



Universidad
ARTURO PRAT
del Estado de Chile

DEPARTAMENTO DE INGENIERIA

**ANALISIS, DISEÑO E IMPLEMENTACION DE pyTOD, UN PROTOTIPO
EXPERIMENTAL PARA REALIZAR DEPURACION OMNISCIENTE A
SCRIPTS ESCRITOS EN EL LENGUAJE DE PROGRAMACION PYTHON**

ACTIVIDAD DE TITULACION PRESENTADA PARA OPTAR A:
GRADO ACADEMICO DE LICENCIADO EN CIENCIAS DE LA INGENIERIA
TITULO DE INGENIERO CIVIL EN COMPUTACION E INFORMATICA

ALUMNO : MILTON INOSTROZA AGUILERA

PROFESOR GUIA : DAVID CONTRERAS AGUILAR

PROFESOR CORRECTOR : WILSON CASTILLO ROJAS

IQUIQUE - CHILE
2008

Resumen

La depuración es la mayor tarea del proceso de construcción de programas de un software, considerándolo en términos de tiempo y costo. Desafortunadamente los depuradores en muchos ambientes de desarrollo de software sólo entregan una asistencia mínima contribuyendo a que la depuración sea algo tedioso y una tarea que consume mucho tiempo.

Existen dos métodos tradicionales para realizar depuración de programas de un software:

1. *log-based*, basado en mensaje, y
2. *breakpoint-based*, basado en punto de quiebre.

La primera propuesta consiste en ir insertando registros de declaraciones dentro del código fuente, en el orden que se produce una huella ad-hoc durante la ejecución del programa. Esta técnica expone la historia actual de ejecución del programa pero:

- Requiere grandes modificaciones del código fuente.
- Esta técnica no es escalar, debido a que el análisis manual de grandes huellas es complicado.

El segundo método consiste en ejecutar el programa bajo un depurador dedicado, cuando el programador permita pausar la ejecución en un punto determinado del software, este podrá examinar el contenido de la memoria, y/o continuar la ejecución paso a paso. A pesar de lo anterior, la depuración basada en *breakpoint* es limitada: cuando la ejecución es pausada, la información sobre el estado y actividad anterior del programa es limitada por la introspección¹ de la pila² de llamada actual.

Aunque los dos métodos anteriormente señalados han sido utilizados ampliamente existe una tercera propuesta llamada depuración omnisciente, también conocida como *atrás-en-el-tiempo* o *después-de-la-muerte*. Esta propuesta supera todas las características de los métodos anteriores. Un depurador *omnisciente* registra los eventos que ocurren durante la ejecución del programa depurado y en seguida entrega al usuario un conveniente navegador por medio del cual obtiene la huella de ejecución. Esta propuesta combina las ventajas de la depuración basada en log – la actividad pasada no se pierde nunca – y también de la depuración basada en breakpoint – fácil navegación, ejecución paso a paso e inspección completa de la pila de ejecución. Un depurador omnisciente puede simular la ejecución paso a paso hacia adelante y hacia atrás, y es posible construir inmediatamente preguntas/respuestas que podrían de una u otra forma exigir un esfuerzo significativo al programador, como ¿En qué punto la variable *x* fue asignada al valor *y*? o ¿En qué estado estaba el objeto *o* cuando se le pasó un argumento al método *foo*?

Es importante señalar que los depuradores omniscientes además de superar todas las características anteriormente señaladas poseen una característica única y es la de identificar la causa inicial de los bugs³.

Existen varias implementaciones de los depuradores omniscientes, pero se basará el caso de estudio en TOD⁴, un depurador omnisciente orientado a la huella de ejecución. TOD depura sólo programas escritos en Java y para esto brinda cuatro características principales:

1. Stepping, ejecución paso a paso del programa a depurar.
2. Estado de reconstitución.
3. Reconstitución del control de flujo.

¹Introspección: Capacidad de un programa de razonar sobre su propia estructura

²Pila: Estructura de datos de tipo LIFO que permite almacenar y recuperar datos

³Bug: Defecto de software

⁴TOD: Trace-Oriented debugger, implementado en el lenguaje de programación Java

4. Identificar la causa inicial de los bugs.

El desafío que se plantea en este trabajo de título es diseñar un prototipo experimental llamado pyTOD que permita realizar depuración omnisciente a scripts escritos en un lenguaje de programación con tipado dinámico llamado Python, basándose en el estudio, análisis y utilización de TOD.

Índice general

1. Introducción	11
2. Propósito general	13
2.1. Herramientas	17
2.2. Proceso de depuración	17
2.3. Objetivo general	18
2.4. Objetivos específicos	18
2.5. Hipótesis del trabajo	19
2.6. Alcance del trabajo	20
2.7. Limitaciones y supuestos	20
3. Metodología del trabajo	21
3.1. Búsqueda bibliográfica	21
3.2. Análisis del Marco teórico	21
3.3. Análisis de TOD	21
3.4. Modelamiento y diseño de pyTOD	22
3.5. Implementación de pyTOD	22
3.6. Evaluación de pyTOD en casos de pruebas	22
4. Marco teórico	23
4.1. Introducción	23
4.1.1. Identificación del bug	23
4.1.2. Réplica del bug	24
4.1.2.1. Situaciones de difícil replicación	25
4.1.3. Comprensión del bug	26
4.1.3.1. Entender el programa	27
4.1.3.2. Encontrar el bug	27
4.1.3.3. Identificar el error	29

ÍNDICE GENERAL	2
4.1.4. Corregir el bug	30
4.2. Técnicas de depuración	30
4.2.1. Depuración basada en mensajes	30
4.2.1.1. Motivación	30
4.2.1.2. Proceso de depuración	31
4.2.1.3. Ventajas/Desventajas	32
4.2.1.4. Herramientas	32
4.2.1.4.1. Log4j	32
4.2.1.4.2. django-logging	33
4.2.2. Depuración basada en puntos de quiebre	34
4.2.2.1. Motivación	34
4.2.2.2. Proceso de depuración	34
4.2.2.3. Ventajas/Desventajas	34
4.2.2.4. Herramientas	35
4.2.2.4.1. Depurador de Eclipse	35
4.2.2.4.2. Depurador Winpdb	36
4.2.3. Depuración Omnisciente	36
4.2.3.1. Motivación	36
4.2.3.1.1. Depuración omnisciente	37
4.2.3.2. Proceso de depuración	37
4.2.3.2.1. Mantener el estado	37
4.2.3.2.2. Presentación	37
4.2.3.2.3. Identificar el estado	37
4.2.3.2.4. Navegación	38
4.2.3.3. Ventajas/Desventajas	38
4.2.3.4. Herramientas	38
4.3. Lenguajes de Programación	39
4.3.1. Clasificación de los lenguajes de programación	39
4.3.1.1. Según su nivel de abstracción	40
4.3.1.1.1. Lenguajes de bajo nivel	40
4.3.1.1.2. Lenguajes de nivel medio	40
4.3.1.1.3. Lenguajes de alto nivel	40
4.3.1.2. Según la forma de ejecución	40
4.3.1.2.1. Lenguajes compilados	40

ÍNDICE GENERAL	3
4.3.1.2.2. Lenguajes interpretados	41
4.3.1.3. Según el paradigma de programación	41
4.3.1.3.1. Paradigma Imperativo	41
4.3.1.3.2. Paradigma Declarativo	42
4.3.1.3.3. Paradigma Estructurado	42
4.3.1.3.4. Paradigma Funcional	42
4.3.1.3.5. Paradigma Lógico	42
4.3.1.3.6. Paradigma Orientado a Objetos	42
4.3.2. Python	43
4.3.2.1. ¿Qué es Python?	43
4.3.2.2. Lenguaje interpretado	43
4.3.2.3. Tipado dinámico	43
4.3.2.4. Fuertemente tipado	44
4.3.3. Comparativa entre Python y Java	44
5. Implementación de depuradores omniscientes	49
5.1. Implementaciones históricas	49
5.1.1. EXDAMS	49
5.1.1.1. Historia	49
5.1.1.2. Motivación	50
5.1.1.3. Características	50
5.1.1.4. Objetivos de diseño	51
5.1.1.5. Depuración y monitoreo	51
5.1.1.5.1. Análisis de flujo	52
5.1.1.6. Arquitectura	53
5.1.1.6.1. Análisis del programa	53
5.1.1.6.2. Compilación	54
5.1.1.6.3. Recolección de historia	54
5.1.1.6.4. Historia del programa	54
5.2. Implementaciones recientes	54
5.2.1. ODB	54
5.2.1.1. La naturaleza de los bug bajo ODB	55
5.2.1.1.1. La serpiente en el pasto	55
5.2.1.1.2. Tamaño	56

ÍNDICE GENERAL	4
5.2.1.1.2.1. Recolección de basura	56
5.2.1.1.2.2. Código seguro	57
5.2.1.1.2.3. Comenzar/Detener registro	57
5.2.1.1.3. Implementación	57
5.3. TOD: Depurador orientado a la huella para Java	58
5.3.1. Contribución	58
5.3.2. Desafíos	59
5.3.2.1. Características	59
5.3.2.2. Desafío de escalabilidad	60
5.3.3. Acerca de TOD	61
5.3.3.1. Arquitectura	61
5.3.3.2. Representación y emisión de eventos	62
5.3.3.3. Bajo nivel de consultas	64
5.3.3.4. Alto nivel de consultas	66
5.3.3.5. Componentes de la interfaz gráfica	67
5.3.4. Soporte para base de datos de alta velocidad	68
5.3.4.1. Jerarquía de indexación de los eventos	69
5.3.4.2. Costo de la creación de un índice	70
5.3.4.3. Costo de recuperación del evento	71
5.3.5. Medidas de rendimiento	72
5.3.5.1. Desempeño de la base de datos	73
5.3.5.2. Sobrecarga por emisión de eventos	76
5.3.5.3. Discusión	77
5.3.6. Trabajando con huellas parciales	78
5.3.6.1. Ejemplo de motivación	78
5.3.6.2. Tratando con información incompleta	79
6. pyTOD	80
6.1. Arquitectura de pyTOD	80
6.1.1. Modelo de componentes	80
6.1.2. Wrapper	81
6.2. Diseño de pyTOD	82
6.3. Implementación de pyTOD	83
6.3.1. Capturador de huella	83

- 6.3.1.1. Sucesos estructurales 85
 - 6.3.1.1.1. Registro de clase 85
 - 6.3.1.1.2. Registro de método 86
 - 6.3.1.1.3. Registro de método especial 87
 - 6.3.1.1.4. Registro de función 87
 - 6.3.1.1.5. Registro de atributo de instancia 87
 - 6.3.1.1.6. Registro de atributo de clase 88
 - 6.3.1.1.7. Registro de variable local 91
 - 6.3.1.1.8. Registro de thread 94
 - 6.3.1.1.9. Registro de probe 94
 - 6.3.1.1.10. Registro de objeto 94
- 6.3.1.2. Eventos 95
 - 6.3.1.2.1. Llamada método/función 95
 - 6.3.1.2.2. Asignación/modificación 96
 - 6.3.1.2.2.1. Atributo de instancia 97
 - 6.3.1.2.2.2. Atributo de clase 98
 - 6.3.1.2.2.3. Variable local 99
 - 6.3.1.2.3. Instanciación 101
 - 6.3.1.2.4. Retorno de método/función 101
 - 6.3.1.2.5. Excepción 102
- 6.3.2. Protocolo Comunicación de pyTOD 105
 - 6.3.2.1. Identificadores 105
 - 6.3.2.1.1. Sucesos 105
 - 6.3.2.1.2. Objetos 105
 - 6.3.2.1.3. Tipo de datos 105
 - 6.3.2.2. Registro de objetos 106
 - 6.3.2.2.1. Función 106
 - 6.3.2.2.2. Variable local 106
 - 6.3.2.2.3. Clase 106
 - 6.3.2.2.4. Método 107
 - 6.3.2.2.5. Atributo de instancia 107
 - 6.3.2.2.6. Atributo de clase 107
 - 6.3.2.2.7. Thread 107
 - 6.3.2.2.8. Probe 107

- 6.3.2.2.9. Objeto 108
 - 6.3.2.2.10. Excepción 108
 - 6.3.2.3. Llamada de objetos 108
 - 6.3.2.3.1. Función 108
 - 6.3.2.3.2. Método 108
 - 6.3.2.4. Instanciación de clase 109
 - 6.3.2.5. Asignación - Modificación de objetos 109
 - 6.3.2.5.1. Variable local 109
 - 6.3.2.5.2. Atributo de instancia 109
 - 6.3.2.5.3. Atributo de clase 110
 - 6.3.2.6. Return 110
- 6.3.3. Estructuras y Clases de pyTOD 110
 - 6.3.3.1. Estructura 110
 - 6.3.3.2. Clases 111
 - 6.3.3.2.1. hunterTrace 111
 - 6.3.3.2.1.1. Atributos 111
 - 6.3.3.2.1.2. Métodos 113
 - 6.3.3.2.2. generatorId 114
 - 6.3.3.2.2.1. Atributos 114
 - 6.3.3.2.2.2. Métodos 114
 - 6.3.3.2.3. Class 114
 - 6.3.3.2.3.1. Atributos 115
 - 6.3.3.2.3.2. Métodos 115
 - 6.3.3.2.4. Method 115
 - 6.3.3.2.4.1. Atributos 115
 - 6.3.3.2.4.2. Métodos 116
 - 6.3.3.2.5. Function 116
 - 6.3.3.2.5.1. Atributos 116
 - 6.3.3.2.5.2. Métodos 116
 - 6.3.3.2.6. Dictionary 117
 - 6.3.3.2.6.1. Atributos 117
 - 6.3.3.2.6.2. Métodos 117
- 6.3.4. Código Principal de pyTOD 117
- 6.3.5. Interfaces principales de pyTOD 119

ÍNDICE GENERAL	7
6.3.5.1. Arrancar el depurador	119
6.3.5.2. Realizando depuración	120
6.4. Aspectos Básicos de Funcionamiento de pyTOD	121
6.4.1. Usuario final	121
6.4.2. Desarrollador	121
6.5. Medidas de rendimiento de pyTOD	125
7. Evaluación de pyTOD en casos de pruebas	127
7.1. Selección de casos de uso	127
7.2. Resultados obtenidos con pyTOD	129
7.2.1. División por cero	129
7.2.2. Fibonacci	131
7.2.3. Manejador de archivo	134
8. Conclusiones	136
9. Trabajo futuro	139
Apéndices	144
A. Código fuente de pyTOD	144

Índice de figuras

4.1. Resultado de depuración con django-logging	33
4.2. Resultado de depuración con Eclipse	35
4.3. Resultado de depuración con Winpdb	36
4.4. Tipado estático	48
4.5. Tipado dinámico	48
5.1. Ventana principal de depuración	55
5.2. Arquitectura de alto nivel de TOD	61
5.3. Eventos y sus atributos	62
5.4. Navegación entre la unión de los eventos y el predicado del cursor . . .	65
5.5. Navegación con TOD en Eclipse	68
5.6. Mural de threads	69
5.7. Jerarquía de los índices	70
5.8. Rendimiento del lanzador	74
5.9. Rendimiento del registro	74
5.10. Eventos emitidos y sobrecarga en tiempo de ejecución.	79
5.11. Reportando una información de flujo de control potencialmente incom- pleta.	79
6.1. Diagrama de componente del capturador de huella	80
6.2. Diagrama de composición wrapper	82
6.3. Diagrama de diseño de pyTOD	82
6.4. Diagrama de clase del capturador de huella	111
6.5. Diagrama de clase de hunterTrace	114
6.6. Diagrama de clase de generatorId	115
6.7. Diagrama de clase de Class	115
6.8. Diagrama de clase de Method	116
6.9. Diagrama de clase de Function	117

6.10. Diagrama de clase de Dictionary 117

6.11. Botón para arrancar el depurador 119

6.12. Proceso Eclipse hijo lanzado 119

6.13. Perspectiva inicial para depuración 119

6.14. Botón de depuración 120

6.15. Interfaz inicial luego de haber depurado el código del usuario 120

6.16. Control de flujo 120

6.17. Línea inspeccionada 121

6.18. Argumentos de GridMaster 122

6.19. Classpath de Gridmaster 123

6.20. Argumentos de StandaloneUI 123

6.21. Classpath de StandaloneUI 124

6.22. Argumentos de TOD Plugin 124

7.1. Excepción lanzada por el script division.py 129

7.2. Eventos sucedidos para el script division.py 130

7.3. Control de flujo para el script division.py 130

7.4. Eventos ocurridos dentro del método division 130

7.5. Estructura de script division.py 131

7.6. Método division con sus variables locales 131

7.7. Resultado de ejecución para el script fibonacci.py 131

7.8. Eventos sucedidos para el script fibonacci.py 132

7.9. Control de flujo para el script fibonacci.py 132

7.10. Control de flujo para fibonacci(5) 133

7.11. Control de flujo para fibonacci(4) 133

7.12. Control de flujo para fibonacci(3) 133

7.13. Control de flujo para fibonacci(2) 133

7.14. Eventos sucedidos para el script manejadorArchivo.py 134

7.15. Control de flujo para el script manejadorArchivo.py 134

7.16. Eventos dentro de la llamada de converter 134

7.17. Excepción dentro del método abrir 135

7.18. Valores de variables cercanas a la excepción 135

Índice de tablas

5.1. Operaciones de los cursores	65
5.2. Consultas de cursores	76
5.3. Consultas de contadores	76
5.4. Sobrecarga en la emisión de eventos	77
6.1. Identificadores de sucesos	105
6.2. Identificadores de objetos	105
6.3. Identificadores de tipo de datos	106
6.4. Registro del objeto función	106
6.5. Registro del objeto variable local	106
6.6. Registro del objeto clase	106
6.7. Registro del objeto método	107
6.8. Registro del objeto atributo de instancia	107
6.9. Registro del objeto atributo de clase	107
6.10. Registro del objeto thread	107
6.11. Registro del objeto probe	107
6.12. Registro del objeto objeto	108
6.13. Registro del objeto excepción	108
6.14. Llamada al objeto función	108
6.15. Llamada al objeto método	108
6.16. Coordenadas	109
6.17. Instanciación de clase	109
6.18. Registro del objeto variable local	109
6.19. Registro del objeto atributo de instancia	110
6.20. Registro del objeto atributo de clase	110
6.21. Registro de return	110

Capítulo 1

Introducción

El proceso de depuración de programas de un software se hace cada vez más necesario debido al crecimiento y evolución de los lenguajes de programación (cambios de paradigmas y características). Es importante que el programador disponga de ambientes adecuados para que de una forma fácil, rápida e intuitiva pueda corregir los defectos del software. Si el programador no cuenta con un ambiente adecuado para poder inspeccionar los programas de su software y corregir ciertas anomalías el costo y tiempo del desarrollo de software aumenta dramáticamente [1]. En muchos casos los programas de un software desarrollado no son expuestos a pruebas exhaustivas para conocer su verdadero comportamiento en diversos ambientes, debido a lo costoso de esto, lo que conlleva a tener comportamientos inesperados en la fase de producción de estos.

El proceso anterior se define como la manera de identificar y eliminar errores dentro de un programa perteneciente a un software y sigue siendo, hoy en día a pesar del avance de la tecnología, en buena medida una actividad manual. Actividad que desafía la paciencia, imaginación e intuición de los programadores. En muchos casos los programadores deben incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables, direcciones de memoria o lo que les sea necesario.

Es deseable que una herramienta de depuración de programas de un software evite que los programadores tengan que introducir líneas adicionales en sus programas para intentar saber el comportamiento de ciertos componentes de su software.

Es de esta forma como la industria del software ha planteado diversas soluciones, las cuales mitigan el pesado proceso de depuración de programas de un software. Las soluciones mayormente utilizadas son los depuradores basados en logs y basados en puntos de quiebre, relegando la solución de los depuradores omniscientes. Es en este

punto donde nacen preguntas fundamentales tales como ¿Conociendo las bondades de los depuradores omniscientes por qué no son ampliamente utilizados?, ¿A los programadores sólo les basta con los depuradores basados en *breakpoint* y/o basados en *log*?, ¿Cuáles son las desventajas que hace impracticable implementar un depurador omnisciente?, ¿No es necesario para el programador conocer la principal causa del defecto de software?.

Enfocándose en la solución de depuración omnisciente, no basta con sólo almacenar la historia del programa depurado, si no que se debe contar con un almacén de datos adecuado que sea escalable y eficiente para guardar esta enorme cantidad de información. Además de esto, un depurador omnisciente debe implementar una interfaz que responda en un tiempo aceptable todas las peticiones que el programador le solicite.

Es por lo anterior que se ha seleccionado como caso de estudio TOD (Trace-oriented Debugger) [2] un depurador omnisciente con almacenamiento escalable. Su interfaz gráfica responde rápidamente a las peticiones del programador y la sobre carga que produce en el ambiente en el cual se ejecuta es menor que otras implementaciones existentes sobre este concepto.

De este modo, este trabajo de memoria se centra en la implementación de un prototipo experimental, el cual realice depuración omnisciente a scripts escritos en el lenguaje de programación Python, utilizando como base de estudio e implementación TOD, un depurador omnisciente escrito en el lenguaje de programación Java.

Capítulo 2

Propósito general

Como concepto general se dice que software es el conjunto de programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación. [3]

La Ingeniería de Software plantea que existen tres grandes procesos para desarrollar software, que se detallan en orden cronológico:

1. Análisis de sistema
2. Diseño de sistema
3. Codificación de software.

En especial nos centraremos en el último proceso en donde se realizan las tareas que comúnmente se conocen como programación; que consiste, esencialmente, en llevar a código fuente, en el lenguaje de programación elegido, todo lo diseñado en la fase anterior. Esta tarea la realiza el programador, siguiendo por completo los lineamientos impuestos en el diseño y en consideración siempre a los requisitos funcionales y no funcionales (ERS) especificados en la primera etapa.

Es común pensar que la etapa de programación o codificación (algunos la llaman implementación) es la que consume la mayor parte del trabajo de desarrollo del software; sin embargo, esto puede ser relativo (y generalmente aplicable a sistemas pequeños) ya que las etapas previas son cruciales y críticas, las que pueden llevar bastante más tiempo. Se suele hacer estimaciones de un 30 % del tiempo total consumido en la programación, pero esta cifra no es consistente ya que depende en gran medida de las características del sistema, el nivel crítico que tenga y el lenguaje de programación elegido. En tanto menor es el nivel del lenguaje de programación mayor será el tiempo de programación requerido, así por ejemplo se tardaría más tiempo en codificar un

algoritmo en ensamblador que el mismo programado en lenguaje C.

Durante la fase de programación, el código puede adoptar varios estados, dependiendo de la forma de trabajo y del lenguaje de programación elegido. Para introducir conceptos fundamentales en este tema de memoria, es importante precisar lo siguiente:

- **Código fuente** : Es el escrito directamente por los programadores en editores de texto, lo cual genera el programa. Contiene el conjunto de instrucciones codificadas en algún lenguaje de programación. Puede estar distribuido en paquetes, procedimientos, librerías fuente, etc.
- **Código objeto** : Es el código binario o intermedio resultante de procesar con un compilador el código fuente. Consiste en una traducción completa y de una sola vez de éste último. El código objeto no es inteligible por el ser humano (normalmente es formato binario) pero tampoco es directamente ejecutable por el computador. Se trata de una representación intermedia entre el código fuente y el código ejecutable, a la espera de un enlace final con las rutinas de librería y entre procedimientos.
 - El código objeto no existe si el programador trabaja con un lenguaje a modo de intérprete puro, en este caso el mismo intérprete se encarga de traducir y ejecutar línea por línea el código fuente (de acuerdo al flujo del programa), en tiempo de ejecución. En este caso tampoco existe él o los archivos de código ejecutable. Una desventaja de esta modalidad es que la ejecución del programa o sistema es un poco más lenta que si se hiciera con un intérprete intermedio, y bastante más lenta que si existe el o los archivos de código ejecutable. Es decir no favorece el rendimiento en velocidad de ejecución. Una gran ventaja de la modalidad intérprete puro, es que esta forma de trabajo facilita enormemente la tarea de depuración del código fuente (frente a la alternativa de hacerlo con un compilador puro). Frecuentemente se suele usar una forma mixta de trabajo (si el lenguaje de programación elegido lo permite), es decir inicialmente trabajar a modo de intérprete puro, y una vez depurado el código fuente (liberado de errores) se utiliza un compilador del mismo lenguaje para obtener el código ejecutable completo, con lo cual se agiliza la depuración y la velocidad de ejecución se optimiza.
- **Código ejecutable** : Es el código binario resultante de enlazar uno o más fragmentos de código objeto con las rutinas y librerías necesarias. Constituye uno o

más archivos binarios con un formato tal que el sistema operativo es capaz de cargarlo en la memoria RAM (eventualmente también parte de este en una memoria virtual), y proceder a su ejecución directa. Por lo anterior se dice que el código ejecutable es directamente *inteligible por la computadora*. El código ejecutable, también conocido como código máquina, no existe si se programa con modalidad de *intérprete puro*.

- **Compilador :** Es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. Usualmente el segundo lenguaje es código máquina, pero también puede ser simplemente texto. Este proceso de traducción se conoce como compilación. De esta manera un programador puede diseñar un programa en un lenguaje mucho más cercano a como piensa un ser humano, para luego compilarlo a un programa manejable por un computador.
- **Intérprete :** Es un programa informático capaz de analizar y ejecutar otros programas, escritos en un lenguaje de programación de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código máquina del sistema destino, los primeros (los intérpretes) sólo realizan la traducción a medida que sea necesario, comúnmente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

Mientras se programa la aplicación, sistema, o software en general, se realizan también tareas de depuración, esto es la labor de ir liberando al código de los errores factibles de ser hallados en esta fase (de semántica, sintáctica y lógica).

De acuerdo con Grace Murray Hopper, uno de los pioneros de la ciencia de la computación, la situación que dio origen al término depuración sucedió a principio de la década de los años 1950, los programadores de la Universidad de Harvard invirtieron semanas en intentos infructuosos para encontrar el error en uno de sus programas. Finalmente, una investigación dentro de las computadoras reveló que un insecto había muerto ahí. Una vez removido el insecto, el programa funcionó correctamente. Desde entonces, el proceso de quitar errores de los programas ha sido llamado *depuración*.

Pero, de acuerdo a Edsger Dijkstra, otro pionero en las ciencias de la computación, el término es irresponsable. Depuración sugiere que el programador no tiene la culpa

por el error. Esto es como si el insecto trepara dentro del código mientras el programador estaba mirando hacia otro lado.

Aunque el término tenga algunas dicidencias podemos decir que depuración es un proceso metódico, el cual encuentra y reduce el número de bug, o defectos, en un programa de computación, haciendo de esta forma que estos se comporten como es esperado. La depuración tiende a ser dura cuando varios subsistemas, que contienen muchos programas, están estrechamente acoplados, y cambios en uno pueden provocar bugs en otros.

La Depuración es, en general, una pesada y cansadora tarea. La habilidad para depurar del programador es probablemente el mayor factor en la capacidad de resolver un problema, pero la dificultad de la depuración de software varía enormemente referente al lenguaje de programación utilizado y a las herramientas disponibles, tales como los depuradores.

Los depuradores son herramientas de software, las cuales permiten al programador monitorear la ejecución de un determinado programa, detenerlo, reactivarlo, poner puntos de quiebre, cambiar valores en memoria e incluso, en algunos casos, ir atrás en el tiempo. El término depurador puede estar además relacionado con la persona que realiza la depuración.

Generalmente, los lenguajes de alto nivel, hacen la depuración fácil, porque ellos tienen características tales como manejo de excepciones que permiten manejar el comportamiento errático de una manera fácil. En los lenguajes de nivel medio como C o bajo nivel como ensamblador, los bugs pueden causar silenciosos problemas, un ejemplo de estos es la corrupción de memoria, y esto frecuentemente dificulta ver donde el problema principal ha ocurrido. En estos casos, las herramientas de depuración de memoria pueden ser útiles.

En determinadas situaciones, herramientas de software de propósito general que son lenguajes específicos pueden ser muy útiles. Estas son llamadas herramientas de análisis estático. Estas herramientas observan problemas conocidos que son muy específicos, algunos comunes y algunos extraños, dentro del código fuente. Cada uno de estos son detectados por estas herramientas y raramente son detectados por el compilador o intérprete, por lo tanto ellos no son correctores sintácticos, si no más bien correctores semánticos.

2.1. Herramientas

Un depurador es un software que es utilizado para probar y depurar otros programas. Una técnica de gran utilidad en esto es la capacidad de suspender el programa depurado, cuando una condición específica es encontrada.

Cuando un programa lanza un error, el depurador muestra la posición en el código original donde se produjo dicho error, esto si es que es un depurador a nivel de código fuente, comúnmente vistos en ambientes de desarrollos integrados. Un lanzamiento de error sucede cuando el programa no puede manejar un comportamiento extraño o desconocido debido a un defecto de software (más conocido como Bug). Por ejemplo, tal vez el programa intentó utilizar una instrucción no disponible en la versión actual de la CPU o intentó acceder a memoria no disponible o protegida.

Comúnmente, los depuradores además ofrecen prestaciones más sofisticadas como ejecutar un programa paso a paso (*step by step*), detener la ejecución (detener la ejecución del programa para examinar el estado actual) en algún tipo de evento mediante un punto de quiebre (*breakpoint*), y seguirle la pista a los valores de determinadas variables. Algunos depuradores tienen la capacidad de modificar el estado del programa mientras este se ejecuta y así no solamente son meros espectadores de la ejecución de éste.

2.2. Proceso de depuración

La depuración comienza tratando de reproducir un problema. Esto puede ser una tarea no trivial, por ejemplo en el caso de procesos paralelos o algún problema inusual de software. Además el ambiente específico del usuario y el uso de la historia puede hacer que sea difícil reproducir el problema.

Después de que el bug es reproducido, la entrada del programa necesita ser simplificada para hacer más fácil el proceso de depuración. Por ejemplo, un bug en un compilador puede hacer que este falle cuando este analizando sintácticamente un programa fuente muy largo. Tal simplificación puede ser realizada manualmente, utilizando un enfoque *divide y vencerás*. El programador intentará sacar algunas partes del caso original de prueba y verificar si el problema aún continúa. Cuando se depura el problema en una interfaz gráfica, el programador tratará de saltarse alguna interacción desde la descripción original del problema y verificar si las acciones restantes son suficientes para que aparezca el bug.

Después de que el caso de prueba es suficientemente simplificado, el programador puede usar el depurador para examinar el estado del programa (valores de las variables, la pila de llamadas) e identificar el origen del problema. De forma alternativa un rastreo puede ser utilizado. En un caso sencillo el rastreo consiste sólo en algunas instrucciones de impresión, las cuales muestran el valor de las variables en determinados puntos del programa en ejecución.

La depuración remota es el proceso de depuración de un programa que está ejecutándose en un sistema diferente que el depurador. Para comenzar la depuración remota, el depurador se conecta a un sistema remoto sobre una red de datos. Una vez conectado, el depurador puede controlar la ejecución del programa sobre un sistema remoto y recobrar la información sobre el estado de éste.

2.3. Objetivo general

El objetivo general de este trabajo de memoria es:

- Construir un prototipo experimental llamado pyTOD para realizar depuración omnisciente a scripts escritos en el lenguaje de programación Python, utilizando como base de estudio TOD, un depurador omnisciente implementado en el lenguaje de programación Java.

Esto implica brindar un ambiente de depuración que sea eficiente para el programador, además de entregar una herramienta básica para inspeccionar programas y poder encontrar las causas de los bugs.

2.4. Objetivos específicos

Para lograr el éxito del objetivo general de este trabajo de título se han definido los siguientes objetivos específicos:

1. *Diseñar e Implementar un sistema que instrumente¹ el código Python de tal forma que envíe eventos a la base de datos de TOD.*

A través de la metodología de *ensayo y error* se implementará un capturador de huella de ejecución ad-hoc, que capturará todos los sucesos ocurridos en el programa objetivo escrito en Python, para luego enviarlos a la base de datos de TOD.

¹Agregar instrucciones al código para que realice determinadas tareas

2. *Analizar la compatibilidad del modelo actual de huella de ejecución¹ de TOD con el lenguaje de programación Python.*

Una vez que el capturador de huellas funcione parcialmente, se comenzará con el estudio de compatibilidad de huella de ejecución de TOD. De ser necesario el modelo actual de huella de ejecución de TOD será modificado para conseguir compatibilidad con pyTOD.

3. *Diseñar e Implementar un protocolo de comunicación entre pyTOD (Python) y TOD (Java)*

Será necesario implementar un protocolo de comunicación el cual transporte la huella de ejecución capturada en pyTOD hacia la base de datos de TOD. Esto se realizará una vez que se haya analizado y modificado el modelo de huella de ejecución de TOD.

4. *Diseñar e Implementar un plugin para eclipse el cual permita conectar la interfaz gráfica del depurador con el ambiente de desarrollo.*

No basta con tener una interfaz gráfica independiente del ambiente de desarrollo del programador, es fundamental que la interfaz de usuario del depurador sea incorporado como plugin dentro del ambiente de desarrollo. Para este caso se considerará como ambiente de desarrollo el IDE Eclipse². Se utilizará el plugin de Python para Eclipse, PyDev como base de extensión para el nuevo plugin el cual permita al programador una interacción transparente entre el ambiente de desarrollo y el ambiente de depuración.

5. *Aplicar pyTOD en un caso de prueba real.*

El resultado de éxito y contribución de este trabajo de título depende mayormente de este punto. Es importante que la herramienta desarrollada pyTOD, sea útil y demuestre con resultados reales que es una aproximación a lo que podría ser un ambiente de depuración para asistir efectivamente a los programadores que utilizan el lenguaje de programación Python.

2.5. Hipótesis del trabajo

¿Es posible utilizar TOD, como base de desarrollo técnico y conceptual, para construir un prototipo experimental que realice depuración omnisciente a scripts escritos en Python?.

¹Secuencia ordenada de eventos heterogéneos de un programa computacional.

²Entorno integrado de desarrollo de software

2.6. Alcance del trabajo

Se plantea construir un prototipo experimental el cual realice depuración básica a scripts escritos en Python. Si es necesario y existe la compatibilidad suficiente, se reutilizarán algunas herramientas que utiliza TOD para realizar este tipo de depuración. En ningún caso este trabajo producirá un Depurador Omnisciente con la implementación completa de las funcionalidades que este implica.

2.7. Limitaciones y supuestos

El prototipo experimental no realizará depuración a programas que hagan uso de multithread.

Para mantener un registro exacto de todos los objetos¹ dentro del programa depurado se necesita marcarlos a cada uno de ellos con un identificador, Python para esto utiliza la función *id*, la cual entrega la dirección de memoria en donde se encuentra el objeto. En el momento que un objeto se destruye esta dirección puede volver a ser utilizada por el lenguaje de programación. Para la tarea de depuración se necesita un identificador único en todo momento y la única manera de realizarlo es modificando la maquina virtual, situación que queda fuera del alcance de este trabajo de memoria. Sin embargo esto no impide del todo implementar las funcionalidades de los depuradores omniscientes.

Todo el desarrollo y pruebas del prototipo se realizarán en un sistema operativo basado en el kernel de Linux, debido a esto no será un objetivo lograr compatibilidad con otros sistemas operativos.

¹Referente a todos los elementos dentro de la ejecución de un programa (métodos, funciones, atributos, instancia de clases, etc.)

Capítulo 3

Metodología del trabajo

3.1. Búsqueda bibliográfica

Esta etapa principalmente se enfocó a la búsquedas de papers relacionados con el tema de depuración en general. Se utilizó como patrón de búsqueda la bibliografía utilizada en el paper de TOD [2] que está relacionada con el concepto de depuración y depuradores omniscientes.

A medida que fue avanzando la memoria, se utilizó la documentación en línea acerca del lenguaje de programación Python.

3.2. Análisis del Marco teórico

Se comenzó conceptualizando las diferencias entre distintos tipos de lenguajes de programación, luego los términos de depuración en general, para luego profundizar en la depuración omnisciente.

Posterior a esto se estudiaron las distintas aproximaciones técnicas (depuradores) que existen para implementar la depuración omnisciente.

3.3. Análisis de TOD

TOD es un depurador omnisciente que tiene como principal característica ser escalable en el sentido de almacenamiento de la huella de ejecución. Para el actual trabajo se deben analizar los siguientes componentes de TOD:

- **Structure Database** : Base de datos escrita en Java que almacena toda la estructura (definiciones) del programa depurado.

- **Event Database** : Base de datos escrita en Java que almacena todos los eventos ocurridos en el programa depurado.
- **Debugger frontend** : Interfaz gráfica que es utilizada por el programador para navegar a través de la huella de ejecución del programa depurado.

3.4. Modelamiento y diseño de pyTOD

Se modeló la arquitectura de pyTOD y los componentes bases para su funcionamiento. En esta etapa se establecieron todas las modificaciones necesarias que se tuvieron que realizar a los componentes de TOD (base de datos estructural, base de datos de sucesos e interfaz gráfica) para que fuera compatible con pyTOD.

3.5. Implementación de pyTOD

Esta sección comprende todo el desarrollo práctico del presente trabajo de memoria, en el cual se implementó el capturador de huella utilizando el paradigma de orientación de objetos, luego la interfaz de conexión con la base de datos de TOD, luego su protocolo de comunicación y finalmente la interfaz gráfica para un ambiente de desarrollo.

3.6. Evaluación de pyTOD en casos de pruebas

Se tomaron tres algoritmos implementados en el lenguaje de programación Python. Algoritmos adecuados para ver el funcionamiento y comportamiento de pyTOD. Los comportamientos que se midieron fueron la capacidad de encontrar bugs y la de registrar la huella de ejecución.

Capítulo 4

Marco teórico

4.1. Introducción

En la práctica, un programador inevitablemente invierte demasiado tiempo encontrando y solucionando bugs. La Depuración, particularmente la realizada a programas escritos por otras personas, es una habilidad independiente a la de escribir programas correctos al primer intento.

Desafortunadamente, mientras la depuración es frecuentemente utilizada, esta es raras veces enseñada. Un curso típico sobre técnicas de depuración consiste únicamente en estudiar el manual de un depurador determinado.

4.1.1. Identificación del bug

Depuración significa remover los bugs de un programa. Un bug es un comportamiento del programa que es inesperado y no deseado.

Ocasionalmente existe una especificación formal que un programa está obligado a seguir, en este caso un bug es la falla en el seguimiento de esta especificación. Frecuentemente la especificación del programa es informal, en este caso las personas pueden discutir si un comportamiento particular es de hecho un bug o no.

Como una de las personas que escriben programas, el programador debe ser la fuente de los reportes de bugs. No simplemente confiando en los usuarios finales o en los dedicados a probar determinadas funcionalidades. Si el programador nota algo extraño mientras el programa se está ejecutando, este puede tentarse a ignorarlo y esperar que todo vaya bien en el programa. Esto es aceptable si el programador en ese momento está trabajando en otra cosa. Si el programador tiene archivos o reportes del bug, es mejor que siempre los guarde. Si tiene tiempo después, debe volver al caso anómalo. Si

esto no es posible, sería recomendable que alguien más estudie este problema para que lo pueda solucionar. Como una de las personas más familiarizadas con el programa, el programador está en la mejor posición para detectar comportamientos inesperados.

Una vez que el programador tiene un reporte de bug, el primer paso para remover este bug es identificarlo. Esto es particularmente importante cuando se está trabajando con un reporte de bug que ha sido producido por otra persona, tales como usuarios o un equipo de pruebas. Algunos bug son relativamente obvios, como cuando el programa termina su ejecución inesperadamente. Otros son difíciles de encontrar, como cuando el programa genera una salida la cual es levemente incorrecta.

