4.74	4.42	267	47
mE2mE1	999	578	4.66	3.98	245.5	31
mE522v1	1250	604	2.91	1.67	14	3.5
mE3	855	553	4.55	4.33	377	57.5

Cuadro 5.1: Tamaños del fuente y los certificados, y tiempos de generación de los certificados

En la Tabla 5.1, se estudian tres aspectos clave para la practicidad de la propuesta: el tamaño del código fuente y de los certificados, el tamaño comparado de los certificados reducidos y completos, y la eficiencia relativa de la generación del

certificado reducido versus la generación del certificado completo. Los experimentos han sido llevados a cabo en un Dell Inspiron 6000 con procesador Pentium a 1.6 Ghz y 1Gb de RAM.

Los programas Java mE1 and mE2* contienen los métodos de los Ejemplos 11, y 14, respectivamente. Los programas Java mE2mE1 y mE2v1E1 contienen métodos que son respectivamente, una composición secuencial de los métodos de los Ejemplos 13 y 11, y una composición secuencial de una variación del método del Ejemplo 13 (con una parte "else") con el método del Ejemplo 11. El programa Java mE522v1 es una variación del método del Ejemplo 12 en la cual la invocación del método decrementing se modifica para cumplir la política de seguridad. El programa mE3 tomado de [Jacobs et al., 2005], es similar al programa mE2mE1 pero usa el operador relacional == en la guarda de la instrucción condicional.

Las columnas "Tamaño Cert. Completo" y "Tiempo Generación Cert. Comp." muestran los tamaños en Kbytes y los tiempos de generación de los certificados completos, respectivamente. De forma similar para las dos columnas "Tamaño Cert. Reducido" y "Tiempo Generación Cert.Redu.". Los tiempos de ejecución están dados en milisegundos y son promedios de un número suficiente de iteraciones.

Estos ejemplos de prueba, están disponibles, juntos con otros más, en la url de la herramienta. Estos experimentos son prometedores, pues reafirman los resultados de los experimentos del Capítulo 4 y muestran que la reducción en el tamaño del certificado es significativa en todos los casos, en un rango entre 8,2 % en mE2* hasta 4,8 % para mE522v1. Cuando se comparan los tiempos de generación de los certificados completos y reducidos, se tiene que el tiempo de generación del certificado reducido toma solo un 12 % del tiempo de generación del certificado completo.

Entonces, también en este caso de la certificación de la no interferencia, al reducir el número de ecuaciones en el certificado, se obtiene un certificado de menor tamaño y más simple.

5.4. Trabajos Relacionados sobre No Interferencia

Las herramientas de verificación para Java que usan JML [Leavens et al., 2006] como lenguaje de especificación de propiedades no soportan la certificación de no interferencia. Algunas políticas sofisticadas de no interferencia pueden expresarse usando extensiones del lenguaje JML y verificarse también con extensiones de la herramienta de verificación Java Krakatoa [Dufay et al., 2005]. Estas extensiones JML fueron desarrolladas para aserciones en el estilo Hoare pero tomando en cuenta la autocomposición secuencial de programas [Barthe et al., 2004], lo cual significa duplicar el código de los programas y requiere así distinguir en la especificación las mismas variables del programa en las dos ejecuciones. Sin embargo, las políticas de no interferencia que requieren el etiquetado de las variables de datos con niveles de confidencialidad no se pueden expresar usando estas extensiones. El aspecto de la confidencialidad de la no interferencia es expresable usando los patrones de especificación JML sugeridos en [Jacobs et al., 2004; Jacobs et al., 2005; Warnier, 2005], donde se usan herramientas estándares de verificación JML y el demostrador de teoremas PVS. Desafortunadamente, los patrones propuestos abusan de la notación al identificar los niveles de confidencialidad con los valores concretos de las variables del programa, y no consideran características importantes de los programas Java como excepciones, retorno de valores en métodos que son funciones, interrupciones de instrucciones condicionales y bucles (con escapes `break`, retornos `return` o `continue`). Más aún, el patrón de especificación de confidencialidad no se puede aplicar en todos los casos como se menciona en [Warnier, 2005].

La extensión a la metodología de certificación de programas Java para la no interferencia, permite considerar algunas características de Java no tratadas en [Warnier, 2005] ni en [Jacobs et al., 2005]: campos de objetos, variables locales y vectores. Así mismo, trata los valores entregados por la instrucción `return`, caso no considerado en [Denning and Denning, 1977; Barbuti et al., 2002a; Jacobs et al., 2005; Hunt and Sands, 2006]. También se consideran instrucciones `return` y `break` dentro de instrucciones condicionales e iteraciones. Finalmente, en las invocaciones a métodos se propaga la etiqueta del contexto tal como se propone

en [Jacobs et al., 2005], donde no lo implementan. En relación con el nivel de confidencialidad en las instrucciones de asignación y en las expresiones unarias con efectos secundarios, se toma en cuenta el flujo indirecto de información mediante la consideración del nivel de confidencialidad del contexto como en [Denning and Denning, 1977; Barbuti et al., 2002a; Sabelfeld and Myers, 2003; Jacobs et al., 2005; Hunt and Sands, 2006], con la diferencia que aquí se maneja con solo una ecuación, que especifica las escrituras en la memoria, y con mayor granularidad, mediante la inferencia durante la actualización de la memoria.

Respecto de los valores entregados por una instrucción `return`, el presente trabajo es similar a [Barbuti et al., 2002b] en cuanto a la especificación de la instrucción bytecode `ret`, y a la evaluación de expresiones y valores en [Francesco and Martini, 2007]. Aunque la no interferencia no ha sido considerada en las actuales implementaciones de PCC, existen algunas propuestas aún no implementadas, que se basan en sistemas de tipos, para subconjuntos de Java [Barthe et al., 2006], de bytecode Java [Rose, 2003; Barthe and Rezk, 2005; Barthe et al., 2007], y algunos lenguajes simples imperativos [Hunt and Sands, 2006; Beringer and Hofmann, 2007]. Sin embargo ninguna de esas propuestas utiliza JML para expresar las políticas de no interferencia. [Barthe et al., 2006] propone un sistema de tipos basado en flujos de información para un fuente Java con objetos, herencia, métodos y excepciones simplificadas, de manera que se pueda desarrollar un compilador que preserve la información de los tipos. [Barthe et al., 2007] define el primer sistema de tipos basado en flujos de información para un lenguaje secuencial tipo JVM con clases, objetos, vectores, excepciones e invocaciones de métodos, que garantiza la no interferencia de programas cuyo tipo sea comprobado. La corrección fué demostrada usando Coq, y de la demostración se extrajo un verificador certificado de no interfererencia de bytecode que puede ser usado del lado del consumidor de código como un validador PCC. Ese trabajo usa un conjunto con múltiples niveles de confidencialidad. Aunque el presente trabajo no usa múltiples niveles de confidencialidad, puede ser extendido fácilmente para manejar múltiples niveles de confidencialidad. Las políticas locales que se manejan en el trabajo de esta tesis son flexibles, pues los niveles de confidencialidad de las variables locales y parámetros de los métodos pueden cambiar temporalmente, mientras en otros trabajos [Hunt and Sands, 2006; Jacobs et al., 2005] son fijas

y limitan el conjunto de programas que se podrían verificar positivamente.

También existen propuestas para la verificación de la no interferencia que se basan en análisis de flujos de información mediante interpretación abstracta [Barbuti et al., 2002a; Barbuti et al., 2002b; Giacobazzi and Mastroeni, 2004; Zanotti, 2002]. Pero, estas propuestas no generan una demostración como resultado de la verificación ni usan JML para expresar las políticas de seguridad. La idea de primero extender la semántica del lenguaje mediante el emparejamiento de cada valor de datos con su nivel de confidencialidad, y posteriormente considerar solo el nivel de seguridad en la abstracción, está también en [Barbuti et al., 2002a]. Una idea similar es desarrollada en [Francesco and Martini, 2007] donde los canales de entrada están asociados con niveles de seguridad. [Zanotti, 2002] diseña un sistema de tipos para análisis de flujos de información en términos de una semántica abstracta que aproxima la semántica de colección ("collecting semantics") de programas imperativos.

[Giacobazzi and Mastroeni, 2004] introduce la noción de no interferencia abstracta, que resulta de relajar la noción estándar de no interferencia parametrizándola en relación con abstracciones de las entradas y salidas. En la no interferencia abstracta los dominios abstractos codifican los flujos permitidos que caracterizan el grado de precisión del conocimiento de un atacante potencial que puede observar los datos de entrada y salida.

Para verificar la no interferencia de programas fuentes Java hay otras propuestas que no usan tampoco JML para especificar políticas de flujos de información sino extensiones al lenguaje Java, como son los compiladores JFlow [Myers, 1999] y Jif [Myers et al., 2001]. Estos compiladores producen código Java seguro para los programas escritos en los lenguajes JFlow and Jif cuyos tipos basados en flujos de información sean comprobados. JFlow hace además análisis dinámico cuando la etiqueta de un valor no se puede determinar estáticamente.

6

Consumo Acotado de Recursos

Estas propiedades se refieren a que el programa, las clases o algunos de los métodos consuman una cantidad de recursos físicos como tiempo de proceso y memoria del "heap"de forma acotada. Inicialmente, esta propiedad solo se consideraba en sistemas de tiempo real o en sistemas embebidos [Gustafsson, 2002; Gustafsson et al., 2006] en los que los recursos físicos estaban limitados, pero en la última década, en las aplicaciones Web también surgió la necesidad de establecer este tipo de políticas dada la frecuente ocurrencia de los ataques a sitios Web denominados "DoS"(Denial of Service) por la respuesta que ocasionan en los servidores Web al saturarlos por el consumo excesivo de sus recursos. Desde PCC ya son varias las aproximaciones a la seguridad del código móvil desde esta perspectiva [Crary and Weirich, 2000; Mok and Yu, 2002b; Yu and Mok, 2004; MacKenzie and Wolverson, 2004; Albert et al., 2005d; Albert et al., 2007a; Chander et al., 2007]. También surgió la necesidad de acotar el consumo de recursos en relación con los dispositivos móviles que cuentan con recursos limitados [Aspinall et al., 2004b].

Para motivar este trabajo consideremos programas Java simples, tomados prestados de la literatura relacionada.

Ejemplo 19. Se considera un programa Java recursivo y simple que computa los números de Fibonacci, del cual se analiza su versión en bytecode en [Albert et al., 2007b]. Este programa consume tiempo de proceso y memoria del "heap"en cada invocación recursiva de forma exponencial en función del parámetro de entrada.

Para expresar la política usamos JML y dos variables de la especificación: `ExecTime` para referirnos al tiempo de ejecución del método, y `MaxMemHeap`, para indicar la máxima cantidad de memoria del "heap"consumida por el método. En

[Barthe et al., 2005] usan extensiones similares de JML para expresar políticas de consumo de memoria de programas bytecode Java. El tiempo de ejecución `ExecTime`, es contabilizado aquí de la misma forma que en [Aspinall et al., 2004a; Aspinall and MacKenzie, 2006], es decir, contando el número de instrucciones bytecode que ejecutaría el método, de acuerdo a la traducción del compilador Java estándar. No se toma en cuenta la optimización que hace el compilador, por ejemplo cuando reemplaza expresiones con operadores constantes por su resultado. Como la política establece una cota superior, es aceptable ésta aproximación. La máxima cantidad de memoria del "heap"se contabiliza también de forma similar a cómo lo hacen en [Aspinall et al., 2004a; Hofmann et al., 2005; Aspinall and MacKenzie, 2006].

```
static  int fib(int i) {
/*@      requires true;
  @       ensures ExecTime <= 25 + 4 * (2 ** (2 *\old(i))) &&
  @                    MaxMemHeap <= \old(i) ;                                    @*/
          if ((i==1)  ||  (i==0))
                  return 1;
          else
                  return ( fib(i-1) + fib(i-2) );
}
```

Ahora se considera otro ejemplo de un método Java que hace operaciones con matrices.

Ejemplo 20. El método Java multiplica dos matrices y también analizado en [Albert et al., 2007b], pero aquí es presentado depurado. En este caso el tiempo de ejecución y el máximo consumo del "heap"son proporcionales a las dimensiones de las matrices.

```
public int[ ][ ] mult(int[ ][ ] a,int[ ][ ] b, int r, int cr, int c)      {
/*@    requires true;
  @    ensures  ( ExecTime <= 10 + 35 * (\old(r) * \old(cr) * \old(c) ) ) &&
  @        ( MaxMemHeap <= 6 + \old(r)* \old(cr) + \old(cr)* \old(c) + \old(r)* \old(c)); @*/
            int[ ][ ] c1 = new int[r][c];
            for(int i=0; i < r;i++)
                for( int j =0; j < c; j++)
                     for (int k=0; k < cr ; k++)
                          c1[i][j] = c1[i][j] + (a[i][k] *b[k][j]);
            return c1;
}
```

6.1. Semántica de Java en Lógica de reescritura con Consumo de Recursos

En esta sección se desarrolla una semántica extendida con consumo de recursos del lenguaje Java en lógica de reescritura, con base en la semántica de Java en lógica de reescritura dada en [Farzan et al., 2007] y presentada en el Capítulo 3.3.

Se describe la versión extendida con consumo de recursos de la semántica de Java en lógica de reescritura con la teoría de reescritura $\mathcal{R}_{\text{Java}^{\text{ERC}}} = (\Sigma_{\text{Java}^{\text{ERC}}}, E_{\text{Java}^{\text{ERC}}}, R_{\text{Java}^{\text{ERC}}})$, $E_{\text{Java}^{\text{ERC}}} = \Delta_{\text{Java}^{\text{ERC}}} \uplus B_{\text{Java}^{\text{ERC}}}$ y su correspondiente relación de reescritura $\rightarrow_{\text{Java}^{\text{ERC}}}$.

En la nueva semántica los estados local y global del programa han sido decorados con el registro del consumo de recursos de cada método, de cada hilo y de todo el programa de forma similar a como se usa en [Aspinall et al., 2004a; Hofmann et al., 2005; Aspinall and MacKenzie, 2006]. El registro de consumo de recursos es una tupla (ResourceTuple) con cuatro elementos Clock × Callc × Invkc × InvkDpth, para el número de instrucciones bytecode, el número de invocaciones de funciones, el número de invocaciones de métodos y la máxima cantidad de memoria utilizada del "heap", respectivamente.

El enfoque consiste en extender $\mathcal{R}_{\text{Java}}$ (aprovechando su modularidad) mediante la complementación adecuada del estado global del programa JavaState para considerar el estado global extendido JavaState × ResourceTuple. El estado local ThreadCtrl es extendido con dos tuplas de consumo de recursos, una para el consumo del hilo y otra con el consumo del método actualmente en ejecución. La tupla de consumo del hilo está decorada con el identificador del hilo correspondiente, que es un número entero: ResourceTuple × Int. La tupla de consumo del método está a su vez decorada con el nombre del método Qid, su clase Type y los valores de los parámetros utilizados en la invocación del género ValueList: ResourceTuple × Type × Qid × ValueList. Así, el estado local extendido es: ThreadCtrl × ResourceTuple × Int × ResourceTuple × Type × Qid × ValueList. Para recordar los consumos de los métodos ejecutados antes del actual, así como los de los hilos terminados, también se extendió el estado global con un conjunto de tuplas de consumos de tales métodos e hilos.

Se introduce el género ResourceTuple para representar la tupla de consumo, y los géneros Clock, Callc, Invkc y InvkDpth, para los cuatro consumos considerados. Los nuevos constructores `clock`, `callc`, `invkc` y `invkdpth` registran cada consumo como números naturales, incluyendo el cero. Los constructores `resourcesM`, `resourcesT`, `resourcesP` registran las tuplas de consumos de los métodos, hilos y del programa, respectivamente. El constructor `resumeResources`, almacena la historia con los consumos de los hilos y métodos ejecutados anteriormente a los actualmente activos. También se dan las versiones apropiadas de las construcciones y operadores Java para el nuevo estado extendido.

Cuando se crea el estado global inicial del programa Java, se inicializa la tupla de consumos del programa en ceros (`resourcesP`), y la historia de consumos (`resumeResources`) de los hilos muertos y de los métodos cuya ejecución haya terminado es creada vacía, como se muestra en la Figura 6.1.

```
eq javaResources((Cl E Vl))
= runResources(t(k(buildS(Cl) -> (E -> stop)) obj(nullo) fstack(noItem) xstack(noItem)
   lstack(noItem) finalblocks(noItem) env(noEnv) id(-1) holds(nil) ) out(noOutput) in(Vl)
   store(noStore) code(Cl) static(onil) busy(noObj) nextLoc(0) nextTid(1)
   resourcesP(< clock(0),callc(0),invkc(0),invkdpth(0) >)
   resumeResources(noInvk)) .
```

Figura 6.1: Ecuación para crear e inicializar el estado global con la tupla de consumo del programa y su historia de consumos

A continuación se muestra en la Figura 6.2 la ecuación de la semántica extendida que termina la inicialización estática y da inicio a la ejecución del hilo principal identificado con el número 0. Allí se crea la tupla del hilo principal, con sus consumos inicializados en ceros.

```
--- the first equation ends the static initialization and starts the main thread
--- with its consumption tuple initialized
eq k(buildS(noClass) -> K) id(-1)
    = k(K) id(0) resourcesT(< clock(0),callc(0),invkc(0),invkdpth(0) >, id(0)) .
```

Figura 6.2: Inicio de la ejecución del hilo principal `id(0)` e inicialización de su tupla de consumo

El consumo de recursos de la invocación al método `main` se registra en la tupla del hilo principal (la evaluación de sus parámetros y la invocación en sí) y tal invocación causa que sea inicializada en ceros la tupla de consumo del método, tal como se muestra en la Figura 6.3.

```
--- main parameters evaluation resource consumption accounting
eq id(I') k(i(I) -> (inita(string[]) -> (call(C, 'main) -> stop)))
       resourcesT(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >, id(I') )
    = id(I') k(int(I) -> (inita(string[]) -> (call(C, 'main) -> stop)))
       resourcesT(< clock(Clock + 1),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >,
       id(I') )  .
--- main invocation accounting and main tuple initialization
eq t(k(Vl -> (call(C, 'main) -> K)) code(Cl)
    resourcesT(< clock(Clock'),callc(Callc'),invkc(Invkc'),invkdpth(InvkDpth') >, id(I)) tc)
     = t(k(findMethod(C, 'main, getTypes(Vl), Cl) -> (call(C, 'main, Vl) -> K))
    code(Cl)  resourcesM(< clock(0),callc(0),invkc(0),invkdpth(0) >, C, 'main, Vl )
    resourcesT(< clock(Clock'),callc(Callc'),invkc(Invkc' + 1),invkdpth(InvkDpth') >,
     id(I)) tc)   [owise] .
```

Figura 6.3: Inicialización de la tupla de consumo del método `main`

La ecuación que construye el ambiente debe ser también modificada para contar cada nueva posición de memoria que es asignada a una variable, tal como se especifica en la Figura 6.4. Nótese que no se contabiliza el uso de posiciones de memoria reutilizadas, debido que se quiere contabilizar el máximo usado tal como también se hace en [Aspinall et al., 2004a; Hofmann et al., 2005; Aspinall and MacKenzie, 2006].

```
--- We do not count reuse of used locations because we want the maximun used
eq  t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) store([L, V', -3] store)
     nextLoc(I')
     resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >, T' ,M, Vl' ) TC)
=  t(k(buildEnv(Pl, Vl) -> K) env([X, L] Env) id(I) store([L, setTid(V, I), I] store)
    nextLoc(I') resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >,
                                    T', M, Vl' ) TC)  .
---  count non used location
eq t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) store(store) nextLoc(I')
    resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >, T' ,M, Vl' ) TC)
 =  t(k(buildEnv(Pl, Vl) -> K) env([X, l(I' + 1)] Env) id(I) store([l(I' + 1), setTid(V, I), I]
     store) nextLoc(I' + 1)
     resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth + 1) >,
                      T', M, Vl' )  TC)  [owise] .
endm
```

Figura 6.4: Ecuación extendida para contar la asignación de una nueva posición de memoria al crear un elemento del ambiente

La escritura de la memoria así como su acceso se contabilizan en las ecuaciones y reglas de las Figuras 6.5 y 6.6 respectivamente. Cada lectura cuesta una instrucción (`push`), y cada escritura cuesta una instrucción (`pop`).

Los operadores Java son extendidos para contabilizar la instrucción bytecode

```
--- TO STORE
---  a location has the tid of -2 only when it is created but has not been assigned
---  to a variable in any thread, e.g., a newly created object
eq t(k([V -> L] -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
       invkdpth(InvkDpth) >, T, M, Vl ) TC) store([L, V', -2] ST)
= t(k(K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
       invkdpth(InvkDpth) >, T, M, Vl ) TC) store([L, V, -2] ST) .
---  a shared location
rl t(k([V -> L] -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V', -1] ST)
=> t(k(K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, shared(V), -1] ST) .
--- a non shared location
eq t(k([V -> L] -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)   store([L, V', I] ST)
= t(k(K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC) store([L, setTid(V, I), I] ST) .
```

Figura 6.5: Ecuaciones extendidas para escribir de la memoria

```
--- TO ACCESS
eq t(k(#(L) -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, -2] ST)
= t(k(V -> K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, -2] ST) .
--- a shared location
rl t(k(#(L) -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, -1] ST)
=> t(k(V -> K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, -1] ST) .
---  a non shared location
eq t(k(#(L) -> K) id(I) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, I] ST)
= t(k(V -> K) id(I) resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),
        invkdpth(InvkDpth) >, T, M, Vl ) TC)  store([L, V, I] ST) .
```

Figura 6.6: Ecuaciones extendidas para leer de la memoria

que requieren, como se muestra en la Figura 6.7 para la suma + binaria. El coste de leer los operandos de la memoria es contabilizado en las ecuaciones o reglas correspondientes de la Figura 6.6 mostrada arriba.