Muchos reportes de bug recibidos de los usuarios son de la forma *Yo hice esto y esto otro, y el programa falló*. Antes de hacer cualquier cosa, el programador debe encontrar que es lo que falló – esto es, el programador debe identificar el bug determinando el comportamiento del programa el cual fue inesperado o no deseado. Cualquier intento de corregir el bug antes de entender que es lo que falló es generalmente tiempo desperdiciado.

Identificar un reporte de bug hecho por un usuario, normalmente requiere obtener la respuesta a estas dos preguntas: *¿Qué hizo el programa?* y *¿Qué esperaba que hiciera el programa?*. El objetivo es determinar precisamente el comportamiento del programa el cual fue inesperado o no deseado.

Una vez que el bug es identificado, la forma más fácil y rápida de solucionarlo es determinar que este no es del todo un bug. Si existe una especificación formal del programa, el programador debe modificar la especificación. En otros casos, el programador debe modificar las expectativas del usuario. Esta rápida solución consiste en llamar a este comportamiento como una *característica no documentada*. A pesar del potencial abuso que se pueda cometer, esto es de hecho la forma correcta de manejar un problema.

Desafortunadamente, la mayoría de los bugs son realmente bugs y requieren de mucho esfuerzo para buscarlos, identificarlos y solucionarlos.

4.1.2. Réplica del bug

El primer paso para solucionar un bug es replicarlo, esto significa recrear el comportamiento no deseado bajo condiciones controladas. El objetivo es encontrar un conjunto de especificaciones precisas de los pasos, que demuestran el bug.

En muchos casos esto es sencillo, el programador ejecuta el programa con una

determinada información de entrada, o presiona un botón en particular o una ventana de diálogo y el bug ocurre. En otros casos, replicar el bug puede ser difícil. Esto puede requerir series de pasos muy largos, o en un programa interactivo como un juego, este puede requerir precisión de tiempo. En los peores casos, la replicación puede ser casi imposible.

Es importante que el programador nunca salte este paso. En la mayoría de los casos los programadores son tentados algunas veces a saltar el paso de replicación, e ir directamente desde el reporte del bug a la eliminación del mismo. Sin embargo, fallar en la replicación del bug significa que es imposible verificar la corrección. La solución propuesta puede provocar un bug diferente, o puede tener un efecto insignificante. Si el bug no ha sido replicado, no hay forma de conocerlo.

Fallar en replicar el bug es un problema real que puede suceder muchas veces. En un programa complejo, es frecuente encontrar algo que solucionar. Esta es la naturaleza del humano, la de asumir que cualquier solución particular resuelve el problema que se tiene. Sin necesitar verificaciones, cualquier solución convincente puede ser aceptada, esté bien o mal. Una solución incorrecta libera futuros problemas. En promedio, saltarse el paso de replicación implica que el programador gastará más tiempo en bugs relacionados a la solución que haya propuesto.

4.1.2.1. Situaciones de difícil replicación

Lejos, la mejor vía para replicar el bug es sobre un sistema que este completamente bajo el control del programador, con una copia del programa construida por el mismo. Replicando el bug sobre su propio programa significa que el puede fácilmente hacer pruebas y nuevos parches para el programa, desafortunadamente en algunos casos esto es imposible.

Una de las razones por lo que la replicación en el sistema del programador puede ser imposible, es debido a que el usuario que reporte el bug puede que tenga un único sistema de configuración, o puede que el usuario sólo use archivos de entradas confidenciales a los cuales el programador no pueda acceder. En estos caso el programador debe tratar de asegurar que el usuario puede realmente replicar el bug, y que el usuario pueda probar el nuevo parche desarrollado. Sin la ayuda del usuario en este sentido, la depuración es reducida a algo más pequeño, que son los supuestos. Esto generalmente no es un mal procedimiento. Si no existe opción, entonces el programador puede tratar al menos de construir algunos casos aleatorios de prueba para construir un parche, con-

firmando que este hace algo útil incluso si el programador no puede verificar que este elimina el bug original.

Un problema mucho más común es que el usuario reporte un bug, pero el programador no pueda replicarlo en su propio sistema, y el programador no sepa por qué. Esta no es una opción definitiva para resolver este tipo de problema, pero a continuación se detallan algunas posibles consideraciones:

- El usuario puede simplemente reportar de forma incorrecta el problema o de la forma en la cual puede ser replicado. El programador debe volver al usuario, confirmar que es lo que el usuario actualmente está escribiendo o donde el usuario actualmente está haciendo clicks, confirmar que es lo que el usuario ve en pantalla. El programador no debe hacer presunciones sobre como el programa está ejecutándose – el usuario puede estar haciendo algo que el programador nunca hubiera imaginado o considerado.
- Algunos programas tienen comportamientos extraños bajo condiciones inusuales. El programador debe verificar si el sistema del usuario tiene poco espacio en el disco duro, o tiene una conexión de red defectuosa, o si está muy sobrecargado. Es recomendable que a su vez verifique si otros programas están fallando inesperadamente y provocan el bug en el programa objetivo.
- El programador además debe revisar la versión de los programas del software que el usuario está ejecutando, revisar la versión del sistema operativo, etc.

En algunos casos un bug puede suceder raramente y aparentemente de forma aleatoria, entonces es difícil imaginarse la forma como estos son lanzados, y como replicarlos. En esta situación frustrante al programador sólo le queda tomar una opción confiable que es la de automáticamente registrar todos los eventos que potencialmente puedan ser relevantes y guardar todos estos registros cuando el bug ocurra. En algunos casos es particularmente útil ser capaz de volver a ejecutar el programa usando los registros, los cuales pueden ser mecanismos poderosos para replicar un bug.

4.1.3. Comprensión del bug

Una vez que el programador es capaz de replicar el bug, debe entender que es lo que lo produjo. Este es generalmente el paso que consume mayor tiempo.

4.1.3.1. Entender el programa

Con el objetivo de entender un bug en un programa, el programador debe entender algo del funcionamiento acerca del programa.

Si el programador escribió el programa, es presumible que ya lo entienda. De caso contrario, entonces el programador tiene un problema más serio.

Si el programador no ha escrito el programa, el necesita basarse en la estructura general del programa. Muchos programas están organizados de una forma sensata. Si el programador tiene suerte, la organización del programa estará documentada, o podrá preguntar a los diseñadores del programa.

Comúnmente, el programador necesitará sacar la estructura del código fuente. La mejor opción es comenzar mirando el código fuente desde el principio del programa. Leer rápidamente el programa, bajando por las funciones, hasta que encuentre el principal centro de acción. En la mayoría de los programas, se trata de algún tipo de ciclo. Esto puede hacerse normalmente muy rápido. La naturaleza de este centro de acción puede decirle donde mirar en el código fuente para cualquier actividad en particular. Esto, además puede decirle en líneas generales como es que el programa actúa.

El peor de los casos son los programas grandes escritos por muchos años y por diferentes desarrolladores. Estos llegan, en algunos casos, a ser un conjunto de ideas diferentes con muy poca consistencia. La situación comúnmente es depresiva. El programador solamente debe hacer lo mejor que pueda.

Un depurador puede además ser útil cuando se trata de entender un programa. Mediante la ejecución del programa bajo un depurador y definiendo puntos de quiebre, el programador puede ser capaz de ver el comportamiento dinámico del programa. Cuando alcanza el punto de quiebre definido por el programador, puede mirar la pila de llamadas para analizar como llegó a la parte en donde está, y mirar algunas variables claves. Incluso si el programador no alcanza el punto de quiebre definido por él, este ya habrá aprendido algo.

4.1.3.2. Encontrar el bug

El siguiente paso es localizar el bug en el código fuente del programa. Existen dos ubicaciones en el código fuente que se debieran considerar por el programador: (a) el código que hace que el comportamiento incorrecto sea visible y (b) el código que en realidad es incorrecto. Es muy común que éstas sean las mismas piezas de código. Sin embargo, es bastante común que éstas estén en diferentes partes del programa. Un

ejemplo típico de esto es cuando existe un error en una parte del programa que causa un acceso a memoria restringida el cual provoca una visión errónea del comportamiento en una parte completamente diferente en el programa. El programador no puede permitirse que en el afán de arreglar el error se confunda y piense que el código que directamente causa el comportamiento erróneo es en realidad la parte incorrecta del programa.

Por lo general, el programador tiene que encontrar el código que produce el comportamiento erróneo. Conociendo el comportamiento erróneo, y entendiendo como el código fuente fue organizado, a menudo esto lo dirige rápidamente a la parte del programa que causa el problema. Algunas veces una rápida lectura del código es suficiente para identificar el código problemático.

De lo contrario, disminuyendo el comportamiento erróneo en una particular pieza de código es donde un depurador puede ser de mucha utilidad. Si el programador tiene bastante suerte tendrá un volcado de memoria, un depurador inmediatamente puede identificar la línea que ha fallado. De lo contrario, juiciosamente el programador deberá ir poniendo puntos de quiebre mientras la replicación del bug puede ser rápidamente acotada sobre el código defectuoso.

Los Depuradores modernos, tienen poderosas capacidades para hacer este proceso más manejable, tales como los puntos de quiebre, los datos de los puntos de observación, ignorar puntos de quiebre en un determinado número de veces. Estas características, muchas veces, son muy útiles para localizar el comportamiento erróneo en el código fuente, y esto sólo falla en circunstancias especiales.

Por supuesto, un depurador a veces no ayuda, en algunos casos, un error no se produce cuando el programa se ejecuta bajo un depurador, a pesar de que se pueden reproducir sin contar con el depurador, lo que generalmente indica un problema que depende del momento preciso de ejecución o del diseño de la memoria (problemas específicos). En otros casos, puede que no tenga acceso a un depurador, o el depurador puede que no sea muy poderoso, lo que puede suceder cuando se trabaja en sistemas empotrados o de otros entornos de programación más restringidos, o cuando se utilizan características de programación que no están soportadas por el depurador (por ejemplo, en algunos ambientes distribuidos).

En estos casos simples las instrucciones de impresión a veces pueden ayudar a localizar la fuente del comportamiento erróneo. Añadir instrucciones de impresión en lugares pertinentes, reconstruir el programa, reproducir el problema con el nuevo programa, y usar las instrucciones de impresión para perfeccionar con exactitud qué código

se está ejecutando cuando se produjo el problema. Lo ideal sería que el programa ya cuente con algún tipo de implementación que permita ser reutilizada. Incluso si no es así, el programador debería considerar la posibilidad de un enfoque sistemático para añadir instrucciones de impresión, de modo que este pueda utilizar posteriormente estos resultados al momento de realizar una nueva depuración al mismo programa. En particular, cada instrucción de impresión debe indicar claramente dónde está localizado en el programa, a fin de que pueda ser rápidamente encontrada más tarde.

Finalmente un método alternativo para localizar la fuente del comportamiento erróneo es la simple inspección del código fuente. Esta es la única opción si el programador no puede reproducir el problema. Una clara comprensión de todo el código fuente del programa es un requisito fundamental para que esta alternativa sea una buena opción. Desafortunadamente, un problema complejo es casi imposible aislar mediante la simple lectura del código fuente. El programador tendrá que seleccionar entre ciertas posibilidades, y tratar de localizar a través del código fuente de forma cuidadosa si está en frente realmente de algún problema.

4.1.3.3. Identificar el error

Ahora que el programador ha encontrado el código que causa el comportamiento erróneo, es necesario identificar el error en la codificación real. A menudo son el mismo código, es decir, la codificación errónea directamente causa el mal comportamiento. Sin embargo, el programador siempre debe considerar la posibilidad de que el error está en otros lugares.

Por ejemplo, la rutina que causa el comportamiento erróneo puede comportarse correctamente, pero puede ser llamada con datos de entrada erróneos, o en el momento equivocado. Un error en la codificación de otros lugares puede causar una estructura de datos que espera valores erróneos. Otra posibilidad es que el usuario ingrese datos incorrectos.

La solución en estos casos pueden ser dos, el programador debe, por supuesto, arreglar el código que llama la rutina de forma incorrecta o de lo contrario restringir los datos de entrada para que no sean incorrectos. En el caso de una mala entrada del usuario, deberá validar la entrada. Además, puede que desee agregar controles al código que utiliza los valores. Se deben comprobar valores que no sean correctos para la entrada, y generar un informe de error u otro manejador con el objetivo de que no se produzca un comportamiento erróneo del programa.

4.1.4. Corregir el bug

El último paso en el proceso de depuración es, por supuesto, eliminar el bug. Este punto es importante detallarlo pero sólo se mencionarán un par de puntos:

- Si el programador desea un programa que pueda mantenerse en el futuro, entonces deberá asegurarse de arreglar el error de forma correcta. Esto significa que la solución encaja con el resto del programa, y que fija todos los aspectos del problema, sin introducir nuevos problemas.
- El programador debe actualizar toda la documentación.
- En algunos casos puede que el programador necesite un parche rápido para solucionar un problema inmediato. No hay nada malo en ello, siempre y cuando el programador se dé el tiempo para después volver atrás y arreglar de forma definitiva el bug.
- Obviamente, siempre fijar cualquier prueba que se haga velando por que ya no se vuelva a repetir el comportamiento erróneo. No debe olvidar asegurarse de que el programa sigue para pasar una serie de ensayos. El programador debe considerar la posibilidad de extender las series de ensayos para detectar el caso que se ha fijado anteriormente, para asegurarse de que éste no vuelva a manifestarse.

A partir de lo anterior, es interesante observar las técnicas que el mercado de software ha desarrollado para satisfacer las necesidades tanto de las empresas que construyen software como la de los desarrolladores independientes.

4.2. Técnicas de depuración

4.2.1. Depuración basada en mensajes

4.2.1.1. Motivación

Insertar instrucciones de mensajes en un código es una técnica que necesita muy poca tecnología para hacer depuración. Esta puede ser la única manera de hacer depuración debido a que los depuradores no siempre están disponibles o son aplicables. Normalmente este es el caso de aplicaciones distribuidas.

La experiencia indica que la acción de registrar determinados comportamientos es un proceso importante dentro del ciclo de desarrollo de software. Estos ofrecen varias ventajas:

- Proporcionan contextos precisos sobre la ejecución del programa. Una vez insertado en el código, la generación del registro de salida no requiere la intervención humana.
- Por otra parte, el registro de salida se debe guardar en un medio persistente, para ser estudiado en otro momento.
- Además de su uso en el ciclo de desarrollo de software, un buen manejo de las instrucciones de mensajes de comportamientos pueden ser vistos como una buena herramienta de auditoría.

Como Brian W. Kernighan y Rob Pike exponen en [4]:

"Como opción personal, tendemos a no utilizar depuradores más allá de obtener un pila de traza o el valor de una o dos variables. Una de las razones es, que es fácil perderse en detalles complicados de estructuras de datos y control de flujo; nos encontramos que ir a través del programa es menos productivo que pensar intensamente y añadir instrucciones de salida que verifiquen por si mismos puntos críticos de este. Al hacer click sobre las declaraciones, llevará más tiempo que el rastreo de la salida que juiciosamente fueron colocadas. Se tarda menos tiempo para decidir dónde colocar las declaraciones de impresión que dar pasos para lograr seccionar la región crítica del código, incluso asumiendo que nosotros sabemos donde está. Más importante aún, las instrucciones de depuración permanecen en el programa, las sesiones de depuración son transitorias".

4.2.1.2. Proceso de depuración

Este proceso consiste en que el programador manualmente vaya insertando instrucciones que impriman datos de interés (ubicación, ambiente, etc.), y estos se guarden en un archivo o se impriman directamente en una consola o terminal. En general esto se hace a nivel de código fuente y lo que se imprime siempre está dirigido para que un humano lo comprenda.

4.2.1.3. Ventajas/Desventajas

Una de las principales ventajas de esta técnica es que se puede aplicar sin la necesidad de contar con una herramienta dedicada para tal labor.

La técnica de mensajes también tiene sus inconvenientes:

- Puede retardar una solicitud que se le realiza al programa.
- Si los mensajes son demasiados extensos pueden causar ceguera de desplazamiento sobre el código fuente original.

Es importante señalar que esta técnica no es escalable.

4.2.1.4. Herramientas

4.2.1.4.1. Log4j Esta herramienta [5] es una biblioteca de código de abierto desarrollada en el lenguaje de programación *Java* por *Apache Software Foundation* que permite a los desarrolladores de software realizar depuración basada en mensajes con distintos niveles de granularidad (nivel de detalle) en tiempo de ejecución. La configuración de salida y granularidad de los mensajes es realizada en tiempo de ejecución mediante el uso de archivos de configuración externos.

Un ejemplo simple de uso es el siguiente:

```
1 import com.foo.Bar;
2 //Importa las clases de log4j.
3 import org.apache.log4j.Logger;
4 import org.apache.log4j.BasicConfigurator;
5 public class MyApp{
6     //Define una variable estatica que tiene una referencia
7     //a una instancia de Logger llamada "MyApp".
8     static Logger logger = Logger.getLogger(MyApp.class);
9
10    public static void main(String[] args){
11        //Crea una simple configuracion a la consola
12        BasicConfigurator.configure();
13        logger.info("Entrando en la aplicacion.");
14        Bar bar = new Bar();
15        bar.doIt();
16        logger.info("Saliendo de la aplicacion.");
17    }
18 }
```

MyApp utiliza la clase *Bar* del paquete *com.foo*, la cual ya no tiene que configurar el log sino únicamente utilizarlo.

```
1 package com.foo;
2 import org.apache.log4j.Logger;
```

```
3
4 public class Bar{
5     static Logger logger = Logger.getLogger(Bar.class);
6     public void doIt(){
7         logger.debug("Dentro de Bar!");
8     }
9 }
```

La salida en el log es:

```
0 [main] INFO MyApp - Entrando en la aplicacion.
36[main]DEBUG com.foo.Bar - Dentro de Bar!
51[main]INFO MyApp - Saliendo de la aplicacion.
```

4.2.1.4.2. django-logging Esta herramienta [6] es un módulo que permite realizar depuración basada en mensajes para aplicaciones web desarrolladas en el framework *Django*. Fue desarrollada para funcionar con *Django* cuando está siendo ejecutado vía *runserver*. Django-logging evita que el programador tenga que imprimir mensajes de depuración vía consola, permitiéndole insertar determinadas instrucciones que serán mostradas en una cómoda vista dentro de la pagina web en la cual han sido insertadas. Un ejemplo sencillo de uso es el siguiente:

```
1 def monthlyReport(aRequest):
2     logging.debug('This is a sample debug message')
3     logging.info('This is a sample informational message')
4     logging.warn('This is a sample warning message')
5     logging.error('This is a sample error message')
6     logging.critical('This is a sample critical message')
7     if aRequest.method == "POST":
8         ....
9     ....
```

El resultado de la introducción de las instrucciones de mensajes son observados a través del navegador web, en el formulario en el cual fueron introducidas:



Figura 4.1: Resultado de depuración con django-logging

4.2.2. Depuración basada en puntos de quiebre

4.2.2.1. Motivación

Un punto de quiebre es una detención intencional o pausa en un lugar del programa o script, puesto en ese lugar con propósitos de depuración. En general, un punto de quiebre es el significado de adquirir conocimiento acerca de un programa durante su ejecución. Durante la interrupción, el programador inspecciona el ambiente de prueba (memoria, archivos, reportes, etc) para averiguar si el programa funciona correctamente.

4.2.2.2. Proceso de depuración

En la práctica, un punto de quiebre consiste en una o más condiciones que determinan cuando la ejecución de un programa debe ser interrumpida. La forma más común de un punto de quiebre es uno, donde la ejecución del programa es interrumpido antes de ejecutar la instrucción especificada por el programador, esto es comúnmente llamado instrucción de punto de quiebre.

Otro tipo de condiciones pueden ser utilizadas, tales como lectura, escritura o modificación de un ubicación específica en el área de memoria. Esto es comúnmente llamado información de punto de quiebre o un punto de observación.

Los puntos de quiebres además pueden ser utilizados para interrumpir la ejecución en un momento determinado, o cuando se presiona determinada tecla.

Muchos procesadores incluyen soporte de hardware para los puntos de quiebre (comúnmente para los puntos de quiebre de instrucción e información). Dicho hardware puede incluir limitantes, por ejemplo no permitir puntos de quiebres o instrucciones ubicadas en sectores reservados por el mismo. Este tipo de restricciones es impuesta por la micro arquitectura del procesador, variando de procesador en procesador.

Sin soporte de hardware, los depuradores tienen que implementar los puntos de quiebre mediante software, lo cual, particularmente para los puntos de quiebre de información, pueden tener un impacto enorme en el rendimiento de la aplicación que está siendo depurada.

4.2.2.3. Ventajas/Desventajas

Los depuradores de punto de quiebre tienen las siguientes ventajas sobre los depuradores basados en mensajes:

- Examinar y modificar la memoria y las variables del programa.
- Examinar el contenido de los registros del procesador.
- Examinar la pila de llamadas que ha desembocado la situación actual.
- Cambiar el punto de ejecución, de manera que el programa continúe su ejecución en un punto diferente al punto en el que fue detenido.
- Ejecutar instrucción a instrucción.
- Ejecutar partes determinadas del código, como el interior de una función, o el resto de código antes de salir de una función.

La principal desventaja de los depuradores de punto de quiebre es que al encontrar el punto de quiebre, no tiene en sus registros la pila de llamadas que se produjeron, perdiendo toda la información anterior al punto de quiebre.

4.2.2.4. Herramientas

4.2.2.4.1. Depurador de Eclipse Eclipse es una plataforma de software compuesta por un framework de aplicaciones extensible, herramientas y una librería de tiempo de ejecución para software. Este está escrito fundamentalmente en Java para entregar a los desarrolladores de software y administradores un ambiente de desarrollo integrado (IDE).

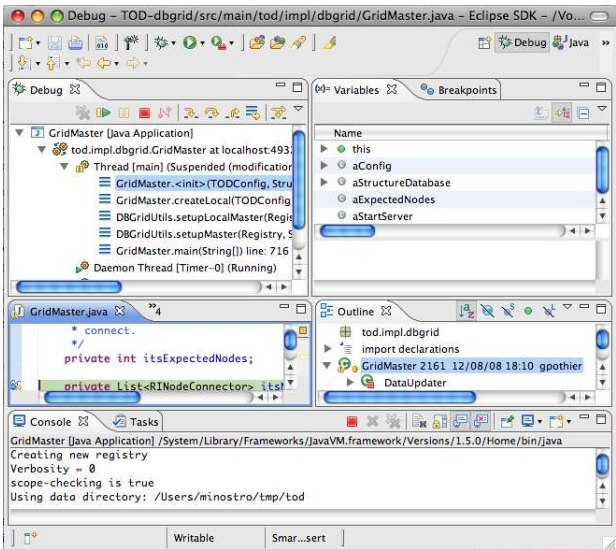


Figura 4.2: Resultado de depuración con Eclipse

Los usuarios de Eclipse pueden extender sus capacidades mediante la instalación de plugins escritos para el framework Eclipse, al igual que herramientas de desarrollo

para otros lenguajes de programación, y pueden escribir y contribuir con sus propios plugins.

Todas las liberaciones de Eclipse están bajo los términos de la licencia pública de Eclipse, Eclipse es una herramienta de software libre y abierta.

4.2.2.4.2. Depurador Winpdb Winpdb es un depurador de punto de quiebre para Python independiente de la plataforma, la cual soporta múltiple hilos de ejecución, modificación del espacio de nombres, depuración embebida, comunicación encriptada y es 20 veces más rápido que pdb (Módulo de Python para realizar depuración).

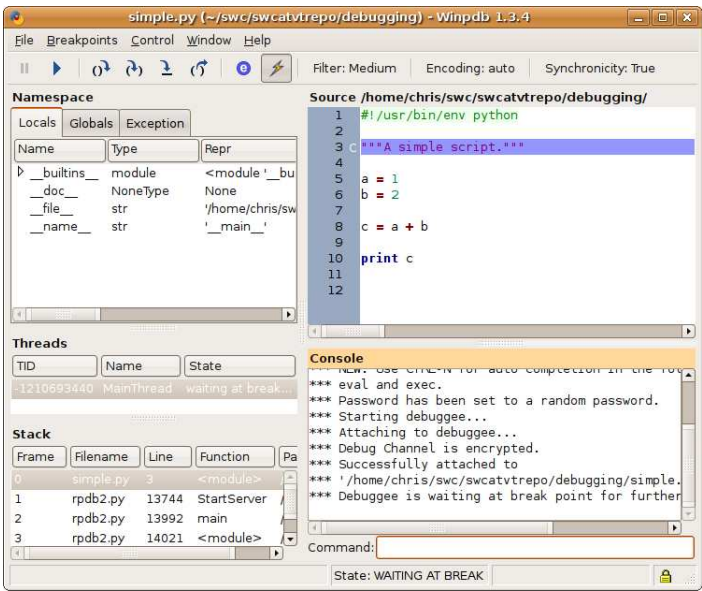


Figura 4.3: Resultado de depuración con Winpdb

4.2.3. Depuración Omnisciente

4.2.3.1. Motivación

Mediante el registro de cada cambio de un programa en ejecución, es posible presentar al programador cualquier información que desee. Esencialmente, esto hace posible depurar el programa yendo *atrás en el tiempo*, simplificando bastante el proceso de depuración.

Los desarrolladores de los depuradores se han concentrado completamente en responder la siguiente pregunta: *¿Qué información podemos entregar a los programadores mientras el programa se está ejecutando?*. Aunque esto no es del todo errado, no es la pregunta correcta. La pregunta que debieran tener que responder es: *¿Qué información ayudaría más al programador?*.

De estudios informales [7] se ha revelado que cientos de programadores, aproxima-

damente el 90 % de ellos, depuran todos sus programas utilizando sólo instrucciones de impresión.

4.2.3.1.1. Depuración omnisciente Un depurador omnisciente trabaja mediante la recolección de eventos generados por cada cambio de estado (cada asignación de variable de cualquier tipo) y cada llamada a un método dentro del programa depurado [2, 7]. Después de terminado el programa es mostrado el depurador (interfaz gráfica) permitiendo al programador ver el estado del programa en el tiempo de ejecución que desee.

El programador puede seleccionar cualquier variable e ir *atrás en el tiempo* y mirar donde ésta fue definida o cual fue su valor.

4.2.3.2. Proceso de depuración

Aunque no existe una definición formal de como llevar a cabo el proceso de depuración en esta técnica, se utiliza la definición de Bil Lewis [7] para explicar el proceso y componentes esenciales que debe tener un depurador omnisciente.

4.2.3.2.1. Mantener el estado Se quiere ser capaz de revertir el programa hacia cualquier estado anterior de ejecución. Esto implica que se debe grabar cada cambio de cada objeto o variable. Se necesita grabar cada asignación en cada hilo de ejecución y definir un orden sobre ellos. La marca de tiempo debe ser el único referente del estado del programa.

4.2.3.2.2. Presentación Cada objeto, variable, instrucción de E/S tendrá un valor conocido para cada marca de tiempo. Cuando *revertimos el depurador* a un tiempo dado, toda la información es actualizada para reflejar los valores para esa marca de tiempo.

Se muestra toda la información de forma adecuada y el programador no debiera nunca preguntarse *¿Esto cambió?* o *¿dónde estoy?*.

4.2.3.2.3. Identificar el estado El programador debe ser capaz de mirar un objeto y saber que objeto es y que valores tienen las variables de instancias. Los objetos de clases son mostrados justo con el nombre de la clase (ej.: Persona), strings y otras primitivas como se esperan (1, True, "soy una cadena").

4.2.3.2.4. Navegación La simple navegación a través de la historia del programa es dotado a través de la selección de línea en un panel o presionando algunos botones. Los botones deben trabajar de una forma estándar para los diferentes paneles que existan.

4.2.3.3. Ventajas/Desventajas

Existen varias razones para hacer un esfuerzo y estudiar la depuración omnisciente:

- La primera y más famosa, es fácil depurar cuando se puede ir hacia atrás. La pregunta más común que los programadores se hacen es *¿Quién definió esta variable?*.
- Elimina los terribles problemas con depuradores de punto de quiebre, con los cuales el programador debe *adivinar* en donde poner el punto de quiebre. No existen pasos extras para depurar, no sucede la situación común de *fui demasiado lejos*. Se eliminan los problemas no determinísticos.
- Entrega al programador una lista única de eventos del programa siendo capaz de ver las huellas de llamadas a los métodos.
- Toda la información es serializada y puede ser analizada remotamente.

La gran desventaja de esta técnica se relaciona con la enorme capacidad de almacenamiento que se necesita para almacenar las huellas de ejecución. En el mejor de los casos es adecuado utilizar un cluster.

4.2.3.4. Herramientas

Algunas de las herramientas que existen son:

- **EXDAMS**, un depurador extensible y sistema de monitoreo
- **ODB**, un depurador omnisciente no escalable para el lenguaje de programación Java
- **ZSTEP**, un depurador omnisciente para el lenguaje de programación Lisp
- **TOD**, un depurador omnisciente escalable para el lenguaje de programación Java

En el capítulo número 5 se individualizarán algunas de estas herramientas.

4.3. Lenguajes de Programación

Un lenguaje de programación es un lenguaje que puede ser utilizado para hacer funcionar o controlar el comportamiento de una máquina, particularmente una computadora. Consiste en un conjunto de símbolos y reglas (sintácticas y semánticas), que definen su estructura y el significado de sus elementos y expresiones.

Aunque muchas veces se usa lenguaje de programación y lenguaje informático como si fuesen sinónimos, no tiene por qué ser así, ya que los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como, por ejemplo, el HTML (lenguaje para el marcado de páginas Web).

Un lenguaje de programación permite a uno o más programadores especificar de manera precisa: sobre qué datos una computadora debe operar, cómo deben ser estos almacenados, transmitidos y qué acciones deben tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural, tal como sucede con el lenguaje Léxico. Una característica relevante de los lenguajes de programación, es precisamente que más de un programador pueda tener un conjunto común de instrucciones que puedan ser comprendidas entre ellos para realizar la construcción del programa de forma colaborativa.

Los procesadores utilizados en las computadoras son capaces de entender y actuar según lo indican programas escritos en un lenguaje fijo llamado lenguaje de máquina. Todo programa escrito en otro lenguaje puede ser ejecutado de dos maneras:

- Mediante un programa que va adaptando las instrucciones conforme son encontradas. A este proceso se lo llama interpretar y a los programas que lo hacen se los conoce como intérpretes.
- Traduciendo este programa al programa equivalente escrito en lenguaje de máquina. A ese proceso se lo llama compilar.

4.3.1. Clasificación de los lenguajes de programación

Los lenguajes de programación se determinan según el nivel de abstracción, según la forma de ejecución y según el paradigma de programación que poseen cada uno de ellos, estos pueden ser:

4.3.1.1. Según su nivel de abstracción

4.3.1.1.1. Lenguajes de bajo nivel Los lenguajes de bajo nivel son lenguajes de programación que se acercan al funcionamiento de una computadora. El lenguaje de más bajo nivel es, por excelencia, el código máquina. A éste le sigue el lenguaje ensamblador, ya que al programar en ensamblador se trabajan con los registros de memoria de la computadora de forma directa.

4.3.1.1.2. Lenguajes de nivel medio Existen lenguajes de programación que son considerados por algunos expertos como lenguajes de medio nivel (como es el caso del lenguaje C), al tener ciertas características que los acercan a los lenguajes de bajo nivel pero teniendo, al mismo tiempo, ciertas cualidades que lo hacen un lenguaje más cercano al humano y, por tanto, de alto nivel.

4.3.1.1.3. Lenguajes de alto nivel Los lenguajes de alto nivel son normalmente fáciles de aprender porque están formados por elementos de lenguajes naturales, como el inglés. En *BASIC*, el lenguaje de alto nivel más conocido, los comandos como *IF contador = 10 THEN STOP* pueden utilizarse para pedir a la computadora que pare si *contador* es igual a 10. Por desgracia para muchas personas esta forma de trabajar es un poco frustrante, dado que a pesar de que las computadoras parecen comprender un lenguaje natural, lo hacen en realidad de una forma rígida y sistemática.

4.3.1.2. Según la forma de ejecución

4.3.1.2.1. Lenguajes compilados Naturalmente, un programa que se escribe en un lenguaje de alto nivel también tiene que traducirse a un código que pueda utilizar la máquina. Los programas traductores que pueden realizar esta operación se llaman compiladores. Éstos, como los programas ensambladores avanzados, pueden generar muchas líneas de código de máquina por cada proposición o instrucción del programa fuente. Se requiere un proceso de compilación antes de procesar los datos de un problema.

Los compiladores son aquellos programas computacionales, cuya función es traducir un programa escrito en un determinado lenguaje a un idioma que la computadora entienda (lenguaje máquina con código binario).

Al usar un lenguaje compilado, el programa desarrollado no se ejecuta mientras existan errores sintácticos, sino hasta que luego de haber compilado el programa, ya no aparecen estos errores en el código.

4.3.1.2.2. Lenguajes interpretados Se puede también utilizar una alternativa diferente de los compiladores para traducir lenguajes de alto nivel. En vez de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante el proceso de compilación para utilizarlo en un proceso de producción futura, el programador sólo carga el programa fuente en la computadora junto con los datos que se van a procesar. Un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición o instrucción del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el proceso de los datos. No se graba el código objeto para utilizarlo posteriormente.

La siguiente vez que se utilice una instrucción, se debe interpretar otra vez y traducir a lenguaje máquina. Por ejemplo, durante el procesamiento repetitivo de los pasos de un ciclo, cada instrucción del ciclo tendrá que volver a ser interpretado cada vez que se ejecute el ciclo, lo cual hace que el programa sea más lento en tiempo de ejecución (porque se va revisando el código en tiempo de ejecución), pero más rápido en tiempo de diseño (porque no se tiene que estar compilando a cada momento el código completo).

El intérprete elimina la necesidad de realizar un proceso de compilación después de cada modificación del programa cuando se quiere agregar funciones o corregir errores; pero es obvio que un programa objeto compilado con antelación deberá ejecutarse con mucha mayor rapidez que uno que se debe interpretar a cada paso durante un proceso de producción.

4.3.1.3. Según el paradigma de programación

Un paradigma de programación representa un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro, sino que cada uno tiene ventajas y desventajas. Dependiendo de la situación un paradigma resulta más apropiado que otro. Atendiendo al paradigma de programación, se pueden clasificar los lenguajes en:

4.3.1.3.1. Paradigma Imperativo Describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa. El código máquina en general está basado en el paradigma imperativo. Su contrario es el paradigma declarativo. En este paradigma se incluye en el paradigma procedimental (procedural) entre otros.

4.3.1.3.2. Paradigma Declarativo No se basa en el cómo se hace algo (cómo se logra un objetivo paso a paso), sino que describe (declara) cómo es algo. En otras palabras, se enfoca en describir las propiedades de la solución buscada, dejando indeterminado el algoritmo (conjunto de instrucciones) usado para encontrar esa solución. Es más complicado de implementar que el paradigma imperativo, tiene desventajas en la eficiencia, pero ventajas en la solución de determinados problemas.

4.3.1.3.3. Paradigma Estructurado La programación se divide en bloques (procedimientos y funciones) que pueden o no comunicarse entre sí. Además la programación se controla con secuencia, selección e iteración. Permite reutilizar código programado y otorga una mejor comprensión de la programación. Es contrario al paradigma inestructurado, de poco uso, que no tiene ninguna estructura, es simplemente un *bloque*, como por ejemplo, los archivos batch o lotes.

4.3.1.3.4. Paradigma Funcional Este paradigma concibe a la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos. Permite resolver ciertos problemas de forma elegante y los lenguajes puramente funcionales evitan los efectos secundarios comunes en otro tipo de programaciones.

4.3.1.3.5. Paradigma Lógico Se basa en la definición de reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas.

4.3.1.3.6. Paradigma Orientado a Objetos La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

Si bien puede seleccionarse la forma pura de estos paradigmas al momento de programar, en la práctica es habitual que se mezclen, dando lugar a la programación multiparadigma.

Actualmente el paradigma de programación más utilizado debido a múltiples ventajas respecto a sus anteriores, es la programación orientada a objetos.

A continuación se detallan algunas características del lenguaje de programación utilizado en el presente trabajo de memoria para realizar la etapa de implementación.

4.3.2. Python

4.3.2.1. ¿Qué es Python?

Python es un lenguaje de programación de alto nivel creado por Guido van Rossum a principios de los años 1990. Es un lenguaje similar a Perl, pero con una sintaxis muy limpia y que favorece un código legible.

Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y multiparadigma (orientación a objetos, estructurada y funcional).

4.3.2.2. Lenguaje interpretado

Es un lenguaje de programación interpretado o de script, esto quiere decir que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora.

La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es semi interpretado. En Python, como en Java y muchos otros lenguajes, el código fuente se traduce, la primera vez que se ejecuta a un pseudocódigo máquina intermedio llamado bytecode, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

4.3.2.3. Tipado dinámico

La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

4.3.2.4. Fuertemente tipado

No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si se tiene una variable que contiene un texto (variable de tipo cadena o string) no se podrá tratar como un número (sumar la cadena "9" y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

4.3.3. Comparativa entre Python y Java

Esta sección muestra similitudes y diferencias entre estos dos lenguajes de programación, evitando principalmente mostrar que uno es mejor que otro, debido a que los lenguajes de programación son herramientas, y diferentes herramientas son apropiadas para distintos trabajos. Como ejemplo podemos ver a un plomero quien sólo tiene en su caja de herramientas un martillo (sin importar el tamaño del martillo), *¿Este será capaz de solucionar todos los problemas de las cañerías y alcantarillados?*. En analogía a este ejemplo podemos ver a una organización de desarrollo de software, que plantea desarrollar todas las soluciones de software con el lenguaje de programación Java, *¿Esta organización será capaz de dar solución eficiente a todos los problemas que se le plantean?*.

Java es un lenguaje de tipado estático, cuyos tipos se fijan en el momento de compilar. La mayoría de los lenguajes de tipado estático fuerzan esto exigiéndole al programador que declare todas las variables con sus tipos antes de usarlas.

Existen tres principales características que los programadores deben tomar en cuenta:

Java	Python
<p><u>Tipado estático</u>: En java, todos los nombres de variables (junto con sus tipos), deben ser explícitamente declarados. El intento de asignar un objeto con un tipo de datos erróneo a un nombre de variable levantará inmediatamente una excepción de tipo.</p> <p>Los objetos contenedores Java (Ejemplo: <i>Vector</i> y <i>ArrayList</i>) esperan objetos del tipo genérico <i>Object</i>, pero no pueden esperar primitivas como un <i>int</i>. Para almacenar un <i>int</i> en un <i>Vector</i>, se debe primero convertir el <i>int</i> a un <i>Integer</i>. Cuando se rescata un objeto desde un contenedor, este no recuerda su tipo, y debe ser explícitamente convertido al tipo deseado.</p>	<p><u>Tipado dinámico</u>: En Python, nunca se declara nada. Una instrucción de asignación vincula un nombre con un objeto, y el objeto puede ser de cualquier tipo. Si un nombre es asignado a otro objeto de un tipo, este puede más tarde ser asignado con un objeto de diferente tipo. Esto quiere decir que Python es un lenguaje de programación de tipado dinámico.</p> <p>Los objetos contenedores Python (Ejemplo: <i>listas</i> y <i>diccionarios</i>) pueden esperar objetos de cualquier tipo, incluyendo números y listas. Cuando se recupera un objeto desde un contenedor, este recuerda su tipo, entonces la conversión no es requerida.</p>
<p><u>Prolijo</u>: Abundante en palabras; utilizando o conteniendo muchas palabras que son necesarias.</p>	<p><u>Conciso</u>: Expresando mucho en pocas palabras. Esto implica brevedad, consiguiendo eliminar cosas superfluas.</p>
<p><u>No es compacto</u></p>	<p><u>Compacto</u>: En The new hacker’s Dictionary [8] entrega la siguiente información de compacto: Compacto adj. de diseño, describe la propiedad valuable que este puede ser retenido en la mente. Esto significa que el objeto fue creado desde el diseño para ser utilizado con mucha facilidad y pocos errores.</p>

A continuación se muestran ejemplos que comparan el uso entre Java y Python:

El clásico programa que ilustra "Hola, mundo!"