Cuando un método diferente a `main` es invocado, la tupla de recursos del método actual es almacenada en la continuación (con el constructor `resourcesK`), y se crea la tupla del método invocado inicializada en ceros, como se especifica en la Figura 6.8. Se diferencia la invocación de un método sin indicar su clase, que se contabiliza como un llamado a función (`callc`), de la invocación de un método indicando su clase, que se contabiliza como un llamado a un método (`invkc`).

```
eq k((int(I), int(I')) -> (+ -> K))
    resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >, T, M, Vl )
 = k(int(I + I') -> K)
    resourcesM(< clock(Clock + 1),callc(Callc),invkc(Invkc),invkdpth(InvkDpth) >, T, M, Vl ) .
```

Figura 6.7: Ecuación del operador suma + extendida para contar la instrucción bytecode requerida

```
--- instance method call
eq t(k(Vl -> (call(o(orig(C') Oattr), M) -> K))
       resourcesM(< clock(Clock'),callc(Callc'),invkc(Invkc'),invkdpth(InvkDpth') >,T',M', Vl')
       TC) code(Cl)
= t(k(findMethod(C', M, getTypes(Vl), Cl) -> (call(o(orig(C') Oattr), M, Vl) ->
     (resourcesK(resourcesM(< clock(Clock'),callc(Callc'),invkc(Invkc' + 1),
       invkdpth(InvkDpth') >,T',M', Vl'))  -> K)))
     resourcesM(< clock(0),callc(0),invkc(0),invkdpth(0) >, C', M, Vl ) TC) code(Cl) .
--- default method call
eq t(k(Vl -> (call(M) -> K)) obj(o(orig(C') Oattr))
      resourcesM(<clock(Clock'),callc(Callc'),invkc(Invkc'),invkdpth(InvkDpth') >,T',M', Vl')
       TC)  code(Cl)
 = t(k(findMethod(C', M, getTypes(Vl), Cl) -> (call(M, Vl) ->  (resourcesK(resourcesM(
      < clock(Clock'),callc(Callc' + 1),invkc(Invkc'),invkdpth(InvkDpth') >,T',M', Vl'))
      ->  K))) obj(o(orig(C') Oattr))
    resourcesM(< clock(0),callc(0),invkc(0),invkdpth(0) >, C', M, Vl ) TC) code(Cl) .
```

Figura 6.8: Ecuaciones extendidas de la invocación de un método distinto de `main`

Otros métodos cuya invocación es manejada de forma particular en la semántica de Java de la Sección 3.3, como son el método `run` invocado de forma automática al crear los hilos, y los constructores de las clases, tienen una especificación similar en la semántica extendida, al mostrado arriba en la Figura 6.8.

Cuando se crea e inicia la ejecución de un nuevo hilo, debe crearse e inicializarse en ceros su correspondiente tupla de consumo de recursos, tal como se especifica en la ecuación de la Figura 6.9.

```
eq t(k(sys(newThread) -> callSys(obj, noVal) -> K) tc) nextTid(I)
= t(k(K) tc) t(k(changeTid(obj, I) -> (call(obj, 'run) -> stop)) obj(obj) fstack(noItem)
     xstack(noItem) lstack(noItem) finalblocks(noItem) env(noEnv) id(I) holds(nil)
     resourcesT(< clock(0),callc(0),invkc(0),invkdpth(0) >, id(I))) nextTid(I + 1) .
```

Figura 6.9: Ecuación para iniciar la ejecución de un nuevo hilo, extendida para inicializar su tupla de recursos

Cuando un método termina su ejecución, su tupla de consumo de recursos debe, por un lado, adicionarse a los del método que lo invocó cuya tupla está en la continuación, y por otro lado, además debe registrarse en la historia del estado

global (`resumeResources`). Esto se muestra en las ecuaciones de la Figura 6.10 para el caso de métodos que retornan valores, recursivos y no recursivos.

```
--- return value from a recursive invocation
eq k(V -> (resourcesK(resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
       invkdpth(InvkDpth)>, T, M, Vl)) -> K)) resourcesM(< clock(Clock'),
       callc(Callc'), invkc(Invkc'), invkdpth(InvkDpth')>, T, M, Vl )
= k(V -> K) resourcesM(< clock(Clock + Clock' + 2),callc(Callc + Callc'),
     invkc(Invkc + Invkc'), invkdpth(InvkDpth + InvkDpth') >, T, M, Vl) .
--- return value from a non recursive invocation
eq t(k(V -> (resourcesK(resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
      invkdpth(InvkDpth)>, T, M, Vl)) -> K))  resourcesM(< clock(Clock'),callc(Callc'),
      invkc(Invkc'), invkdpth(InvkDpth')>, T',M', Vl') TC) resumeResources(TC')
= t(k(V -> K) resourcesM(< clock(Clock + Clock' + 2),callc(Callc + Callc'),
      invkc(Invkc + Invkc'), invkdpth(InvkDpth + InvkDpth') >, T, M, Vl) TC)
      resumeResources(resourcesM(<clock(Clock'),callc(Callc'),invkc(Invkc'),
      invkdpth(InvkDpth') >,T',M', Vl') TC')  [owise] .
```

Figura 6.10: Ecuaciones para la terminación de métodos que retornan valores

Cuando termina la ejecución de un hilo al encontrarse el constructor de continuación `stop`, su tupla de consumo de recursos debe ser adicionada al consumo de recursos del programa, y almacenada en la historia (`resumeResources`), como se especifica en las Figuras 6.11 y 6.12 para el caso de retorno de valor y no retorno de valor, respectivamente.

```
--- value
eq t(id(I) k(V -> stop) resourcesM(< clock(Clock),callc(Callc),invkc(Invkc),
      invkdpth(InvkDpth) >, T, M, Vl)  resourcesT(< clock(Clock''),callc(Callc''),
      invkc(Invkc''),invkdpth(InvkDpth'') >, id(I)) TC)
      resourcesP(< clock(Clock'),callc(Callc'),invkc(Invkc'),invkdpth(InvkDpth') >)
      out(Output) resumeResources(TC')
 = resourcesP(< clock(Clock'' + Clock' + Clock),callc(Callc'' + Callc' + Callc),
      invkc(Invkc'' + Invkc' + Invkc), invkdpth(InvkDpth'' + InvkDpth' + InvkDpth) >)
      out(Output, r(V)) resumeResources(resourcesT(< clock(Clock + Clock''),
      callc(Callc + Callc''),invkc(Invkc + Invkc''), invkdpth(InvkDpth + InvkDpth'') >,
      id(I) ) TC') .
```

Figura 6.11: Ecuación para la terminación de la ejecución de un hilo con retorno de valor

La mecanización de la semántica extendida para el consumo de recursos se ilustra con el siguiente ejemplo

Ejemplo 21. Se considera el programa Java del Ejemplo 19, junto con la invocación desde la función `main`:

```
--- no value
eq t(id(I) k(stop) resourcesM(< clock(Clock'') ,callc(Callc'') ,invkc(Invkc'') ,
       invkdpth(InvkDpth'') >, T, M, Vl) resourcesT(< clock(Clock) ,callc(Callc) ,
       invkc(Invkc) ,invkdpth(InvkDpth) >, id(I)) tc)
    resourcesP(< clock(Clock') ,callc(Callc') ,invkc(Invkc') ,invkdpth(InvkDpth') >)
    resumeResources(TC') State
= resourcesP(< clock(Clock'' + Clock' + Clock) ,callc(Callc'' + Callc' + Callc) ,
        invkc(Invkc'' + Invkc' + Invkc) , invkdpth(InvkDpth'' + InvkDpth' + InvkDpth) >)
   resumeResources(resourcesT(< clock(Clock + Clock'') ,callc(Callc + Callc'') ,
        invkc(Invkc + Invkc'') , invkdpth(InvkDpth + InvkDpth'') >, id(I) )
   resourcesM(< clock(Clock'') ,callc(Callc'') ,invkc(Invkc'') ,invkdpth(InvkDpth'') >,
       T, M, Vl) TC') State .
```

Figura 6.12: Ecuación para la terminación de la ejecución de un hilo sin retorno de valor

```
public static void main(String[] args)   {
         System.out.println(fib(5));
}
```

En el comando de reescritura `rew` abajo, se pregunta por el valor retornado por la función Java `main` del Ejemplo 19 y los consumos de recursos correspondientes al programa completo, el hilo principal y los métodos invocados `main`, `System.out.println` y `fib`.

```
rew in PGM-SEMANTICS-RESOURCE-TUPLE : javaResources((preprocess(default
    class t('Safe1Cost5) imports nil extends Object implements none {(default
    static) int 'fib(int d('n))throws(noType) {12 @ (if 'n == i(1) || 'n == i(
    0) 10 @ return i(1) ; else 12 @ return 'fib < 'n - i(1) > + 'fib < 'n - i(
    2) > ; fi)} (public static) void 'main(t('String)[] d('args))throws(noType)
    {5 @ ('System . 'out . 'println < 'fib < i(5) > > ;)}}) noType . 'main <
    new string [i(0)] > noVal)) .

rewrites: 1538 in 1039ms cpu (837ms real) (6 rewrites/second)
result JavaState: runResources(out(pl(int(8))) in(noVal) store([l(1),a(string,
    anil),0] [l(2),int(5),-3] [l(3),int(3),-3] [l(4),int(1),-3] [l(5),int(0),
    -3] [l(6),int(0),-3]) code(default class t('Safe1Cost5) extends Object
    implements none {(default static) int 'fib(int d('n))throws(noType) {12 @ (
    if 'n == i(1) || 'n == i(0) 10 @ return i(1) ; else 12 @ return 'fib < 'n -
    i(1) > + 'fib < 'n - i(2) > ; fi)} (public static) void 'main(string[] d(
    'args))throws(noType) {5 @ ('System . 'out . 'println < 'fib < i(5) > > ;)}
    public t('Safe1Cost5)(noPara)throws(noType) super(noExp){nop}}) static([t(
    t('Safe1Cost5)),f(noEnv)]) busy(noObj) nextLoc(6) nextTid(1)

    resourcesP(<clock(211),callc(15),invkc(2),invkdpth(6) >)
```