■ Java

```
1 public class HelloWorld
2 {
3     public static void main (String[] args)
4     {
5         System.out.println("Hola Mundo!");
6     }
7 }
```

■ Python

```
1 print "Hola Mundo!"
```

En el siguiente ejemplo, se inicializa un entero con cero, luego este se convierte a string, luego se verifica si es que este está vacío. Es importante notar que la declaración es necesaria en Java pero no en Python.

■ Java

```
1  int myCounter = 0;
2  String myString = String.valueOf(myCounter);
3  if (MyString.equals("0"))
4  ...
5  //imprimir los enteros del 1 al 9
6  for(int i = 1; i < 10; i++){
7      System.out.println(i);
8  }
```

■ Python

```
1  myCounter = 0
2  myString = str(myCounter)
3  if myString == 0
4  ...
5  for i in range(1,10):
6      print i
```

Imagine que su aplicación tiene 15 clases. (Más precisamente, esta tiene 15 clases públicas de nivel superior).

Java	Python
Cada nivel superior de clases públicas debe ser definida en su propio archivo. Si la aplicación tiene 15 clases, esta tiene 15 archivos.	Múltiples clases pueden ser definidas en un archivo. Si la aplicación tiene 15 clases, la aplicación completa puede ser almacenada en un solo archivo, aunque quizás se desee dividir de forma sensata las clases entre 4, 5, o 6 archivos.

En una aplicación, el método A llama al método B, el método B llama al método C, el método C llama al método D, el método D llama al método E, el método E llama al método F. Se descubre que el método F debe lanzar la excepción `SpecialException`, y esta debe ser capturada por el método A.

Java	Python
Se debe lanzar <code>SpecialException</code> en el método F, y capturarla en A, y se debe añadir <i>throws SpecialException</i> en las firmas de los métodos B, C, D, E y F.	Se debe levantar <code>SpecialException</code> en el método F, y capturarla en A. Las excepciones serán propagadas hacia los niveles superiores automáticamente.

En una aplicación se tiene una clase *Empleado*. Cuando una instancia de *Empleado* es creada, al constructor debe ser pasado uno, dos o tres argumentos.

Si se está programando en Java, esto significa que se tiene que escribir tres constructores, con tres diferentes firmas. Si se está programando en Python, sólo se escribe un constructor, con valores por defecto para los argumentos opcionales.

■ Java

```
1 public class Empleado
2 {
3     private String Nombre;
4     private int Impuesto;
5     private String estadoCivil;
6     //-----constructor #1-----
7     public Empleado(String aEmpleado)
8     {
9         this(aEmpleado,1);
10    }
11    //-----constructor #2-----
12    public Empleado(String aEmpleado, int aImpuesto)
13    {
14        this(aEmpleado, aImpuesto, "soltero");
15    }
16    //-----constructor #3-----
17    public Empleado(String aEmpleado, int aImpuesto,
18                    String aEstadoCivil)
19    {
20        this.Nombre = aEmpleado;
21        this.Impuesto = aImpuesto;
22        this.estadoCivil = aEstadoCivil;
23    }
24 }
```

■ Python

```
1 class Empleado():
2
3     def __init__(self, aNombre, aImpuesto=2,
4                 aEstadoCivil="soltero"):
5         self.nombre = aNombre
6         self.impuesto = aImpuesto
7         self.estadoCivil = aEstadoCivil
```

Para graficar de mejor manera el tipado estático versus el tipado dinámico se muestran los siguientes gráficos:

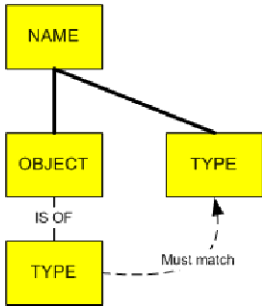


Figura 4.4: Tipado estático

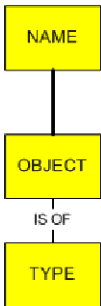


Figura 4.5: Tipado dinámico

Capítulo 5

Implementación de depuradores omniscientes

A lo largo de la historia de la Ciencia de la Computación han existido muchos esfuerzos por implementar depuradores omniscientes, es por esto que se encuentra importante conocer las distintas perspectivas de diferentes equipos de implementación. A continuación se muestra, dividida por etapas, las distintas implementaciones existentes al día de hoy.

5.1. Implementaciones históricas

En esta sección se considera EXDAMS - Extendable Debugging And Monitoring System, como la implementación histórica más importante.

5.1.1. EXDAMS

5.1.1.1. Historia

Con la llegada de los lenguajes algebraicos de alto nivel, la industria de la computación esperaba ser aliviada de los detalles de programación requeridos en el nivel de los lenguajes ensamblador. Esta expectativa ha sido en gran parte cumplida y muchos sistemas son ahora construídos en lenguajes de alto nivel.

Sin embargo, la habilidad de depurar programas tiene avances que son pequeños en relación al incremento en el uso de estos lenguajes de alto nivel. Como Evans and Darlay [9] indican: *"Nosotros encontramos que, hablando en términos generales, un análisis cercano de casi todas las principales técnicas de depuración de los lenguajes en ensamblador, existe al menos un sistema de depuración perteneciente a algún*

lenguaje de alto nivel. Sin embargo, las facilidades de la depuración en línea para los lenguajes de alto nivel son en general menos desarrolladas y menos utilizadas (relativo al uso de estos lenguajes) que sus contrapartes para lenguajes en ensamblador".

En general, los sistemas construidos son meramente copias de los depuradores en línea de los lenguajes en ensamblador, más que diseños de facilidades totalmente nuevas para los lenguajes de alto nivel. Ellos no han creado ni formatos gráficos en los cuales se presente información acerca de la depuración, no han entregado una manera razonable con la cual los usuarios puedan especificar el proceso requerido en cualquier depurador de información disponible.

EXDAMS, es un intento por quebrar este impasse entregando un ambiente simple en el cual los usuarios pueden facilmente añadir nuevas funcionalidades a un depurador en línea sin tener que modificar mayormente el compilador a nivel de código fuente, ni EXDAMS, o sus programas para que sean depurados.

5.1.1.2. Motivación

EXDAMS, es un sistema depurador y monitor en línea diseñado para facilitar la experimentación con nuevas herramientas de ayuda que cumplan estas mismas características, y para proveer alternancia de forma flexible entre estas ayudas. Es importante señalar que en esta herramienta el tiempo de ejecución es controlado en cualquiera de los sentidos adelante o atrás.

5.1.1.3. Características

Es un poderoso depurador y monitor a nivel de código fuente para lenguajes de programación. Sus facilidades son de dos tipos: estático, el cual hace referencia a un punto específico en tiempo de ejecución; y movimiento, el cual cambia en tiempo de ejecución y puede ser visto, en tiempo de ejecución, sin importar la dirección atrás o adelante (ejemplo: Un usuario puede ver la ejecución de su programa, en reversa, yendo hacía atrás a algún estado anterior).

Adicionalmente, de estas facilidades, las características de ambiente de EXDAMS son:

1. Habilidad para alternar, en cualquier punto de ejecución, entre el espacio de los datos (*¿Qué está sucediendo?*) y el espacio de control (*¿Cómo sucedió esto?*), de esta forma asociando una acción del programa con la instrucción exacta que produce la acción.

2. Fácil extensión para nuevas herramientas de depuración y monitoreo definidas por el usuario.

5.1.1.4. Objetivos de diseño

EXDAMS, fue diseñado para satisfacer tres necesidades:

- Como un vehículo para probar algunas propuestas, pero sin aplicarlas, para facilidades de depuración y monitoreo en línea.
- Como facilidades extensibles las cuales los nuevos depuradores y monitores pueden añadir fácilmente y luego probarlas.
- Como un sistema que provee algunas medidas de independencia de no sólo una máquina en particular sobre la cual está siendo ejecutado este, y la particular implementación de un lenguaje que está siendo depurado y/o monitoreado, sino que además de muchos lenguajes fuentes en los cuales los programas de los usuarios pueden ser escritos y depurados, y/o monitoreados.

En una situación perfecta para hacer depuración, de acuerdo con la filosofía de EXDAMS, el usuario primero establece *que es lo que está sucediendo*, luego decide si ese comportamiento es correcto, y finalmente, si este no es correcto, determina *como* el programa efectuó esa operación, al mismo tiempo que busca el error en el programa o en la información. De esta manera, cualquier sistema de depuración y monitoreo exhaustivo debe incluir facilidades poderosas en los espacios de información y control, como también proveer una manera simple de alternancia entre los puntos correspondientes en cualquier espacio, como las necesidades del usuario o preferencias personales le indiquen.

5.1.1.5. Depuración y monitoreo dentro de EXDAMS

EXDAMS contiene dos tipos de asistencia para la depuración y monitoreo:

- **Asistencia estática :** Muestra información que es invariable en relación al tiempo de ejecución (una marca de tiempo es incrementada como cada instrucción del código es ejecutado y usado para referir un punto particular en la ejecución del programa), como son los valores de variables al momento de ocurrir un error, una lista de todos los valores de las variables que fueron dadas en tiempo de ejecución, o una muestra de una porción del código fuente.

- **Asistencia dinámica :** Esta asistencia es sensible al tiempo de ejecución, esto es, la información que ella muestra puede variar con el tiempo de ejecución. Esta asistencia dinámica incluye el último valor de n de un conjunto de variables, la instrucción actual y la subrutina, además del valor actual del conjunto de variables. El usuario puede ejecutar la asistencia dinámica en cualquiera de las direcciones, adelante y atrás, controlando el tiempo de ejecución.

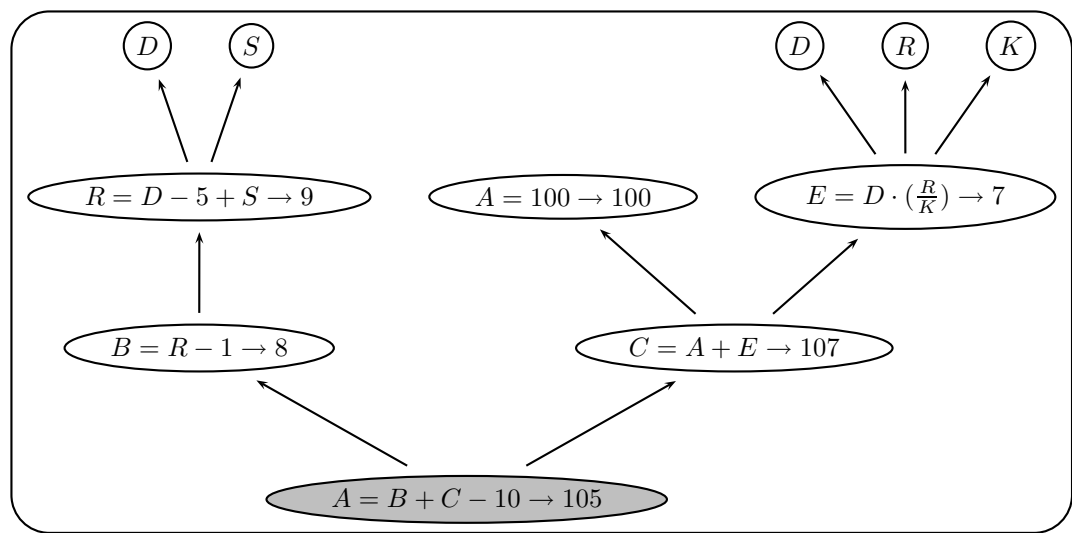
Las características más importantes, desde el punto de vista del usuario son:

- La habilidad para controlar el tiempo de ejecución de su programa, moviendo una variable rápidamente hacia atrás o adelante, mientras el depurador y/o monitor actualiza constantemente su despliegue de información.
- La habilidad de detenerse en cualquier punto en tiempo de ejecución del programa, cambiando al otro asistente, y examinando concienzudamente el comportamiento de su programa.

5.1.1.5.1. Análisis de flujo invertido Mediante la llamada de *FLOWBACK FOR* (instrucción reservada de EXDAMS) y especificando un valor en particular, el usuario solicitará a EXDAMS analizar *¿Cómo la información fluyó a través de su programa para producir el valor especificado?*. Este análisis es presentado en la forma de un árbol invertido, con el nodo de más abajo correspondiente al valor del cual el análisis inverso fue solicitado. Cada nodo consiste en una instrucción de asignación a nivel de código de lenguaje que produjo el valor, el valor mismo, y conectado con los nodos del siguiente nivel. Estos nodos corresponden a valores no constantes dentro de la instrucción de asignación mostrada que está conectada con esos nodos. Esos nodos tienen la misma forma como el original y están conectados para todos los valores no constantes utilizados en una instrucción de asignación en particular produciendo su valor. De esta manera, la siguiente figura muestra en análisis de flujo invertido para un valor particular de A .

Esta figura muestra que la instrucción de asignación $A = B + C - 10$; produjo un valor específico de A , y este valor fue 105. Los valores de B y C usados en la asignación de A fue 8 y 107, respectivamente, y fueron producidas mediante las instrucciones de asignación $B = R - 1$; y $C = A + E$; respectivamente. Cada uno de los otros nodos se explican de la misma manera.

A continuación se muestra un ejemplo de flujo invertido:



5.1.1.6. Arquitectura

EXDAMS es un sistema que cuenta de cuatro fases. Estas fases son: análisis del programa, compilación, recolección de historia y depuración en tiempo de ejecución, con navegación a través de la historia del programa.

5.1.1.6.1. Análisis del programa La primera fase analiza el programa fuente del usuario como efectúa las cuatro funciones, la más importante de las cuales es la creación de un modelo para este programa. Este modelo, el corazón de la fase depuración en tiempo de ejecución con navegación a través de la historia del programa, es la manera por la cual los valores recolectados sobre la cinta de la historia son interpretados y por la cual porciones del código fuente son recuperados, y es el repositorio de toda la información estructural conocida acerca del programa.

En general, la historia contiene toda la información dinámica necesaria para actualizar en tiempo de ejecución cuando se vaya en las direcciones hacia atrás o adelante, mientras el modelo contiene todo lo necesario sobre la información estática.

El modelo contiene la información estática de control y la información variable de alteración del programa del usuario. La información de control consiste en la estructura del programa acerca de *CALL*, *GOTO*, *IF-THEN-ELSE* y *DO-END*, mientras que la información variable de alteración consiste en los nombres de las variables sobre el lado izquierdo los cuales son afectados por instrucciones de asignación y las cuales a la vez pueden ser alteradas por instrucciones de entrada.

Las instrucciones de depuración añadidas al programa pasan la información relevante de tiempo de ejecución hacia la recolección de historia en tiempo de ejecución.

5.1.1.6.2. Compilación El procesador estándar del lenguaje fuente compila el programa fuente actualizado durante el análisis del programa.

5.1.1.6.3. Recolección de historia en tiempo de ejecución La versión compilada del programa actualizado es ejecutada con un conjunto de rutinas en tiempo de ejecución que este llama. Estas rutinas reúnen la información dinámica acerca del comportamiento del programa.

Esta información es completada en un buffer que es escrito cuando se llena. Esta es la cinta de historia del comportamiento del programa y, junto con la tabla simbólica y el modelo, es suficiente para recrear el comportamiento del programa en cualquiera de las direcciones (atrás o adelante) en tiempo de ejecución.

5.1.1.6.4. Depuración en tiempo de ejecución con navegación a través de la historia del programa. Esta fase contiene a los asistentes para depuración y monitoreo los cuales presentan la información histórica hacia el usuario de una manera usable en su pantalla. Esto además interpreta los comandos del usuario para mostrar alternativas y/o variaciones en tiempo de ejecución, y entrega la capacidad de editar para modificar bug descubiertos y para retornar el programa modificado a las cuatras fases para un nuevo proceso de depuración.

5.2. Implementaciones recientes

En esta sección consideraremos ODB - Omniscient Debugger, como la implementación reciente más importante.

5.2.1. ODB

ODB es una implementación Java que recolecta información mediante la instrumentación del bytecode del programa objetivo, al momento que éste es cargado. Utiliza un mecanismo simple de alto nivel para ordenar los eventos de diferentes hilos. ODB ha sido probado en MacOS, UNIX, y Windows sobre una gran variedad de programas.

Para un depurador gráfico como este, existen dos temas de vital importancia:

1. Presentación de la información; *¿Cómo el programador puede obtener desde el depurador, el estado del programa que le interese?*
2. Navegación; *¿Cómo el programador reconoce el estado?*

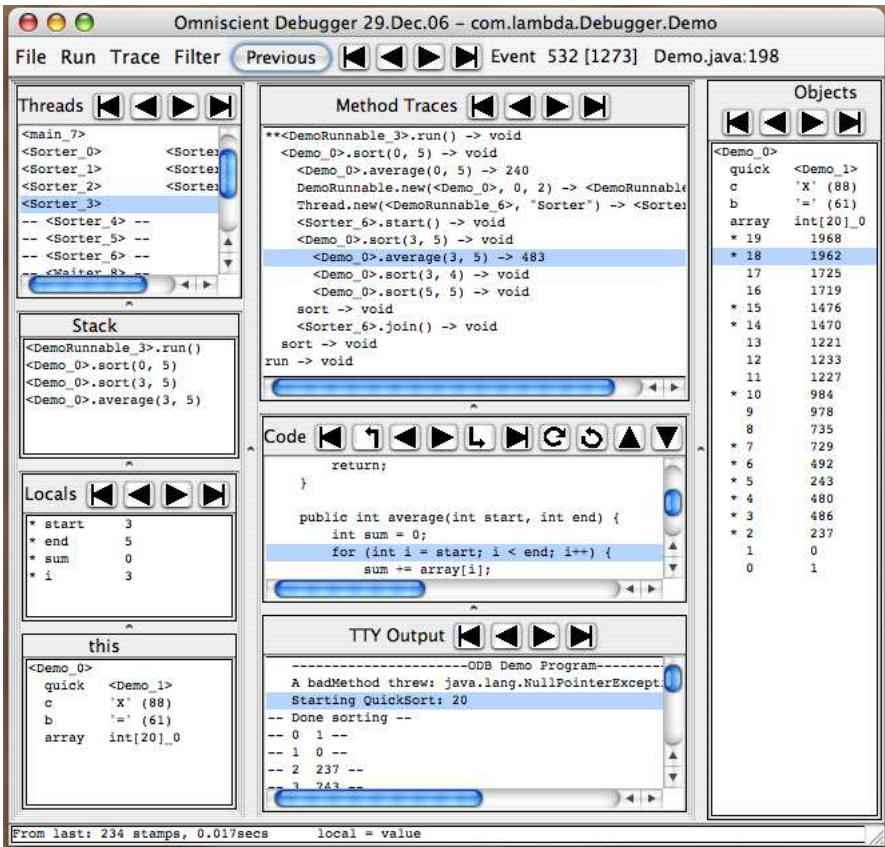


Figura 5.1: Ventana principal de depuración

5.2.1.1. La naturaleza de los bug bajo ODB

ODB divide a los bug en dos dimensiones: ¿Todos los eventos registrados para el bug caben en memoria? y ¿El bug produce una información errónea o este falla al producir una información correcta?.

5.2.1.1.1. La serpiente en el pasto Si el programa imprime la respuesta 41 en vez de 42, entonces se tiene un manejador sobre el bug. Se observa la serpiente en el pasto y se puede agarrar su cola. Ahora si el programa falla al producir la respuesta correcta, entonces no se tiene un manejador directo del bug, esta es la serpiente en el pasto que no podemos ver.

Para programas en los cuales todos sus eventos son registrados en memoria y en los cuales se pueda ver la serpiente en el pasto, entonces ODB es absolutamente efectivo. Es posible comenzar sobre la salida errónea e ir hacia atrás, encontrando el valor inapropiado en cada punto, y luego seguir este valor hasta su causa. (Si se tiene la cola de una serpiente y se tira lo suficientemente fuerte, se obtendrá su cabeza.). Es común iniciar el depurador, seleccionando la salida errónea, y navegar por todo el código fuente en 60 segundos, con absoluta confianza.

Los depuradores basados en punto de quiebre sufren del problema *La lagartija en el pasto*. Incluso cuando ellos pueden ver la lagartija y tirar de su cola, la lagartija

quebrará su cola y se irá. Luego ellos vuelven a otro problema *La lagartija perdida en el pasto*.

Con ODB, existe una versión de este problema. Cuando la producción incorrecta es causada por la falla de un determinado objeto cuando está siendo asignado, entonces se tiene que responder la pregunta *¿Quién pudo haber echo esto?*, la cual es una pregunta mucho más difícil que *¿Quién hizo esto?* (Esta es la lagartija sin pata en el problema del pasto). Se piensa que tenemos una serpiente, pero después de tirar su cola por un momento, esta la quiebra de cualquier forma.

Para problemas relativamente pequeños (digamos 10.000 eventos), esto es fácil y basta con sólo examinar por completo las huellas de asignación de los métodos. Para problemas más grandes, la *serpiente perdida en el pasto*, se tornan más problemáticos. Se tiene una gran cantidad de información acerca de la asignación y una idea no muy cierta de lo que estamos buscando.

Afortunadamente, el problema anterior es un caso profundamente analizado, ODB incorpora la función *get()* de Ducassé [10].

5.2.1.1.2. Tamaño Otra dimensión importante es el tamaño. Dentro de un espacio de direccionamiento de 31-bit, existe espacio para almacenar alrededor de 10 millones de eventos. Un gran porcentaje de los bug reales se encuentran dentro de este espacio, un buen porcentaje no. Existe un número de opciones para atacar este problema:

- Recolector de basura, tirar a la basura a los eventos antiguos
- Instrumentar pocos métodos
- Registrar por poco tiempo

5.2.1.1.2.1. Recolección de basura Esto es bueno porque no requiere esfuerzo y efectivamente mantiene una ventana de eventos circundante al bug. Por el contrario esto es malo porque no se elimina basura, si no que sólo eventos antiguos que pudieran ser importantes. Además es malo porque permite el programa correr lo bastante para que el rendimiento se convierta en un tema importante. Añade un 50 % extra sobre el mayor de los costos promedio para un evento.

Cuando la fuente del bug está dentro de los 10 millones de eventos en el momento cuando la grabación se apague, entonces se vuelve al bien conocido problema de *la serpiente en el pasto*. Cuando este va más lejos, nosotros perdemos la fuente del bug, y el recolector de basura pierde su valor.

5.2.1.1.2.2. Código seguro Es común que para un programa exista un gran porcentaje de código fuente seguro (Cualquier método recomputable, o métodos que no se quieren ver al interior de ellos). Mediante la solicitud que estos métodos no sean instrumentados, muchos eventos que no son interesantes son eliminados. ODB permite al programador seleccionar arbitrariamente un conjunto de métodos, clases, o paquetes, los cuales pueden ser instrumentados o no.

5.2.1.1.2.3. Comenzar/Detener registro Si el programador sospecha que determinado evento conocido siempre ocurre antes del bug (Ejemplo: Cuando presiona un botón y el programa se cae), entonces el registro puede ser habilitado en ese punto y apagado después del bug. ODB permite control manual (Existe un botón inicio/fin registro). Este además permite control automático de la manera que se irá examinando un gran conjunto de eventos por un patrón particular, el cual Comenzará/Detendrá el registro. Una vez más, esto es precisamente el problema de los analizadores de eventos para lo cual ODB utiliza la función *get()* de Ducassé.

5.2.1.1.3. Implementación ODB mantiene un array sencillo de marca de tiempos. Cada marca de tiempo es un entero de 32 bit, conteniendo un índice de hilo (8 bits), un índice de línea (20 bits) y un índice de tipo (4 bits). Un evento para cambiar una instancia de una variable necesita tres palabras: una marca de tiempo, la variable que ha cambiado, y el nuevo valor. Un evento para una llamada de método necesita: objeto, nombre del método, los argumentos y el valor de retorno (o excepción), junto con un pequeño conjunto de variables internas propias del depurador. Esto añade cerca de 20 palabras, la instrucción *return line* además generará 10 palabras más.

Cada variable tiene un historial asociado que es sólo una lista de pares marca de tiempo/valor. Cuando ODB quiere saber el valor de una variable en el tiempo 102, este justo toma el valor que está más cerca al valor previo. La lista de historia para variables locales y argumentos cuelgan sobre la línea de huella.

Siempre que ocurra un evento, este bloquea por completo el depurador, una nueva marca de tiempo es generada y registrada dentro de la lista, y estructuras asociadas son construidas. Los valores de retorno y excepciones son agregadas dentro de la apropiada línea de huella como estas fueran generadas.

La inserción en el código es muy simple, el bytecode fuente es examinado, y un código instrumentado es insertado antes de cada asignación y cerca de cada llamada de

método.

Una inserción típica luce de esta forma:

```
289 aload 4    // nuevo valor
291 astore_1 // variable local ie
292 ldc_w #404 <String "Micro4:Micro4.java:64">
295 ldc_w #416 <String "ie">
298 aload_1 // ie
299 aload_2 // Huella padre
300 invokestatic #14 <change(String, String, Object, Trace)>
```

Donde el código original son las líneas 289 y 291, asignando un valor a la variable local *ie*, la instrumentación crea un evento registra que sobre la línea 64 de *Micro.java*(#292), la variable local *ie*(#295), de quien su lista de historia puede ser encontrada sobre la línea de huella en el registro 2 (#299), le fue asignada al valor en el registro 1 (#298), los otros tipos de eventos son similares.

5.3. TOD: Depurador orientado a la huella para Java

5.3.1. Contribución

La contribución de este trabajo es mostrar que la depuración omnisciente puede ser realizada a través del uso de diferentes técnicas mejorando factores como la eficiencia, escalabilidad y usabilidad. Lo planteado es validado por TOD, un depurador portable orientado a la huella para Java integrado dentro de ECLIPSE [11]. Las características de TOD son:

- **Eficiente generación de eventos:** Basado en un compacto modelo de huella, un uso de codificación binaria de eventos, y un rápido tejedor de bajo nivel.
- **Máquina especializada de base de datos distribuida:** Para un almacenamiento escalable, rápido y consultas sobre eventos, que aprovecha la muy limitada naturaleza de la ejecución huellas. En unos 10 nodos dedicados en cluster, TOD mantiene una tasa sustancial de entrada de 170.000 eventos por segundo y cientos de consultas por segundo.
- **Soporte para huellas parciales:** Ofreciendo mecanismos estáticos y dinámicos para la generación selectiva de la huella, y un reporte adecuado para la información incompleta.

- **GUI informativa:** Debido a la eficiencia del procesamiento de consultas, TOD es utilizado para depurar una aplicación compleja como eclipse preservando su interactividad.
- **Componentes especializados de GUI:** Entregando una vista de alto nivel en huellas grandes de eventos para una navegación mucho más efectiva, como el mural de hilos.

5.3.2. Desafíos de la depuración omnisciente

Se presentan las características principales de un depurador omnisciente comparado con los depuradores tradicionales, y el perfil de los desafíos de escalabilidad de la depuración omnisciente.

5.3.2.1. Características de un depurador omnisciente

Un debugger omnisciente(OD; DO) entrega cuatro características principales:

- **Paso (stepping):** En depuradores basados en punto de quiebre, stepping consiste en la ejecución del programa objetivo una instrucción a la vez. Existen dos variaciones del stepping:
 - Step over ejecuta el método sin estar esperando dentro del llamado del mismo, mientras que el
 - Step into espera a la primera instrucción (comienzo) de la llamada del método.
- **Estado de reconstitución:** Reconstitución del estado consiste en ir entregando al programador un objeto de estado inspector cuando el programa objetivo está esperando.
- **Reconstitución del control del flujo:** Reconstitución del control de flujo permite obtener una vista en la llamada de la pila actual del programa, con ligadura a variables y objetos.
- **Encontrar la causa principal:** OD's extiende estas tres características con completa libertad en relación al tiempo: stepping puede ser realizado tanto hacia adelante como hacia atrás, los programadores pueden inspeccionar el estado de objetos como estaban en cualquier punto del tiempo, y pueden libremente navegar por el árbol de control.

La última es una característica única de los depuradores omniscientes, mientras que las otras son características de los depuradores típicos.

Finalmente, una de las características más usadas de OD's es su habilidad para encontrar *dónde y en qué contexto*, a un campo o variable en particular le fue entregado un determinado valor. Frecuentemente los bugs se manifiestan mucho después de suceder su causa principal. Para ejemplo, tratar de referirnos a una referencia *NULL* obtenida desde un campo dado causa una caída, esto es el síntoma del bug. La información que el programador necesita saber es dónde el campo fue definido con el valor *NULL*. Con un depurador basado en punto de quiebre, incluso si la ejecución es pausada justo antes del imperfecto referido, la causa raíz del bug pudo haberse perdido. Ejemplo: Porque el código que causó el problema ya no está más en la pila de llamadas.

5.3.2.2. Desafío de escalabilidad

Destacando las características presentadas sobre la necesidad de generar y registrar las huellas de ejecución. El potencial crecimiento de estas huellas plantean diversos desafíos de escalabilidad, cuando son la principal razón que afecta a la calidad de producción de OD's.

Los eventos deben ser grabados rápidamente, preferentemente en tiempo real, como que (a) la depuración puede comenzar inmediatamente después que el programa objetivo haya terminado o caído, y (b) el desbordamiento en tiempo de ejecución es minimizado para preservar sobre todo el desempeño del programa depurado y la interactividad cuando se necesite.

El depurador puede causar una interferencia mínima al programa objetivo en el sentido de no afectar su comportamiento. En particular, la administración del espacio de direcciones y de memoria del proceso objetivo no puede ser alterado.

La capacidad del almacén de eventos de un depurador omnisciente debe estar alineado con un número esperado de eventos dentro del uso total de la huella; en procesadores con capacidad de procesamiento de GHZ, cientos de millones de eventos pueden ser generados en solamente pocos minutos de ejecución.

Consultas dentro la huella de ejecución deben ejecutarse a una velocidad compatible con la interacción del usuario; por ejemplo: en décimas de segundos para operaciones como el stepping. La información debe ser presentada de una manera que trata que la carga cognoscitiva de la navegación a través de la enorme huella de los eventos sea la menor posible, permitiendo una rápida identificación de los errores.

Este trabajo se encamina en el uso optimizado de las representaciones de los eventos e indexación agresiva, un modelo simple de consulta, soporte para una base distribuida, soporte para huellas parciales y componentes de presentación e interacción especializados.

5.3.3. Acerca de TOD

TOD es un depurador omnisciente orientado a la huella para Java, que se dirige sobre la identificación del uso de la escalabilidad. El objetivo es dirigir esos usos en el orden para obtener un depurador omnisciente que es prácticamente aplicable. Esta sección da un panorama de TOD en el sentido de su arquitectura, el modelo de evento, y los componentes de la GUI.

5.3.3.1. Arquitectura

TOD es diseñado basándose en dos ideas centrales:

- Desacoplar el núcleo del depurador desde el programa objetivo en ejecución.
- Ser portable.

Esto es hecho sobre un árbol de componentes (Figura 5.2): el objetivo (JVM) en el cual el programa depurado se ejecuta y emite eventos, el núcleo del depurador que implementa las principales funcionalidades de TOD, y la interfaz del depurador a través del cual el usuario a la misma vez consulta y navega en la huella de ejecución.

La racionalidad para almacenar los eventos en una base de datos preferentemente que están en memoria como huellas en otros depuradores omniscientes [7, 12, 13] está precisamente dirigido a algunos de los desafíos discutidos en la sección 5.3.2.2: almacenar eventos en el espacio de dirección de la aplicación objetivo no es escalable para unos cuantos cientos de megabytes de información de la huella, e interfiere con el administrador de memoria, en particular con el colector de basura. El costo de captura

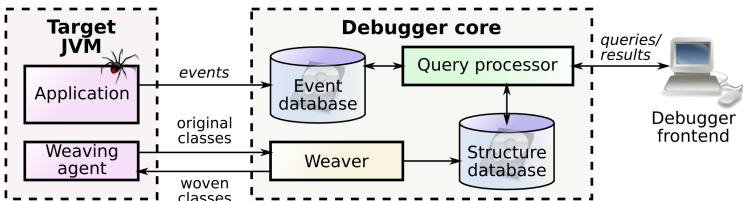


Figura 5.2: Arquitectura de alto nivel de TOD

se incrementa por el uso de una base de datos que es compensado por una mejor escalabilidad y sin generar intrusión. Como efecto secundario esto permite a los depuradores *post-mortem* tener la habilidad de serializar la huella de ejecución, lo que abre una interesante perspectiva para las compañías de software dispuesta a ofrecer software con alta calidad de soporte: observando el costo de almacenamiento, una huella de ejecución navegable es una información lejos mucho más relevante para reportar un bug que un texto descriptivo apropiado.

	kind	ts	tid	depth	pev	loc	fid	bid	vid	idx	val	ret	tgt	exc	args
Field write (FW)	✓	✓	✓	✓	✓	✓	✓				✓		✓		
Local var. write (VW)	✓	✓	✓	✓	✓	✓			✓		✓				
Array write (AW)	✓	✓	✓	✓	✓	✓				✓	✓		✓		
Exception (Ex)	✓	✓	✓	✓	✓	✓		✓						✓	
Behavior call (BC)	✓	✓	✓	✓	✓	✓		✓					✓		✓
Behavior enter (Bn)	✓	✓	✓	✓	✓	✓		✓					✓		✓
Behavior exit (Bx)	✓	✓	✓	✓	✓	✓		✓				✓	✓		

Figura 5.3: Eventos y sus atributos

De la figura debemos destacar que la cabecera de las filas son tipos de eventos y la cabecera de las columnas son posibles argumentos (ts: tiempo, tid: identificador del hilo, depth: llamada a la pila en profundidad, pev: puntero al padre del evento, LOC: localidad del código fuente, fid: identificador de celda, bid: identificador del método, vid: identificador de variable local, idx: índice, val: valor, ret: valor de retorno, tgt: objetivo, vxc: excepción; args: argumentos).

Durante la ejecución, la aplicación objetivo emite eventos que son enviados al núcleo del depurador, donde ellos son grabados e indexados en una base de datos de eventos. El tema de los eventos que son emitidos es discutido posteriormente. La base de datos de los eventos contiene las particularidades del curso del evento y el conjunto restrictivo de posibles consultas que entrega, entre el alto rendimiento de grabación y el buen desempeño de las consultas. El núcleo del depurador contiene otra base de datos, la base de datos estructural, que contiene información estática sobre la aplicación objetivo. En particular realiza un seguimiento de identificadores de 32 bits que son asignados a elementos estructurales del programa objetivo (por ejemplo: clases, métodos y campos(celdas)). Las consultas son efectuadas por el usuario basándose entre el evento y la base de datos estructural.

5.3.3.2. Representación y emisión de eventos

Introducimos ahora a la representación de los eventos y las huellas de estos, así como también los eventos son emitidos por la aplicación depurada en TOD.

- **Modelo de evento y huella:** Un evento es una estructura caracterizada por un número de atributos elegidos entre el conjunto $A = a_0, \dots, a_k$. Observemos $e.a_j$ el valor del atributo a_j del evento e . Para cada j perteneciente $[0 \dots k]$, dejamos D_j como el dominio de a_j , por ejemplo el conjunto de todos los valores distintos que pueden ser tomados por a_j , para cualquier evento dentro de la huella. Una huella del evento $T = \langle e_1, \dots, e_n \rangle$ es una secuencia ordenada de n eventos heterogéneos.

El atributo a_0 corresponde a la marca de tiempo del evento; este es caracterizado por el hecho que (a) todos los eventos tienen un valor para a_0 , (b) allí existe un completo orden sobre D_0 y (c) los eventos en T están ordenados por sus valores de a_0 . La figura 5.3 muestra cuales eventos concretamente son capturados y cuáles son sus atributos.

- **Emisión de los eventos** El núcleo del depurador de TOD captura los eventos emitidos por la aplicación objetivo (Figura 5.2). Existen tres vías en las cuales los eventos pueden ser emitidos: puertos especializados de la huella del hardware [14], máquina virtual o interpretador de instrumentación [15], y aplicación de instrumentación [7, 13]. TOD utiliza la última, aunque no es tan rápida como las pruebas de hardware y significativamente utiliza más espacio que el nivel de instrumentación de máquina virtual, en términos de tamaño de código, la instrumentación de aplicación es mucho más portable y fácil de implementar.

En TOD, la JVM¹ que está alojada en la aplicación objetivo está configurada para usar el agente JVMTI². El agente intercepta los eventos cargados por la clase y reemplaza las definiciones originales por la versión instrumentada. La Instrumentación es realizada por el tejedor en el núcleo del depurador:

- El agente envía el bytecode original al núcleo
- El tejedor implementa la clase y almacena estructuradamente la información en la base de datos estructural
- La clase modificada es enviada nuevamente a la JVM objetivo donde es eventualmente cargada (Figura 5.2). El agente toma la clase instrumentada del disco duro para reducir el número de turnos de los procesos internos. Esto es particularmente usado por el uso frecuente de las clases como algunas

¹Maquina Virtual de Java

²Interfaz de herramienta de la máquina virtual de Java, parte de la plataforma de Java 5.

en el JDK.

La instrumentación se efectúa utilizando la librería de bytecode ASM [16]: el código de captura del evento es añadido antes y/o después de un patrón específico de bytecode en el código original, tal como un campo de escritura o una llamada a un método. Cuando el código instrumentado es ejecutado, los eventos son contruidos a partir de sus atributos, serializados en un formato binario personalizado, y enviados a través de un socket a la base de datos de eventos.

- **Sincronización no ambigua del evento:** Aunque las marcas de tiempo del evento son obtenidos a través del servicio de precisión en nano segundos de Java, es potencialmente carente de exactitud, haciendo posible que varios eventos del mismo hilo tengan el mismo valor de marca de tiempo. Como esto es incompatible con el esquema de indexación usado por TOD, se cambió el valor original del timestamp por unos bits a la izquierda y se usan los bits libres para diferenciar eventos del mismo hilo que sea parte de la misma marca de tiempo. Cuando se compara la marca de tiempo de los eventos de diferentes hilos, se usa la marca de tiempo original para preservar el orden de los hilos internos(inter-thread).
- **Alcance de la captura de huella:** El esquema de instrumentación descrito anteriormente es selectivo, esto es, posible de proveer filtros definidos para los usuarios que limite el número de eventos emitidos.
- **Identificación del objeto:** El agente JVMTI de TOD asigna un único identificador para cada objeto en la aplicación objetivo. Como una excepción a este mecanismo las instancias de los objetos que representan valores primitivos (por ejemplo entero, decimal, etc) tales como las cadenas y excepciones son pasadas por valor.

5.3.3.3. Bajo nivel de consultas: cursores y contadores

Todas las características presentadas en la sección 5.3.2.1 (paso a paso, reconstitución de estado, reconstitución del control de flujo y encontrar la causa raíz) pueden ser expresados en términos de dos niveles de consultas de bajo nivel: cursores y contadores, que se introduce a continuación. Ambos basados en el filtrado de eventos de la huella de acuerdo a algunas condiciones de sus atributos. Condiciones que pueden ser cualquier combinación booleana de simples predicados de la forma *attribute = value*,

donde *value* es una constante. Para la instancia (*kind* = *FW* o *kind* = *BC*) o *target* = *obj145*. Si *Q* es una condición y *e* es un evento, definimos el predicado de la función *Q(e)* cuyo valor es verdadero si *e* verifica la condición *Q*.