El comando `rew` retorna que el resultado de la instrucción Java "`System.out.println(fib(5));`" es el valor Java `int(8)`, y que el programa completo consume 211 instrucciones bytecode, hace 2 invocaciones a métodos, 15 a

funciones (son las invocaciones recursivas de `fib`) y utiliza 6 posiciones de memoria del "heap".

```
resumeResources(

resourcesT(<clock(211),callc(15),invkc(2),invkdpth(6) >, id(0))

resourcesM(< clock(0), callc(0),invkc(0),invkdpth(0) >, System, 'println, int(8))

resourcesM(< clock(7),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(1))
resourcesM(< clock(7),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(1))
resourcesM(< clock(7),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(1))
resourcesM(< clock(7),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(1))
resourcesM(< clock(7),callc( 0),invkc(0),invkdpth(1) >, t('Safe1Cost5), 'fib, int(1))
resourcesM(<clock(10),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(0))
resourcesM(< clock(10),callc(0),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(0))
resourcesM(< clock(10),callc(0),invkc(0),invkdpth(0) >, t( 'Safe1Cost5), 'fib, int(0))
resourcesM(< clock(37),callc(2),invkc(0), invkdpth(0) >, t('Safe1Cost5), 'fib, int(2))
resourcesM(< clock(37),callc( 2),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(2))
resourcesM(<clock(37),callc(2),invkc(0),invkdpth(2) >, t('Safe1Cost5), 'fib, int(2))
resourcesM(< clock(64),callc(4),invkc(0),invkdpth(0) >, t('Safe1Cost5), 'fib, int(3))
resourcesM(< clock(64),callc(4),invkc(0),invkdpth(3) >, t('Safe1Cost5), 'fib, int(3))
resourcesM(< clock(121),callc(8),invkc(0),invkdpth(4) >, t('Safe1Cost5), 'fib, int(4))
resourcesM(< clock(205),callc(14),invkc(0),invkdpth(5) >, t('Safe1Cost5), 'fib, int(5))

resourcesM(< clock(210),callc(15),invkc(1),invkdpth(6) >, t('Safe1Cost5), 'main,
                  a(string, anil))))
```

Además, se obtienen todos los consumos de recursos de las invocaciones a `fib`, incluyendo el consumo de la invocación inicial desde `main fib(5)`, que es `resourcesM(< clock(205),callc(14),invkc(0),invkdpth(5) >, t('Safe1Cost5), 'fib, int(5))`, lo que significa que se ejecutaron 205 instrucciones de bytecode, 14 invocaciones a funciones y se usaron máximo 5 posiciones de memoria del "heap".

Por otra parte, en el comando de búsqueda `search` abajo, se pregunta si como resultado de la ejecución de la función Java `main` existe alguna invocación del método `fib` en la que el tiempo de ejecución (medido como el número de instrucciones bytecode ejecutadas) es superior a la cota señalada en la cláusula JML, usando la semántica extendida con tuplas de consumo de recursos.

Nótese el uso de la condición que define el estado final inseguro `NC:Nat > 25 + 4 * 2 ^(I * 2) = true`, que corresponde exactamente con la negación de la cláusula JML que establece la cota superior del consumo.

```
search in PGM-SEMANTICS-RESOURCE-TUPLE : javaResources((preprocess(default
    class t('Safe1Cost5) imports nil extends Object implements none {(default
```

```
    static) int 'fib(int d('n))throws(noType) {12 @ (if 'n == i(1) || 'n == i(
    0) 10 @ return i(1) ; else 12 @ return 'fib < 'n - i(1) > + 'fib < 'n - i(
    2) > ; fi)} (public static) void 'main(t('String)[] d('args))throws(noType)
    {5 @ ('System . 'out . 'println < 'fib < i(5) > > ;)}}) noType . 'main <
    new string [i(0)] > noVal))
 =>! runResources(JS:JavaState resumeResources( TC resourcesM(<
    clock(NC:Nat),callc(NCa:Nat),invkc(NInC:Nat),invkdpth( NMaxMem:Nat) >,
    t('Safe1Cost5), 'fib, int(I))))

such that NC:Nat > 25 + 4 *  2 ^ (I * 2) = true .

No solution.
states: 2  rewrites: 1583 in 7723379976ms cpu (35ms real) (0 rewrites/second)
```

El comando `search` retorna que el consumo del tiempo de ejecución de ninguna de las invocaciones al método `fib`, incluida la invocación inicial "`System.out.println(fib(5));`", es superior a la cota indicada.

6.2. La Semántica Abstracta de Java para Consumo de Recursos en Lógica de Reescritura

La semántica abstracta de Java para consumo de recursos no ha sido implementada todavía. Hasta el momento, se ha explorado cómo hacer la abstracción en la literatura relacionada, encontrándose que se podrían utilizar como valores abstractos intervalos de enteros [Cousot and Cousot, 1976; Cousot and Cousot, 1978; Engblom et al., 2003] y poliedros [Cousot and Halbwachs, 1978; Lisper, 2003; Besson et al., 2007].

7

La Certificación de código fuente Java

Los Ejemplos 2, 8, 9, 10 de la Sección 4.1 y el Ejemplo 18 de la Sección 5.2 arriba presentados, ilustran la manera como la metodología propuesta genera un certificado de seguridad, el cual esencialmente consiste del conjunto de demostraciones de reescritura (abstractas) de la forma $t_1 \rightarrow^{r_1}_{\text{Java}\#} t_2 \cdots \rightarrow^{r_{k-1}}_{\text{Java}\#} t_k$ que describen los estados del programa que pueden o no alcanzarse desde un estado (abstracto) inicial dado. Debido a que estas demostraciones corresponden a la ejecución de la especificación de la semántica abstracta de Java, que está disponible al consumidor del código, el certificado puede ser validado por parte del consumidor de forma rápida con un motor de reescritura estandard mediante un proceso de reescritura que puede ser muy simplificado. Es suficiente con comprobar que cada paso de reescritura abstracta del certificado es válido y ninguna otra reescritura válida ha sido descartada, lo que esencialmente se puede hacer usando la infraestructura de ajuste "matching"del motor de reescritura. Debe notarse que de acuerdo con el tratamiento diferente de las reglas y las ecuaciones en Maude, en el que solo las transiciones causadas por las reglas crean nuevos estados en el espacio de estados, se puede entregar un certificado muy reducido mediante el solo registro de los pasos de reescritura dados con las reglas omitiendo las reescrituras con las ecuaciones.

La metodología de certificación presentada aquí ha sido implementada en Maude y está disponible públicamente en `http://www.dsic.upv.es/users/elp/toolsMaude/rewritingLogic.html`.

La herramienta puede entregar tres certificados:

- El certificado completo con todas las ecuaciones y reglas aplicadas.
- El certificado reducido con solo las reglas aplicadas.
- El certificado únicamente con las etiquetas de las reglas aplicadas.

En el desarrollo y despliegue del sistema, se han atendido los siguientes requerimientos: 1) definir la arquitectura del sistema tan simple como fuera posible, 2) hacer disponible el servicio de certificación a todo solicitante vía Internet, y 3) ocultar los detalles técnicos a los usuarios. La herramienta prototipo Web ofrece un servicio de certificación de programas Java basado en reescritura, que está en la capacidad de analizar propiedades de seguridad del código Java relacionadas con el uso seguro de tipos, y propiedades de confidencialidad de datos secretos desde el punto de vista de la no interferencia. Las principales características de la herramienta son:

- El lenguaje de programación Maude, que implementa la lógica de reescritura, provee una infraestructura de análisis formal (tal como la búsqueda primero-en-anchura en el espacio de estados) con eficiencia competitiva (véase [Farzan et al., 2004a]). Maude podría ser usado como la infraestructura requerida en el lado del consumidor del código para la validación del certificado.
- La semántica operacional de Java en lógica de reescritura que hemos utilizado es modular y tiene 2635 líneas de código en 4 ficheros [Farzan et al., 2007]. Para la semántica relacionada con las propiedades basadas en tipos (Capítulo 4) modificamos 15 líneas de código del fichero principal de la semántica original; para la semántica relacionada con las propiedades de no interferencia (Capítulo 5) modificamos 19 líneas de código, y para la semántica relacionada con las propiedades de consumo acotado de recursos (Capítulo 6), modificamos 82 líneas de código. Las semánticas abstractas de los Capítulos 4 y 5, fueron desarrolladas como una transformación fuente-a-fuente en lógica de reescritura y consisten de 608 y 509 líneas extras de código, respectivamente. Esto nos permite decir que en nuestro sistema actual la *base de código fiable* (TCB) es de un tamaño menor que la mitad del tamaño de la semántica original de Java (y al menos menor en un orden

de magnitud que el tamaño de la infraestructura estándar de reescritura y aún mucho más pequeña que otros sistemas de PCC).

- Para automatizar el procesamiento de las cláusulas JML, la herramienta usa la herramienta de construcción de compiladores Javacc y un subconjunto de la gramática JML. Se generan automáticamente la codificación de la abstracción de las variables anotadas, así como el comando de búsqueda `search` (primero-en-amplitud) de Maude para realizar el análisis de alcanzabilidad, que contiene el programa Java con el método `main` que invoca el método por certificar. La herramienta también usa el programa JavaWrapper, disponible en [Farzan et al., 2007], que transforma el programa Java completo en un término Maude para contruir el estado inicial del comando `search`.

- La interfaz Web permite poner disponible públicamente la técnica de certificación abstracta y a la vez, ocultar los detalles técnicos internos a cualquier posible usuario.

El certificado que se entrega se genera en dos pasos: (i) la carga del programa fuente Java con las anotaciones JML, y (ii) la ejecución del análisis de alcanzabilidad abstracto. Ahora se describe la funcionalidad de la herramienta, cuya interfaz se muestra en la Figura 7.1, con la certificación del Ejemplo 11.

i. Carga del programa y fijación de dominios abstractos. El código Java puede ser cargado de un fichero que contenga el programa anotado con JML. La herramienta permite también seleccionar uno de varios ejemplos predefinidos. El fichero "`Safe1NonInterferenceC.java`", que corresponde al Ejemplo 11, es un ejemplo predefinido que se muestra en la Figura 7.1. La carga del programa de forma implícita provee la validación de su sintaxis usando el compilador estándar de Java. Las anotaciones JML también son validadas con el uso de un módulo que fué generado usando la herramienta para desarrollo de compiladores Javacc.

ii. Análisis de alcanzabilidad abstracto. El certificado de seguridad se genera automáticamente, mediante el pinchado en el botón *Certify!*. El proceso

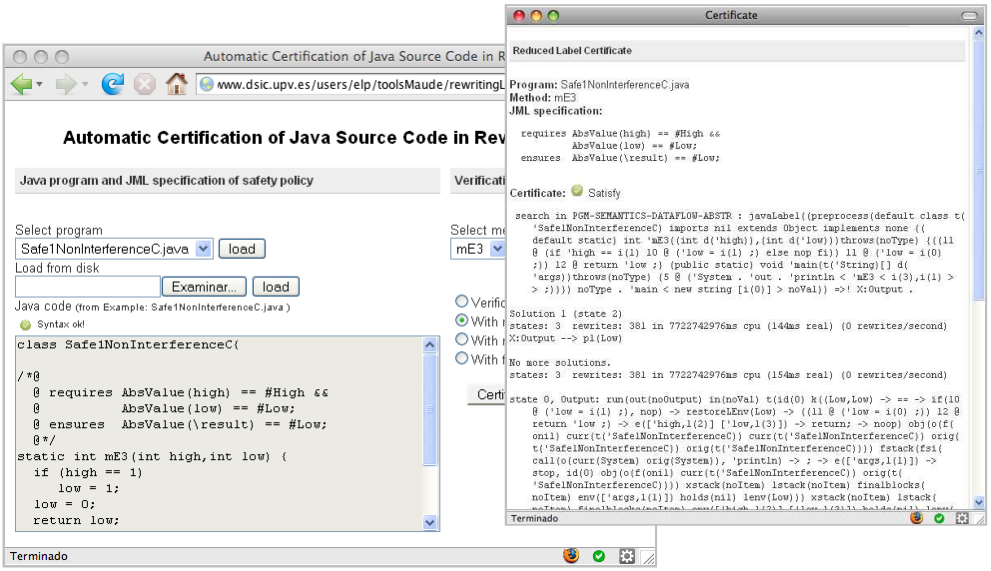

Figura 7.1: Interfaz Web del Prototipo con el Ejemplo 11 y su certificado

de generación se lleva a cabo de la siguiente manera: (i) se crea una nueva clase Java con base en el código del método suministrado y la cláusula JML `ensures`; (ii) el programa JavaWrapper transforma la clase Java en un término Maude que contiene el estado inicial (véase [Farzan et al., 2007] para los detalles de como construir un estado inicial Java); y (iii) al fichero generado con el JavaWrapper, se adicionan las abstracciones de las variables indicadas en las cláusulas JML `requires` y `assert` , junto con el comando *search* que se sintetiza de la cláusula JML `ensures`. y finalmente (iv) se ejecuta en Maude el archivo generado y se obtiene el certificado, que puede entonces entregarse al usuario.

Una vista de la versión anterior de la herramienta relacionada con la certificación del Ejemplo 5 se muestra en la Figuras 7.2 y 7.3. En dicha versión, el usuario debía asociar las variables con sus dominios abstractos mediante el pinchado de botones, y escribir la invocación al método así como el resultado esperado en cajas de texto.

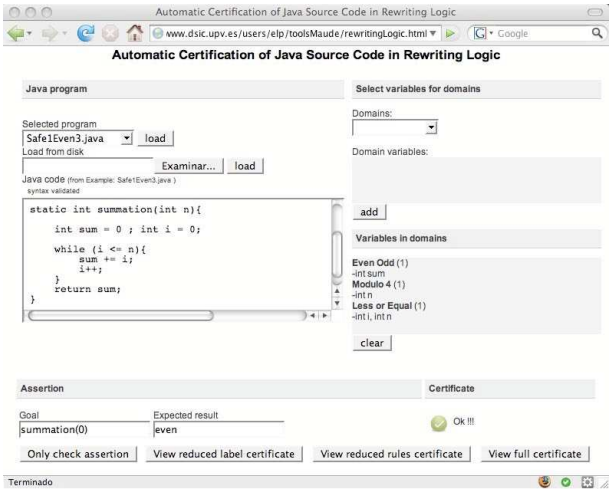

Figura 7.2: Interfaz Web del Prototipo con el Ejemplo 5

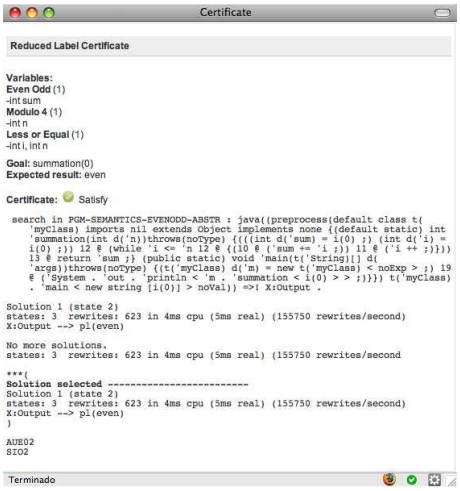

Figura 7.3: Interfaz Web del Prototipo con un certificado del Ejemplo 5 .

8

Conclusiones

8.1. Diferencias con otros enfoques de ACC

El enfoque adoptado difiere de otros enfoques PCC basados en interpretación abstracta en varios aspectos. Con respecto al enfoque ACC (*abstraction carrying code*) de [Albert et al., 2005c] para programas lógicos con restricciones, se comparte la alta flexibilidad debida a la parametrización respecto de diferentes dominios abstractos, la eficiencia del validador de las demostraciones del lado del consumidor, y el hecho de que ambas técnicas están definidas a nivel de lenguaje fuente (en el caso de ACC es Ciao-Prolog). Sin embargo, su certificado es producido mediante un analizador estático y toma la forma de un subconjunto particular del resultado del análisis de punto fijo, que el consumidor valida mediante un interpretador abstracto simple. Nuestro certificado es principalmente una codificación de demostraciones abstractas de inalcanzabilidad, que está más cercano al enfoque original de PCC [Necula, 1997; Necula and Lee, 1996] en el que el certificado es una demostración en lógica de primer orden. En relación con el enfoque de ACCEPT/C [Xia and Hook, 2004] también se comparte que ambas técnicas están definidas a nivel de lenguaje fuente (en el caso de ACCEPT/C es C). En este caso, el certificado es una función booleana que está codificada como anotaciones de tipos en el programa compilado en lenguaje ensamblador. La desventaja de este enfoque es el gran tamaño de la base de código fiable que debería usar el consumidor y la menor eficiencia durante la validación del certificado. [Besson et al., 2006; Besson et al., 2007] también aplican interpretación abstracta sin usar herramientas de análisis de tipos, pero sus certificados toman la forma de estrategias para reconstruir un punto fijo. En este caso, la interpretación abstracta

se usa para reducir las demostraciones que son generadas y validadas mediante el demostrador de teoremas Coq, para un subconjunto de bytecode Java usando una técnica de compresión de punto fijo. Considerando el trabajo [Chang et al., 2006], que se sale del marco de PCC, pues el productor del código no suministra una demostración del cumplimiento de una política del consumidor, sino un verificador desarrollado usando el marco formal que se propone en el trabajo, para precisamente implementar y certificar análisis de programas. En este caso, usan la interpretación abstracta para obtener los teoremas que deben ser demostrados para cada analizador que sea certificado. El analizador impone la política del consumidor.

8.2. Propiedades Aritméticas basadas en tipos

Por lo que se sabe, el enfoque desarrollado es la primera aproximación correcta, y completa de una herramienta completamente automática para la verificación de propiedades aritméticas basadas en tipos no verificables por un compilador estándar, de programas Java (código fuente). La completitud es relativa a la abstracción, pero puede obviarse al mejorar la aproximación, como se ilustró al introducir el dominio abstracto `AbstLeqN` para la certificación del Ejemplo 5.

La escalabilidad a código de tamaño y complejidad industriales depende de las abstracciones disponibles como ocurrió al considerar el Ejemplo 5, para el cual los dominios abstractos *EvenOdd* y $Mod4$ no tenían la precisión suficiente. Esto quiere decir que para certificar un programa Java dado, es posible que se necesite definir un nuevo dominio abstracto. Si se quiere certificar la política de resultados *EvenOdd*, el nuevo dominio posiblemente sea una refinación de los existentes *EvenOdd*, $Mod4$ y *AbstLeqN*.

8.3. No Interferencia

Esta es la primera aproximación correcta y completa, de una herramienta completamente automática para la verificación de no interferencia, de programas Java (código fuente).

Sin embargo, hay programas no realistas pero cuyo cumplimiento de la política

de no interferencia no se puede certificar, porque la semántica extendida con análisis de flujos de información señala que hay flujos ilícitos, cuando no los hay. Por ejemplo consideramos el siguiente código Java, tomado de [Warnier, 2005].

Ejemplo 22. Considérese el método Java junto con la invocación a la función `main` mostrados a continuación:

```
static int m521(int high, int low) {
/*@      requires AbsValue(high) == High && AbsValue(low) == Low;
  @       ensures AbsValue(\result) == Low;                                    @*/
           low = high;
           low = low - high;
           return low;
}

public static void main(String[] args)     {
          System.out.println(m521(h,l));
}
```

En el comando `search` abajo, se pregunta por todos los posibles valores retornados por la función Java `main` con la invocación al método `m521`.

```
search in PGM-SEMANTICS-LABELED : javaLabel((preprocess(default class t(
    'Safe1NonInterference1b) imports nil extends Object implements none {(
    default static) int 'm521((int d('high)),(int d('low)))throws(noType) {((8
    @ ('low = 'high ;)) 9 @ ('low = 'low - 'high ;)) 10 @ return 'low ;} (
    public static) void 'main(t('String)[] d('args))throws(noType) {5 @ (
    'System . 'out . 'println < 'm521 < i(1),i(0) > > ;)}}) noType . 'main <
    new string [i(0)] > noVal)) =>! X:Output .

Solution 1 (state 1)
states: 2  rewrites: 253 in 976ms cpu (13ms real) (0 rewrites/second)
X:Output --> pl(< int(0),High >)

No more solutions.
states: 2  rewrites: 253 in 976ms cpu (24ms real) (0 rewrites/second)
```

El comando `search` retorna que el resultado de la instrucción Java "`System.out.println(m521(h,l));`" es el valor extendido `< int(0),High >`, que significa que hay flujo ilícito cuando no lo hay, pues el valor de la variable `low` no tiene nada que ver con el valor de la variable secreta `high`.

Este resultado se debe a que a pesar de que el valor de la variable `low` depende del valor de la variable `high` por las dos instrucciones de asignación, en realidad no es así, pues esas dos intrucciones tienen el mismo efecto de la siguiente instrucción permitida `low = low;`.

La semántica de no interferencia, puede extenderse fácilmente para tratar procedimientos (métodos que no retornan valores), excepciones, "heaps"y multihilos, ya que esas características son consideradas en la semántica Java en lógica de reescritura que se usa. Por otra parte, como se hereda de Maude y de la semántica Java en lógica de reescritura sus altas prestaciones (ver [Farzan et al., 2004a]), y utiliza solo un dominio abstracto, tenemos una técnica escalable que puede refinarse aún más para certificar programas complejos de uso industrial.