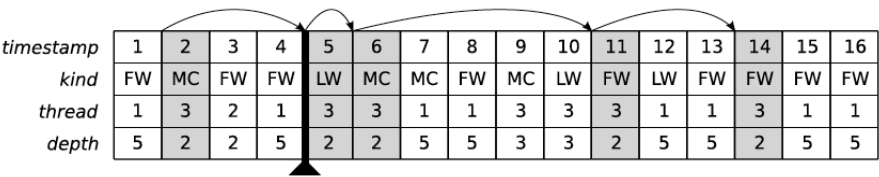


Figura 5.4: Navegación entre la unión de los eventos y el predicado del cursor

De la imagen se debe notar que la posición actual del cursor está representado por la línea negra entre los eventos cuatro y cinco. Los Eventos que juntan el predicado del cursor están en gris. Se llaman sucesivamente a *next()* retornando los eventos 5,6,11 y 14; llamando *posNext(11)* la posición del cursor estará entre los eventos 10 y 11; llamando *posPrev(11)* las posiciones estarán entre los eventos 11 y 12.

Operación	Semántica (significado)
next() / prev()	Retorna el siguiente/previo unión de eventos y mueve el cursor hacia adelante/atrás.
posNext(t) / posPrev(t)	Mueve de modo que la siguiente llamada a next()/prev() retorna el primero/ultimo evento cuya marca de tiempo es mejor/peor que o igual que t.
posNext(ev) / posPrev(ev)	Mueve de modo que la siguiente llamada a next()/prev() retorne el evento dado.

Tabla 5.1: Operaciones de los cursores

- **cursores** Definimos *cursor(Q)* como un iterador sobre los eventos que reúnen la condición *Q* (Figura 5.4). Los cursores tienen una posición actual que está situada entre dos eventos consecutivos (o hacia el principio o el término de la huella). Un cursor soporta un número de operaciones de navegación, como las mostradas en la tabla 5.1.
- **contadores** Entregan un intervalo de tiempo $[t1, t2]$ dividido en *s* porciones de largo $\delta t = (t1, t2)/s$ cada uno, y una condición *Q* para los atributos del evento, una consulta de contador retorna un arreglo de *s* enteros tales que $s[i] = |e \in T : dentro(e, t1 + i * \delta t) \wedge Q(e)|$ donde $dentro(e, t) \iff e.ts \geq t \wedge e.ts < t + \delta t$. Cada posición del arreglo contiene el número de eventos unidos *Q* que ocurrieron durante la correspondiente porción de tiempo.

5.3.3.4. Alto nivel de consultas

A continuación se explica como los cursores y los contadores son combinados algorítmicamente para implementar el alto nivel de las características descritas anteriormente.

- **Stepping:** Definimos stepper como un objeto que tiene un evento actual ev y soporta operaciones paso hacia adelante y hacia atrás, paso dentro y paso sobre. Por ejemplo, paso hacia adelante dentro es definido como sigue:

$$c \leftarrow \text{cursor}(\text{thread} = ev.\text{thread})$$

$$c.\text{posPrev}(ev); ev \leftarrow c.\text{next}()$$

Paso hacia adelante sobre, cambia la condición del cursor $a : \text{thread} = ev.\text{thread} \wedge \text{depth} = ev.\text{depth}$. Paso atrás es simétrico al paso hacia adelante.

- **Reconstitución de estado:** El valor v de una celda f de un objeto en particular o con tiempo t puede ser recuperado como sigue:

$$c \leftarrow \text{cursor}(\text{kind} = FW \wedge \text{fid} = f \wedge \text{target} = o)$$

$$c.\text{posPrev}(t); v \leftarrow c.\text{prev}().\text{val}$$

El estado de un objeto puede ser recuperado realizando la misma operación para cada uno de los campos. La pila de marcos son reconstituidos en una manera similar, usando eventos de escritura variables instanciados de la celda de eventos escritos.

- **Reconstitución del control de flujo:** Los eventos que ocurren en el alto nivel del control de flujo de una llamada dada del método e son recuperados de la siguiente manera:

$$c \leftarrow \text{cursor}(\text{thread} = e.\text{thread} \wedge \text{depth} = e.\text{depth} + 1)$$

$$c.\text{posPrev}(e.ts); cflow = \langle \rangle$$

repeat

$$ev = c.\text{next}(); cflow \leftarrow cflow \sqcup \langle ev \rangle$$

until $ev.\text{kind} = BEx$

- **Buscador de la causa inicial:** Determinando como a un campo ha sido asignado un valor indeseado, es tan directo como la consulta de la reconstrucción del estado: en lugar de obtener el valor del evento de la celda escrita que asignó el valor

al campo, el evento por sí mismo se hace el actual, dando acceso para el contexto en ese momento. La exploración atrás en el tiempo es la causa que pueda ir sobre esto, encima de la causa inicial.

5.3.3.5. Componentes de la interfaz gráfica

La interfaz de TOD puede ser usada independiente o como una añadidura para el IDE de Java llamado Eclipse (Figura 5.5). El navegador de usuario entre diferentes vistas utiliza las ampliamente conocidas metáforas de los navegadores Web (hiper vínculos, botón hacia atrás, etc). Las vistas disponibles son: inspector de objetos, control de flujo, y mural.

- **Vista de inspección de objetos:** Muestra la reconstitución de objetos, y permite encontrar la causa inicial por valor de celdas a través de un conveniente ¿por qué? link a cada campo siguiente.
- **Vista de control de flujo:** Muestra una constitución del flujo de control y permite las operaciones de stepping así como encontrar la raíz de la causa para valores de variables locales.
- **Murals:** Las descripciones del nivel alto son útil para marcar patrones anormales del comportamiento. Sin embargo representando un número fuerte de eventos en un limitado número de píxeles es difícil. Jerding y Stasko introdujeron la información del mural [17] como una *reducción de la representación de la información del espacio entero a uno que se ajusta completamente dentro de una ventana mostrada*. Las características de los eventos murals de TOD, los cuales son gráficas que muestran la evolución de la densidad del evento, o un número de eventos por unidad de tiempo, en un periodo dado. Por ejemplo, TOD puede mostrar la densidad del evento por cada uno de los hilos para el conjunto de ejecución de la aplicación objetivo (Figura 5.6), o la densidad de llamadas a métodos de un objeto en particular. Las densidades son obtenidas a través de los contadores, donde el largo de las tajadas de tiempo correspondiente al espacio utilizado por un simple pixel de la barra en el mural. El usuario puede hacer un acercamiento y ver el mural; cuando el nivel de acercamiento permite distinguir los eventos individuales del usuario y puede seleccionar un evento y ver el contexto dentro de una vista de paso. El mural de hilos tiene distintas aplicaciones, por ejemplo: Entender la interoperación entre los hilos.

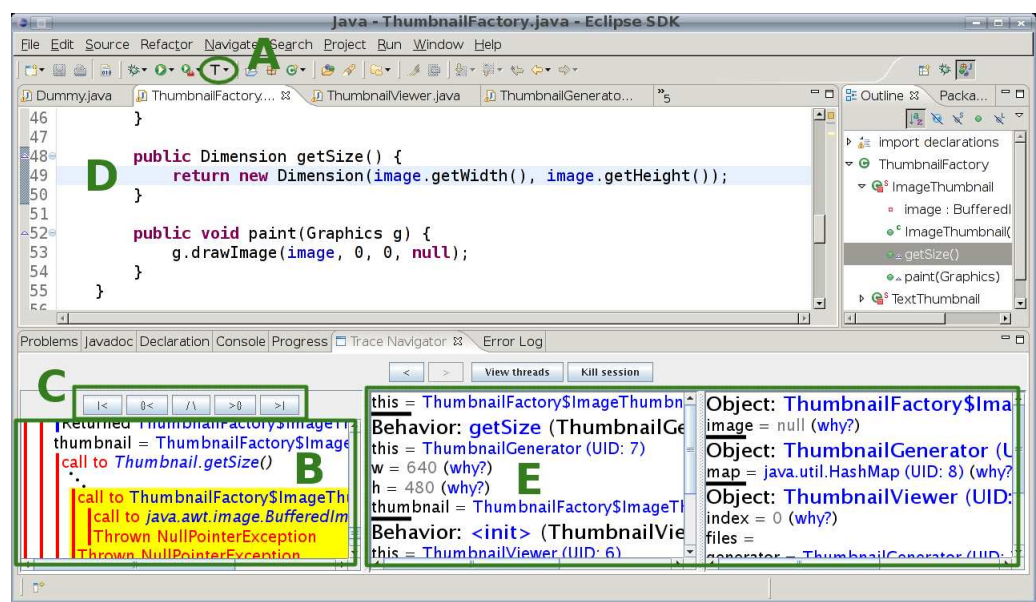


Figura 5.5: Navegación con TOD en Eclipse

De la figura anterior se debe notar que el botón (A) lanza el programa con la traza registrando la huella. El usuario navega en el control de flujo (B) utilizando los botones de paso (C), o por medio de un click sobre el evento. La línea correspondiente al evento actual está resaltada en la ventana del código (D). El estado de la pila de marcos y el objeto actual es mostrado en la ventana (E). El usuario puede saltar de una instrucción que defina el valor actual de una variable o campo por medio de un click en *¿Por qué?* siguiendo a este.

5.3.4. Soporte para base de datos de alta velocidad

Ahora se describe y se analiza el esquema de indexación de TOD, el cual permite la ejecución eficiente de consultas mientras es lo suficientemente rápida para permitir un alto rendimiento de grabación.

La necesidad de desarrollar un cimiento de base de datos especializada para TOD fue motivado por el bajo rendimiento del uso profundo de la administración de sistemas de base de datos para los propósitos de TOD: PostgreSQL y Oracle sólo soportan almacenar eventos a una tasa de 50 y 500 eventos por segundo respectivamente, mientras que en los registros de TOD la tasa es del orden de los cientos de millones de eventos por segundo [18]. Este alto rendimiento especializado en los cimientos de la base de datos permiten las siguientes especificaciones para el flujo del evento de una huella de ejecución: (a) el flujo del evento es de sólo lectura, (b) los eventos al llegar son ordenados a partir de su marca de tiempo y (c) las consultas son limitadas por el filtrado.

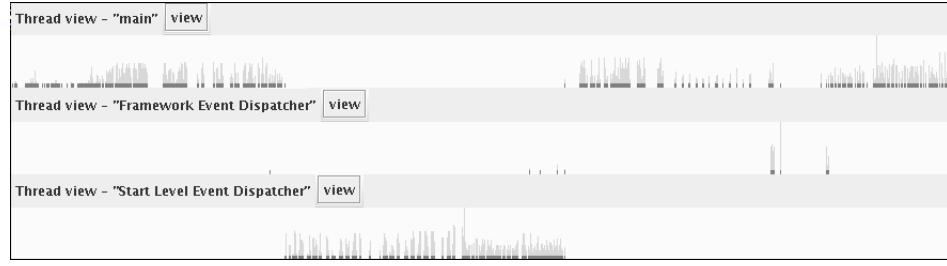


Figura 5.6: Mural de threads

El gráfico muestra la densidad de los eventos a lo largo del eje del tiempo.

5.3.4.1. Jerarquía de indexación de los eventos

TOD adopta un esquema de indexación jerárquica que permite recuperar al evento juntando un predicado en orden según su marca de tiempo sin tener que acceder realmente a los eventos en si, reduciendo así los costos del procesamiento de consultas.

- **Indices en los valores de los atributos:** Usando la notación definida anteriormente, se define el índice de T en a_j como una función $I_j : D_j \rightarrow (D_o, N)^*$ para $j \in [1..k]$ de modo que ningún mapa I_j tenga cualquier posible valor v de a_j una secuencia de índices de entrada de la forma (ts, i) . Una entrada se encuentra en el índice $I_j(v) \Leftrightarrow e_i.a_j = v \wedge e_i.a_0 = ts$, donde e_i es el i -ésimo evento de T . Adicionalmente, las entradas son ordenadas por ts . Estos índices pueden ser usados directamente para recuperar todos los eventos que correspondan a una consulta simple de la forma $attribute = value$.
- **Jerarquía de indexación por marca de tiempo:** Debido a que las consultas en TOD no consisten sólo en encontrar eventos correspondientes que ocurrieron antes o después de un momento, los índices son ordenados por sus valores de ts , entonces es posible presentarlos en una búsqueda binaria para encontrar la marca de tiempo deseada.

Esto es siempre mucho más eficiente para extender el índice de estructura en una manera jerárquica (Figura 5.7). Cada índice jerárquico para el valor v del atributo a_j contiene un número de niveles; El índice $I_j(v)$ describe sobre la constitución del nivel 0. La entrada (ts, i) índice del nivel 0 son almacenados en el disco duro en pequeñas páginas, donde cada una contiene un número de entradas pertenecientes al mismo índice. Cuando una página está llena, una entrada de la forma (ts, pid) es creada en el índice del nivel 1 : ts es tomado desde la primera entrada (ts, i) de la reciente página llena, y pid es un puntero a esa página. Las entradas

del Nivel 1 están acumuladas en una página; cuando una página del Nivel 1 es llenada, una entrada en el Nivel 2 es creada, y así sucesivamente.

El nivel más alto siempre contiene una pagina sencilla, llamada la página raíz. El número de niveles sobre el Nivel 0 de un índice es llamado la altura del índice.

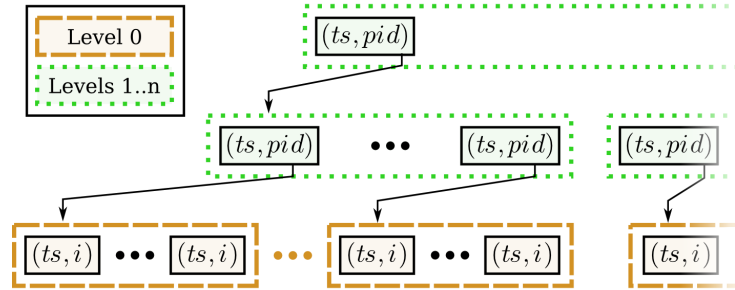


Figura 5.7: Jerarquía de los índices

5.3.4.2. Costo de la creación de un índice

Los experimentos realizados en el desarrollo de TOD muestran que el tamaño promedio de un evento es $\|e\| = 50$ bytes. El tamaño de una entrada de Nivel 0 es $(ts, i) = 16$ bytes (dos enteros de 64 bits). El tamaño de las entradas de nivel superior es $(ts, pid) = 12$ bytes (pid está en 32 bits). Las experiencias realizadas en el desarrollo de TOD han determinado que el tamaño de una página óptima es $P = 4096$ bytes, por lo tanto los índices de las páginas de Nivel 0 contienen 256 entradas, en los niveles superiores las páginas contienen 341 entradas y las páginas de eventos contienen 81 eventos en promedio. La altura h de los índices es logarítmica con respecto al número de entradas y en la práctica nunca excedieron 5 (un índice de altura 5 permite para $341^5 \approx 4 \cdot 10^{12}$ entradas) Asumiendo que la actual página de eventos puede residir en memoria, la página de eventos sólo causa una página escrita cada 81 eventos.

Para cada evento que entre a la base de datos a lo más existirá $k = |A| - 1$ índices que actualizar (como no hay índice separado en a_0). Los experimentos indican que el promedio $k = 10$. Dado ciertos eventos que llegan en orden con respecto a_0 , actualizando un índice sólo pensando en añadir una entrada en el final de la página en el Nivel 0, y en el final de las páginas de nivel superior sólo cuando una página de nivel inferior es llenada. La entrada/salida y costos de memoria de esta operación son los siguientes:

- Si la página de índices actual para cada nivel puede residir en memoria, un costo de E/S es realizado sólo cuando una página está llena. El número promedio de

páginas accesadas por el evento entrante es:

$$\frac{\|e\|}{P} + k \cdot \sum_{i=1}^h \frac{\|(ts, i)\|}{P} \cdot \left(\frac{\|(ts, i)\|}{P} \right)^i \simeq 0,05$$

- Si sólo la página de índices está en el nivel 0 puede ser mantenida en memoria, cuando una página es llenada debe ser escrita, El nivel de la página leída es el 1, actualizado y escrito en el disco. Analizando, el nivel más alto, tiene aproximadamente 0,13 accesos por evento.
- Si la pagina no puede ser mantenida en memoria, cualquier actualización implicará las tres operaciones anteriores, entregando 20 accesos por evento.

La cantidad de memoria dedicada al buffering de página es por lo tanto lo más importante: Hay una diferencia de 400 directorios entre la situaciones extremas anteriores.

Ahora se estima el número total de índices, $\sum_{j=1}^k |D_j|$, en orden de determinar cuántas páginas de buffer de memoria se necesita para permanecer en el caso donde al menos la página actual del nivel 0 para que cada índice quepa en memoria. Los atributos de los eventos pueden ser divididos en dos categorías. El dominio de los atributos estáticos (por ejemplo: behavior id, field id, type id, ubicación, etc.) depende solamente de la estructura del programa, no del tamaño de la huella. En las pruebas con la captura de las huellas en Eclipse, se observó que estos acumulan cerca de 200.000 distintos valores. Hay dos atributos dinámicos: thread id y object id. Su dominio puede ser enorme dado que el programa objetivo puede repetitivamente crear y destruir objetos e hilos durante su ejecución. Sin embargo, sólo una fracción de ellos puede ser usado en cualquier punto dado en el tiempo: Todos los objetos que viven deben ajustarse dentro de la JVM's objetivo que está disponible en memoria y todos los hilos que viven deben ser razonablemente ejecutados por la CPU. Así es solamente necesario tener un espacio para el índice del buffer de la pagina para los objetos e hilos usados actualmente.

El dominio de los object id's claramente domina todos los dominios acumulados. Si la aplicación objetivo regularmente usa un millón de objetos, cerca $P \cdot 10^6 = 4 \text{ GB}$ de espacio es necesario para el buffer.

5.3.4.3. Costo de recuperación del evento

Ahora se presenta los algoritmos que permiten recuperar eventos buscando un predicho arbitrario en la línea de tiempo con respecto al tamaño de los índices involu-

crados. Los algoritmos son recuperados en orden respecto a su marca de tiempo; la recuperación al revés tiene el mismo costo.

- **condiciones simples:** Para una condición simple de la forma $a_j = C$ donde C es una constante, podemos recuperar eventos buscados ordenados por marca de tiempo, simplemente obteniendo las entradas (ts, i) desde $I_j(C)$. Si el evento actual es requerido (por ejemplo: por los cursores), estos son directamente recuperados desde la huella como e_i ; si no (por ejemplo: para los contadores) el evento no necesita ser accesado. En cualquier caso, todas las entradas pueden ser recuperadas en la línea de tiempo, debido a que el índice se explora simplemente una vez.
- **condiciones conjuntivas:** Para una conjunción booleana de una condición simple de la forma $a_{j_1} = C_1 \wedge \dots \wedge a_{j_m} = C_m$, se utiliza una variante del algoritmo sort merge join [19], extensamente utilizado en la administración de los sistemas de bases de datos, para identificar los eventos buscados sin accesarlos (Algoritmo 1): obtenemos el $I_{j_l}(C_l)$ para cada una de las condiciones simples, y por cada uno mantenemos un puntero a la entrada actual (ts_l, i_l) . Entonces iteramos: en cada paso verificamos si todos los i_l son iguales, en este caso añadimos cualquier resultado a las entradas actuales: El hecho que todos ellos hacen referencia al mismo evento significa que el evento coincide con todas las condiciones. Entonces se avanza el puntero de índice cuya entrada actual tenga el mínimo valor para la marca de tiempo. Como cada índice es revisado una sola vez y no existen ciclos anidados, merge join corre en tiempo lineal con respecto a la suma de los tamaños de los índices considerados.

5.3.5. Medidas de rendimiento

Se presentan reportes sobre el primer conjunto de medidas de rendimientos evaluando diferentes aspectos de TOD:

- Base de datos distribuida, en términos de registro de eventos y evaluación de consultas.
- Sobrecarga efectuada por la emisión de eventos en la aplicación depurada.

Algorithm 1 MERGE-JOIN

Entrada: $(S, j_1, \dots, j_m, C_1, \dots, C_m)$

$result \leftarrow 0$

para $l = 1$ to m **do**

$index[l] \leftarrow I_{j_l}(C_1), pos[l] \leftarrow 1$

fin para

mientras there are more elements **do**

$match \leftarrow \mathbf{true}, refI \leftarrow -1$

$minL \leftarrow -1, minTS \leftarrow +\infty$

para $l = 1$ to m **do**

$(curTS, curI) \leftarrow index[l][pos[l]]$

si $refI = -1$ **entonces**

$refI \leftarrow curI$

sino si $curI \neq refI$ **entonces**

$match \leftarrow \mathbf{false}$

fin si

si $curTS < minTS$ **entonces**

$minTS \leftarrow curTS, minL \leftarrow l$

fin si

fin para

si $match$ **entonces**

$result \leftarrow result \cup \{s_{refI}\}$

fin si

$pos[minL] \leftarrow pos[minL] + 1$

fin mientras

5.3.5.1. Desempeño de la base de datos

Para evaluar el rendimiento de la base de datos distribuida de TOD, se han realizado varias mediciones de rendimientos en relación al registro y consultas bajo distintas configuraciones. Se capturó una gran huella de ejecución de una sesión de Eclipse donde el usuario realizó un simple secuencia de pasos: Abrió un archivo escrito en Java, lo editó utilizando completación automática, creando una nueva clase y la editó. La huella capturada comprende aproximadamente 516 millones de eventos y pesa 20GB. Luego se importó esta huella dentro de la base de datos de TOD, que utiliza 16 nodos dedicados dentro de un cluster. En este experimento hasta 10 nodos estaban para ser utilizados como nodos de base de datos, y 1 como disparador y agregador de consultas. Cada nodo es un Intel Itanium de 1.60GHZ con 2GB de RAM y 7GB espacio disponible en el disco duro local, corriendo un kernel Linux 2.6.9 y BEA JRockit 1.5.0 06 JVM. Los nodos están conectados a través de un adaptador de red de 1Gbps. Desafortunadamente esta no es una configuración ideal para TOD.

La primera medida de rendimiento es el tiempo tomado para importar la huella de ejecución dentro de la base de datos. La segunda medida de rendimiento es la tasa en la cual los eventos individuales coinciden con una condición arbitraria pueden ser recuperados utilizando un cursor, y la tercera medida de rendimiento es el tiempo tomado para computar los contadores de eventos de estos mismos para la duración completa de la huella.

- **Registro:** Primero se determinó la tasa de transferencia máxima del disparador mediante la desactivación del procesador de eventos en relación a los nodos de la base de datos: El disparador puede manejar hasta 200.000 eventos por segundo, independientemente del número de nodos (Figura 5.8) Los resultados actuales muestran que una nodo de base de datos es capas de manejar aproximadamente 50.000 eventos/segundo, y que con 10 nodos se obtiene limitar al disparador cerca de los 170.000 eventos/segundo (Figura 5.9). El rendimiento incrementa con el uso de más nodos, aunque no muy linealmente. La comparación es sin embargo basada por el hecho que con pocos nodos, menos eventos fueron importados. Se puede conjeturar que el rendimiento obtenido con pocos nodos habría sido menor si se hubiera sido capas de importar la huella completa.

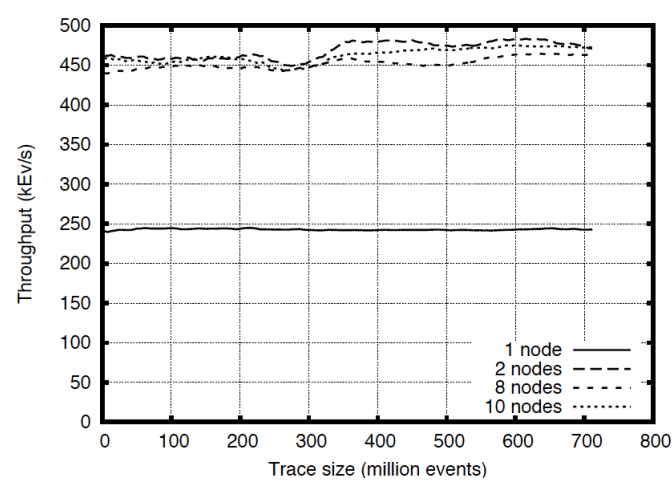


Figura 5.8: Rendimiento del lanzador

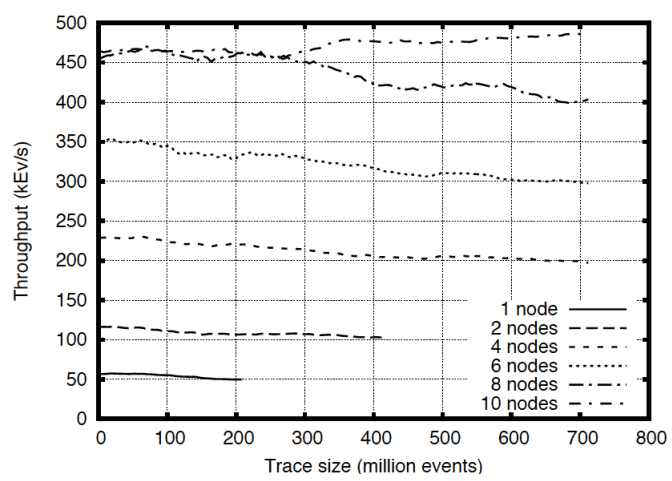


Figura 5.9: Rendimiento del registro

- **Cursores de consultas :** Se midió la velocidad de ejecución de dos tipos de cursores de consultas stepping-related: seek y step. Ambas están basadas sobre en una condición compuesta generada aleatoriamente que se ajusta con eventos de determinado hilo de ejecución y profundidad de la llamada. La búsqueda de las

peticiones de las consultas de un cursor con una condición, posiciona una marca de tiempo elegida al azar en el lapso de ejecución de la traza y obtiene el siguiente evento que corresponda. La operación es repetida 1.000 veces, cada tiempo con diferentes condiciones y marcas de tiempos. Las consultas de paso (step) además solicitan un cursor similar y se ubican en una marca de tiempo aleatoria, para luego traer los siguientes 1.000 eventos correspondientes. La operación es repetida ciento de veces.

Los resultados se muestran en Tabla 5.2. En este experimento se utilizaron un tamaño fijo de eventos de huellas de 80 millones de eventos las cuales pueden ser importadas en una base de datos usando 3 de los 10 nodos. La eficiencia de consultas de búsqueda no mejora, e incluso decrece, cuando más nodos de base de datos son utilizados. Este es porque cada consulta de búsqueda recupera sólo un evento, el cual obviamente está presente en un solo nodo. El tiempo de buscar ese evento es por lo tanto igual al máximo del tiempo tomado por cada nodo para encontrar el siguiente evento que corresponda, entregando la condición de la consulta y la marca de tiempo. Las consultas de paso son un poco más rápidas cuando más nodos son utilizados pero sorprendentemente la mejora está lejos de ser lineal. Aún no se ha encontrado una explicación para este resultado inesperado. A pesar de una débil capacidad de ampliación, la posición del cursor de consultas se ejecuta en décimas de segundos, lo suficientemente rápido para ser utilizado interactivamente a través de la interfaz de depuración de TOD.

- **Contadores de consultas:** Se midió la velocidad de ejecución de los contadores de consultas y se compararon dos métodos: merge counts y fast counts. Se utilizó una porción de la huella de ejecución de Eclipse descrita anteriormente, que contenía 80 millones de eventos distribuidos en 27 hilos de ejecución. Se solicitaron al contador de eventos por cada hilo sobre el lapso de la huella, dividiendo $n = 1.000$ subintervalos. Los resultados son entregados en Tabla 5.3. Varios puntos que son importantes de destacar:
 - Contadores rápidos proveen una aproximación bastante precisa, con una distorsión por bajo del 2 % de comparación con los resultados reunidos de los contadores.
 - Contadores rápidos son mucho más rápido que los contadores de fusión, pero no mejoran en relación al aumento de nodos en la base de datos, debido

a que cada nodo registra menos eventos, el algoritmo de contador rápido debe recurrir frecuentemente a índices de niveles inferiores.

- Los contadores de fusión escalan linealmente, en términos de números de nodos.

Nodes	seek(q/s)	step(q/s)
3	6.85	267
6	6.05	273
8	5.29	299
10	5.04	353

Tabla 5.2: Consultas de cursores

Nodes	merge(s)	fast(s)	speedup	dist.(%)
3	527.2	9.6	54.9x	0.4
5	288.6	9.7	29.8x	0.7
8	177.1	10.8	16.4x	1
10	133.8	8.9	15x	1.21

Tabla 5.3: Consultas de contadores

5.3.5.2. Sobrecarga por emisión de eventos

En el sentido de evaluar la sobrecarga de la emisión de los eventos, se compara el tiempo de ejecución de un programa (a) independiente, (b) con TOD y (c) con ODB, otro depurador omnisciente para Java. Como en esta medida de rendimiento se desea medir la sobrecarga por emisión de eventos cuasado por TOD y no el rendimiento de su base de datos, los eventos simplemente son escritos en el disco sin indexación.

El programa objetivo es un programa de prueba que hace uso intensivo de CPU, el cual crea 100 instancias de objetos y luego itera 10 millones de veces en un ciclo que toma uno de estos objetos de forma aleatoria y le pasa un método que efectúa una simple operación aritmética sobre su código hash. El programa no llama a ningún método no instrumentado y por eso que cada operación emite un evento.

Esta prueba de rendimiento fue realizada en un Pentium M de 2GHz notebook con 1GB de ram corriendo el kernel 2.6.17 de Linux y la JVM 1.5.0 08 de Sun. Los resultados son presentados en la Tabla 5.4. Como los eventos en ODB son almacenados en la pila de la JVM del programa objetivo; los eventos antiguos son eliminados cuando la pila está llena. Se realizaron dos pruebas con ODB, cambiando el tamaño de la pila de la JVM. Con 500MB la pila fue capas de registrar 5 millones de eventos de

110 millones emitidos durante la ejecución del programa. Con 64MB se pudo registrar solo 500.000 eventos. Por otra parte con TOD se fue capas de registrar 90 millones de eventos emitidos sin interferir con la pila de la JVM. La huella de ejecución generada pesa 3.6GB. La sobrecarga de la emisión de eventos es similar en TOD y en ODB: alrededor de 115 veces la de TOD, donde los eventos son serializados y escritos en disco.

Se debe hacer notar que esta medida de rendimiento representa el peor escenario. Se midió la sobrecarga de uso intensivo de CPU, instrumentando completamente un programa, mientras que en las situaciones típicas de depuración algunas partes de los programas son excluidos de la instrumentación, como se expondrá posteriormente.

Setup	RAM	time	emit.	rec.	rate	ovh.
None	16	1.53	N/A	N/A	N/A	1x
ODB1	500	179	110m	5m	614k	116x
ODB2	64	188	110m	530k	585k	122x
TOD	16	173	90m	90m	520k	113x

Tabla 5.4: Sobrecarga en la emisión de eventos

5.3.5.3. Discusión

Los resultados experimentales presentados anteriormente muestran que es factible registrar y consultar grandes huellas de ejecución para el propósito de la depuración omnisciente. Se fue capas de registrar una huella de ejecución de 20GB correspondiente a una sesión de trabajo en Eclipse, un IDE bastante complejo orientado a Java, y se importó sobre los 400 millones de eventos de esa huella en la base de datos distribuidas para eventos a una tasa de 170.000 eventos/segundo, cercano a tres veces más lento que la tasa máxima de eventos observada, 520.000 eventos/segundo. La base de datos fue también capas de servir entre 5 y 350 cursores de consultas por segundo y producir contadores globales de consultas para 27 hilos de ejecución en menos de 10 segundos; cada tiempo de respuesta es compatible con los requerimientos de interactividad de la interfaz de usuario del depurador. Se utilizó la base de datos distribuida sobre un cluster dedicado y realizando medidas utilizando 1 hasta 10 nodos de la base de datos. La base de datos demostró escalabilidad por cada evento registrado y consultas, pero no tanto para los cursores de las consultas. Por lo que respecta a la emisión de eventos la sobrecarga es preocupante, se observó que un intensivo uso de CPU, de un programa completamente instrumentado ejecutado bajo TOD disminuye alrededor de 115 veces

y emite alrededor de 520.000 de eventos/segundo. Este caso es similar al obtenido con ODB, un depurador omnisciente que almacena eventos en un espacio de direcciones en el programa objetivo y de esta forma no es tan escalable. En esta implementación el disparador de eventos impone un embotellamiento de 200.000 eventos/segundo; Una mejor implementación del disparador (presumiblemente en C) es necesario si las huellas de ejecución son grabadas en tiempo real.

5.3.6. Trabajando con huellas parciales

Aunque TOD está diseñado para soportar grandes huellas de ejecución, no siempre es practicable grabar cada evento: El tiempo de ejecución mayor de una captura de evento es importante, como también lo son los requerimientos de almacenamiento. La idea de las huellas parciales es que se puede afirmar el echo que durante el desarrollo de una pieza de software, algunos componentes son confiables, por ejemplo: maduros y bien probados, y por esto no es necesario generar y almacenar eventos para las actividades internas de esos componentes. Esta sección muestra como el alcance de la captura de la huella puede facilitar la depuración y como TOD hace posible el trabajo con huellas parciales.

5.3.6.1. Ejemplo de motivación: Depurando el plugin de TOD en eclipse

Consideremos como un ejemplo de depuración el mismo plugin de TOD en eclipse. Este ejemplo es bastante representativo de un componente de desarrollo existente, probado, frameworks o arquitecturas de plugin. Aquí, se está fuertemente interesado en dos tipos de bugs: aquellos que son internos del plugin y aquellos relacionados con la interacción entre el plugin y la plataforma. En el primer caso, no se necesita capturar eventos que ocurren dentro de la plataforma ECLIPSE porque eso es considerado como probado. En el segundo caso, se tiene que grabar eventos que ocurren dentro de la plataforma, pero no necesariamente todos: puede que sean bastantes los eventos que se deban grabar de la Java Tooling(JDT), o solamente alguna parte de ella, por ejemplo la interfaz de usuario.

Figura 5.10 muestra el impacto de diferentes alcances de estrategias de la huella en comparación con el número de eventos emitidos y el exceso del tiempo de ejecución, durante diferentes etapas de la ejecución del plugin de TOD. En este pequeño experimento se puede ver que por apropiado que sea el alcance de la captura de la huella, existe sobre cinco ordenes de magnitud de diferencia en el número de eventos emitidos,

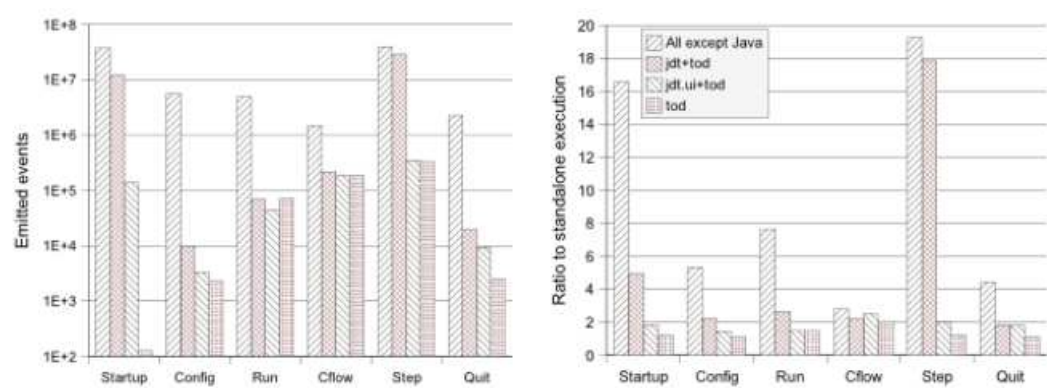


Figura 5.10: Eventos emitidos y sobrecarga en tiempo de ejecución.

y que los aumentos en el exceso del tiempo de ejecución puede ser sobre 20 tiempos, realza enormemente la aplicabilidad de TOD.

5.3.6.2. Tratando con información incompleta

El inconveniente de ignorar algunos eventos es que la captura de la huella de ejecución es incompleta, y por lo tanto, alguna información es precaria para reconstruir la historia completa del programa. El soporte para huellas parciales de ejecución en TOD es conseguida mediante reportes sistemáticos, sobre la información perdida, al usuario para que el pueda razonar sólidamente acerca de la información disponible. La información perdida se manifiesta en dos áreas: cuando el código no instrumentado es llamado desde un código instrumentado, y en el turno de las llamadas de código instrumentado, alguna información sobre el flujo de control es perdida; en este caso TOD entrega indicadores visuales en los lugares apropiados, como se muestra en la Figura 5.11 Segundo, en el estado de reconstitución: si una clase tiene un atributo no privado que es escrito por código no instrumentado, el valor de este campo en un punto determinado del tiempo no puede ser determinado de forma exacta. TOD representa estos campos con un color distinto en las correspondientes vistas.

Los pequeños puntos indican que la información del flujo de control pudo ser perdida: El método Collections.sort no está instrumentado pero llama al método de comparación de la clase instrumentada Comp durante su ejecución

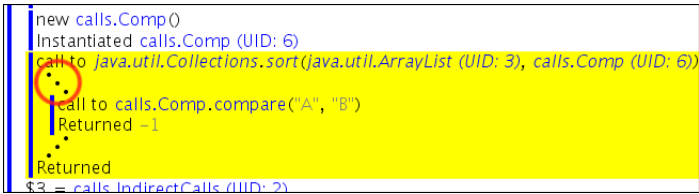


Figura 5.11: Reportando una información de flujo de control potencialmente incompleta.

pyTOD

6.1.1. Modelo de componentes

Para tener una mejor perspectiva de los componentes que componen pyTOD, es que se utiliza el diagrama de componentes del lenguaje unificado de modelamiento UML. Es importante señalar que este diagrama en ningún caso es un diagrama de bajo nivel.

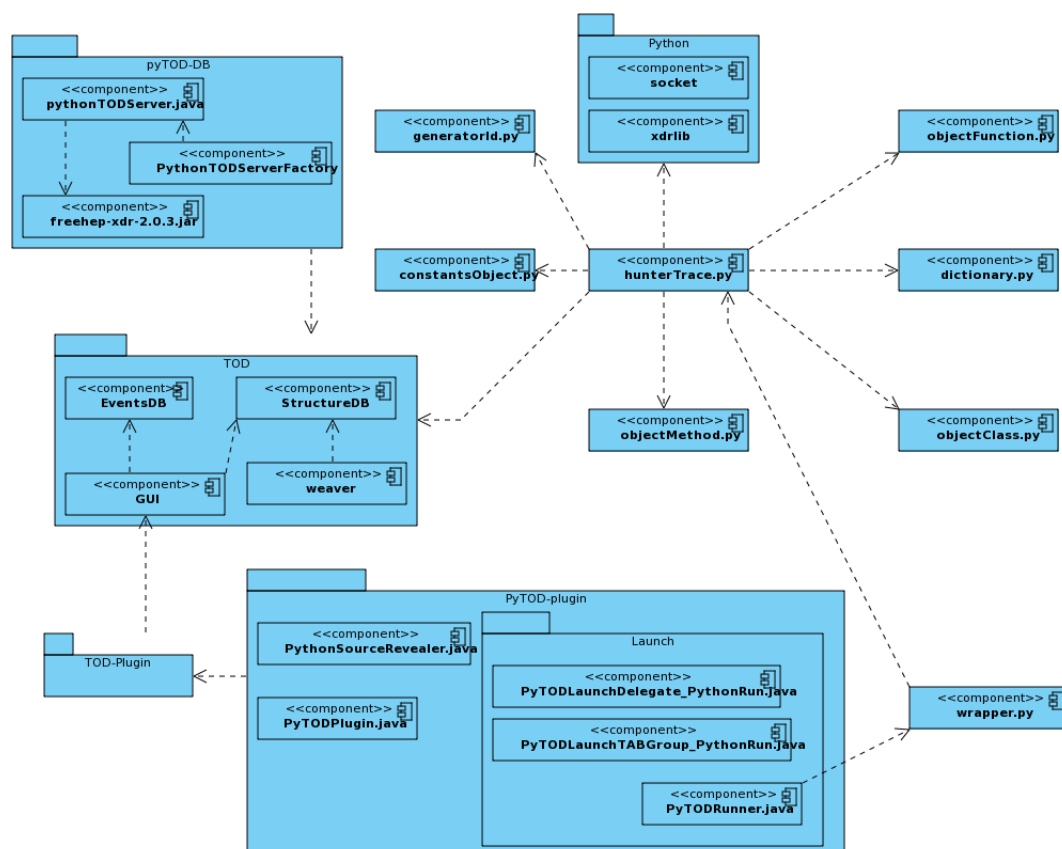


Figura 6.1: Diagrama de componente del captador de huella

6.1.2. Wrapper

Por un asunto de usabilidad, se ha construido un wrapper con el objetivo que el programador no deba modificar ninguna línea de los script que desee depurar. A continuación se muestran dos ejemplos, el primero sin utilizar el wrapper y el segundo utilizando el wrapper.