8.4. Generales

La metodología propuesta se caracteriza por sus atributos de calidad (fiabilidad, seguridad, y también altas prestaciones) a través de mecanismos rigurosos que integran un amplio rango de técnicas de programación maduras y bien establecidas (interpretación abstracta, semántica de programas, metaprogramación, etc.).

La metodología se basa en la especificación de la semántica Java en lógica de reescritura de la versión completa 1.4 del lenguaje [Farzan et al., 2007], y por consiguiente funciona con todo el lenguaje Java 1.4. Sin embargo, como dicha especificación no incluye de la biblioteca de clases Java sino algunos métodos de las clases Java para entrada y salida, actualmente solo se pueden certificar programas que usen esos métodos únicamente.

La metodología propuesta puede ser ampliada a otros lenguajes de programación convencionales o lenguajes de bajo nivel (por ejemplo bytecode Java) simplemente reemplazando la semántica concreta que se usa en este trabajo, por la semántica del lenguaje de programación en cuestión (por ejemplo, una semántica en lógica de reescritura para bytecode Java puede encontrarse en [Farzan et al., 2004a]).

Se pueden definir diferentes políticas de seguridad usando diferentes términos abstractos que denotan los estados que no deberían alcanzarse. Tales políticas de seguridad son certificadas por el productor de código, y fácilmente validadas por el consumidor del código, utilizando un proceso de reescritura que puede ser muy simple. Los certificados codificados como secuencias abstractas de reescritura pueden comprobarse del lado del consumidor con la semántica abstracta escrita

en Maude mediante reducción estándar.

En la actualidad estamos investigando el consumo acotado de recursos físicos[1] y el uso de la comprobación de modelos abstracta para considerar propiedades temporales de los programas Java y así producir una metodología más potente.

[1]De lo cual se ha mostrado un resultado parcial en el Capítulo 6

Bibliografía

[Alba-Castro et al., 2008] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2008, Automatic certification of java source code in rewriting logic, in *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, Vol. 4916 of *Lecture Notes in Computer Science*, pp 200–217, Springer

[Albert et al., 2007a] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D.: 2007a, Cost analysis of java bytecode, in *16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science

[Albert et al., 2007b] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D.: 2007b, *Electron. Notes Theor. Comput. Sci.* **190(1)**, 67

[Albert et al., 2006] Albert, E., Arenas, P., Puebla., G., and Hermenegildo, M.: 2006, Reduced certificates for abstraction-carrying code, in *Logic Programming, 22nd International Conference (ICLP 2006)*, Vol. 4079 of *Lecture Notes in Computer Science*, pp 163–178

[Albert et al., 2004a] Albert, E., Puebla, G., and Hermenegildo, M.: 2004a, Poster presentation: Abstract interpretation-based mobile code certification, in *20th International Conference Logic Programming (ICLP 2004)*, Vol. 3132 of *Lecture Notes in Computer Science*, pp 446–447

[Albert et al., 2005a] Albert, E., Puebla, G., and Hermenegildo, M.: 2005a, An abstract interpretation-based approach to mobile code safety, in *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004)*, Electronic Notes in Theoretical Computer Sc., pp 113–129

[Albert et al., 2005b] Albert, E., Puebla, G., and Hermenegildo, M.: 2005b, *Electr. Notes Theor. Comput. Sci.* **132(1)**, 113

[Albert et al., 2005c] Albert, E., Puebla, G., and Hermenegildo, M.: 2005c, Abstraction-carrying code, in *LPAR 2005*, Vol. 3452 of *Lecture Notes in Computer Science*, pp 380–397, Springer

[Albert et al., 2004b] Albert, E., Puebla, G., Hermenegildo, M., and López-García, P.: 2004b, Some techniques for automated, resource-aware distributed and mobile computing in a multi-paradigm programming system, in *EURO-PAR 2004 Conference*, Vol. 3149 of *Lecture Notes in Computer Science*, pp 21–37

[Albert et al., 2005d] Albert, E., Puebla, G., Hermenegildo, M., and López-García, P.: 2005d, Abstraction carrying code and resource-awareness, in *Proc. of 7th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP' 05), Lisboa Portugal*, pp 1–11

[Appel, 2001] Appel, A.: 2001, Foundational proof-carrying code, in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (SLCS 2001)*, pp 247–258

[Appel et al., 2003] Appel, A., Michael, N., Stump, A., and Virga, R.: 2003, *Journal of Automated Reasoning* **31(3-4)**, 231

[Appel and Felten, 2001] Appel, A. W. and Felten, E. W.: 2001, *Models for security policies in proof-carrying code*, Technical Report TR-636-01, Princeton University

[Appel and Felty, 2000] Appel, A. W. and Felty, A. P.: 2000, A semantic model of types and machine instuctions for proof-carrying code., in *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2000)*, pp 243–253

[Appel and Mcallester, 2001] Appel, A. W. and Mcallester, D.: 2001, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **23(5)**, 657

[Aspinall et al., 2004a] Aspinall, D., Beringer, L., Hofmann, M., Loidl, H., and Momigliano, A.: 2004a, A program logic for resource verification, in *Proceedings Theorem Proving in Higher Order Logics 17th International Conference (TPHOLs 2004), Park City, Utah, USA, September 14-17, 2004*, Vol. 3223 of *Lecture Notes in Computer Science*, pp 34–49

[Aspinall et al., 2004b] Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I.: 2004b, Mobile resource guarantees for smart devices, in *Proc. of the International Workshop CASSIS 2004*, Vol. 3362 of *Lecture Notes in Computer Science*, pp 1–26

[Aspinall and MacKenzie, 2006] Aspinall, D. and MacKenzie, K.: 2006, Mobile resource guarantees and policies, in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices Second International Workshop (CASSIS 2005), Nice, France, March 8-11, 2005*, Vol. 3956 of *Lecture Notes in Computer Science*, pp 16–36

[ATIS, 2001] ATIS: 2001, *Alliance for Telecommunications Industry Solutions (ATIS) Telecom Glossary 2001*, Available in `http://www.atis.org/tg2k/t1g2k.html`, Approved February 28, 2001 by American National Standards Institute, Inc.

[Avvenuti et al., 2003] Avvenuti, M., C.Bernardeschi, and Francesco, N. D.: 2003, *ACM SIGPLAN Notices* **38(12)**, 20

[Barbuti et al., 2002a] Barbuti, R., Bernarddeschi, C., and Francisco, N. D.: 2002a, *Information Processing Letters* **83(22)**, 101

[Barbuti et al., 2002b] Barbuti, R., Bernarddeschi, C., and Francisco, N. D.: 2002b, Checking security of java bytecode by abstract interpretation, in *Proceedings of the 2002 ACM symposium on Applied computing (SAC 2002), Madrid, Spain*, pp 229–236

[Barthe, 2005] Barthe, G.: 2005, *MOBIUS -Securing the Next Generation of Java Based Global Computers*, ERCIM news 63

[Barthe et al., 2004] Barthe, G., D'Argenio, P., and Rezk, T.: 2004, Secure information flow by self-composition, in *Proceedings of the 17th IEEE workshop on Computer Security Foundations (CSFW '04)*, pp 100–114

[Barthe et al., 2006] Barthe, G., Naumann, D., and Rezk, T.: 2006, Deriving an information flow checker and certifying compiler for java, in *2006 IEEE Symposium on Security and Privacy*, pp 230–242

[Barthe et al., 2005] Barthe, G., Pavlova, M., and Schneider, G.: 2005, Precise analysis of memory consumption using program logics, in *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 05)*, pp 86–95

[Barthe et al., 2007] Barthe, G., Pichardie, D., and Rezk, T.: 2007, A certified lightweight non-interference java bytecode verifier, in *Proc. 16th European Symposium on Programming (ESOP 2007)*, Vol. 4421 of *Lecture Notes in Computer Science*, pp 125–140

[Barthe and Rezk, 2005] Barthe, G. and Rezk, T.: 2005, Non-interference for a jvm-like language, in *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation (TLDI '05)*, pp 103–112

[Beringer and Hofmann, 2007] Beringer, L. and Hofmann, M.: 2007, Secure information flow and program logics, in *20th IEEE Computer Security Foundations Symposium (CSF '07)*, pp 233–248

[Beringer et al., 2003] Beringer, L., Mackenzie, K., and Stark, I.: 2003, *ENTCS* **85(1)**, 21

[Besson et al., 2005] Besson, F., Jensen, T., and Pichardie, D.: 2005, *A PCC Architecture based on Certified Abstract Interpretation*, Technical Report RR-5751, INRIA Rennes

[Besson et al., 2007] Besson, F., Jensen, T., Pichardie, D., and Turpin, T.: 2007, *Result Certification for relational program analysis*, Technical Report 6333, INRIA

[Besson et al., 2006] Besson, F., Jensen, T. P., and Pichardie, D.: 2006, *Theor. Comput. Sci.* **364(3)**, 273

[Bishop, 2004] Bishop, M.: 2004, *Introduction to Computer Security*, Addison-Wesley Professional

[Burdy et al., 2005] Burdy, L., Y.Cheon, Cok, D., Ernst, M., Kiniry, J., Leavens, G., Rustan, K., Leino, M., and Poll, E.: 2005, *International Journal on Software Tools for Technology Transfer* **7(3)**, 212

[Chalin et al., 2005] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E.: 2005, Beyond assertions: Advanced specification and verification with JML and ESC/Java2., in *FMCO*, pp 342–363

[Chander et al., 2005a] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2005a, Enforcing resource bounds via static verification of dynamic checks, in *European Symposium on Programming (ESOP2005)*

[Chander et al., 2005b] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2005b, Jver: A java verifier, in *procs. Conference on Computer Aided Verification (CAV 2005)*, Vol. 3576 of *Lecture Notes in Computer Science*, pp 144–147

[Chander et al., 2007] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2007, *ACM Transactions on Programming Languages and Systems* 29(5)

[Chang et al., 2006] Chang, B. E., Chlipala, A., and Necula, G. C.: 2006, A framework for certified program analysis and its applications to mobile-code safety, in *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI 2006)*, Vol. 3855 of *Lecture Notes in Computer Science*, pp 174–189

[Chang et al., 2005] Chang, B. E., Chlipala, A., Necula, G. C., and Schneck, R. R.: 2005, The open verifier framework for foundational verifiers, in *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation (TLDI '05)*, pp 1–12

[Chen et al., 2006] Chen, F., Hills, M., and Roşu, G.: 2006, *A Rewrite Logic Approach to Semantics Definition,Design and Analysis of Object-Oriented Languages*, Technical Report UIUCDCS-R-2006-2702, University of Illinois Urbana-Champaign Department of Computer Science

[Clavel et al., 2002] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F.: 2002, *Theoretical Computer Science* **285(2)**, 187

[Clavel et al., 2003] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2003, The maude 2.0 system, in R. Nieuwenhuis (ed.), *Rewriting Techniques and Applications (RTA 2003)*, No. 2706 in Lecture Notes in Computer Science, pp 76–87, Springer-Verlag

[Clavel et al., 2005] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2005, *Maude Manual (version 2.2)*, SRI International

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2007, *All About Maude: A High-Performance Logical Framework*, Vol. 4350 of *Lecture Notes in Computer Science*, Springer-Verlag

[Colby et al., 2000] Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M., and Cline, K.: 2000, A certifying compiler for java, in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), Vancouver, British Columbia, Canada*, pp 95–107, ACM Press

[Condit et al., 2007] Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. C.: 2007, Dependent types for low-level programming, in *Programming Languages and Systems, Proceedings 16th European Symposium on Programming (ESOP 2007)*, No. 4421 in Lecture Notes in Computer Science, pp 520–535

[Condit et al., 2005] Condit, J., Harren, M., McPeak, S., Weimer, W., and Necula, G. C.: 2005, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **27(3)**, 477

[Cousot and Cousot, 1976] Cousot, P. and Cousot, R.: 1976, Static determination of dynamic properties of programs, in *Proceedings of the second international symposium on Programming, Paris, France*, pp 106–130

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R.: 1977, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth ACM SYmposium on Principles of Programming Languages*, pp 238–252

[Cousot and Cousot, 1978] Cousot, P. and Cousot, R.: 1978, Static determination of dynamic properties of recursive procedures, in *IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B., Canada, 1977*, pp 237–277

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R.: 1979, Systematic Design of Program Analysis Frameworks, in *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pp 269–282

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N.: 1978, Automatic discovery of linear restraints among variables of a program, in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1978)*, pp 84–97

[Crary and Weirich, 2000] Crary, K. and Weirich, S.: 2000, Resource bound certification, in *(POPL 2000)*, pp 184–198

[Denning and Denning, 1977] Denning, D. E. and Denning, P. J.: 1977, *Commun. ACM* **20(7)**, 504

[Dufay et al., 2005] Dufay, G., Felty, A., and Matwin, S.: 2005, Privacy-sensitive information flow with jml, in *Proc. of the Conferen. on Automated Deduction (CADE20)*, Vol. 3622 of *Lecture Notes in Computer Science*, pp 116–130

[Engblom et al., 2003] Engblom, J., Ermedahl, A., Sjodin, M., Gustafsson, J., and Hansson, H.: 2003, *International Journal Software Tools Technology Transfer* **4**, 437

[Farzan et al., 2004a] Farzan, A., Chen, F., Meseguer, J., and Rosu, G.: 2004a, Formal analysis of Java programs in JavaFAN., in R. Alur and D. Peled (eds.), *Proceedings of 16th International Conference on Computer Aided Verification (CAV 2004)*, Vol. 3114 of *Lecture Notes in Computer Science*, pp 501–505, Springer

[Farzan et al., 2007] Farzan, A., Chen, F., Meseguer, J., and Rosu, G.: 2007, *JavaRL: The Rewriting Logic Semantics of Java*, Available at `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`

[Farzan et al., 2004b] Farzan, A., Meseguer, J., and Rosu, G.: 2004b, Formal JVM code analysis in JavaFAN., in C. Rattray, S. Maharaj, and C. Shankland (eds.), *Proceedings 10th International Conference on Algebraic Methodology and Software Technology (AMAST 2004)*, Vol. 3116 of *Lecture Notes in Computer Science*, pp 132–147, Springer

[Felty, 2005] Felty, A. P.: 2005, A tutorial example of the semantic approach to foundational proof-carrying code., in J. Giesl (ed.), *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, Vol. 3467 of *Lecture Notes in Computer Science*, pp 394–406, Springer

[Francesco and Martini, 2007] Francesco, N. D. and Martini, L.: 2007, *International Journal of Information Security* **6(2-3)**, 85

[FSL, 2006] FSL: 2006, *Formal Systems Laboratory (FSL) Semantics Research Homepage*, Formal Systems Laboratory Web page: `http://fsl.cs.uiuc.edu/index.php/`, Department of Computer Science, University of Illinois, Urbana-Champaign

[Giacobazzi and Mastroeni, 2004] Giacobazzi, R. and Mastroeni, I.: 2004, Abstract non-interference: Parameterizing non-interference by abstract interpretation, in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'04), Venice, Italy*, pp 186–197

[Gilmore and Prowse, 2005] Gilmore, S. and Prowse, M.: 2005, *Elect Notes in Thoretical Computer Science* **141(1)**, 3

[Goguen and Meseguer, 1982] Goguen, J. A. and Meseguer, J.: 1982, Security policies and security models, in *IEEE Symposium on Research in Security and Privacy*, pp 11–20

[Grossman and Morrisett, 2001] Grossman, D. and Morrisett, G.: 2001, Scalable certification for typed assembly language, in *Third International Workshop*

on Types in Compilation (TIC 2000), Montreal, Canada, September 21, 2000, Selected Papers, Vol. 2071 of *Lecture Notes in Computer Science*, pp 117–146

[Gustafsson, 2002] Gustafsson, J.: 2002, Worst case execution time analysis of object-oriented programs, in *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pp 71–76

[Gustafsson et al., 2006] Gustafsson, J., Lisper, B., Kirner, R., and Puschner, P.: 2006, *Real-Time Systems* **32(3)**, 253

[Hamid, 2005] Hamid, N.: 2005, *Ph.D. thesis*, Yale University

[Hamid et al., 2002] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z.: 2002, A syntactic approach to foundational proof-carrying code, in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pp 89–100

[Hamid et al., 2003] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z.: 2003, *Journal of Automated Reasoning* **31(3-4)**, 191

[Hamid and Shao, 2004] Hamid, N. A. and Shao, Z.: 2004, Interfacing hoare logic and type systems for foundational proof-carrying code, in *Proc. 17th International Conference on the Applications of Higher Order Logic Theorem Proving (TPHOLs'04)*, Vol. 3223 of *Lecture Notes in Computer Science*, pp 118–135

[Harren and Necula, 2005] Harren, M. and Necula, G. C.: 2005, Using dependent types to certify the safety of assembly code, in *Static Analysis Symposium (SAS 2005)*, No. 3672 in Lecture Notes in Computer Science, pp 155–170

[Harren, 2007] Harren, M. T.: 2007, *Ph.D. thesis*, University of California, Berkeley

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G., Sutre, G., and Weimer, W.: 2002, Temporal-safety proofs for systems code, in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, Vol. 2404 of *Lecture Notes in Computer Science*, pp 526–538

[Hofmann and Jost, 2003] Hofmann, M. and Jost, S.: 2003, Static prediction of heap space usage for first order functional programs (extended version), in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03)*, pp 185–197

[Hofmann et al., 2005] Hofmann, M., Loidl, H.-W., and Beringer, L.: 2005, Certification of quantitative properties of programs, in *Logical Aspects of Secure Computer Systems. Lecture Notes of the Marktoberdorf Summer School 2005*, Lecture Notes of the Marktoberdorf Summer School 2005. To appear

[Hunt and Sands, 2006] Hunt, S. and Sands, D.: 2006, On flow-sensitive security types, in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, pp 79–90

[Jacobs et al., 2004] Jacobs, B., Oostdijk, M., and Warnier, M.: 2004, *Journal of Logic and Algebraic Programming JLAP* **58(1-2)**, 107

[Jacobs et al., 2005] Jacobs, B., Pieters, W., and Warnier, M.: 2005, Statically checking confidentiality via dynamic labels, in *Proceedings of the 2005 workshop on Issues in the theory of security (WITS '05)*, pp 50–56

[Jim et al., 2002] Jim, T., Morriset, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y.: 2002, Cyclone: A safe dialect of c, in *USENIX Annual Technical Conference, Monterrey, CA, USA*, pp 275–288

[Leavens et al., 2006] Leavens, G., Baker, A., and Ruby, C.: 2006, *ACM SIGSOFT Software Engineering Notes* **31(3)**, 1

[Lee and Necula, 1997] Lee, P. and Necula, G. C.: 1997, Research on proof carrying code for mobile code security. a position paper, in *DARPA Workshop on Foundations for Secure Mobile Code*

[Lisper, 2003] Lisper, B.: 2003, Fully automatic, parametric worst-case execution time analysis, in *Proc. of Third International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2003)*, pp 99–104

[MacKenzie and Wolverson, 2004] MacKenzie, K. and Wolverson, N.: 2004, Camelot and grail: Resource-aware functional programming for the jvm, in

TFP 2004 Fifth Symposium on Trends in Functional Programming, Ludwig-Maximilians University, Munich, Germany, Vol. 4, pp 29–46

[Manna and Pnueli, 1995] Manna, Z. and Pnueli, A.: 1995, *Temporal Verification of Reactive Systems: safety*, Springer-Verlag

[Meseguer, 1992] Meseguer, J.: 1992, *Theoretical Computer Science* **96(1)**, 73

[Meseguer and Rosu, 2007] Meseguer, J. and Rosu, G.: 2007, *Theoretical Computer Science* **373(3)**, 213

[Mobius, 2005] Mobius: 2005, *Mobius Mobility, Ubiquity and Security*, Web site

[Mok and Yu, 2002a] Mok, A. and Yu, W.: 2002a, Tinman: A resource bound security checking system for mobile code, in *Proceedings 7th European Symposium on Research in Computer Security (ESORICS 2002) , Zurich, Switzerland*, Vol. 2502 of *Lecture Notes in Computer Science*, pp 178–193

[Mok and Yu, 2002b] Mok, A. K. and Yu, W.: 2002b, Enforcing resource bound safety for mobile snmp agents, in *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC)*, pp 69–77

[Morrisett et al., 1999a] Morrisett, G., K.Crary, N.Glew, D.Grossman, R.Samuels, F.Smith, D.Walker, S.Weirich, and S.Zdancewic: 1999a, Talx86: A realistic typed assembly language, in *ACM SIGPLAN Workshop on Compiler Support for System Software,Atlanta, GA, USA*, pp 25–35

[Morrisett et al., 1999b] Morrisett, G., Walker, D., Crary, K., and Glew, N.: 1999b, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **21(3)**, 527

[MRG, 2005] MRG: 2005, *A Users Guide for the MRG PCC Infrastructure*, MRG: Mobile Resource Guarantees Project web site, http://groups.inf.ed.ac.uk/mrg/publications/mrg/Users-Guide/sec-users.html