- Código sin wrapper

```
1  import sys
2  sys.path.append('/Volumes/archivos/eclipse/workspace/
   python-project/src')
3  from debugger.pytod.core.hunterTrace import hT
4
5  class miClase(object):
6      def __init__(self, y):
7          self.condicion = True
8          self.cantidad = 1
9          self.metodo(self.z, 1, 2, 3)
10         return
11
12     def metodo(self, h, i, j, k):
13         self.cantidad = 1 + h
```

Como se puede observar el programador añadió tres líneas (líneas desde la uno hasta la tres) a su código, tema que se vuelve tedioso al momento de modificar cientos de archivos en los cuales está distribuido su software. Para evitar esto es que se construyó el wrapper, el cual evita que el programador tenga que introducir líneas de código ajenas a su programa.

- Código con wrapper

El programador sólo escribe su programa, sin tener que adaptarlo para que funcione con el depurador.

```
1  class miClase(object):
2      def __init__(self, y):
3          self.condicion = True
4          self.cantidad = 1
5          self.metodo(self.z, 1, 2, 3)
6          return
7
8      def metodo(self, h, i, j, k):
9          self.cantidad = 1 + h
```

A continuación se muestra el código fuente del wrapper:

```
1 import sys
2 from debugger.pytod.core.hunterTrace import hT
3
4 print "PyTOD wrapper v1"
5 if __name__ == '__main__':
6     execfile('\%s\'%(sys.argv[1]),locals(),globals())
```

Wrapper implementa las líneas que en el caso anterior el programador tuvo que agregar a su script, además de esto, wrapper se encarga de ejecutar el script del programador. Wrapper es utilizado en el plugin construido para Eclipse.

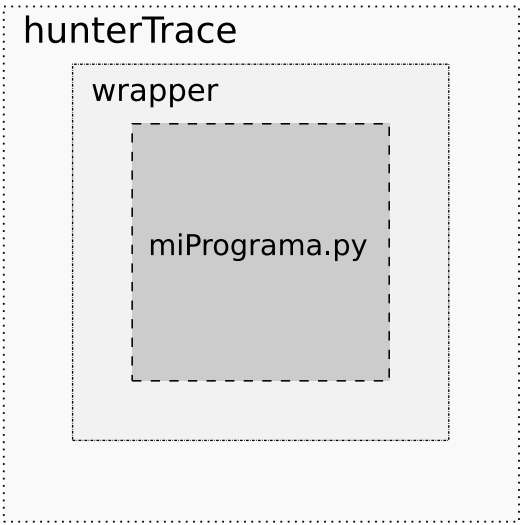


Figura 6.2: Diagrama de composición wrapper

La figura anterior describe de que forma *wrapper* se posiciona en la arquitectura del depurador.

6.2. Diseño de pyTOD

En esta sección se muestra una perspectiva del diseño de pyTOD, separando los componentes entre *Máquina virtual*, *Base de datos* y *Vista final del usuario*

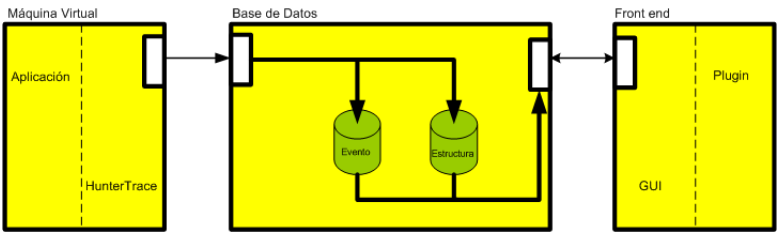


Figura 6.3: Diagrama de diseño de pyTOD

6.3. Implementación de pyTOD

6.3.1. Capturador de huella

El capturador de huella es el encargado de:

- Registrar la estructura del programa objetivo (clases, métodos, funciones, atributos de instancia, atributos de clase, variables locales, threads, probes)
- Capturar los eventos que el programa objetivo genere en tiempo de ejecución (Llamadas a métodos/funciones, asignaciones/modificaciones, instanciaciones, retornos de métodos/funciones, excepciones).

El capturador de huella tiene una dependencia directa con la función *settrace* [20] perteneciente al módulo estándar *sys*. Esta función por definición fue creada para facilitar la creación de depuradores de código.

Por lo anterior, en el presente trabajo de memoria se utilizó esta función para satisfacer ciertos requerimientos de registro estructural del programa objetivo. Es importante señalar que la función *settrace* no cubre todas las necesidades de este trabajo de memoria, ejemplo: no es posible estar enterado de la creación o modificación de una variable de instancia o una variable de clase.

A continuación se muestra un ejemplo a nivel introductorio como esta función trabaja:

- Programación estructurada

```
1  import sys
2
3  def trace(aFrame, aEvent, aArg):
4      if aEvent == 'call':
5          print "Llamada a", aFrame.f_code.co_name
6          return trace
7      elif aEvent == 'line':
8          print "Ejecutando linea", aFrame.f_lineno,
9          return trace
10     elif aEvent == 'return':
11         print "Saliendo de", aFrame.f_code.co_name,
12         print "con valor", aArg
13     elif aEvent == 'exception':
14         pass
15
16     sys.settrace(trace)
17
18     def miFuncion(aPrimero, aSegundo):
19         if aPrimero > aSegundo:
20             theResultado = aPrimero + aSegundo
```

```
21     else:
22         theResultado = aPrimero - aSegundo
23     return theResultado
24
25 if __name__ == '__main__':
26     theResultado = miFuncion(10,5)
27     print "Mi resultado es:", theResultado
```

El resultado obtenido al ejecutar este código es:

```
peregrino:~ minostro$ python miTrace.py
Llamada a miFuncion
Ejecutando linea 20
Ejecutando linea 21
Ejecutando linea 24
Saliendo de miFuncion con valor 15
Mi resultado es: 15
```

■ Programación Orientada a Objetos

```
1  import sys
2
3  def trace(aFrame, aEvent, aArg):
4      if aEvent == 'call':
5          print "Llamada a", aFrame.f_code.co_name
6          return trace
7      elif aEvent == 'line':
8          print "Ejecutando linea", aFrame.f_lineno,
9          print "indice bytecode", aFrame.f_lasti
10         return trace
11     elif aEvent == 'return':
12         print "Saliendo de", aFrame.f_code.co_name,
13         print "con valor", aArg
14     elif aEvent == 'exception':
15         pass
16
17 sys.settrace(trace)
18
19 class miClase(object):
20
21     def __init__(self):
22         pass
23
24     def miMetodo(self, aPrimero, aSegundo):
25         if aPrimero > aSegundo:
26             theResultado = aPrimero + aSegundo
27         else:
28             theResultado = aPrimero - aSegundo
29         return theResultado
30
31 if __name__ == '__main__':
32     theClase = miClase()
33     theResultado = theClase.miMetodo(5, 10)
34     print "Mi resultado es:", theResultado
```

El resultado obtenido al ejecutar este código es:

```
peregrino:~ minostro$ python miTrace.py
Llamada a miClase
Ejecutando linea 18
Ejecutando linea 20
Ejecutando linea 23
Saliendo de miClase con valor {'miMetodo': <function miMetodo at 0x82eb0>, '
__module__': '__main__', '__init__': <function __init__ at 0x82e70>}
Llamada a __init__
Ejecutando linea 21
Saliendo de __init__ con valor None
Llamada a miMetodo
Ejecutando linea 24
Ejecutando linea 27
Ejecutando linea 28
Saliendo de miMetodo con valor -5
Mi resultado es: -5
```

El ejemplo anterior nos muestra que es importante dejar en claro la diferencia entre los sucesos estructurales y los eventos. A continuación se detalla cada uno de éstos, se explica su implementación y ejecución dentro del capturador de huella:

6.3.1.1. Sucesos estructurales

6.3.1.1.1. Registro de clase El objeto clase se registra junto a sus métodos en el momento que el programador termina de definirla. El capturado de huella al detectar el evento *return*, consultará en el diccionario *frame.locals* si es que la función *__init__* está presente como atributo en este diccionario, de ser así, el capturador de huella registrará la definición de la clase. Ejemplo:

```
1 | class miClase(object):
2 |
3 |     def __init__(self):
4 |         pass
5 |
6 |     def miMetodo(self, aPrimero, aSegundo):
7 |         if aPrimero > aSegundo:
8 |             theResultado = aPrimero + aSegundo
9 |         else:
10 |             theResultado = aPrimero - aSegundo
11 |         return theResultado
```

Es importante señalar que la clase se registra cuando existe un evento *return* debido a que Python al momento de finalizar la construcción de la clase, carga todo el diccionario del *frame* en el atributo *aArg* del evento *return* [21]

```
"""
Contenido de aArg
"""
{
    'miMetodo': <function miMetodo at 0x82eb0>,
    '__module__': '__main__',
    '__init__': <function __init__ at 0x82e70>
}
```

Como se puede observar en esta estructura de datos se encuentran todos los métodos de la clase que serán ligadas el objeto en el momento de instanciación. Luego de registrar la clase en el diccionario de la clase *hunterTrace*, se envía un mensaje a través de un socket a la base estructural de TOD para crear un registro relacionado con la clase:

```
Registrando clase miClase
```

Continuando con el registro completo de la clase se guardan los métodos definidos en ella. Los métodos son almacenados en una estructura de datos especial perteneciente a la clase *objectClass*.

Finalmente, se registran los atributos de clase de la misma forma que se registran los métodos. Estos atributos son guardados en una estructura de dato especial de la clase *objectClass*. Este caso se trata con mayor complejidad que el anterior, debido a que la estructura de la base de datos TOD no permite registrar un atributo de clase que esté fuera de un método de la clase, situación que en Python es posible dado que se definen como atributos de la clase a cualquier variable que esté dentro del cuerpo de la definición de la clase y fuera del cuerpo de los métodos definidos para ésta. Bajo esta situación se tomó la decisión de crear en la parte Java de pyTOD un método artificial llamado *classStaticMethod*, al cual se le asociarán todos los atributos de clase que se encuentren en la situación descrita anteriormente.

6.3.1.1.2. Registro de método El registro de método se realiza en el momento que se detecta un evento del tipo *call*. Se busca en el diccionario *locals* del frame la llave *self*, si está en el diccionario, se utiliza para obtener la clase a la cual pertenece el método. Para obtener la clase asociada al método se realiza lo siguiente:

```
type(self).__name__
```

Con este nombre se busca la clase en *hunterTrace* y se obtiene el identificador único del método para poder registrarlo en la base de datos estructural de TOD:

```
Registrando el metodo miMetodo id = 105
```

Es importante señalar que al momento de registrar un método en la base de datos estructural, se deben enviar los argumentos de éste. Por definición Python pone como primer argumento la referencia a la instancia, la forma estándar de nombrarlo es *self*, este argumento no es registrado como argumento en la base de datos estructural.

Los argumentos son registrados utilizando como identificador la posición que ocupan en `frame.f_code.co_varnames`, que por definición son siempre los primeros elementos que contiene esta estructura de datos.

6.3.1.1.3. Registro de método especial De la misma forma que se registra un método, se registra este método especial. Como anteriormente se señala este método artificial se utiliza al momento de registrar un atributo de clase. Este método siempre se encontrará por el nombre *ClassSpecialMethod* siendo *Class* el nombre de la clase a la cual pertenece.

A este método no se le registran argumentos ni variables locales. Este método es registrado en la base de datos estructural de TOD la primera vez que se registra un atributo de clase.

```
Registrando el metodo especial miClaseStaticMethodid = 106
```

6.3.1.1.4. Registro de función En el caso de este registro se busca en *locals* el atributo *self*, de no encontrarse se verifica que el objeto sea una función utilizando la función *isfunction* del módulo *inspect*. De ser una función esta se registra en la base de datos de TOD.

```
Registrando la funcion miFuncion id = 115
```

Respecto a los atributos la única diferencia es que no se tiene que omitir el atributo *self* ya que para funciones este no existe.

6.3.1.1.5. Registro de atributo de instancia La función de sistema *settrace* no entrega ningún mecanismo para estar notificado de la creación de un atributo de instancia. Es por esto que se utiliza el mecanismo llamado `__setattr__` que es un método interno de Python [22] que permite estar notificado en el momento que se asigna un valor o se modifica una variable de instancia. Lo que se hace es sobrescribir este método para todas las clases que el programador haya escrito.

Se muestra un ejemplo de funcionamiento de este método:

```
1 | class Descriptor(object):
2 |     def __setattr__(self, aName, aValue):
3 |         """
4 |         Metodo de interes que permite auditar los movimientos
5 |         de las variables de instancia
6 |         """
```

```

7         if aName in self.__dict__:
8             print "Modificacion de", aName, "por el valor",
              aValue
9         else:
10            print "Registro de", aName, "con el valor", aValue
11            object.__setattr__(self, aName, aValue)
12
13 class miClase(Descriptor):
14
15     def __init__(self):
16         self.itsEdad = 15
17         self.itsSexo = "Masculino"
18         self.itsColor = ("Amarillo", "Verde", "Azul")
19
20     def modificaEdad(self, aEdad):
21         self.itsEdad += aEdad
22
23 if __name__ == "__main__":
24     theClase = miClase()
25     theClase.modificaEdad(20)

```

El resultado de la ejecución es la siguiente:

```

peregrino:~ minostro$ python miDescriptor.py
Registro de itsEdad con el valor 15
Registro de itsSexo con el valor Masculino
Registro de itsColor con el valor ('Amarillo', 'Verde', 'Azul')
Modificacion de itsEdad por el valor 35

```

Como se puede ver funciona todo de forma correcta, pero se detectó un problema de usabilidad de cara al programador. Cada vez que el programador defina una clase deberá heredar de *Descriptor*, cosa que para un par de clases no es molesto pero cuando este número se incrementa puede llegar a ser incomodo para el programador.

Como solución a lo anterior se agrega directamente al diccionario local de la clase escrita por el programador, en el momento de la definición, el método `__setattr__` de la clase *Descriptor*

```

theLocals.update(
    {
        '__setattr__':Descriptor.__dict__['__setattr__']
    }
)

```

Finalmente el atributo de instancia es registrado en la base de datos.

```

h0 - fieldWrite (thread: 101, p.ts: 1217048770743326976, depth: 2, ts:
1217048770744072960, fid: 118 (x), target: UID: 117, val: 6

```

6.3.1.1.6. Registro de atributo de clase Para realizar este registro se deben diferenciar dos situaciones importantes:

1. Si el atributo de clase se encuentra definido en el cuerpo de la clase pero fuera del cuerpo de los métodos, se utilizará el mecanismo de registro señalado para esta situación en el registro de clase 6.3.1.1.1.
2. Si el atributo de clase está definido dentro de un método de la clase, se utilizará el paradigma de metaprogramación.

Se muestra un ejemplo para mostrar su funcionamiento:

- Atributo de clase definido dentro del cuerpo de la clase pero fuera del cuerpo de los métodos

```
1  class MetaDescriptor(type):
2
3      def __setattr__(self, aName, aValue):
4          """
5              Metodo de interes que permite auditar los
6              movimientos
7              de las variables de clase
8              """
9          if aName in self.__dict__:
10              print "Modificacion de", aName, "por el valor",
11                  , aValue
12          else:
13              print "Registro de", aName, "con el valor",
14                  aValue
15              super(MetaDescriptor, self).__setattr__(aName,
16                  aValue)
17
18  class miClase:
19
20      __metaclass__ = MetaDescriptor
21
22      temperatura = 75
23
24      def __init__(self):
25          self.__class__.itsEdad = 73
26          self.__class__.itsColor = "Rojo"
27
28  if __name__ == "__main__":
29      theClase = miClase()
30      miClase.theSexo = "Femenino"
31      miClase.itsEdad += 1
```

El resultado de la ejecución del código es el siguiente:

```
peregrino:~ minostro$ python miMetaDescriptor.py
Registro de temperatura con el valor 75
Registro de itsEdad con el valor 73
Registro de itsColor con el valor Rojo
Registro de theSexo con el valor Femenino
Modificacion de itsEdad por el valor 74
```

- Atributo de clase definido dentro del cuerpo de los métodos

```

1  class MetaDescriptor(type):
2
3      def __setattr__(self, aName, aValue):
4          """
5          Metodo de interes que permite auditar los
6          movimientos
7          de las variables de clase
8          """
9          if aName in self.__dict__:
10             print "Modificacion de", aName, "por el valor"
11                 , aValue
12         else:
13             print "Registro de", aName, "con el valor",
14                 aValue
15             super(MetaDescriptor, self).__setattr__(aName,
16                 aValue)
17
18  class miClase:
19
20      __metaclass__ = MetaDescriptor
21
22      def __init__(self):
23          self.__class__.itsEdad = 73
24          self.__class__.itsColor = "Rojo"
25
26  if __name__ == "__main__":
27      theClase = miClase()
28      miClase.theSexo = "Femenino"
29      miClase.itsEdad += 1

```

El resultado de la ejecución del código es el siguiente:

```

peregrino:~ minostro$ python miMetaDescriptor.py
Registro de itsEdad con el valor 73
Registro de itsColor con el valor Rojo
Registro de theSexo con el valor Femenino
Modificacion de itsEdad por el valor 74

```

Al igual que en el caso del registro de atributos de instancia es incomodo para el programador personalizar su clase para que pueda comportarse de la manera que se desea, es por esto que en el momento de definir la clase se agrega al diccionario locals el atributo `__metaclass__` con valor `MetaDescriptor`.

```

theLocals.update(
    {
        '__metaclass__':MetaDescriptor
    }
)

```

Es importante que para ambas situaciones se logra registrar el atributo de clase en la base estructural de TOD.

Registrando un atributo estatico con id 108

6.3.1.1.7. Registro de variable local El registro de una variable local puede realizarse en dos situaciones:

1. Si el programador escribe la instrucción *return* en el método o en la función, todos los registros de las variables locales se realizarán cuando suceda el evento *line*.
2. Si el programador no escribe la instrucción *return*, en el método o en la función, los registros de las variables locales se realizarán cuando sucedan los eventos *line* y *return* [21].

En ambos casos se asegura el registro de la variable local en la base de datos estructural de TOD.

```
1 | Registrando variable local: y
```

La forma de capturar la creación de una variable local, es inspeccionando el bytecode de Python en búsqueda de la instrucción *STORE_FAST*. [21]

Es importante señalar que al momento de registrar el método o la función se crea un *lnotab* [23] propio, que es una estructura de datos que indica los rangos de los índices del bytecode, para luego ser consultados al momento de inspeccionar el bytecode. Esto se realiza para hacer un poco más eficiente el proceso de búsqueda dentro del bytecode.

Se muestra un ejemplo introductorio para graficar la forma en que se registran los variables locales.

```
1 | import sys
2 | import dis
3 |
4 | class miTrace:
5 |     def __init__(self):
6 |         self.itsLnotab = None
7 |
8 |     def __createlnnotab__(self, aCode):
9 |         theLnotab = {}
10 |         if hasattr(aCode, 'co_lnotab'):
11 |             table = aCode.co_lnotab
12 |             index = 0
13 |             last_index = None
14 |             for i in range(0, len(table), 2):
15 |                 index = index + ord(table[i])
16 |                 if last_index == None:
```

```

17         last_index = index
18     else:
19         theLnotab.update({index:tuple([last_index,
20                                     index-1])})
21         last_index = index
22     theLnotab.update(
23     {
24         len(aCode.co_code)-1:tuple(
25             [last_index,len(aCode.co_code)-1])
26     })
27     return theLnotab
28
29 def __getpartcode__(self, aCode, aLimits):
30     theLower = aLimits[0]
31     theUpper = aLimits[1]
32     theCode = aCode.co_code
33     theStoreFast = {}
34     while theLower < theUpper:
35         theOp = ord(theCode[theLower])
36         theNameOp = dis.opname[theOp]
37         theLower = theLower + 1
38         if theOp >= dis.HAVE_ARGUMENT:
39             theValue = ord(theCode[theLower])
40             theValue += ord(theCode[theLower+1])*256
41             theLower = theLower + 2
42             if theOp in dis.haslocal and \
43                 theNameOp == 'STORE_FAST':
44                 theArgumentValue = aCode.co_varnames[
45                     theValue]
46                 theStoreFast.update({theArgumentValue:
47                                     theValue})
48     return theStoreFast
49
50 def trace(self, aFrame, aEvent, aArg):
51     theCode = aFrame.f_code
52     if aEvent == 'call':
53         print "Llamada a", aFrame.f_code.co_name
54         print "Byte Code asociado:"
55         dis.disassemble(theCode)
56         print "1. Generando lnotab = ",
57         self.itsLnotab = self.__createlnnotab__(theCode)
58         print self.itsLnotab
59         print "2. Comienza inspeccion"
60         return self.trace
61     elif aEvent == 'line':
62         if self.itsLnotab.has_key(aFrame.f_lasti):
63             theByteCodeLocals = self.__getpartcode__(
64                 theCode,self.itsLnotab[aFrame.f_lasti])
65             print "[line] Examinando rango:", self.
66                 itsLnotab[aFrame.f_lasti],
67             print ", resultado de inspeccion:",
68                 theByteCodeLocals
69             return self.trace
70     elif aEvent == 'return':
71         if self.itsLnotab.has_key(aFrame.f_lasti):
72             theByteCodeLocals = self.__getpartcode__(
73                 theCode,self.itsLnotab[aFrame.f_lasti])
74             print "[return] Examinando rango:", self.
75                 itsLnotab[aFrame.f_lasti],

```



```
68         print ", resultado de inspeccion:",
           theByteCodeLocals
69         print "Saliendo de", aFrame.f_code.co_name,
70         print "con valor", aArg
71     elif aEvent == 'exception':
72         pass
73
74 sys.settrace(miTrace().trace)
```

Código donde el programador escribe la instrucción *return*:

```
1 | def miFuncion():
2 |     thePrimera = 10
3 |     theSegunda = 35
4 |     return theSegunda
5 |
6 | if __name__ == '__main__':
7 |     miFuncion()
```

El resultado de la ejecución de este código es el siguiente:

```
peregrino:~ minostro$ python miCapturadorLocales.py
Llamada a miFuncion

Byte Code asociado:
85          0 LOAD_CONST          1 (10)
           3 STORE_FAST           0 (thePrimera)

86          6 LOAD_CONST          2 (35)
           9 STORE_FAST           1 (theSegunda)

87         12 LOAD_FAST           1 (theSegunda)
          15 RETURN_VALUE

1. Generando lnotab = {12: (6, 11), 6: (0, 5), 15: (12, 15)}
2. Comienza inspeccion
  [line] Examinando rango: (0, 5) , resultado de inspeccion: {'thePrimera': 0}
  [line] Examinando rango: (6, 11) , resultado de inspeccion: {'theSegunda': 1}
  [return] Examinando rango: (12, 15) , resultado de inspeccion: {}

Saliendo de miFuncion con valor 35
```

Código donde el programador no escribe la instrucción *return*:

```
1 | def miFuncion():
2 |     thePrimera = 10
3 |     theSegunda = 35
4 |
5 | if __name__ == '__main__':
6 |     miFuncion()
```

El resultado de la ejecución de este código es el siguiente:

```
peregrino:~ minostro$ python miCapturadorLocales.py
Llamada a miFuncion

Byte Code asociado:
85          0 LOAD_CONST          1 (10)
           3 STORE_FAST           0 (thePrimera)

86          6 LOAD_CONST          2 (35)
           9 STORE_FAST           1 (theSegunda)
          12 LOAD_CONST          0 (None)
```

```
15 RETURN_VALUE

1. Generando lnotab = {6: (0, 5), 15: (6, 15)}
2. Comienza inspeccion
  [line] Examinando rango: (0, 5) , resultado de inspeccion: {'thePrimera': 0}
  [return] Examinando rango: (6, 15) , resultado de inspeccion: {'theSegunda': 1}

Saliendo de miFuncion con valor None
```

El método `__getpartcode__` entrega las variables locales que han sido definidas en un trozo de bytecode determinado, las que luego en este ejemplo su muestran por salida estándar.

6.3.1.1.8. Registro de thread El registro de thread se realiza al momento de entrar a *settrace*. Para conocer la identificación de thread se utiliza el método *get_ident* del módulo *thread*. Si se detecta que el thread no ha sido registrado en la estructura interna de *hunterTrace*, este se registra y se envía a la base de datos estructural de TOD.

```
Registrando thread id 101
```

6.3.1.1.9. Registro de probe El registro de probe ocurre en cualquier momento antes que se produzca un evento. Probe se comporta como una sonda la cual nos indica en que método o en que función sucedió el evento, el índice del bytecode y número de linea del código donde se encuentra la instrucción del evento ocurrido.

Por ejemplo para una asignación de atributo de instancia se tendrá el siguiente registro de probe en la base de datos estructural de TOD:

```
Registrando probe id 116
```

6.3.1.1.10. Registro de objeto El registro de objeto ocurre en cualquier momento antes que se produzca un evento que involucre la utilización de:

- argumentos de funciones o métodos
- variables locales
- atributos de instancia o atributos de clase.

Este registro se realiza para optimizar el paquete de datos que se envía a través del socket cuando ocurre un evento.

El registro de objeto consiste en tomar el valor correspondiente del objeto (señalado anteriormente), asignarle un identificador único el cual es generado por la función *id*

de Python y registrar el objeto en la base de datos estructural de TOD con su valor original.

```
Registrando un nuevo objeto UID 10956000 valor "integer division or modulo by zero"
```

Al momento de utilizar el valor de uno de estos objetos en un evento sólo se envía el identificador del objeto y no su valor original.

```
h0 - exception (thread: 101, p.ts: 1217118691038720000, depth: 3, ts: 1217118691041134080, pid: 116, exc.: UID: 10956000)
```

6.3.1.2. Eventos

6.3.1.2.1. Llamada método/función Este evento ocurre cuando el capturador de huella detecta el evento *call*. Se envía a la base de datos de eventos la información que identifica a esta llamada de método o de función.

```
h0 - methodCall (thread: 101, p.ts: 1217207598479134208, depth: 2, ts: 1217207598480425984, direct: true, c.bid: -1, e.bid: 104 (miClase.miMetodo), target: UID: 117, args: null)
```

A continuación se muestran dos ejemplos en los cuales las llamadas son capturadas:

■ Llamada a función

```
1 import sys
2
3 def trace(aFrame, aEvent, aArg):
4     if aEvent == 'call':
5         print "Llamada a", aFrame.f_code.co_name
6         return trace
7     elif aEvent == 'line':
8         return trace
9     elif aEvent == 'return':
10        pass
11    elif aEvent == 'exception':
12        pass
13
14 sys.settrace(trace)
15
16 def miFuncion(aPrimero, aSegundo):
17     if aPrimero > aSegundo:
18         theResultado = aPrimero + aSegundo
19     else:
20         theResultado = aPrimero - aSegundo
21     return theResultado
22
23 if __name__ == '__main__':
24     theResultado = miFuncion(10,5)
25     print "Mi resultado es:", theResultado
```

El resultado de la ejecución de este código es:

```
peregrino:~ minostro$ python miLlamadaFuncion.py
Llamada a miFuncion
Mi resultado es: 15
```

■ Llamada a método

```
1  import sys
2
3  def trace(aFrame, aEvent, aArg):
4      if aEvent == 'call':
5          print "Llamada a", aFrame.f_code.co_name
6          return trace
7      elif aEvent == 'line':
8          return trace
9      elif aEvent == 'return':
10         pass
11     elif aEvent == 'exception':
12         pass
13
14 sys.settrace(trace)
15
16 class miClase(object):
17
18     def __init__(self):
19         pass
20
21     def miMetodo(self, aPrimero, aSegundo):
22         if aPrimero > aSegundo:
23             theResultado = aPrimero + aSegundo
24         else:
25             theResultado = aPrimero - aSegundo
26         return theResultado
27
28 if __name__ == '__main__':
29     theClase = miClase()
30     theResultado = theClase.miMetodo(5, 10)
31     print "Mi resultado es:", theResultado
```

El resultado de la ejecución de este código es:

```
peregrino:~ minostro$ python miLlamadaMetodo.py
Llamada a miClase
Llamada a __init__
Llamada a miMetodo
Mi resultado es: -5
```

Es importante señalar que *miClase* no es un método sino que un objeto del tipo *class*, asunto que para este ejemplo no se discrimina para no complicar innecesariamente el código de la implementación de *trace*.

6.3.1.2.2. Asignación/modificación

6.3.1.2.2.1. Atributo de instancia De igual forma que en el registro de atributo de instancia 6.3.1.1.5, se utiliza el método `__setattr__` para estar notificados de todos los cambios de estos atributos. En el momento que el programador modifique el valor de un atributo de instancia el capturador de huella llamará al método `__addAttribute__` de la estructura de datos interna *itsClass*, la cual se encarga de manejar los cambios en los atributos de instancia.

```

1 class Descriptor(object):
2     def __setattr__(self, aName, aValue):
3         if aName in self.__dict__:
4             print "Modificacion de", aName, "por el valor",
              aValue
5         else:
6             print "Registro de", aName, "con el valor", aValue
7             object.__setattr__(self, aName, aValue)
8
9 class miClase(Descriptor):
10
11     def __init__(self):
12         self.itsEdad = 15
13         self.itsSexo = "Masculino"
14         self.itsColor = ("Amarillo", "Verde", "Azul")
15
16     def modificaEdad(self, aEdad):
17         self.itsEdad += aEdad
18
19     def modificaSexo(self, aSexo):
20         self.itsSexo = aSexo
21
22 if __name__ == "__main__":
23     theClase = miClase()
24     theClase.modificaEdad(20)
25     theClase.modificaSexo("Femenino")

```

El resultado de la ejecución de este código es:

```

peregrino:~ minostro$ python miModificacionAtributoInstancia.py
Registro de itsEdad con el valor 15
Registro de itsSexo con el valor Masculino
Registro de itsColor con el valor ('Amarillo', 'Verde', 'Azul')
Modificacion de itsEdad por el valor 35
Modificacion de itsSexo por el valor Femenino

```

Por simplicidad no se ha utilizado el método `__addAttribute__` y la estructura *itsClass*, pero el ejemplo presentado anteriormente grafica con toda claridad lo que se realiza al momento de detectar la modificación de una variable de instancia.

Finalmente, se registra la modificación del valor en la base de datos de eventos de TOD.

```

h0 - fieldWrite (thread: 101, p.ts: 1217207598470598912, depth: 2, ts:
1217207598471888896, fid: 118 (itsEdad), target: UID: 117, val: 6

```

6.3.1.2.2.2. Atributo de clase De la misma forma que en el registro de los atributos de clase (incluyendo los dos casos) 6.3.1.1.5, se realiza el registro de modificación de éstos. En el momento que el programador modifique el valor de un atributo de clase el capturador de huella llamará al método `__addStaticAttribute__` de la estructura de datos interna `itsClass`, la cual se encarga de manejar los cambios en los atributos de clase.

```

1  class MetaDescriptor(type):
2      def __setattr__(self, aName, aValue):
3          if aName in self.__dict__:
4              print "Modificacion de", aName, "por el valor",
5                  aValue
6          else:
7              print "Registro de", aName, "con el valor", aValue
8              super(MetaDescriptor, self).__setattr__(aName, aValue)
9
10 class miClase:
11
12     __metaclass__ = MetaDescriptor
13
14     def __init__(self):
15         self.__class__.itsEdad = 73
16         self.__class__.itsColor = "Rojo"
17
18
19 if __name__ == "__main__":
20     theClase = miClase()
21     miClase.theSexo = "Femenino"
22     miClase.itsEdad += 1
23     miClase.itsColor = "Azul"

```

El resultado de la ejecución de este código es:

```

peregrino:~ minostro$ python miModificacionAtributoClase.py
Registro de itsEdad con el valor 73
Registro de itsColor con el valor Rojo
Registro de theSexo con el valor Femenino
Modificacion de itsEdad por el valor 74
Modificacion de itsColor por el valor Azul

```

Por simplicidad no se ha utilizado el método `__addStaticAttribute__` y la estructura `itsClass`, pero el ejemplo presentado anteriormente grafica con toda claridad lo que se realiza al momento de detectar la modificación de una variable de clase.

Finalmente, se registra la modificación del valor en la base de datos de eventos de TOD.

```

h0 - fieldWrite (thread: 101, p.ts: 1217207598468965888, depth: 1, ts:
1217207598469940992, fid: 115 (itsEdad), target: null, val: 6

```

Aparentemente este registro es igual al de los atributos de instancia, pero la dife-

rencia está en que este registro tiene su *target* con valor *null*

6.3.1.2.2.3. Variable local El registro de la modificación del valor de una variable local se realiza de la misma forma que el registro de la misma 6.3.1.1.7.

Al momento de detectar un evento del tipo *line* el método del capturador de huella `__localWrite__` se encarga de buscar a que objeto (método/función) pertenece la variable local y luego envía la información necesaria para crear el registro asociado a esta modificación en la base de datos de eventos de TOD.

```
h0 - localWrite (thread: 101, p.ts: 1217207598474802944, depth: 3, ts:
1217207598477009920, vid: 4, val: 3)
```

A continuación se muestra como se realiza la captura de la modificación del valor de una variable local:

```
1 | import sys
2 | import dis
3 |
4 | class miTrace:
5 |     def __init__(self):
6 |         self.itsLnotab = None
7 |         self.itsLocals = {}
8 |
9 |     def __createlnnotab__(self, aCode):
10 |         theLnotab = {}
11 |         if hasattr(aCode, 'co_lnotab'):
12 |             table = aCode.co_lnotab
13 |             index = 0
14 |             last_index = None
15 |             for i in range(0, len(table), 2):
16 |                 index = index + ord(table[i])
17 |                 if last_index == None:
18 |                     last_index = index
19 |                 else:
20 |                     theLnotab.update({index:tuple([last_index,
21 |                                                     index-1])})
22 |                     last_index = index
23 |                 theLnotab.update(
24 |                     {
25 |                         len(aCode.co_code)-1:tuple(
26 |                             [last_index,len(
27 |                                 aCode.co_code)
28 |                                 -1]
29 |                         )
30 |                     })
31 |         return theLnotab
32 |
33 |     def __getpartcode__(self, aCode, aLimits):
34 |         theLower = aLimits[0]
35 |         theUpper = aLimits[1]
36 |         theCode = aCode.co_code
37 |         theStoreFast = {}
```

```

35         while theLower < theUpper:
36             theOp = ord(theCode[theLower])
37             theNameOp = dis.opname[theOp]
38             theLower = theLower + 1
39             if theOp >= dis.HAVE_ARGUMENT:
40                 theValue = ord(theCode[theLower]) + ord(theCode
41                     [theLower+1])*256
42                 theLower = theLower + 2
43                 if theOp in dis.haslocal and theNameOp == '
44                     STORE_FAST':
45                     theArgumentValue = aCode.co_varnames[
46                         theValue]
47                     theStoreFast.update({theArgumentValue:
48                         theValue})
49             return theStoreFast
50
51     def __localWrite__(self, aByteCodeLocals, aLocals):
52         for theKey, theValue in aByteCodeLocals.iteritems():
53             print "Modificando variable local", theKey,
54             print "con el valor", aLocals[theKey]
55
56     def __registerLocals__(self, aByteCodeLocals):
57         for theKey, theValue in aByteCodeLocals.iteritems():
58             if not self.itsLocals.has_key(theKey):
59                 print "Registrando variable local", theKey
60                 self.itsLocals.update({theKey:theValue})
61
62     def trace(self, aFrame, aEvent, aArg):
63         theCode = aFrame.f_code
64         if aEvent == 'call':
65             print "Llamada a", aFrame.f_code.co_name
66             print "Byte Code asociado:"
67             dis.disassemble(theCode)
68             self.itsLnotab = self.__createLnotab__(theCode)
69             return self.trace
70         elif aEvent == 'line':
71             if self.itsLnotab.has_key(aFrame.f_lasti):
72                 theByteCodeLocals = self.__getpartcode__(
73                     theCode,
74                     self.itsLnotab[
75                         aFrame.
76                         f_lasti]
77                     )
78                 self.__registerLocals__(theByteCodeLocals)
79                 self.__localWrite__(theByteCodeLocals, aFrame.
80                     f_locals)
81                 return self.trace
82         elif aEvent == 'return':
83             if self.itsLnotab.has_key(aFrame.f_lasti):
84                 theByteCodeLocals = self.__getpartcode__(
85                     theCode,
86                     self.itsLnotab[
87                         aFrame.
88                         f_lasti]
89                     )
90                 self.__registerLocals__(theByteCodeLocals)
91                 self.__localWrite__(theByteCodeLocals, aFrame.
92                     f_locals)
93                 print "Saliendo de", aFrame.f_code.co_name,

```



```
84         print "con valor", aArg
85     elif aEvent == 'exception':
86         pass
87
88
89 sys.settrace(miTrace().trace)
90
91 def miFuncion():
92     thePrimera = 10
93     theSegunda = 35
94     thePrimera = 24
95     theSegunda = 54
96     return theSegunda
97
98 if __name__ == '__main__':
99     miFuncion()
```

Al ejecutar este código da como resultado:

Llamada a miFuncion

Byte Code asociado:

100	0 LOAD_CONST	1 (10)
	3 STORE_FAST	0 (thePrimera)
101	6 LOAD_CONST	2 (35)
	9 STORE_FAST	1 (theSegunda)
102	12 LOAD_CONST	3 (24)
	15 STORE_FAST	0 (thePrimera)
103	18 LOAD_CONST	4 (54)
	21 STORE_FAST	1 (theSegunda)
104	24 LOAD_FAST	1 (theSegunda)
	27 RETURN_VALUE	

Registrando variable local thePrimera

Modificando variable local thePrimera con el valor 10

Registrando variable local theSegunda

Modificando variable local theSegunda con el valor 35

Modificando variable local thePrimera con el valor 24

Modificando variable local theSegunda con el valor 54

Saliendo de miFuncion con valor 54

6.3.1.2.3. Instanciación El evento de instanciación ocurre en el momento que el programador crea un objeto de instancia (instancia de una clase). En el evento de tipo *call* el capturador de huella consulta por el método `__init__`, si es la llamada de este método, se registra la instanciación de clase en la base de datos de eventos TOD.

```
h0 - instantiation(thread: 101, p.ts: 0, depth: 1, ts: 1217207598470598912, direct:
      true, c.bid: -1, e.bid: 105 (miClase.__init__), target: UID: 117, args: [Ljava.
      lang.Object;@8a2006])
```

6.3.1.2.4. Retorno de método/función El evento de retorno de método o función sucede cuando ocurre un evento del tipo *return* dentro del capturador de huella. El evento de retorno ocurre independientemente si el programador escribe la instrucción *return* dentro del cuerpo del método/función.

```
1 import sys
2
3 def trace(aFrame, aEvent, aArg):
4     if aEvent == 'call':
5         return trace
6     elif aEvent == 'line':
7         return trace
8     elif aEvent == 'return':
9         print "Saliendo de", aFrame.f_code.co_name,
10        print "con valor", aArg
11    elif aEvent == 'exception':
12        pass
13
14 sys.settrace(trace)
15
16 def miFuncion(aPrimero, aSegundo):
17     if aPrimero > aSegundo:
18         theResultado = aPrimero + aSegundo
19     else:
20         theResultado = aPrimero - aSegundo
21     return theResultado
22
23 if __name__ == '__main__':
24     theResultado = miFuncion(10,5)
25     print "Mi resultado es:", theResultado
```

El resultado de la ejecución de este código es:

```
Saliendo de miFuncion con valor 15
Mi resultado es: 15
```

Finalmente se registra la salida del método/función en la base de datos de eventos de TOD.

```
h0 - behaviorExit (thread: 101, p.ts: 1217207598474802944, depth: 3, ts:
1217207598478802944, bid: 102 (miClase.miMetodo), thrown: false, ret: 3)
```

6.3.1.2.5. Excepción El evento excepción es registrado por el capturador de huella cuando este detecta el evento del tipo *exception*. Es importante señalar que existen dos situaciones en donde puede ocurrir una excepción:

- Excepción sin bloque *try/except*

Cuando ocurre una excepción en el código del programador, ésta debe ser propagada por todos los niveles de la pila de frames. En este caso, como el programador no ha manejado la excepción, propagarla por todos los niveles de la pila es lo más sensato.

```
1 def miPrimeraFuncion():
```

```
2         miSegundaFuncion( )
3
4     def miSegundaFuncion( ):
5         miTerceraFuncion( )
6
7     def miTerceraFuncion( ):
8         miCuartaFuncion( )
9
10    def miCuartaFuncion( ):
11        y = 1/0
12
13    if __name__ == '__main__':
14        miPrimeraFuncion( )
```

El resultado de la ejecución de este código es:

```
Saliendo de miCuartaFuncion con excepcion integer division or modulo by zero
Saliendo de miTerceraFuncion con excepcion integer division or modulo by zero
Saliendo de miSegundaFuncion con excepcion integer division or modulo by zero
Saliendo de miPrimeraFuncion con excepcion integer division or modulo by zero
Traceback (most recent call last):
  File "/Volumes/archivos/eclipse/workspace/python-project/src/testcase/
memoryTrace.py", line 49, in <module>
    miPrimeraFuncion()
  File "/Volumes/archivos/eclipse/workspace/python-project/src/testcase/
memoryTrace.py", line 37, in miPrimeraFuncion
    miSegundaFuncion()
  File "/Volumes/archivos/eclipse/workspace/python-project/src/testcase/
memoryTrace.py", line 40, in miSegundaFuncion
    miTerceraFuncion()
  File "/Volumes/archivos/eclipse/workspace/python-project/src/testcase/
memoryTrace.py", line 43, in miTerceraFuncion
    miCuartaFuncion()
  File "/Volumes/archivos/eclipse/workspace/python-project/src/testcase/
memoryTrace.py", line 46, in miCuartaFuncion
    y = 1/0
ZeroDivisionError: integer division or modulo by zero
```

■ Excepción con bloque *try/except*

En este caso no tiene sentido propagar la excepción por todos los niveles de la pila de frame, ya que el programador ha manejado correctamente la excepción, esto implica que sólo debe propagarse hasta el nivel en donde se ha realizado el manejo de excepción.

```
1     def miPrimeraFuncion( ):
2         try:
3             miSegundaFuncion( )
4         except:
5             y = 5
6         return y
7
8     def miSegundaFuncion( ):
9         miTerceraFuncion( )
10
11    def miTerceraFuncion( ):
12        miCuartaFuncion( )
13
14    def miCuartaFuncion( ):
15        y = 1/0
```

```
16
17 if __name__ == '__main__':
18     miPrimeraFuncion()
```

El resultado de la ejecución de este código es:

```
Saliendo de miCuartaFuncion con excepcion integer division or modulo by zero
Saliendo de miTerceraFuncion con excepcion integer division or modulo by zero
Saliendo de miSegundaFuncion con excepcion integer division or modulo by zero
Saliendo de miPrimeraFuncion con valor 5
```

Los casos anteriores son implementados de la siguiente manera en el capturador de huellas:

```
1 import sys
2 import dis
3
4 class miTrace(object):
5     def __init__(self):
6         self.FLAG_THROWN = False
7
8     def trace(self, aFrame, aEvent, aArg):
9         theCode = aFrame.f_code
10        if aEvent == 'call':
11            return self.trace
12        elif aEvent == 'line':
13            return self.trace
14        elif aEvent == 'return':
15            if self.FLAG_THROWN == True:
16                self.FLAG_THROWN = False
17            return
18            print "Saliendo de", aFrame.f_code.co_name,
19            print "con valor", aArg
20        elif aEvent == 'exception':
21            for theTuple in dis.findlinestarts(theCode):
22                if aFrame.f_lineno in theTuple:
23                    theIndex = theTuple[0]
24                    theOp = ord(theCode.co_code[theIndex-3])
25                    theInstruction = dis.opname[theOp]
26                    if theInstruction == 'SETUP_EXCEPT':
27                        return self.trace
28                    print "Saliendo de",aFrame.f_code.co_name,
29                    print "con excepcion", aArg[1]
30                    self.FLAG_THROWN = True
31
32    sys.settrace(miTrace().trace)
```

Finalmente, la excepción es registrada en la base de datos de eventos de TOD.

```
h0 - exception      (thread: 101, p.ts: 1217207598480425984, depth: 3, ts:
1217207598480540928, pid: 115, exc.: UID: 10955936)
```

6.3.2. Protocolo Comunicación de pyTOD

Se utilizó sockets para comunicar pyTOD con TOD. Se confeccionaron mensajes para cada tipo de registro y para cada tipo de evento que se genera en el programa objetivo.

Es importante señalar que se utilizó la librería *XDRLib* [24] de python, basada en el estándar *xdrlib*[25], para construir el protocolo de comunicación entre el lenguaje de programación Python y Java.

A continuación se detalla el protocolo de comunicación creado.

6.3.2.1. Identificadores

6.3.2.1.1. Sucesos La siguiente tabla muestra que cada suceso tiene un identificador en el sistema de captación de huella.

Suceso	Identificador
Registro	0
Llamada	1
Asignación	2
Retorno	3
Instanciación	4

Tabla 6.1: Identificadores de sucesos

6.3.2.1.2. Objetos La siguiente tabla muestra que cada objeto tiene un identificador en el sistema de captura de huella.

Id Objeto	Identificador
Clase	0
Método	1
Atributo	2
Función	3
Variable local	4
Probe	5
Thread	6
Atributo de clase	7
Objeto	8
Excepción	9
Método estático	10

Tabla 6.2: Identificadores de objetos

6.3.2.1.3. Tipo de datos La siguiente tabla muestra que cada tipo de datos tiene un identificador en el sistema de captura de huella.

Tipo	Identificador
int	0
str	1
float	2
long	3
bool	4
tuple	5
list	6
dict	7

Tabla 6.3: Identificadores de tipo de datos

6.3.2.2. Registro de objetos

A continuación se muestra el formato que tienen el registro de los diferentes objetos dentro del capturador de huellas:

6.3.2.2.1. Función Se describe el registro del objeto función:

eventId	objectId	functionId	functionName	argsCount	{ argName;	argId;}	fileName
int	int	int	str	int	str	int	str

Tabla 6.4: Registro del objeto función

6.3.2.2.2. Variable local Se describe el registro del objeto variable local:

eventId	objectId	localId	parentId	localName
int	int	int	int	str

Tabla 6.5: Registro del objeto variable local

6.3.2.2.3. Clase Se describe el registro del objeto clase:

eventId	objectId	classId	className	classBases
int	int	int	str	— ¹

Tabla 6.6: Registro del objeto clase

¹No se registran las super clases que pueda tener la clase.

6.3.2.2.4. Método Se describe el registro del objeto método:

eventId	objectId	methodId	classId	methodName	argsCount	{argName <i>i</i>	argId <i>i</i> }	fileName
int	int	int	int	str	int	str	int	str

Tabla 6.7: Registro del objeto método

6.3.2.2.5. Atributo de instancia Se describe el registro del objeto atributo de instancia:

eventId	objectId	attributeId	parentId	attributeName
int	int	int	int	str

Tabla 6.8: Registro del objeto atributo de instancia

6.3.2.2.6. Atributo de clase Se describe el registro del objeto atributo de clase:

eventId	objectId	attributeId	parentId	attributeName
int	int	int	int	str

Tabla 6.9: Registro del objeto atributo de clase

6.3.2.2.7. Thread Se describe el registro del objeto thread:

eventId	objectId	threadId	sysId
int	int	int	int

Tabla 6.10: Registro del objeto thread

6.3.2.2.8. Probe Se describe el registro del objeto probe:

eventId	objectId	Id	parentId	currentLasti	currentLineno
int	int	int	int	int	int

Tabla 6.11: Registro del objeto probe

6.3.2.2.9. Objeto Se describe el registro del objeto objeto:

eventId	objectId	typeId	Id	currentTimestamp
int	int	int	double	double

Tabla 6.12: Registro del objeto objeto

6.3.2.2.10. Excepción Se describe el registro del objeto excepción:

eventId	objectId	typeId	theValue
int	int	int	value or valueId ¹

Tabla 6.13: Registro del objeto excepción

6.3.2.3. Llamada de objetos

A continuación se muestra el formato que tienen las llamadas de los objetos función y método dentro del capturador de huellas:

6.3.2.3.1. Función Se describe la llamada al objeto función:

eventId	objectId	functionId	argsCount	{ typeId _{<i>i</i>}	argValue _{<i>i</i>} }
int	int	int	int	int	value or valueId ¹

Tabla 6.14: Llamada al objeto función

6.3.2.3.2. Método Se describe la llamada al objeto método:

eventId	objectId	methodId	targetId	argsCount	{ typeId _{<i>i</i>}	argValue _{<i>i</i>} }
int	int	int	int	int	int	value or valueId ¹

Tabla 6.15: Llamada al objeto método

Es importante señalar que todas estas llamadas estan acompañadas de los siguientes datos que se describen a continuación:

¹Si el objeto es un entero o un booleano se pasa por valor, de caso contrario se pasa el id del objeto registrado anteriormente

probeId	parentTimeStampFrame	depth	currentTimeStamp	threadId
int	double	int	double	int

Tabla 6.16: Coordenadas

6.3.2.4. Instanciación de clase

A continuación se muestra el formato que tiene la instanciación de clase dentro del capturador de huellas:

eventId	behaviorId	targetId	argsCount	{typeId _{<i>i</i>}	argValue _{<i>i</i>} }
int	int	int	int	int	value or valueId ¹

Tabla 6.17: Instanciación de clase

Es importante señalar que la instanciación está acompañada de los siguientes datos

Tabla 6.16.

6.3.2.5. Asignación - Modificación de objetos

A continuación se muestra el formato que tienen las asignaciones/modificaciones de los objetos variable local, atributo de instancia y atributo de clase dentro del capturador de huellas:

6.3.2.5.1. Variable local Se describe la asignación/modificación al objeto variable local:

eventId	objectId	localId	typeId	value
int	int	int	int	value or valueId ¹

Tabla 6.18: Registro del objeto variable local

6.3.2.5.2. Atributo de instancia Se describe la asignación/modificación al objeto atributo de instancia:

¹Si el objeto es un entero o un booleano se pasa por valor, de caso contrario se pasa el id del objeto registrado anteriormente

eventId	objectId	attributeId	targetId	typeId	value
int	int	int	int	int	value or valueId ¹

Tabla 6.19: Registro del objeto atributo de instancia

6.3.2.5.3. Atributo de clase Se describe la asignación/modificación al objeto atributo de clase:

eventId	objectId	staticFieldId	typeId	value
int	int	int	int	value or valueId ¹

Tabla 6.20: Registro del objeto atributo de clase

Es importante señalar que todas estas asignaciones/modificaciones están acompañadas de los siguientes datos Tabla 6.16.

6.3.2.6. Return

A continuación se muestra el formato que tiene el return dentro del capturador de huellas:

Se describe return:

eventId	behaviorId	typeId	value	hasThrown	probeId
int	int	int	value or valueId ¹	bool	int

Tabla 6.21: Registro de return

Es importante señalar return está acompañado de los siguientes datos 6.16.

6.3.3. Estructuras y Clases de pyTOD

6.3.3.1. Estructura

Para mostrar la estructura de pyTOD se utiliza un diagrama de clases resumido en el cual sólo se muestran las relaciones entre las clases, métodos y atributos más importantes de esta. Lo anterior con el fin de mostrar una estructura clara y concisa. En la sección posterior se individualizará cada una de las clases aquí mostradas.

¹Si el objeto es un entero o un booleano se pasa por valor, de caso contrario se pasa el id del objeto registrado anteriormente

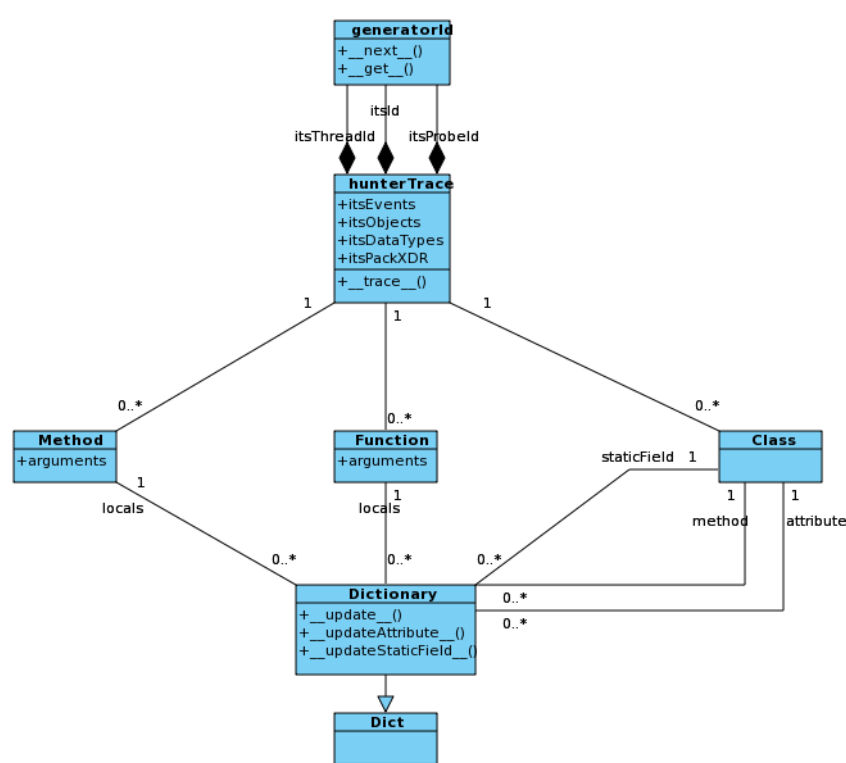


Figura 6.4: Diagrama de clase del capturador de huella

6.3.3.2. Clases

En esta sección se muestran todos los atributos y métodos de las clases que inter-vienen en pyTOD. Se explica cual es su uso y se explica sus métodos y métodos más importantes.

6.3.3.2.1. hunterTrace Esta es la clase principal de pyTOD ya que contiene refe-rencias a las demás clases y es la clase que contiene al método `__trace__`

6.3.3.2.1.1. Atributos Entre los atributos más importantes se tienen:

- **itsClass** : Diccionario que contiene a todas las clases escritas por el programador.
- **itsFunction** : Diccionario que contiene a todas las funciones escritas por el pro-gramador.
- **itsMethod** : Diccionario que contiene a todos los métodos escritos por el progra-mador.
- **itsProbe** : Diccionario que almacena todos los probes que se generan en el de-purador. El objetivo de cada probe es identificar un lugar específico dentro del código depurado en donde ha ocurrido un evento.
- **itsEvent** : Diccionario que contiene todos los eventos que suceden dentro del depurador:

- register
- call
- return
- instantiation

- **itsObjects** : Diccionario que contiene todos los tipos de objetos que pueden encontrarse dentro del programa depurado:

- class
- method
- attribute
- function
- local
- probe
- thread
- classAttribute
- object
- exception
- specialMethod

- **itsPackXDR** : Diccionario que es utilizado para reconocer el tipo de datos de un objeto. Al momento que el tipo de datos es reconocido se devuelve el identificador para que la librería XDRLib sepa con que tipo de objeto está tratando. El contenido de este diccionario es:

- int
- string
- float
- double
- bool

- **itsDataTypes** : Diccionario que se utiliza para identificar el tipo de dato que se está enviando desde Python hacia Java, que es donde se almacenan en la base de datos de TOD. El contenido de este diccionario es:

- `int.__name__`
- `str.__name__`
- `float.__name__`
- `long.__name__`
- `bool.__name__`
- `tuple.__name__`
- `list.__name__`
- `dict.__name__`

- **FLAG_DEBUGG** : Atributo utilizado para mostrar cierta información de los eventos o de las registraciones, cuando su valor es *True*.
- **FLAG_THROWN** : Atributo utilizado para controlar la propagación de excepciones. La utilización de este es explicado en la sección 6.3.1.2.5.
- **itsCurrentDepth** : Atributo utilizado para guardar el nivel actual de llamadas, se utiliza para el evento de modificación de un atributo de clase.

6.3.3.2.1.2. Métodos Entre los métodos más importantes se tienen:

- **__createnotab__** : Método que crea una estructura de datos la cual almacena la estructura de bytecode del objeto que se está registrando en el depurador. Este método sólo se aplica a los objetos *class*, *method* y *function*.
- **__getPartCode__** : Método que se utiliza para inspeccionar el bytecode por bloques (inicio, fin) con el objetivo de extraer todas las variables locales definidas en este bloque.
- **__trace__** : Método más importante de la clase y de pyTOD. Éste permite estar notificados de los siguientes eventos dentro de un programa Python:
 - `call`
 - `line`
 - `return`
 - `exception`

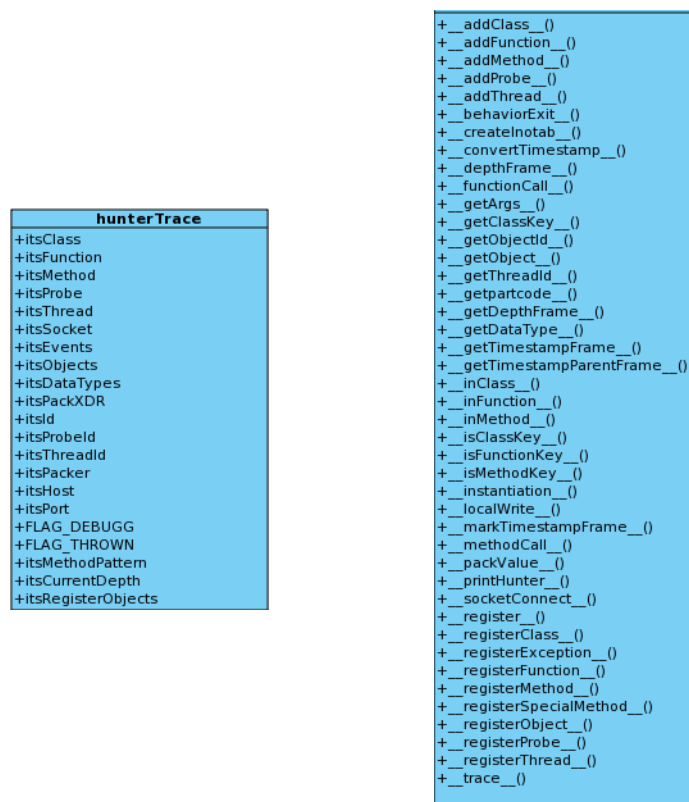


Figura 6.5: Diagrama de clase de hunterTrace

En este método se centra la mayoría de los movimientos para la captura de la huella de ejecución. Es importante señalar que por si sólo este método no tiene ninguna utilidad, ya que debemos indicar a Python que asigne `__trace__` como función al módulo `sys` mediante su método `settrace`.

6.3.3.2.2. generatorId Esta clase sirve para implementar todos los generadores de identificadores que se requieren para que el depurador funcione correctamente.

6.3.3.2.2.1. Atributos

- **Id** : Atributo que mantiene el valor actual del identificador. El rango al que el identificador pertenece es `[101 . . . n]`. Esto debido a que TOD tiene reservado el rango `[1 . . . 100]` para objetos internos.

6.3.3.2.2.2. Métodos

- **__get__** : Retorna el valor actual del atributo *Id*.
- **__next__** : Método que aumenta en una unidad el atributo *Id*.

6.3.3.2.3. Class Esta clase representa a cada una de las clases que ha definido el programador en su script.

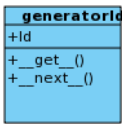


Figura 6.6: Diagrama de clase de generatorId

6.3.3.2.3.1. Atributos Los atributos más importantes son:

- **staticField**: Diccionario que guarda todos los atributos estáticos que pertenecen a la clase.
- **code** : Atributo que guarda el objeto code [26] de la clase original que definió el programador.
- **specialBehaviorId** : Atributo que sirve como contador para identificar a cada método estático registrado en la clase

6.3.3.2.3.2. Métodos

- **__getLnotab__** : Retorna el *lnotab* correspondiente de la clase.
- **__addAttribute__** : Agrega un nuevo atributo de instancia a la clase.
- **__addMethod__** : Agrega un nuevo método a la clase.

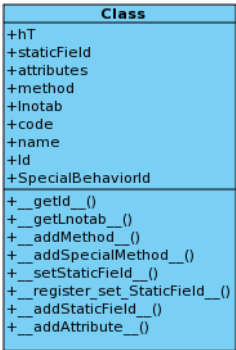


Figura 6.7: Diagrama de clase de Class

6.3.3.2.4. Method Esta clase representa a cada método que el programador haya definido en cualquiera de sus clases.

6.3.3.2.4.1. Atributos

- **locals** : Diccionario que representa a todas las variables locales definidas en el método.

- **argument** : Diccionario que representa todos los argumentos que tiene el método.
- **idClass** : Atributo que indica a que clase pertenece el método.

6.3.3.2.4.2. Métodos

- **__getLocals__** : Método que retorna todas las variables locales del método.
- **__getArgValues__** : Retorna el valor de todos los argumentos que se encuentran en *frame.f_locals*.

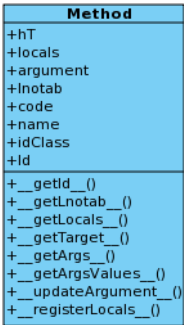


Figura 6.8: Diagrama de clase de Method

6.3.3.2.5. Function Esta clase representa a cada función que el programador haya definido en cualquiera de sus clases.

6.3.3.2.5.1. Atributos

- **locals** : Diccionario que representa a todas las variables locales definidas en la función.
- **argument** : Diccionario que representa todos los argumentos que tiene la función.

6.3.3.2.5.2. Métodos

- **__getLocals__** : Método que retorna todas las variables locales de la función.
- **__getArgValues__** : Retorna el valor de todos los argumentos que se encuentran en *frame.f_locals*.

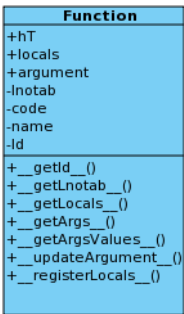


Figura 6.9: Diagrama de clase de Function

6.3.3.2.6. Dictionary Esta clase hereda del objeto nativo de Python *dict*. Esta clase se utiliza para almacenar los atributos de instancia, atributos de clase y variables locales de métodos y funciones.

6.3.3.2.6.1. Atributos

- **hT** : Atributo que hace referencia al *hunterTrace* que pertenece.

6.3.3.2.6.2. Métodos

- **__update__** : Método que actualiza los elementos del diccionario, si el elemento no existe notifica a la base de datos estructural de TOD su creación y si el elemento existe notifica a la base de datos de eventos de TOD la modificación.

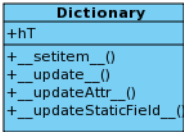


Figura 6.10: Diagrama de clase de Dictionary

6.3.4. Código Principal de pyTOD

En esta sección se mostrará el código principal de pyTOD utilizando el lenguaje de pseudocódigo, para facilitar el entendimiento de este. El código que se muestra son sólo las partes más importantes dentro del método `__trace__`, detallando los eventos del tipo *call*, *line*, *return* y *exception*.

Algorithm 2 Código principal de pyTOD

Entrada: *aFrame, aEvent, aArg*

```
1  theCode  $\leftarrow$  aFrame.f_code
2  theLocals  $\leftarrow$  aFrame.f_locals
3  theGlobals  $\leftarrow$  aFrame.f_globals
4  aFrame  $\leftarrow$  __markTimestampFrame__()
5  si aEvent == ' call ' entonces
6    si self  $\in$  theLocals entonces
7      __registerMethod__()
8      si theCode.co_name == ' __init__ ' entonces
9        asignar identificador unico a instancia de clase
10       __registerInstantiation__()
11     sino
12       __methodCall__()
13     fin si
14   sino
15     si theObject es del tipo function entonces
16       __functionCall__()
17     fin si
18   fin si
19   retorna __trace__
20 sino si aEvent == ' line ' entonces
21   theObject  $\leftarrow$  __getObject__()
22   theLnotab  $\leftarrow$  __getLnotab__()
23   si aFrame.f_lasti  $\in$  theLnotab entonces
24     theByteCodeLocals  $\leftarrow$  __getPartCode__()
25     __registerLocals__()
26     __localWrite__()
27   fin si
28   retorna __trace__
29 sino si aEvent == ' return ' entonces
30   si ' __init__ '  $\in$  theLocals entonces
31     si code  $\notin$  class entonces
32       __registerClass__()
33     fin si
34   sino
35     theObject  $\leftarrow$  __getObject__()
36     si FLAG_THROWN == true entonces
37       FLAG_THROWN  $\leftarrow$  false
38     fin si
39     theObject  $\leftarrow$  __getObject__()
40     theLnotab  $\leftarrow$  __getLnotab__()
41     si aFrame.f_lasti  $\in$  theLnotab entonces
42       theByteCodeLocals  $\leftarrow$  __getPartCode__()
43       __registerLocals__()
44       __localWrite__()
45     fin si
46     __behaviorExit__()
47   fin si
48 sino si aEvent == ' exception ' entonces
49   __registerException__()
50   si excepción es manejada en el bloque == true entonces
51     retorna __trace__
52   fin si
53   __behaviorExit__()
54   FLAG_THROWN  $\leftarrow$  true
55 fin si
```

6.3.5. Interfaces principales de pyTOD

Esta sección describe las interfaces principales que encontrará el usuario en pyTOD. Se describe desde arrancar el depurador hasta como interactuar con éste.

6.3.5.1. Arrancar el depurador

Luego de abrir una sesion en Eclipse se debe arrancar el depurador:



Figura 6.11: Botón para arrancar el depurador

Al momento de presionar este botón se mostrará un proceso hijo de Eclipse, el cual contiene al ambiente completo para comenzar a realizar la depuración.

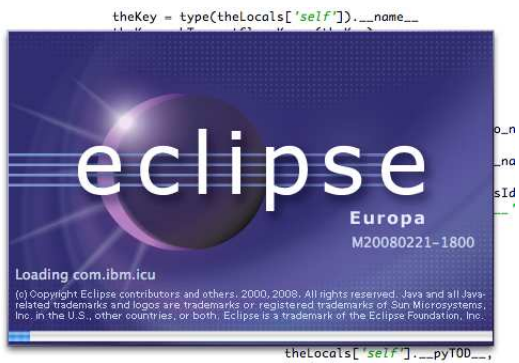


Figura 6.12: Proceso Eclipse hijo lanzado

Al terminar la carga completa del proceso Eclipse hijo, se muestra la perspectiva para realizar depuración.

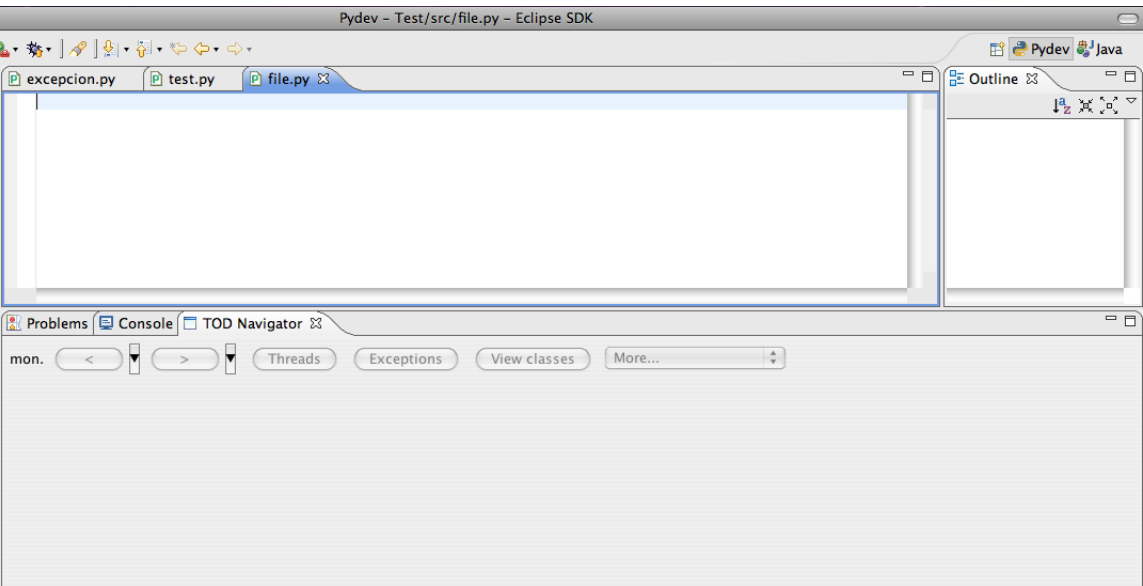


Figura 6.13: Perspectiva inicial para depuración

6.3.5.2. Realizando depuración

El usuario después de escribir su código lo puede depurar presionando el siguiente botón:

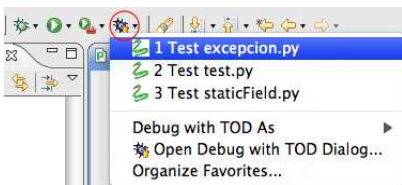


Figura 6.14: Botón de depuración

Después de ejecutar el depurador se le mostrará la siguiente interfaz al usuario:

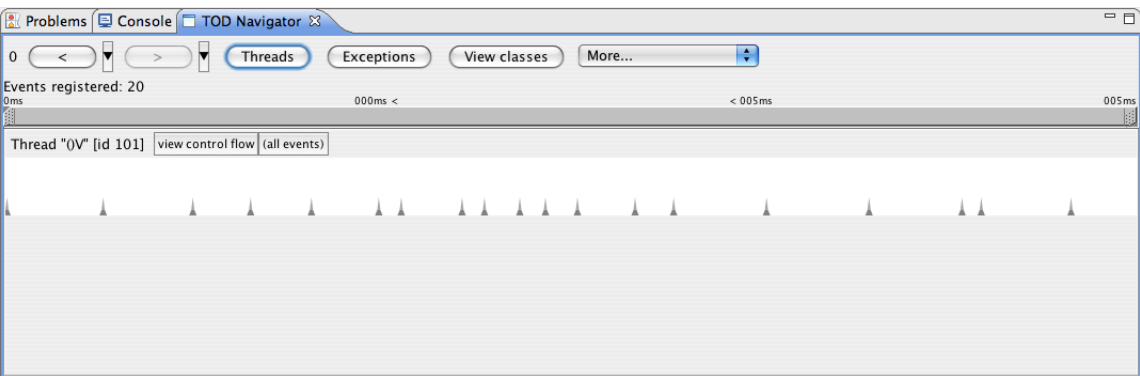


Figura 6.15: Interfaz inicial luego de haber depurado el código del usuario

Lo que se muestra aquí son todos los eventos ocurridos en el script del programador, quien haciendo doble click se encuentra con la siguiente interfaz:

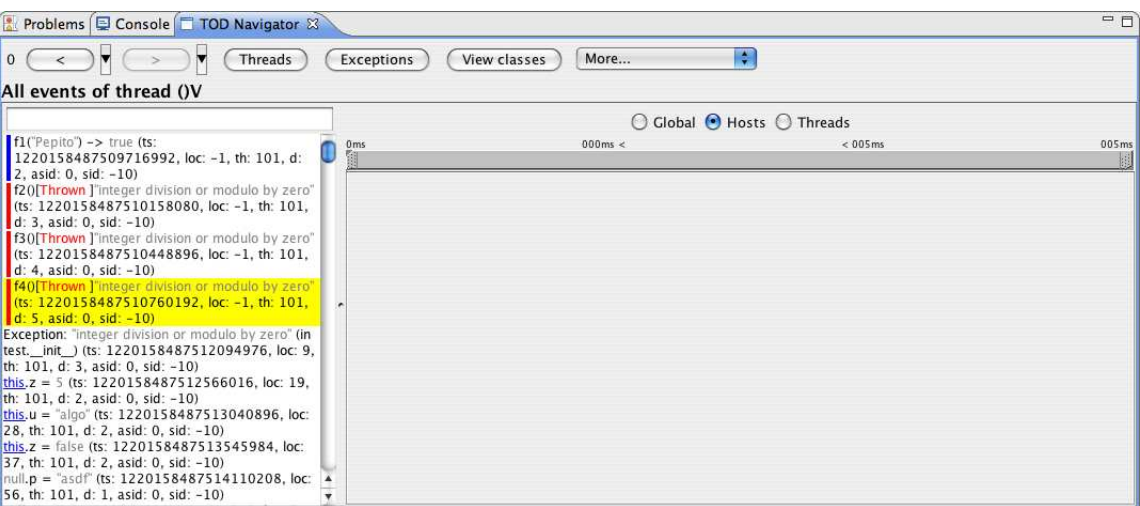


Figura 6.16: Control de flujo

Como se puede observar en esta figura se ha seleccionado la línea en la que el método *f4* lanza una excepción. Como se puede observar en la siguiente figura, en el momento que el usuario selecciona la línea el depurador inmediatamente ubica en el código fuente la línea que se está examinando.

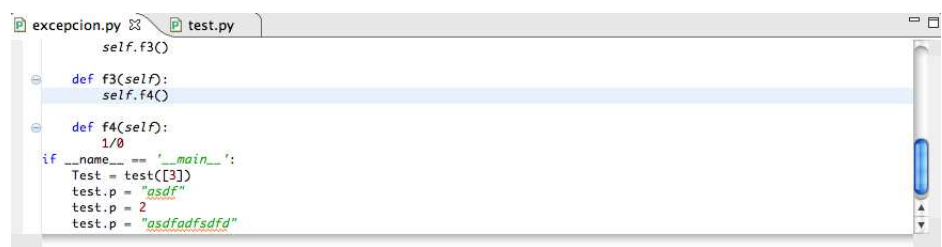


Figura 6.17: Línea inspeccionada

6.4. Aspectos Básicos de Funcionamiento de pyTOD

En esta sección se detallarán los pasos necesarios a seguir para obtener el depurador TOD y pyTOD.

6.4.1. Usuario final

Usuario final se refiere al usuario que sólo realizará depuración omnisciente en sus scripts y no tendrá la intención de realizar mejoras o implementar nuevas características. Este usuario sólo debe visitar <http://pleiad.dcc.uchile.cl/files/tod/releases/> y descargar la última versión liberada. Es importante señalar que los únicos requerimientos que se necesitan son:

- Eclipse 3.2 o superior.
- Java 1.5 o superior.
- Python 2.5.
- Sistema operativo basado en el kernel de Linux 2.6.x

Para mayor información puede visitar <http://pleiad.dcc.uchile.cl/tod/download.html>.

6.4.2. Desarrollador

Para que el desarrollador pueda trabajar comodamente y pueda aportar mejoras e implementar nuevas características, debe seguir los siguientes pasos para instalar la versión de desarrollo:

- Instalar Eclipse 3.2 o superior.
- Instalar Java 1.5 o superior.
- Instalar Python 2.5
- Descargar mediante svn los siguientes proyectos:

- <http://stgo.dyndns.org/svn/gpothier/devel/zz.utils>
- <http://stgo.dyndns.org/svn/gpothier/devel/zz.eclipse.utils>
- <http://pleiad.dcc.uchile.cl/svn/tod/pytod/trunk/java-proyect>
- <http://pleiad.dcc.uchile.cl/svn/tod/dbgrid/trunk>
- <http://pleiad.dcc.uchile.cl/svn/tod/agent/trunk>
- <http://pleiad.dcc.uchile.cl/svn/tod/tod.plugin.pytod/trunk>
- <http://pleiad.dcc.uchile.cl/svn/tod/tod.plugin.ajdt>
- <http://pleiad.dcc.uchile.cl/svn/tod/tod.plugin/trunk>
- <http://pleiad.dcc.uchile.cl/svn/tod/core/trunk>
- <http://pleiad.dcc.uchile.cl/svn/tod/pytod/trunk/python-project>
- <http://pleiad.dcc.uchile.cl/svn/tod/evdb1/trunk>

Dentro de eclipse se debe modificar las configuraciones de ejecución de los siguientes archivos:

■ **GridMaster**

- *Argumentos*

Para ejecutar este programa Java se debe notar que existen dos tipos de argumentos: los argumentos del programa y los argumentos de la maquina virtual.

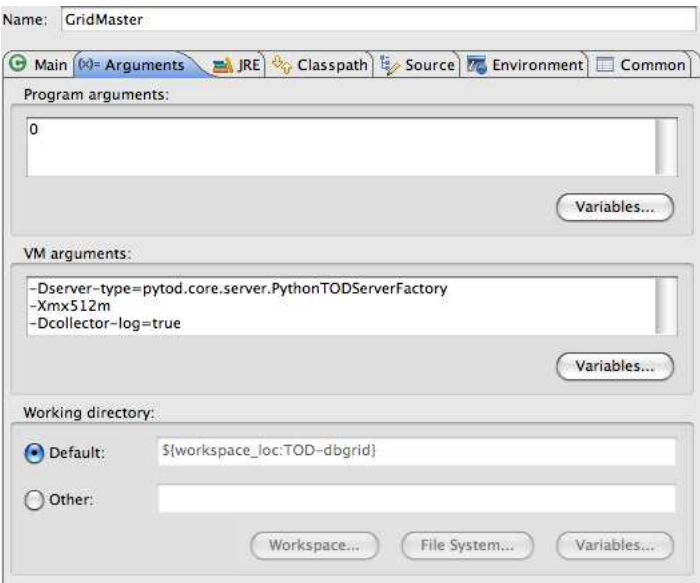


Figura 6.18: Argumentos de GridMaster

- *Classpath*

Es importante tener bien configurada esta sección ya que varios proyectos dependen de otros y si no se establecen las relaciones entre ellos, el depurador no funcionará.

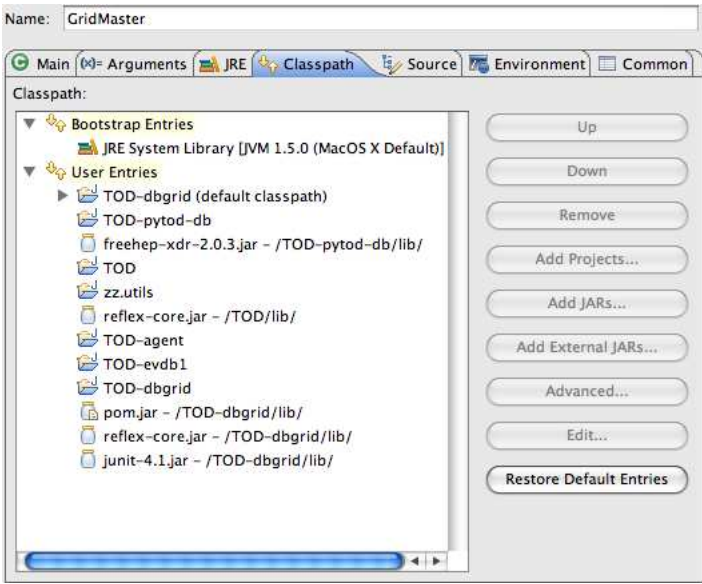


Figura 6.19: Classpath de Gridmaster

- **StandaloneUI**

- *Argumentos*

Para ejecutar este programa Java se debe notar que existen dos tipos de argumentos: los argumentos del programa y los argumentos de la maquina virtual.