[Myers, 1999] Myers, A.: 1999, Jflow: Practical mostly-static information flow control, in *Proceedings of the 26th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL 1999)*, pp 228–241

[Myers et al., 2001] Myers, A., Nystrom, N., Zheng, L., and Zdancewic, S.: 2001, *Jif: Java information flow*, Software release. Available at:*http://www. cs. cornell. edu/jif*

[Necula, 1997] Necula, G. C.: 1997, Proof carrying code, in *Proceedings of the 24th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL 1997), Paris, France*, pp 106–119

[Necula, 2001a] Necula, G. C.: 2001a, A scalable architecture for proof-carrying code, in *Functional and Logic Programming: Proceedings: 5th International Symposium (FLOPS 2001), Tokyo, Japan*, Vol. 2024 of *Lecture Notes in Computer Science*, pp 21–39

[Necula, 2001b] Necula, G. C.: 2001b, *TouchStone, a certifying compiler for Java*, TouchStone Web Page. `http://raw.cs.berkeley.edu/touchstone.html`

[Necula and Lee, 1996] Necula, G. C. and Lee, P.: 1996, Safe kernel extensions without run time checking, in *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI 1996), Seattle, Washington, United States*, Vol. 30, pp 229–243

[Necula and Lee, 1997a] Necula, G. C. and Lee, P.: 1997a, *Efficient Representation and Validation of Logical Proofs*, Technical Report CMU-CS-97-172, Carnegie Mellon University

[Necula and Lee, 1997b] Necula, G. C. and Lee, P.: 1997b, Research on proof-carrying code for untrusted code security, in *Proceedings 1997 IEEE Symposium on Security and Privacy*, p. 204

[Necula and Lee, 1998a] Necula, G. C. and Lee, P.: 1998a, The design and implementation of a certifying compiler, in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI 1998), Montreal, Quebec, Canada*, Vol. 33, pp 333 – 344

[Necula and Lee, 1998b] Necula, G. C. and Lee, P.: 1998b, Safe untrusted agents using proof carrying code, in *Special Issue on Mobile Agent Security*, Vol. 1419 of *Lecture Notes in Computer Science*, pp 61 – 91, Springer-Verlag

[Necula and Rahul, 2001] Necula, G. C. and Rahul, S.: 2001, Oracle-based checking of untrusted software, in *Proceedings of the 28th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL 2001) , London, United Kingdom*, pp 142 – 154

[Necula and Schneck, 2002] Necula, G. C. and Schneck, R. R.: 2002, A gradual approach to a more trustworthy, yet scalable, proof-carrying code, in *Proceedings of the 18th International Conference on Automated Deduction (CADE ' 02), Copenhagen,*, Vol. 2392, pp 47 – 62

[Necula and Schneck, 2003a] Necula, G. C. and Schneck, R. R.: 2003a, Proof-carrying code with untrusted proof rules, in *Software Security - Theories and Systems Mext-NSF-JSPS Revised Papers of International Software Security Symposium (ISSS 2002) Tokyo, Japan, 2002*, Vol. 2609 of *Lecture Notes in Computer Science*, pp 283–298

[Necula and Schneck, 2003b] Necula, G. C. and Schneck, R. R.: 2003b, A sound framework for untrusted verification-condition generators, in *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pp 248–260

[Ni and Shao, 2006] Ni, Z. and Shao, Z.: 2006, Certified assembly programming with embedded code pointers, in *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, Vol. 41, pp 320–333

[Pavlova et al., 2004] Pavlova, M., Barthe, G., Burdy, L., Huisman, M., and Lanet, J.-L.: 2004, Enforcing high-level security properties for applets, in *Proceedings of CARDIS'04, Toulouse, France*, pp 1–16

[Robby et al., 2006] Robby, Rodriguez, E., Dwyer, M., and Hatcliff, J.: 2006, *Int. Journal on Software Tools for Technology* **8(3)**, 280

[Rose, 2003] Rose, E.: 2003, *J. Autom. Reason.* **31(3-4)**, 303

[Rosu, 2005] Rosu, G.: 2005, *K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation*, Technical Report

UIUCDCS-R-2005-2672, Department of Computer Science, University of Illinois at Urbana- Champaign

[Sabelfeld and Myers, 2003] Sabelfeld, A. and Myers, A.: 2003, *IEEE Journal on Selected Areas in Communications* **21(1)**, 5

[Sannella et al., 2005] Sannella, D., Hofmann, M., Aspinall, D., Gilmore, S., Stark, I., Beringer, L., Loidl, H.-W., K.Mackenzie, Momigliano, A., and Shkaravska, O.: 2005, Mobile resource guarantees (project evaluation paper), in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, pp 211–226

[Sekar et al., 2001] Sekar, R., Ramakrishnan, C., Ramakrishnan, I., and Smolka, S.: 2001, Model-carrying code (mcc): A new paradigm for mobile-code security, in *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, pp 23–30

[Sekar et al., 2003] Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., and DuVarney, D.: 2003, Model-carrying code: A practical approach for safe execution of untrusted, in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp 15–28

[Shao, 1997] Shao, Z.: 1997, An overview of the flint/ml compiler, in *Proc. ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, pp 85–98

[Shao and Appel, 1995] Shao, Z. and Appel, A. W.: 1995, A type-based compiler for standard ml, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pp 116–129

[Smith and Volpano, 1998] Smith, G. and Volpano, D.: 1998, Secure information flow in a multi-threaded imperative language, in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 98)*, pp 355–364

[Tarditi et al., 1996] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P.: 1996, Til: A type-directed optimizing compiler for ml, in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, Vol. 31 of *ACM/SIGPLAN Notices*, pp 181–192

[Tarditi et al., 2004] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P.: 2004, Til: A type-directed optimizing compiler for ml (with retrospective), in *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pp 554–567

[TeReSe, 2003] TeReSe (ed.): 2003, *Term Rewriting Systems*, Cambridge University Press, Cambridge

[TILT, 2006] TILT: 2006, *The TILT Compiler Project*, Web Page. `http://www.cs.cornell.edu/home/jgm/tilt.html`

[Tsukada, 2000] Tsukada, Y.: 2000, Mobile codes with interactive proofs: an approach to provably safeevolution of distributed software systems, in *Proceedings International Symposium on Principles of Software Evolution,2000*, pp 23–27

[Tsukada, 2005] Tsukada, Y.: 2005, *Automated Software Engineering* **12(2)**, 237

[Warnier, 2005] Warnier, M.: 2005, *Ph.D. thesis*, Radboud University Nijmegen

[Whitehead et al., 2004] Whitehead, N., Abadi, M., and Necula, G.: 2004, By reason and authority: A system for authorization of proof-carrying code, in *Procs. 17th IEEE Computer Security Foundations WorkShop (CSF 2004)*, pp 236–250

[Wildmoser et al., 2005] Wildmoser, M., Chaieb, A., and Nipkow, T.: 2005, Bytecode analysis for proof carrying code, in *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation*, Vol. 141 of *Electronic Notes in Theoretical Computer Science*, pp 19–34

[Wildmoser and Nipkow, 2005] Wildmoser, M. and Nipkow, T.: 2005, Asserting bytecode safety, in *Procs of the 15th Symp. On Programming ESOP05*, Vol. 3444 of *Lecture Notes in Computer Science*, pp 326–341

[Wildmoser et al., 2004] Wildmoser, M., Nipkow, T., Klein, G., and Nanz, S.: 2004, Prototyping proof carrying code., in J.-J. Lévy, E. W. Mayr, and J. C. Mitchell (eds.), *3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, pp 333–348, Kluwer

[Wu et al., 2003] Wu, D., Appel, A., and Stump, A.: 2003, Foundational proof checkers with small witnesses, in *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP 2003)*, pp 264–274

[Xia and Hook, 2003a] Xia, S. and Hook, J.: 2003a, *Abstraction-carrying Code: a New Method to Certify Temporal Properties*, informal proceedings available at `http://www.cs.ru.nl/~erikpoll/ftfjp/2003/FTfJP2003.pdf`, Presented at the Workshop Formal Techniques for Java-like Programs, co-located with the 17th European Conference on Object-Oriented Programming (ECOOP 2003)

[Xia and Hook, 2003b] Xia, S. and Hook, J.: 2003b, *Electr. Notes Theor. Comput. Sci.* **89(3)**, 17, Workshop on Software Model Checking (SoftMC 2003)

[Xia and Hook, 2003c] Xia, S. and Hook, J.: 2003c, *An Implementation of Abstraction-carrying Code*, Foundations of Computer Security"(proceedings of the LICS'03 workshop on Foundations of Computer Security), Technical Report TR-2003-04, Department of Computer Science, University of Ottawa, Ottawa, Canada, 26-27 June 2003.

[Xia and Hook, 2004] Xia, S. and Hook, J.: 2004, Certifying temporal properties for compiled c programs, in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, Vol. 2937 of *Lecture Notes in Computer Science*, pp 161–174

[Yu and Mok, 2004] Yu, W. and Mok, A.: 2004, Formal specification and verification of resource bound security using pvs, in *Software Security Theories and Systems*, Vol. 3233 of *Lecture Notes in Computer Science*, pp 113–133

[Zanotti, 2002] Zanotti, M.: 2002, Security typings by abstract interpretation, in *Proc. Symposium on Static Analysis (SAC'02)*, Vol. 2477 of *Lecture Notes in Computer Science*, pp 360–375

A

Trabajos desarrollados en el marco de esta tesis

- **Internacionales**

  - M. Alba-Castro, M. Alpuente y S. Escobar
    Automatic certification of java source code in rewriting logic.
    In Post Proc. of *12th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007), Berlin, Alemania, Julio, 2007*, LNCS 4916:200–217. Springer, 2008.

  - M. Alba-Castro, M. Alpuente y S. Escobar
    Automated certification of Non-Interference in Rewriting Logic.
    13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008), l'Aquila, ITALY, September 15-16, 2008. *Aceptado.*

- **Nacionales**

  - M.Alba-Castro, M. Alpuente, S. Escobar, P.Ojeda y D. Romero
    A Tool for Automated Certification of Java Source Code in Maude
    VIII Jornadas sobre Programación y Lenguajes, PROLE'08, Gijón, Asturias, España, 7-10 de Octubre de 2008. *Aceptado.*

  - M. Alba-Castro, M. Alpuente y S. Escobar
    Automatic Certification of Source Java Code in Rewriting Logic
    Technical Report. February 2007. DSIC, Technical University of Valencia.

- M. Alba-Castro, M. Alpuente y S. Escobar
  Verificación Automática de Propiedades de Seguridad en Protocolos de Seguridad y en Código Móvil: Una Aproximación Unificada al Estado del Arte. Informe Técnico. Febrero 2007. DSIC, Universidad Politécnica de Valencia.