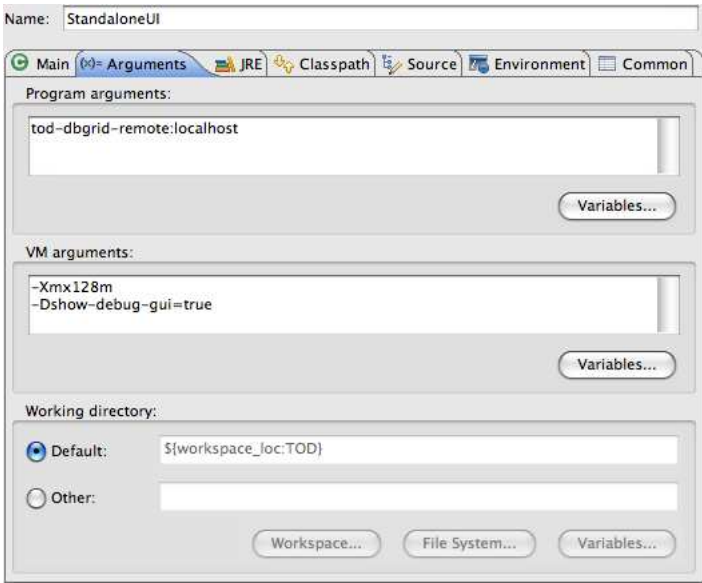


Figura 6.20: Argumentos de StandaloneUI

- *Classpath*

Es importante tener bien configurada esta sección ya que varios proyectos dependen de otros y se no se establecen las relaciones entre ellos, el depurador no funcionará.

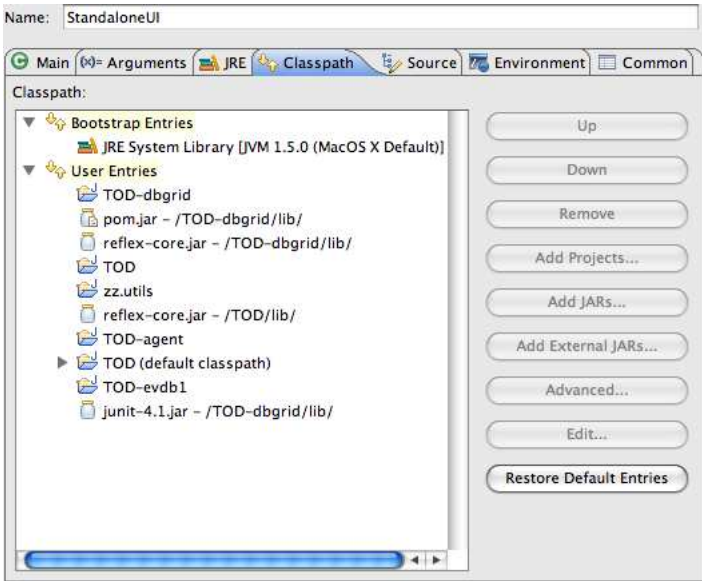


Figura 6.21: Classpath de StandaloneUI

- **Plugin**

- *Argumentos*

Para ejecutar este programa Java se debe notar que existen dos tipos de argumentos: los argumentos del programa y los argumentos de la maquina virtual.

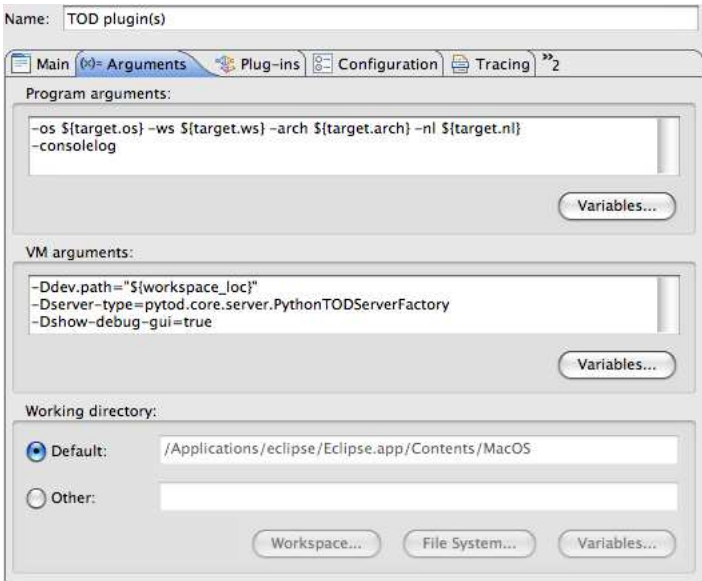


Figura 6.22: Argumentos de TOD Plugin

6.5. Medidas de rendimiento de pyTOD

Para medir la sobrecarga que realiza pyTOD sobre los scripts depurados se ha medido el tiempo de ejecución de los scripts sin depurar y luego el resultado del tiempo de ejecución cuando son depurados. Los tiempos que se muestran a continuación son los mejores tiempos obtenidos de cinco tiempos de ejecución de cada script. Las características del computador en el que se realizaron las mediciones son: Procesador de 2.4 GHz Intel Core 2 Duo, memoria ram de 2 GB 667 MHz DDR2, sistema operativo de Ubuntu 8.04.

El primer algoritmo en el cual se realizaron mediciones es el de *ordenamiento por burbuja*:

```

1  class Sort(object):
2
3      def __init__(self, aArreglo=None):
4          if aArreglo is None:
5              aArreglo = []
6          self.aArreglo = aArreglo
7
8      def burble(self):
9          for i in range(len(self.aArreglo)):
10             for j in range(i, len(self.aArreglo)):
11                 if self.aArreglo[i] > self.aArreglo[j]:
12                     theAux = self.aArreglo[i]
13                     self.aArreglo[i] = self.aArreglo[j]
14                     self.aArreglo[j] = theAux
15
16  if __name__ == '__main__':
17      theArreglo = list(range(5000))
18      theSort = Sort(theArreglo)
19      theSort.burble()
```

El mejor tiempo de ejecución para este script sin el depurador fue de 4,77015805244 segundos, y el mejor tiempo de ejecución con el depurador 1.878,3465. Presentando una sobrecarga aproximada de 393 veces. Es importante señalar que el depurador en este caso ha registrado el total de 12.507.506 eventos.

El segundo algoritmo en el cual se realizaron mediciones es en *fibonacci*:

```

1  class Fibonacci(object):
2
3      def __init__(self):
4          self.resultado = -1
5
6      def fibonacci(self, n):
7          if n == 1 or n == 2:
8              self.resultado = 1
9          elif n > 2:
```

```
10         self.resultado = self.fibonacci(n-1) + self.  
11             fibonacci(n-2)  
12     return self.resultado  
13 if __name__ == '__main__':  
14     theNumero = 32  
15     theFibonacci = Fibonacci()  
16     print theFibonacci.fibonacci(theNumero)
```

El mejor tiempo de ejecución para este script sin el depurador fue de 3,5871169 segundos, y el mejor tiempo de ejecución con el depurador 2.846,12946486. Presentando una sobrecarga aproximada de 793 veces. Es importante señalar que el depurador en este caso ha registrado el total de 13.069.854 eventos.

Capítulo 7

Evaluación de pyTOD en casos de pruebas

7.1. Selección de casos de uso

En esta sección se seleccionan tres casos de uso: división por cero, fibonacci y manejador de archivo. El primero se utilizar para mostrar el registro correcto de excepciones sin manejar por parte del programador, el segundo se utiliza para mostrar la capacidad de pyTOD para la captura de huella y posterior navegación en ella y el tercero para mostrar la capacidad de pyTOD para encontrar excepciones manejadas por parte del programador y encontrar la causa inicial del problema.

- **División por cero:** Este es un sencillo script que toma un valor a de forma aleatoria, utilizando la función *random* de la librería *random*, luego toma este valor e itera en el rango $[1, \dots, 100]$ dividiendo al numerador por la sustracción $\text{rango}_i - a$.

El algoritmo que se utiliza es el siguiente:

```
1  #! /usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from random import random
5
6  class Division(object):
7
8      def __init__(self, aRandomValue):
9          self.theRandomValue = aRandomValue
10
11     def division(self):
12         for theIndex in range(1,101):
13             theResult = 1/(theIndex - self.theRandomValue)
14             pass
```

```

15
16 if __name__ == '__main__':
17     theDivision = Division(int(random() * 100))
18     theDivision.division()

```

- **Fibonacci** : Este algoritmo, ampliamente conocido en el ámbito de la computación, construye una serie aritmética sumando los términos t_{i-1} y t_{i-2} para formar el término i -ésimo de la serie. Para la implementación utilizada $i \in [2, \dots, n]$ y los términos t_0 y t_1 tienen los valores 1 y 1 respectivamente.

El algoritmo que se utiliza es el siguiente:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import sys
5
6  class Fibonacci(object):
7
8      def __init__(self):
9          self.resultado = -1
10
11      def fibonacci(self, n):
12          if n == 0 or n == 1:
13              self.resultado = 1
14          elif n > 1:
15              self.resultado = self.fibonacci(n-1) + self.
16                             fibonacci(n-2)
17          return self.resultado
18
19  if __name__ == '__main__':
20      theNumero = 5
21      theFibonacci = Fibonacci()
22      print theFibonacci.fibonacci(theNumero)

```

- **Manejador de archivos** : Este algoritmo toma tres nombres de archivos, con un directorio dado. Los nombres son tomados uno por uno, el algoritmo abre el archivo y lo renombra para luego guardarlo en el mismo directorio. Claramente este algoritmo fue mal diseñado por el desarrollador, ya que, en el manejo de excepciones deja que silenciosamente estas no sean mostradas al usuario final, dando la impresión que la tarea ha sido completada con éxito.

El algoritmo que se utiliza es el siguiente:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  class ExampleFile(object):
4
5      def __init__(self):

```

```
6         self.theDirectory = 'images'
7
8     def abrir(self, aName):
9         try:
10             theFile = open(self.theDirectory + '/' + aName
11                             )
12             theNewFile = open(self.theDirectory + '/' + '
13                             NEW' + aName, 'w+')
14             theNewFile.close()
15         except:
16             return -1
17
18     def converter(self, aFile):
19         im = self.abrir(aFile)
20
21 if __name__ == '__main__':
22     theExample = ExampleFile()
23     theNames = ['tree_branches_and_roots_01.png',
24                 'acorn_jonathan_dietrich_01.png',
25                 'claveles.png']
26     for theName in theNames:
27         try:
28             theExample.converter(theName)
29         except:
30             pass
```

7.2. Resultados obtenidos con pyTOD

En esta sección se muestran los resultados de depuración obtenidos de los tres casos de uso seleccionados.

7.2.1. División por cero

En el momento de ejecutar este script, Eclipse lanza la siguiente excepción:

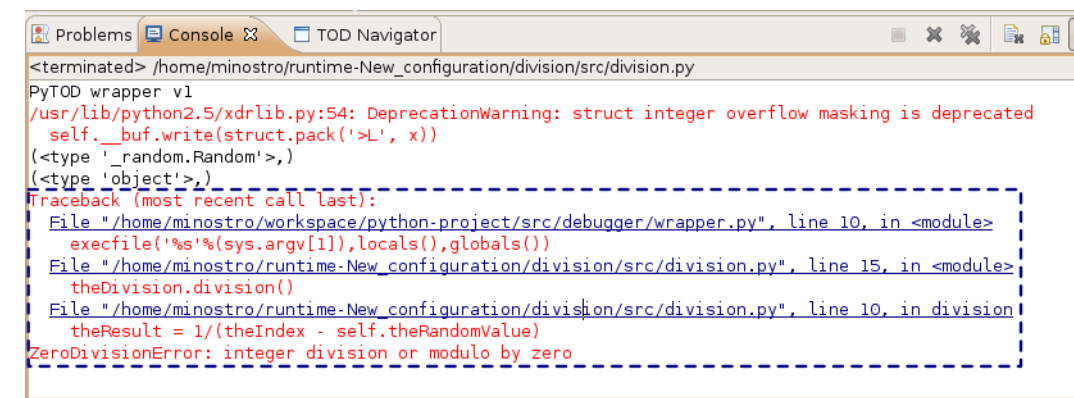


Figura 7.1: Excepción lanzada por el script division.py

De lo único que se puede estar enterado es que el script falló en la línea 10, cosa que es útil pero no es de gran ayuda.

En la misma ejecución del script se puede ingresar a la perspectiva de pyTOD. Lo primero que se vera es el mural de hilos de ejecución con sus eventos respectivos.

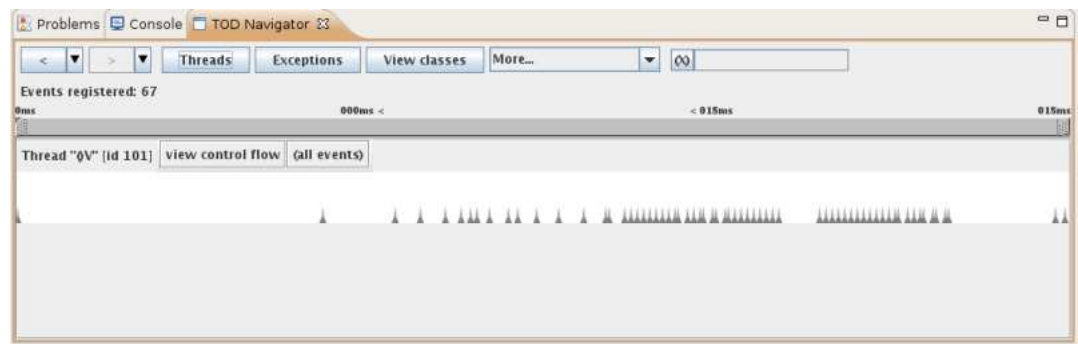


Figura 7.2: Eventos sucedidos para el script division.py

Al presionar dos veces el mural en el hilo que nos interesa o al presionar el botón *ver control de flujo* se mostrarán todas las llamadas a métodos e instanciaciones a objetos.

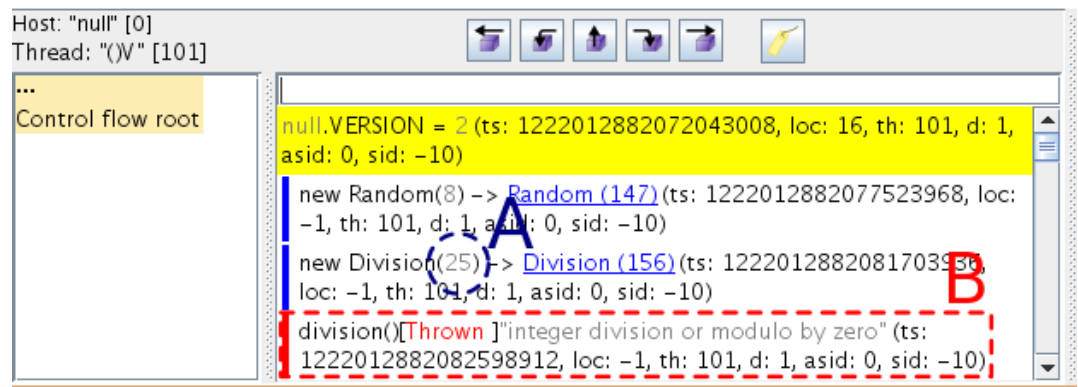


Figura 7.3: Control de flujo para el script division.py

Se puede observar dos cosas de gran interés: (A) la clase fue instanciada con el valor 25 y (B) el método *division* tiene como valor de salida la excepción *integer division or modulo by zero*.

Al presionar dos veces el método *division* se mostrarán todos los eventos ocurridos dentro de este.

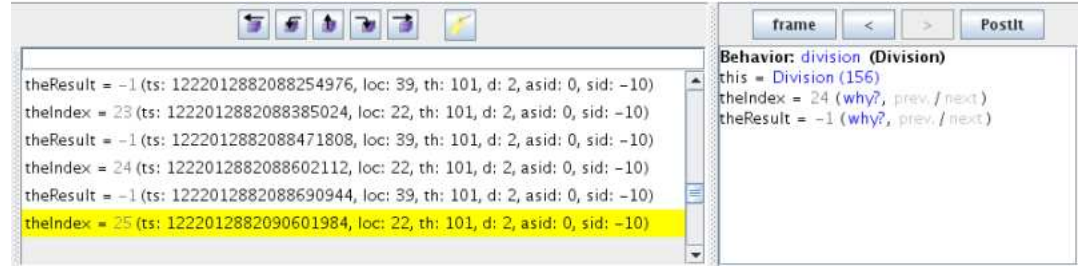


Figura 7.4: Eventos ocurridos dentro del método division

Por comodidad se muestran los últimos eventos en donde se puede ver que el último valor para *theIndex* fue 25. Sabiendo que el valor con que la clase fue construida, por simple inspección se sabe que el valor para el cual el método lanza la excepción es el valor 25.

Además el programador puede verificar la estructura de su clase presionando el botón *ver clases*.

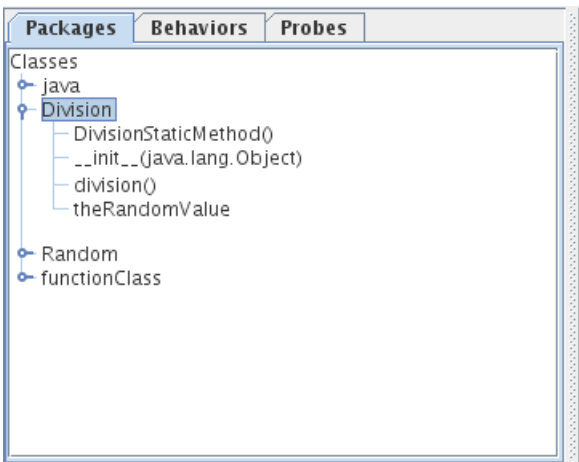


Figura 7.5: Estructura de script division.py

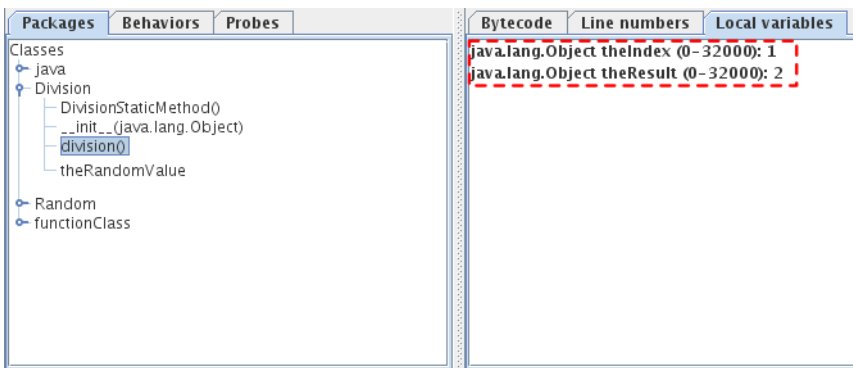


Figura 7.6: Método division con sus variables locales

7.2.2. Fibonacci

Al momento de ejecutar este script, termina de forma correcta, esto debido a que sólo se quiere mostrar la capacidad de captura de huella de pyTOD. La salida que se muestra es la siguiente:

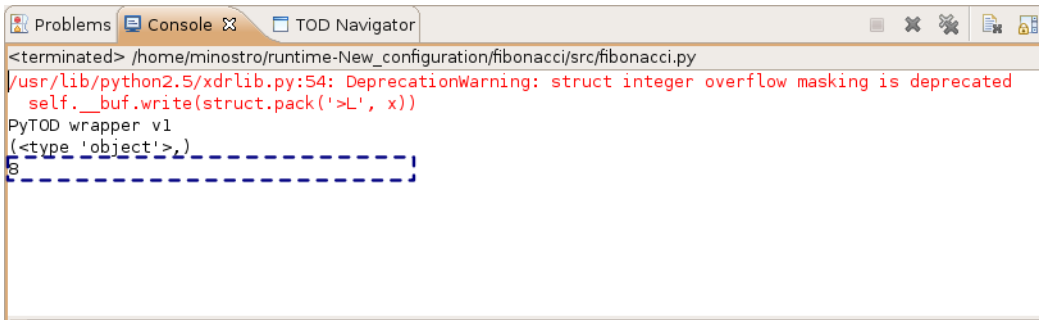


Figura 7.7: Resultado de ejecución para el script fibonacci.py

Inspeccionando el mural de hilos de ejecución se pueden ver todos los eventos generados.

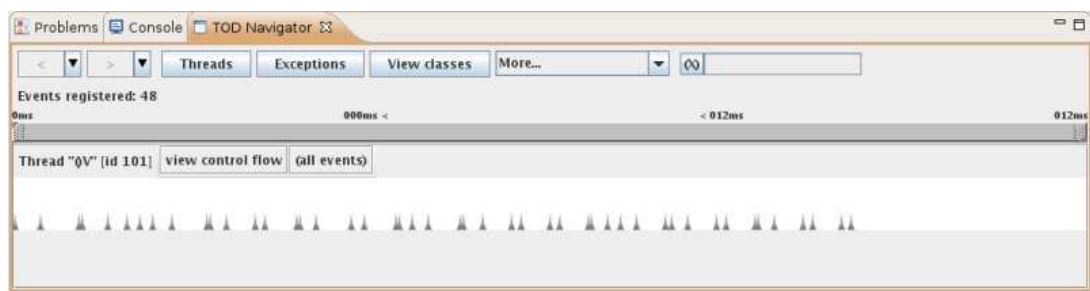


Figura 7.8: Eventos sucedidos para el script fibonacci.py

Al presionar dos veces sobre el mural de hilos de ejecución se muestra el flujo de control principal.

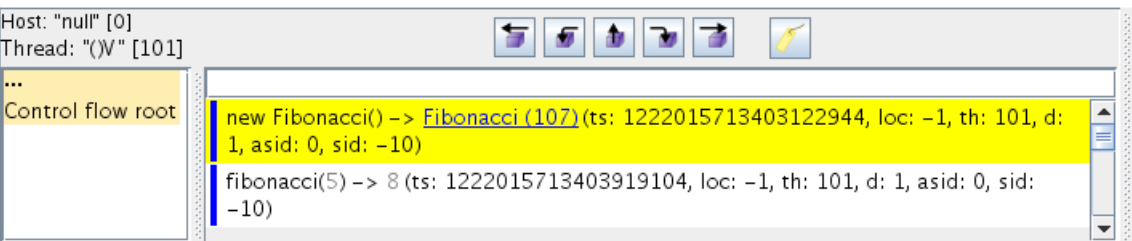
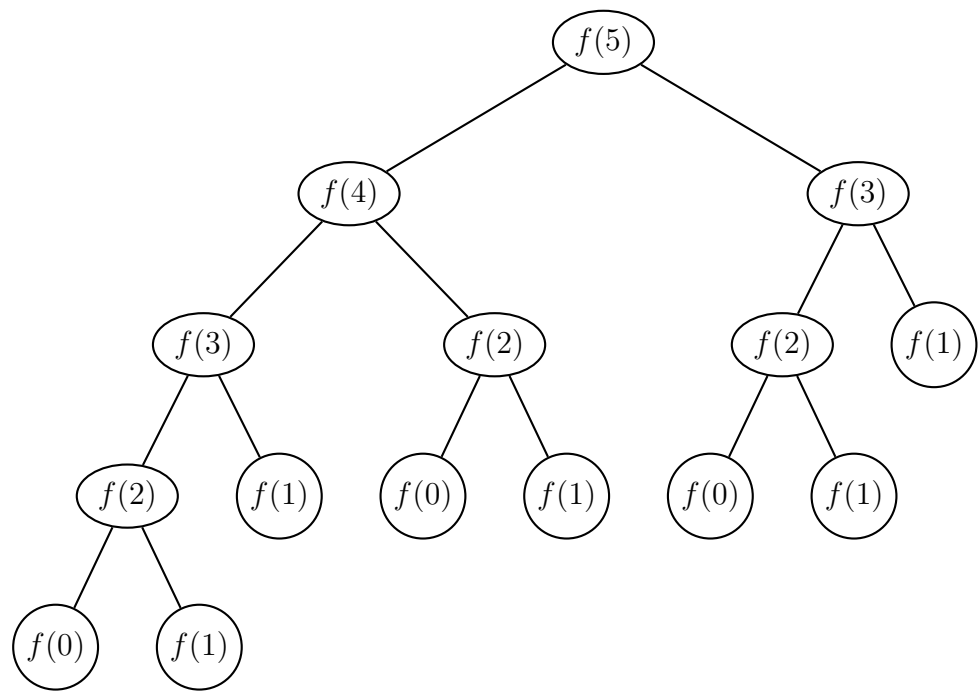


Figura 7.9: Control de flujo para el script fibonacci.py

Lo que es de mayor interés para este caso es que la llamada al método *fibonacci* con valor 5 para su argumento, retorna el valor 8.

Se muestra el siguiente árbol de ejecución para este algoritmo, con el fin de verificar posteriormente el resultado de pyTOD.



Se sabe que $f(0)$ y $f(1)$ devuelven el valor 1 respectivamente.

Al ingresar al llamado de *fibonacci(5)* se tiene:

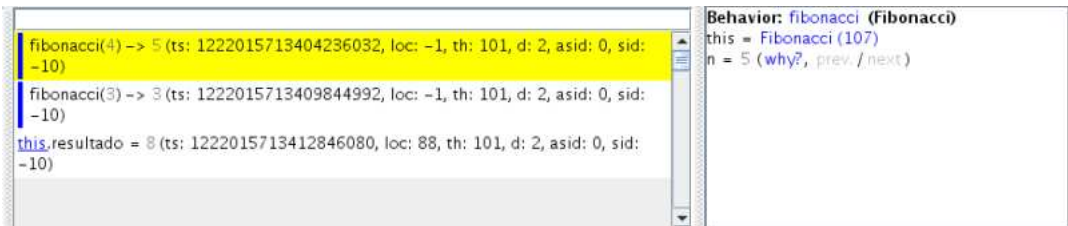


Figura 7.10: Control de flujo para fibonacci(5)

Por un tema de facilidad sólo se mostrará el recorrido inorden del árbol anteriormente mostrado.

Se ingresar a *fibonacci(4)* que como se puede observar retorna el valor 5.

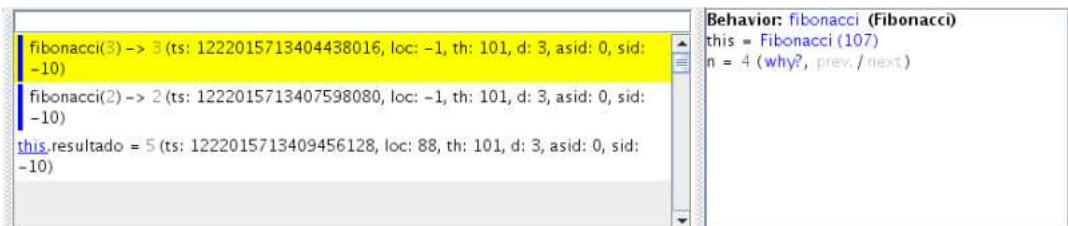


Figura 7.11: Control de flujo para fibonacci(4)

Se ingresar a *fibonacci(3)* que como se puede observar retorna el valor 3.

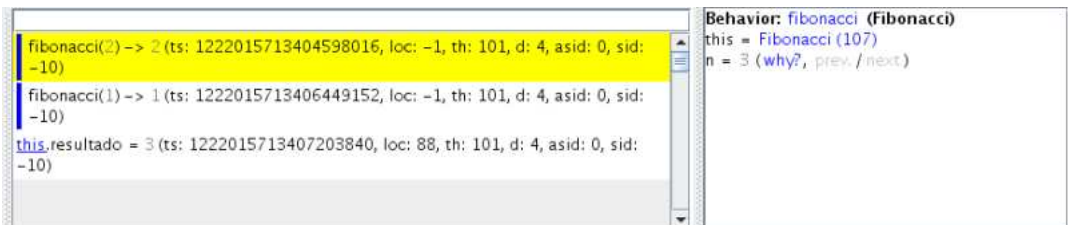


Figura 7.12: Control de flujo para fibonacci(3)

Se ingresar a *fibonacci(2)* que como se puede observar retorna el valor 2.

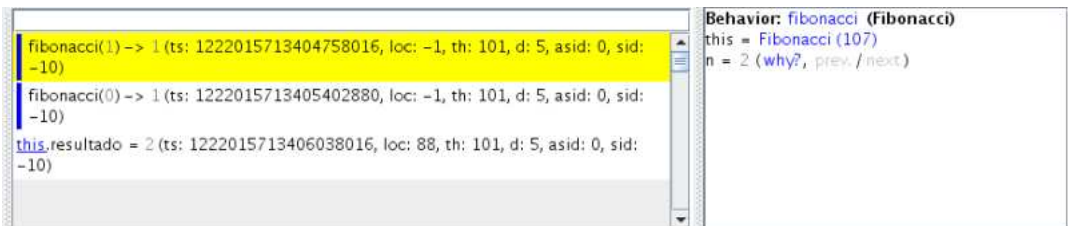


Figura 7.13: Control de flujo para fibonacci(2)

Como se puede observar pyTOD es capas de capturar correctamente la huella de ejecución.

El programador al igual que en el ejemplo anterior podrá ver la estructura del script, sus métodos y variables locales.

7.2.3. Manejador de archivo

Al ejecutar este script la salida por consola indica que todo ha ido bien (no se muestran mensajes).

Inspeccionando el mural de hilos de ejecución se pueden ver todos los eventos.

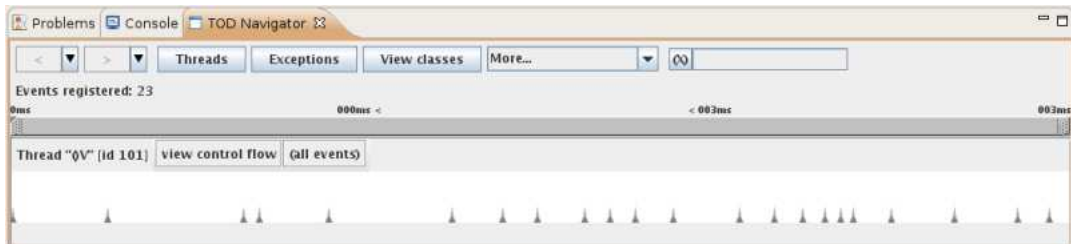


Figura 7.14: Eventos sucedidos para el script manejadorArchivo.py

Al presionar dos veces en el mural de hilo de ejecución se muestra el control de flujo principal.

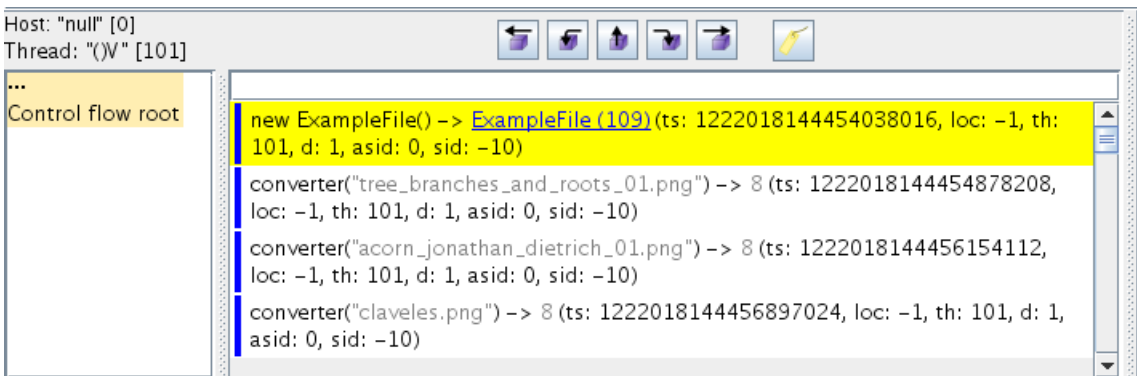


Figura 7.15: Control de flujo para el script manejadorArchivo.py

A simple vista todas las llamadas a métodos han retornado un valor váido y ninguno ha levantado excepción alguna. Sin embargo se sabe que la llamada a *converter* con valor *claveles.png* en su argumento debiera haber lanzado una excepción, ya que, este archivo no existe.

A continuación se inspecciona la llamada a este *converter*.

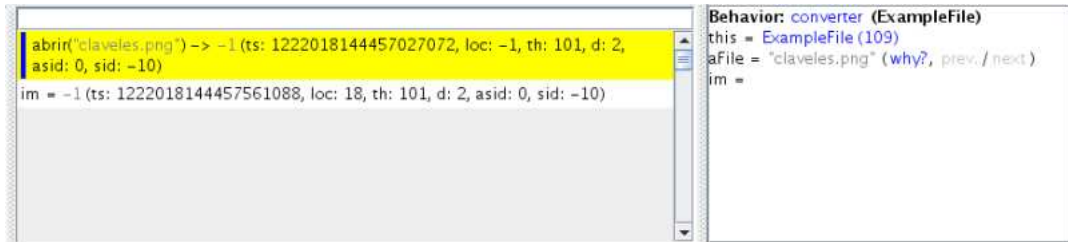


Figura 7.16: Eventos dentro de la llamada de converter

En este nivel tampoco se observa directamente que algo ha ido mal. Con el fin de seguir inspeccionando se ingres a la llamada del método *abrir* con valor en su argumento de *claveles.png*

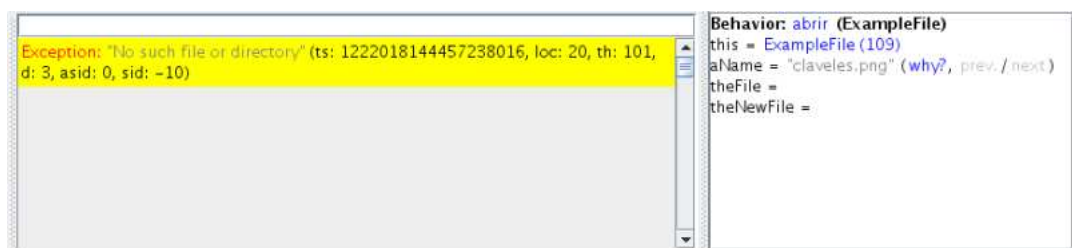


Figura 7.17: Excepción dentro del método abrir

Como se puede observar se ha encontrado una excepción con un valor *No such file or directory*, en el momento que la variable local *aName* tenía como valor *claveles.png*



Figura 7.18: Valores de variables cercanas a la excepción

El programador de la misma forma que los casos anteriores puede observar la estructura del programa.

Capítulo 8

Conclusiones

Las herramientas de depuración de código, son las herramientas que prestan más ayuda a los desarrolladores de software tanto en la etapa de desarrollo como en la etapa de puesta en marcha de un proyecto informático. Es por esto que existía una gran motivación de encontrar un área, dentro de la depuración, en la cual se pudiera realizar un trabajo a nivel de ingeniería que fuera un aporte interesante. En la actualidad la formación del Ingeniero Civil en Computación e Informática no contempla asignaturas en las que se enseñen los distintos enfoques y arquitecturas de los depuradores. Debido a esto al comienzo del trabajo de memoria existía incertidumbre sobre los posibles problemas que se pudieran encontrar en esta área totalmente desconocida, incertidumbre que a través del desarrollo de este trabajo de título fue desapareciendo.

El análisis del marco teórico se volvió fundamental ya que sienta las bases para poder conocer las principales ventajas y desventajas de los tres distintos enfoques que existen para realizar depuración de código:

- Basado en mensajes.
- Punto de quiebre.
- Omnisciente.

El resultado de este análisis condujo el presente trabajo por el enfoque de los depuradores omniscientes debido a sus características que superan a las de los depuradores de punto de quiebre y los basados en mensajes. Esto a pesar de que este tipo de depuradores no ha podido posicionarse como el enfoque de facto para realizar depuración de software. La pregunta principal que se planteó en ese momento fue *¿Por qué los depuradores omniscientes no son los más utilizados?*.

Con el objetivo de responder la pregunta anteriormente planteada, se decidió buscar y estudiar las implementaciones existentes del enfoque de depuración omnisciente. La búsqueda de las implementaciones fue compleja y se encontró poca cantidad de implementaciones de este enfoque, comparado con la cantidad encontrada de implementaciones que tienen los enfoques de punto de quiebre y basado en mensajes. El principal problema que se argumentaba dentro de los mismos desarrolladores de los depuradores omniscientes, era la gran capacidad de almacenamiento que se requería para almacenar la huella de ejecución completa de una sesión de depuración y el complejo manejo de los datos que esto implicaba.

La poca cantidad de depuradores omniscientes desarrollados fue una señal potente de que implementar un depurador omnisciente no era una tarea fácil y menos una tarea que la pudiera desarrollar un individuo en si. Entendiendo que no era posible desarrollar un depurador omnisciente desde cero, se enfocó el estudio en un depurador omnisciente llamado TOD. Esta herramienta llamó particularmente la atención en el momento del estudio de implementaciones del enfoque de depuración omnisciente, debido a que era la única herramienta que en su definición ofrecía una estructura de base de datos que permitía almacenar por completo la huella de ejecución de una sesión de depuración sobre una arquitectura escalable, solucionando el principal problema que se le atribuyen a estos tipos de depuradores.

Al analizar las posibles contribuciones o mejoras que se pudieran realizar en TOD, se encontró un punto de principal interés. TOD al ser un depurador para el lenguaje de programación Java, sólo realizaba depuración a un lenguaje de programación que es de tipado dinámico. Esto por si solo dejó muchas interrogantes abiertas *¿Cómo se comportaría TOD al realizar depuración a un lenguaje dinámico?, ¿Qué cambios habría que realizar en TOD para dar soporte a este tipo de lenguajes?, ¿Se pueden reutilizar algunos componentes de TOD para este objetivo?, ¿Cómo comunicar los componentes de TOD con la nueva arquitectura de depuración?, entre otras.*

A partir de todas estas interrogantes es que se tomó a TOD como base de estudio y plataforma de desarrollo para la construcción de pyTOD, un prototipo experimental para realizar depuración omnisciente a scripts escritos en el lenguaje de programación Python, lenguaje de tipado dinámico.

En la etapa de diseño del prototipo se consideró necesario que pyTOD estuviera compuesto por los siguientes elementos:

- Capturador de huella.

- Protocolo de comunicación.
- Mecanismo de conexión a nivel de base de datos.

Una vez implementados estos elementos y realizando pequeñas modificaciones en algunos de los componentes de TOD, se pudo efectuar una sesión de depuración omnisciente a scripts escritos en un lenguaje de programación de tipado dinámico, logrando alcanzar con esto el objetivo general del presente trabajo de título.

Es importante señalar que además de demostrar que se pudo realizar depuración omnisciente a scripts escritos en un lenguaje de tipado dinámico utilizando como plataforma tecnológica a TOD, también permitió lo siguiente:

- Notar que el diseño de base de datos de TOD no requirió de mayores modificaciones.
- Hacer más general el mecanismo de conexión a la base de datos de TOD.
- Demostrar que TOD es fácilmente extensible para realizar depuración omnisciente a otros lenguajes de programación.
- Ampliar el público objetivo de TOD, ya que ahora realiza depuración a más de un lenguaje de programación.

Al finalizar este trabajo de título se tiene la satisfacción de haber realizado una pequeña y real contribución en el área de la depuración, específicamente en la depuración omnisciente. De la misma forma se está muy satisfecho de haber planteado y de haber dado solución a la situación de utilizar TOD como plataforma tecnológica para realizar depuración a scripts de lenguajes de tipado dinámico.

Capítulo 9

Trabajo futuro

Siendo pyTOD un prototipo experimental, su desempeño y prestaciones pueden ser mejoradas ampliamente, esto si se desea utilizarlo de forma real en la depuración de código.

Principalmente lo que se puede optimizar bastante es el capturador de huella, ya que no se utilizó el enfoque clásico de instrumentación de código si no que se utilizó la captura de la huella de ejecución mediante un mecanismo propio del lenguaje de programación. Cuando se habla de optimización no se refiere sólo a mejorar los tiempos de ejecución sino que también (a) Poder capturar objetos nativos de Python como son las tuplas, diccionarios, listas entre otras y capturar las llamadas a sus métodos internos (b) Implementar un sistema de identificadores únicos el cual sea aplicado a todos los objetos dentro del capturador de huella.

Es importante además mejorar la adaptabilidad en los diferentes sistemas operativos, como también mejorar el soporte para depuración de múltiples hilos de ejecución.

Bibliografía

- [1] A study of the effect of imperfect debugging on software development cost.
John Shafer, Rakesh Agrawal, Manish Mehta, 1996.
- [2] TOD, a scalable Omniscient Debugger
Guillaume Pothier, Eric Tanter, Jose Piquer, 2007.
- [3] IEEE Std, IEEE Software Engineering Standard: Glossary of Software Engineering Terminology.
IEEE Computer Society Press, 1993
- [4] The Practice Of Programming, Addison-Wesley, Inc.
Brian W. Kernighan, Rob Pike, 1999.
- [5] <http://logging.apache.org/log4j/1.2/manual.html>
- [6] <http://www.djangoproject.com/>
- [7] Bil Lewis. Debugging backwards in time. In M. Ronsse and K. De Bosschere, editors, Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), Ghent, Belgium, 2003.
- [8] The New Hacker's Dictionary, Third Edition. Eric. S.Raymond.
The MIT Press. 1996 (1993). ISBN: 0262680920. Inglés.
- [9] Evans, Thomas G., and D. Lucille Darlay, On-Line Debugging Techniques: A survey
AFIPS Conference Proceedings (1966 FJCC), Vol. 29, Spartan Books, Washington, D.C., 1966, pp.37-50
- [10] M. Ducassé. Opium: An Extendable trace analyser for Prolog.
The Journal of Logic programming, 39:177-223, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

- [11] Jim des Rivieres and John Wiegand. Eclipse: A platform for integrating development tools.
IBM Systems Journal, 43(2):371–383, 2004.
- [12] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Implementing a backward-in-time debugger.
In Proceedings of NODe’06, volume P-88, pages 17–32. Lecture Notes in Informatics, 2006.
- [13] Henry Lieberman and Christopher Fry. ZStep 95: A reversible, animated source code stepper.
In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, Software Visualization — Programming as a Multimedia Experience, pages 277–292, Cambridge, MA-London, 1998. The MIT Press.
- [14] Charles R. Hill. A real-time microprocessor debugging technique.
In SIGSOFT ’83: Proceedings of the symposium on High-level debugging, pages 145–148, New York, NY, USA, 1983. ACM Press.
- [15] Henry Lieberman. Reversible object-oriented interpreters.
In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, ECOOP, volume 276 of Lecture Notes in Computer Science, pages 11–19. Springer, 1987.
- [16] Éric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems.
In Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002: Adaptable and extensible component systems, November 2002.
- [17] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces.
IEEE Trans. Vis. Comput. Graph., 4(3):257–271, 1998.
- [18] Guillaume Pothier. Benchmarks of COTS database management systems.
Technical Report TR/DCC-2006-16, University of Chile, October 2006.
- [19] M. Blasgen and K. Eswaran. Storage and access in relational databases.
IBM Systems Journal, 16(4):363, 1977.
- [20] <http://docs.python.org/lib/debugger-hooks.html>

[21] <http://docs.python.org/lib/bytecodes.html>

[22] <http://docs.python.org/ref/attribute-access.html>

[23] <http://docs.python.org/ref/types.html#12h-143>

[24] <http://docs.python.org/lib/module-xdrlib.html>

[25] <http://www.faqs.org/rfcs/rfc1832.html>

[26] <http://docs.python.org/ref/types.html#12h-142>

Apéndice

Apéndice A

Código fuente de pyTOD

A.1. Código fuente de hunterTrace.py

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  __author__ = "Milton Inostroza Aguilera"
5  __email__ = "minoztro@gmail.com"
6  __all__ = ['hT']
7  th = True
8  import sys
9  import dis
10 import re
11 import types
12 import time
13 import thread
14 import xdrlib
15 import socket
16 import inspect
17 from constantsObjects import events, objects, dataTypes,
       packXDRLib
18 from generatorId import generatorId
19 from objectClass import Class
20 from objectMethod import Method
21 from objectFunction import Function
22 if th:
23     from threading import settrace
24
25
26 class hunterTrace(object):
27
28     def __init__(self, aId, aProbeId, aThreadId, aPacker, aHost
   , aPort):
29         self.itsClass = {}
30         self.itsFunction = {}
31         self.itsMethod = {}
32         self.itsProbe = {}
33         self.itsThread = {}
34         self.itsSocket = None
35         self.itsEvents = events
36         self.itsObjects = objects
```

```

37         self.itsDataTypes = dataTypes
38         self.itsPackXDR = packXDRLib
39         self.itsId = aId
40         self.itsProbeId = aProbeId
41         self.itsThreadId = aThreadId
42         self.itsPacker = aPacker
43         self.itsHost = aHost
44         self.itsPort = aPort
45         self.FLAG_DEBUGG = False
46         self.FLAG_THROWN = False
47         self.itsMethodPattern = "\A__.*(__)$"
48         #trick for MetaDescriptor's depth
49         self.itsCurrentDepth = 0
50         #trick for Objects {strings, tuple, dict, list}
51         self.itsRegisterObjects = []
52         self.__socketConnect__()
53
54     def __addClass__(self, aId, aLnotab, aCode):
55         objectClass = Class(self, aId, aCode, aLnotab)
56         self.itsClass.update({aCode: objectClass})
57         return objectClass
58
59     def __addFunction__(self, aId, aLnotab, aCode, aArgs):
60         self.itsFunction.update({aCode: Function(self, aId, aCode,
61             aLnotab, aArgs)})
62
63     def __addMethod__(self, aId, aLnotab, aCode, idClass, aArgs
64 ):
65         self.itsMethod.update({aCode: Method(self, aId, aCode,
66             aLnotab, idClass, aArgs)})
67
68     def __addProbe__(self, aProbeId, currentLasti, parentId):
69         self.itsProbe.update({(currentLasti, parentId): aProbeId
70             })
71
72     def __addThread__(self, aThreadId, aThreadSysId):
73         self.itsThread.update({aThreadSysId: aThreadId})
74
75     def __behaviorExit__(self,
76         aFrame,
77         aArg,
78         aDepth,
79         aParentTimestampFrame,
80         aThreadId,
81         aHasThrown):
82         theBackFrame = aFrame.f_back
83         theBackFrameCode = theBackFrame.f_code
84         theParentId = self.__getObjectId__(theBackFrameCode)
85         behaviorId = self.__getObjectId__(aFrame.f_code)
86         theCurrentLasti = aFrame.f_lasti
87         theDepth = aDepth + 1
88         if not self.itsProbe.has_key((theCurrentLasti,
89             theParentId)):
90             theProbeId = self.__registerProbe__(theCurrentLasti
91                 ,
92                 theParentId,
93                 aFrame.f_lineno
94             )
95         else:

```

```

89         theProbeId = self.itsProbe[(theCurrentLasti,
90                                     theParentId)]
91     theCurrentTimestamp = self.__convertTimestamp__(time.
92         time())
93     if aHasThrown:
94         self.__registerObject__(aArg, theCurrentTimestamp)
95         self.itsPacker.reset()
96         self.itsPacker.pack_int(self.itsEvents['return'])
97         self.itsPacker.pack_int(behaviorId)
98         self.itsPacker.pack_int(theDataType)
99         self.itsPacker.pack_int(id(aArg))
100        self.itsPacker.pack_int(1) #True
101        self.itsPacker.pack_hyper(aParentTimestampFrame)
102        self.itsPacker.pack_int(theDepth)
103        self.itsPacker.pack_hyper(theCurrentTimestamp)
104        self.itsPacker.pack_int(aThreadId)
105    else:
106        theDataType = self.__getDataType__(aArg)
107        if theDataType == 1:
108            if not id(aArg) in self.itsObjects:
109                self.__registerObject__(aArg,
110                    theCurrentTimestamp)
111            self.itsPacker.reset()
112            self.itsPacker.pack_int(self.itsEvents['return'])
113            self.itsPacker.pack_int(behaviorId)
114            self.itsPacker.pack_int(theDataType)
115            if theDataType == 1:
116                thePackValue = id(aArg)
117                self.itsPacker.pack_int(id(aArg))
118            else:
119                thePackValue = self.__packValue__(theDataType,
120                    aArg)
121            self.itsPacker.pack_int(0) #False
122            self.itsPacker.pack_int(theProbeId)
123            self.itsPacker.pack_hyper(aParentTimestampFrame)
124            self.itsPacker.pack_int(theDepth)
125            self.itsPacker.pack_hyper(theCurrentTimestamp)
126            self.itsPacker.pack_int(aThreadId)
127        if self.FLAG_DEBUGG:
128            print self.itsEvents['return'],
129            print behaviorId,
130            print theDataType,
131            print thePackValue,
132            print aHasThrown,
133            print theProbeId,
134            print aParentTimestampFrame,
135            print theDepth,
136            print theCurrentTimestamp,
137            print aThreadId
138            raw_input()
139        try:
140            self.itsSocket.sendall(self.itsPacker.get_buffer())
141        except:
142            print 'TODO: Murmiendo :-( - Salida de behavior
143                ', aFrame.f_code.co_name
144
145    def __createlnTAB__(self, aCode):

```

```

143         theLnotab = {}
144         if hasattr(aCode, 'co_lnotab'):
145             table = aCode.co_lnotab
146             index = 0
147             last_index = None
148             for i in range(0, len(table), 2):
149                 index = index + ord(table[i])
150                 if last_index == None:
151                     last_index = index
152                 else:
153                     theLnotab.update({index:tuple([last_index,
154                                                     index-1])})
154                     last_index = index
155             theLnotab.update({len(aCode.co_code)-1:tuple([
156                 last_index,len(aCode.co_code)-1])})
157         return theLnotab
158
159     def __convertTimestamp__(self,aTimestamp):
160         #the timestamp is converted to long
161         return long(aTimestamp*1000000000)
162
163     def __depthFrame__(self, aFrame):
164         theBackFrame = aFrame.f_back
165         if theBackFrame.f_locals.has_key('__depth__'):
166             theCurrentDepth = theBackFrame.f_locals['__depth__']
167             aFrame.f_locals['__depth__'] = theCurrentDepth + 1
168         else:
169             aFrame.f_locals['__depth__'] = 1
170         return aFrame.f_locals['__depth__']
171
172     def __functionCall__(self,
173                         aCode,
174                         aFrame,
175                         aDepth,
176                         aCurrentTimestamp,
177                         aParentTimestampFrame,
178                         aThreadId):
179         theObject = self.__getObject__(aCode)
180         theFunctionId = theObject.__getId__()
181         theArgsValue = theObject.__getArgsValues__(aFrame.
182             f_locals)
183         theBackFrame = aFrame.f_back
184         theBackFrameLasti = theBackFrame.f_lasti
185         theBackFrameCode = theBackFrame.f_code
186         theParentId = self.__getObjectId__(theBackFrameCode)
187         theCurrentLasti = aFrame.f_lasti
188         if not self.itsProbe.has_key((theCurrentLasti,
189             theParentId)):
190             theProbeId = self.__registerProbe__(theCurrentLasti
191                 ,
192                 theParentId,
193                 aFrame.f_lineno
194             )
195         else:
196             theProbeId = self.itsProbe[(theCurrentLasti,
197                 theParentId)]
198         #preguntamos si el argumento es del tipo string
199         #y lo registramos debidamente

```

```

194         for theValue in theArgsValue:
195             if type(theValue) == types.StringType:
196                 if not id(theValue) in ht.itsRegisterObjects:
197                     self.__registerObject__(theValue,
198                                     aCurrentTimestamp)
199
200         self.itsPacker.reset()
201         self.itsPacker.pack_int(self.itsEvents['call'])
202         self.itsPacker.pack_int(self.itsObjects['function'])
203         self.itsPacker.pack_int(theFunctionId)
204         self.itsPacker.pack_int(len(theArgsValue))
205         thePrintArg = " "
206         for theValue in theArgsValue:
207             theDataType = self.__getDataType__(theValue)
208             self.itsPacker.pack_int(theDataType)
209             thePrintArg += str(theDataType)
210             thePrintArg += " "
211             if theDataType == 1:
212                 self.itsPacker.pack_int(id(theValue))
213                 thePrintArg += str(id(theValue))
214                 thePrintArg += " "
215             else:
216                 thePrintArg += str(self.__packValue__(
217                     theDataType, theValue))
218                 thePrintArg += " "
219         self.itsPacker.pack_int(theProbeId)
220         self.itsPacker.pack_hyper(aParentTimestampFrame)
221         self.itsPacker.pack_int(aDepth)
222         self.itsPacker.pack_hyper(aCurrentTimestamp)
223         self.itsPacker.pack_int(aThreadId)
224         if self.FLAG_DEBUGG:
225             print self.itsEvents['call'],
226             print self.itsObjects['function'],
227             print theFunctionId,
228             print len(theArgsValue),
229             print thePrintArg,
230             print theProbeId,
231             print aParentTimestampFrame,
232             print aDepth,
233             print aCurrentTimestamp,
234             print aThreadId
235             raw_input()
236         try:
237             self.itsSocket.sendall(self.itsPacker.get_buffer())
238         except:
239             print 'TODAY+FFFFD durmiendo :-( - Llamado a funcion'
240                 , aCode.co_name
241
242     def __getArgs__(self, aCode):
243         return aCode.co_varnames[:aCode.co_argcount]
244
245     def __getClassKey__(self, aNameClass):
246         for theKey, theValue in self.itsClass.iteritems():
247             if theKey.co_name == aNameClass:
248                 return theKey
249         return None
250
251     def __getObjectId__(self, aCode):
252         if self.__isClassKey__(aCode):
253             return self.itsClass[aCode].__getId__()

```



```

250         elif self.__isFunctionKey__(aCode):
251             return self.itsFunction[aCode].__getId__()
252         elif self.__isMethodKey__(aCode):
253             return self.itsMethod[aCode].__getId__()
254         return -1
255
256     def __getObject__(self, aCode):
257         if self.__isFunctionKey__(aCode):
258             return self.itsFunction[aCode]
259         elif self.__isMethodKey__(aCode):
260             return self.itsMethod[aCode]
261         return None
262
263     def __getThreadId__(self, aThreadSysId):
264         if not hT.itsThread.has_key(aThreadSysId):
265             theThreadId = self.__registerThread__(aThreadSysId)
266         else:
267             theThreadId = self.itsThread[aThreadSysId]
268         return theThreadId
269
270     def __getpartcode__(self, aCode, aLimits):
271         theLower = aLimits[0]
272         theUpper = aLimits[1]
273         theCode = aCode.co_code
274         theStoreFast = {}
275         while theLower < theUpper:
276             theOp = ord(theCode[theLower])
277             theNameOp = dis.opname[theOp]
278             theLower = theLower + 1
279             if theOp >= dis.HAVE_ARGUMENT:
280                 theValue = ord(theCode[theLower]) + ord(theCode
281                     [theLower+1])*256
282                 theLower = theLower + 2
283                 if theOp in dis.haslocal and theNameOp == '
                STORE_FAST':
284                     theArgumentValue = aCode.co_varnames[
285                         theValue]
286                     theStoreFast.update({theArgumentValue:
287                         theValue})
288             return theStoreFast
289
290     def __getDepthFrame__(self, aFrame):
291         try:
292             return aFrame.f_locals['__depth__']
293         except:
294             return -1
295
296     def __getDataType__(self, aValue):
297         theDataType = 8
298         try:
299             if self.itsDataTypes.has_key(aValue.__class__.
300                 __name__):
301                 theDataType = self.itsDataTypes[aValue.
302                     __class__.__name__]
303             except:
304                 return theDataType
305             finally:
306                 return theDataType

```

```

303     def __getTimestampFrame__(self, aFrame):
304         if aFrame.f_locals.has_key('__timestampFrame__'):
305             return aFrame.f_locals['__timestampFrame__']
306         return 0
307
308     def __getTimestampParentFrame__(self, aFrame):
309         theBackFrame = aFrame.f_back
310         if theBackFrame.f_locals.has_key('__timestampFrame__'):
311             return theBackFrame.f_locals['__timestampFrame__']
312         return 0
313
314     def __inClass__(self, aClass):
315         if self.itsClass.has_key(aClass):
316             return True
317         return False
318
319     def __inFunction__(self, aFunction):
320         if self.itsFunction.has_key(aFunction):
321             return True
322         return False
323
324     def __inMethod__(self, aMethod):
325         if self.itsMethod.has_key(aMethod):
326             return True
327         return False
328
329     def __isClassKey__(self, aClassCode):
330         for theKey in self.itsClass.iterkeys():
331             if theKey == aClassCode:
332                 return self.itsClass[theKey]
333         return None
334
335     def __isFunctionKey__(self, aFunctionCode):
336         for theKey in self.itsFunction.iterkeys():
337             if theKey == aFunctionCode:
338                 return self.itsFunction[theKey]
339         return None
340
341     def __isMethodKey__(self, aMethodCode):
342         for theKey in self.itsMethod.iterkeys():
343             if theKey == aMethodCode:
344                 return self.itsMethod[theKey]
345         return None
346
347     def __instantiation__(self,
348                          aCode,
349                          aFrame,
350                          aInstantiationId,
351                          aDepth,
352                          aCurrentTimestamp,
353                          aParentTimestampFrame,
354                          aThreadId):
355         theBehavior = self.__getObject__(aCode)
356         theBehaviorId = theBehavior.__getId__()
357         theClassId = theBehavior.__getTarget__()
358         theArgsValue = theBehavior.__getArgsValues__(aFrame.f_locals)
359         theBackFrame = aFrame.f_back
360         theFrameLasti = theBackFrame.f_lasti

```

```

361         theBackFrameCode = theBackFrame.f_code
362         theParentId = self.__getObjectId__(theBackFrameCode)
363         theCurrentLasti = aFrame.f_lasti
364         if not self.itsProbe.has_key((theCurrentLasti,
365                                     theParentId)):
366             theProbeId = self.__registerProbe__(theCurrentLasti
367                                                 ,
368                                                 theParentId,
369                                                 aFrame.f_lineno
370                                                 )
371
372         else:
373             theProbeId = self.itsProbe[(theCurrentLasti,
374                                         theParentId)]
375
376         #preguntamos si el argumento es del tipo string
377         #y lo registramos debidamente
378         for theValue in theArgsValue:
379             if type(theValue) == types.StringType:
380                 if not id(theValue) in hT.itsRegisterObjects:
381                     self.__registerObject__(theValue,
382                                             aCurrentTimestamp)
383
384         self.itsPacker.reset()
385         self.itsPacker.pack_int(self.itsEvents['instantiation'
386         ])
387         self.itsPacker.pack_int(theBehaviorId)
388         self.itsPacker.pack_int(aInstantiationId)
389         self.itsPacker.pack_int(len(theArgsValue))
390         thePrintArg = " "
391         for theValue in theArgsValue:
392             theDataType = self.__getDataType__(theValue)
393             self.itsPacker.pack_int(theDataType)
394             thePrintArg += str(theDataType)
395             thePrintArg += " "
396             if theDataType == 1:
397                 self.itsPacker.pack_int(id(theValue))
398                 thePrintArg += str(id(theValue))
399                 thePrintArg += " "
400             else:
401                 thePrintArg += str(self.__packValue__(
402                     theDataType, theValue))
403                 thePrintArg += " "
404         self.itsPacker.pack_int(theProbeId)
405         self.itsPacker.pack_hyper(aParentTimestampFrame)
406         self.itsPacker.pack_int(aDepth)
407         self.itsPacker.pack_hyper(aCurrentTimestamp)
408         self.itsPacker.pack_int(aThreadId)
409         if self.FLAG_DEBUGG:
410             print self.itsEvents['instantiation'],
411             print aInstantiationId,
412             print len(theArgsValue),
413             print thePrintArg,
414             print theProbeId,
415             print aParentTimestampFrame,
416             print aDepth,
417             print aCurrentTimestamp,
418             print aThreadId
419         raw_input()
420
421         try:
422             self.itsSocket.sendall(self.itsPacker.get_buffer())
423         except:

```

```

413         print 'Todavía durmiendo :-( - Instancia de ' + aCode.co_name
414
415     def __localWrite__(self,
416                        aCode,
417                        aBytecodeLocal,
418                        aLocals,
419                        aObject,
420                        aCurrentLasti,
421                        aCurrentLineno,
422                        aDepth,
423                        aParentTimestampFrame,
424                        aThreadId):
425         theLocalVariables = aObject.__getLocals__()
426         theBehaviorId = self.__getObjectId__(aCode)
427         theDepth = aDepth + 1
428         for theValue in aBytecodeLocal.iterkeys():
429             if not theLocalVariables.has_key(theValue) or \
430                not aLocals.has_key(theValue):
431                 return
432             if not self.itsProbe.has_key((aCurrentLasti,
433                                           theBehaviorId)):
434                 theProbeId = self.__registerProbe__(
435                                     aCurrentLasti,
436                                     theBehaviorId,
437                                     aCurrentLineno
438                                 )
439             else:
440                 theProbeId = self.itsProbe[(aCurrentLasti,
441                                             theBehaviorId)]
442             self.itsPacker.reset()
443             self.itsPacker.pack_int(self.itsEvents['set'])
444             self.itsPacker.pack_int(self.itsObjects['local'])
445             self.itsPacker.pack_int(theLocalVariables[theValue])
446             self.itsPacker.pack_int(theBehaviorId)
447             theDataType = self.__getDataType__(aLocals[theValue])
448             self.itsPacker.pack_int(theDataType)
449             thePackValue = self.__packValue__(theDataType,
450                                               aLocals[theValue])
451             self.itsPacker.pack_int(theProbeId)
452             self.itsPacker.pack_hyper(aParentTimestampFrame)
453             self.itsPacker.pack_int(theDepth)
454             theCurrentTimestamp = self.__convertTimestamp__(
455                 time.time())
456             self.itsPacker.pack_hyper(theCurrentTimestamp)
457             self.itsPacker.pack_int(aThreadId)
458             if self.FLAG_DEBUGG:
459                 print self.itsEvents['set'],
460                 print self.itsObjects['local'],
461                 print theLocalVariables[theValue],
462                 print theBehaviorId,
463                 print theDataType,
464                 print thePackValue,
465                 print theProbeId,
466                 print aParentTimestampFrame,
467                 print theDepth,

```

```

462         print theCurrentTimestamp,
463         print aThreadId
464         raw_input()
465     try:
466         self.itsSocket.sendall(self.itsPacker.
                                get_buffer())
467     except:
468         print 'TODO: Durmiendo :-( - Local write',
              aCode.co_name
469
470 def __markTimestampFrame__(self, aFrame):
471     if not aFrame.f_locals.has_key('__timestampFrame__'):
472         aFrame.f_locals['__timestampFrame__'] = self.
            __convertTimestamp__(
473                                     time
                                     .
                                     time
                                     ()
                                     )
474     return
475
476 def __methodCall__(self,
477                    aCode,
478                    aFrame,
479                    aTargetId,
480                    aDepth,
481                    aCurrentTimestamp,
482                    aParentTimestampFrame,
483                    aThreadId):
484     theObject = self.__getObject__(aCode)
485     theMethodId = theObject.__getId__()
486     #classId = theObject.__getTarget__()
487     theArgsValue = theObject.__getArgsValues__(aFrame.
        f_locals)
488     theBackFrame = aFrame.f_back
489     theBackFrameLasti = theBackFrame.f_lasti
490     theBackFrameCode = theBackFrame.f_code
491     theParentId = self.__getObjectId__(theBackFrameCode)
492     theCurrentLasti = aFrame.f_lasti
493     if not self.itsProbe.has_key((theCurrentLasti,
        theParentId)):
494         theProbeId = self.__registerProbe__(theCurrentLasti
        ,
495                                             theParentId,
496                                             aFrame.f_lineno
        )
497     else:
498         theProbeId = self.itsProbe[(theCurrentLasti,
        theParentId)]
499     #preguntamos si el argumento es del tipo string
500     #y lo registramos debidamente
501     for theValue in theArgsValue:
502         if type(theValue) == types.StringType:
503             if not id(theValue) in hT.itsRegisterObjects:
504                 self.__registerObject__(theValue,
                    aCurrentTimestamp)
505     self.itsPacker.reset()
506     self.itsPacker.pack_int(self.itsEvents['call'])
```

```

507         self.itsPacker.pack_int(self.itsObjects['method'])
508         self.itsPacker.pack_int(theMethodId)
509         self.itsPacker.pack_int(aTargetId)
510         self.itsPacker.pack_int(len(theArgsValue))
511         thePrintArg = " "
512         for theValue in theArgsValue:
513             theDataType = self.__getDataType__(theValue)
514             self.itsPacker.pack_int(theDataType)
515             thePrintArg += str(theDataType)
516             thePrintArg += " "
517             if theDataType == 1:
518                 self.itsPacker.pack_int(id(theValue))
519                 thePrintArg += str(id(theValue))
520                 thePrintArg += " "
521             else:
522                 thePrintArg += str(self.__packValue__(
523                     theDataType, theValue))
524                 thePrintArg += " "
525         self.itsPacker.pack_int(theProbeId)
526         self.itsPacker.pack_hyper(aParentTimestampFrame)
527         self.itsPacker.pack_int(aDepth)
528         self.itsPacker.pack_hyper(aCurrentTimestamp)
529         self.itsPacker.pack_int(aThreadId)
530         if self.FLAG_DEBUGG:
531             print self.itsEvents['call'],
532             print self.itsObjects['method'],
533             print theMethodId,
534             print aTargetId,
535             print len(theArgsValue),
536             print thePrintArg,
537             print theProbeId,
538             print aParentTimestampFrame,
539             print aDepth,
540             print aCurrentTimestamp,
541             print aThreadId
542             raw_input()
543         try:
544             self.itsSocket.sendall(self.itsPacker.get_buffer())
545         except:
546             print 'TODO: Durmiendo :-( - Call metodo',
547                   aCode.co_name
548
549     def __packValue__(self, aDataType, aValue):
550         if self.itsPackXDR.has_key(aDataType):
551             theMethodName = self.itsPackXDR[aDataType]
552             #bool no funciona bien con XDRLib entre java y
553             python
554             if aDataType == 4:
555                 if aValue:
556                     aValue = 1
557                 else:
558                     aValue = 0
559             #trick for string objects (it's send the Id)
560             if aDataType == 1:
561                 self.itsPacker.pack_int(id(aValue))
562                 return id(aValue)
563             getattr(self.itsPacker, 'pack_%s' % theMethodName)(
564                 aValue)

```

```

562         return aValue
563     else:
564         #en estos momentos en U+FFFD es el tipo de dato
565         #TODO: debieramos en U+FFFD el id del objeto
566         self.itsPacker.pack_int(aDataType)
567         return aDataType
568
569     def __printHunter__(self):
570         #cerrar socket
571         #TODO: encontrar una manera mejor de hacer esto
572         self.itsSocket.close()
573         print
574         print 'clases'
575         for theKey, theValue in hT.itsClass.iteritems():
576             print theValue.__dict__
577             print
578         print '======'
579
580         print 'metodos'
581         for theKey, theValue in hT.itsMethod.iteritems():
582             print v.__dict__
583             print
584         print '======'
585
586         print 'funcion'
587         for theKey, theValue in hT.itsFunction.iteritems():
588             print theValue.__dict__
589             print
590         print '======'
591
592     def __socketConnect__(self):
593         self.itsSocket = socket.socket(socket.AF_INET,socket.
594             SOCK_STREAM)
595         try:
596             self.itsSocket.connect((self.itsHost, self.itsPort)
597                 )
598         except:
599             print "TOD, esta durmiendo :("
600
601     def __register__(self, aObject, aLocals):
602         aObject.__registerLocals__(aLocals)
603
604     def __registerClass__(self, aCode, aLocals,
605         aParentTimestamp, aDepth, aFrameLineNo):
606         theClassId = self.itsId.__get__()
607         theClassName = aCode.co_name
608         #HINT: ver como recuperar las herencias de esta clase
609         theClassBases = None
610         self.itsPacker.reset()
611         self.itsPacker.pack_int(self.itsEvents['register'])
612         self.itsPacker.pack_int(self.itsObjects['class'])
613         self.itsPacker.pack_int(theClassId)
614         self.itsPacker.pack_string(theClassName)
615         self.itsPacker.pack_int(0)
616         if self.FLAG_DEBUGG:
617             print self.itsEvents['register'],
618             print self.itsObjects['class'],
619             print theClassId,
620             print theClassName,

```

```

618         print theClassBases
619         raw_input()
620     try:
621         self.itsSocket.sendall(self.itsPacker.get_buffer())
622     except:
623         print 'TODO: HURMIENDO :-( - Registrando Clase'
624         , aCode.co_name
625         theObjectClass = self.__addClass__(
626             theClassId,
627             self.
628                 __createInotab__(
629                     aCode),
630                     aCode)
631
632         self.itsId.__next__()
633         theObjectClass.__addMethod__(aCode,aLocals)
634         theObjectClass.__register_set_StaticField__(aLocals,
635                                                     aFrameLineNo
636                                                     ,
637                                                     aParentTimestamp
638                                                     ,
639                                                     aDepth,
640                                                     aCode.
641                                                         co_filename
642                                                         )
643
644     def __registerException__(self,
645                             aFrame,
646                             aArg,
647                             aDepth,
648                             aParentTimestampFrame,
649                             aThreadId):
650         theBackFrame = aFrame.f_back
651         theBackFrameCode = theBackFrame.f_code
652         theParentId = self.__getObjectId__(theBackFrameCode)
653         #behaviorId = self.__getObjectId__(aFrame.f_code)
654         #theParentId = self.__getObjectId__(aFrame.f_code)
655         theCurrentLasti = aFrame.f_lasti
656         theDepth = aDepth + 1
657         if not self.itsProbe.has_key((theCurrentLasti,
658                                     theParentId)):
659             theProbeId = self.__registerProbe__(theCurrentLasti
660                                                 ,
661                                                 theParentId,
662                                                 aFrame.f_lineno
663                                                 )
664
665         else:
666             theProbeId = self.itsProbe[(theCurrentLasti,
667                                         theParentId)]
668         theCurrentTimestamp = self.__convertTimestamp__(time.
669                                                         time())
670         if not id(aArg) in self.itsObjects:
671             self.__registerObject__(aArg,theCurrentTimestamp)
672         self.itsPacker.reset()
673         self.itsPacker.pack_int(self.itsEvents['register'])
674         self.itsPacker.pack_int(self.itsObjects['exception'])
675         theDataType = self.__getDataType__(aArg)
676         self.itsPacker.pack_int(theDataType)
677         self.itsPacker.pack_int(id(aArg))
678         self.itsPacker.pack_int(theProbeId)

```



```

665         self.itsPacker.pack_hyper(aParentTimestampFrame)
666         self.itsPacker.pack_int(theDepth)
667         self.itsPacker.pack_hyper(theCurrentTimestamp)
668         self.itsPacker.pack_int(aThreadId)
669         if self.FLAG_DEBUGG:
670             print self.itsEvents['register'],
671             print self.itsObjects['exception'],
672             print theDataType,
673             print id(aArg),
674             print theProbeId,
675             print aParentTimestampFrame,
676             print theDepth,
677             print theCurrentTimestamp,
678             print aThreadId
679         raw_input()
680     try:
681         self.itsSocket.sendall(self.itsPacker.get_buffer())
682     except:
683         print 'TODO: Elurmiendo :-( - Salida de behavior
        ', aFrame.f_code.co_name
684
685     def __registerFunction__(self, aCode):
686         theFunctionId = self.itsId.__get__()
687         aArgs = self.__getArgs__(aCode)
688         theLineNumbers = 0
689         self.itsPacker.reset()
690         self.itsPacker.pack_int(self.itsEvents['register'])
691         self.itsPacker.pack_int(self.itsObjects['function'])
692         self.itsPacker.pack_int(theFunctionId)
693         self.itsPacker.pack_string(aCode.co_name)
694         self.itsPacker.pack_int(len(aArgs))
695         thePrintArg = " "
696         for theValue in range(len(aArgs)):
697             if not aArgs[theValue] == 'self':
698                 thePrintArg += str(aArgs[theValue])
699                 thePrintArg += " "
700                 self.itsPacker.pack_string(aArgs[theValue])
701                 thePrintArg += str(theValue)
702                 thePrintArg += " "
703                 self.itsPacker.pack_int(theValue)
704         self.itsPacker.pack_string(aCode.co_filename)
705         #agregamos setup del módulo para que el plugin
706         #funcione correctamente
707         #de seguro que esto se puede hacer mejor
708         for theTuple in dis.findlinestarts(aCode):
709             theLineNumbers += 1
710         self.itsPacker.pack_int(len(aCode.co_code))
711         self.itsPacker.pack_int(theLineNumbers)
712         thePrintLines = " "
713         for theStartPc, theLineNumber in dis.findlinestarts(
            aCode):
714             thePrintArg += str(theStartPc)
715             thePrintArg += " "
716             self.itsPacker.pack_int(theStartPc)
717             thePrintArg += str(theLineNumber)
718             thePrintArg += " "
719             self.itsPacker.pack_int(theLineNumber)
720         if self.FLAG_DEBUGG:
721             #if True:

```

```

722         print self.itsEvents['register'],
723         print self.itsObjects['function'],
724         print theFunctionId,
725         print aCode.co_name,
726         print len(aArgs)-1,
727         print thePrintArg,
728         print aCode.co_filename
729         print len(aCode.co_code)
730         print theLineNumbers
731         print thePrintLines
732         raw_input()
733     try:
734         self.itsSocket.sendall(self.itsPacker.get_buffer())
735     except:
736         print 'TODO: [U+FFFF] Durmiendo :-( - Registrando
              Funcion', aCode.co_name
737     self.__addFunction__(
738         theFunctionId,
739         self.__createInotab__(aCode),
740         aCode,
741         aArgs)
742     self.itsId.__next__()
743
744
745     def __registerMethod__(self, aCode, aMethodId, aClassId,
746                           aArgs):
747         theLineNumbers = 0
748         self.itsPacker.reset()
749         self.itsPacker.pack_int(self.itsEvents['register'])
750         self.itsPacker.pack_int(self.itsObjects['method'])
751         self.itsPacker.pack_int(aMethodId)
752         self.itsPacker.pack_int(aClassId)
753         self.itsPacker.pack_string(aCode.co_name)
754         #argumento viene con self, se le debe restar uno a la
755         #cantidad de
756         #elementos
757         self.itsPacker.pack_int(len(aArgs)-1)
758         thePrintArg = " "
759         for theValue in range(len(aArgs)):
760             if not aArgs[theValue] == 'self':
761                 thePrintArg += str(aArgs[theValue])
762                 thePrintArg += " "
763                 self.itsPacker.pack_string(aArgs[theValue])
764                 thePrintArg += str(theValue)
765                 thePrintArg += " "
766                 self.itsPacker.pack_int(theValue)
767         self.itsPacker.pack_string(aCode.co_filename)
768         #agregamos setup del [U+FFFF] para que el plugin
769         #funcione correctamente
770         #de seguro que esto se puede hacer mejor
771         for theTuple in dis.findlinestarts(aCode):
772             theLineNumbers += 1
773             self.itsPacker.pack_int(len(aCode.co_code))
774             self.itsPacker.pack_int(theLineNumbers)
775             thePrintLines = " "
776             for theStartPc, theLineNumber in dis.findlinestarts(
              aCode):
              thePrintArg += str(theStartPc)
              thePrintArg += " "

```

```

777         self.itsPacker.pack_int(theStartPc)
778         thePrintArg += str(theLineNumber)
779         thePrintArg += " "
780         self.itsPacker.pack_int(theLineNumber)
781     if self.FLAG_DEBUGG:
782         print self.itsEvents['register'],
783         print self.itsObjects['method'],
784         print aMethodId,
785         print aClassId,
786         print aCode.co_name,
787         print len(aArgs)-1,
788         print thePrintArg,
789         print aCode.co_filename
790         print len(aCode.co_code)
791         print theLineNumbers
792         print thePrintLines
793         raw_input()
794     try:
795         self.itsSocket.sendall(self.itsPacker.get_buffer())
796     except:
797         print 'TODO: Durmiendo :-( Registrando Metodo',
798               aCode.co_name
799     self.__addMethod__(
800         aMethodId,
801         self.__createNotab__(aCode),
802         aCode,
803         aClassId,
804         aArgs)
805
806     def __registerSpecialMethod__(self, aName, aMethodId,
807                                  aClassId, aFileName):
808         self.itsPacker.reset()
809         self.itsPacker.pack_int(self.itsEvents['register'])
810         self.itsPacker.pack_int(self.itsObjects['specialMethod'])
811         self.itsPacker.pack_int(aMethodId)
812         self.itsPacker.pack_int(aClassId)
813         self.itsPacker.pack_string(aName)
814         self.itsPacker.pack_string(aFileName)
815         #agregamos setup del método para que el plugin
816         #funcione correctamente
817         #de seguro que esto se puede hacer mejor
818     if self.FLAG_DEBUGG:
819         print self.itsEvents['register'],
820         print self.itsObjects['specialMethod'],
821         print aMethodId,
822         print aClassId,
823         print aName,
824         print aFileName
825         raw_input()
826     try:
827         self.itsSocket.sendall(self.itsPacker.get_buffer())
828     except:
829         print 'TODO: Durmiendo :-( Registrando Metodo',
830               aName
831         """
832     self.__addMethod__(
833         aMethodId,

```

```

832         self.__createlnTAB__(aCode),
833         aCode,
834         aClassId,
835         aArgs)
836     """
837
838     def __registerObject__(self, aValue, aCurrentTimestamp):
839         self.itsRegisterObjects.append(id(aValue))
840         self.itsPacker.reset()
841         self.itsPacker.pack_int(self.itsEvents['register'])
842         self.itsPacker.pack_int(self.itsObjects['object'])
843         #enviar el tipo de dato por mientras solo es string
844         self.itsPacker.pack_int(1) #string
845         self.itsPacker.pack_int(id(aValue))
846         self.itsPacker.pack_string(aValue)
847         self.itsPacker.pack_hyper(aCurrentTimestamp)
848         if self.FLAG_DEBUGG:
849             print self.itsEvents['register'],
850             print self.itsObjects['object'],
851             print 1,
852             print id(aValue),
853             print aValue,
854             print aCurrentTimestamp
855             raw_input()
856         try:
857             self.itsSocket.sendall(self.itsPacker.get_buffer())
858         except:
859             print 'TODO:U+FFFDDurmiendo :-( - Registrando Probe'
860
861     def __registerProbe__(self, aCurrentLasti, aBehaviorId,
862                          aCurrentLineno):
863         theProbeId = self.itsProbeId.__get__()
864         self.__addProbe__(theProbeId,aCurrentLasti,aBehaviorId)
865         self.itsPacker.reset()
866         self.itsPacker.pack_int(self.itsEvents['register'])
867         self.itsPacker.pack_int(self.itsObjects['probe'])
868         self.itsPacker.pack_int(theProbeId)
869         self.itsPacker.pack_int(aBehaviorId)
870         self.itsPacker.pack_int(aCurrentLasti)
871         self.itsPacker.pack_int(aCurrentLineno)
872         if self.FLAG_DEBUGG:
873             print self.itsEvents['register'],
874             print self.itsObjects['probe'],
875             print theProbeId,
876             print aBehaviorId,
877             print aCurrentLasti,
878             print aCurrentLineno
879             raw_input()
880         try:
881             self.itsSocket.sendall(self.itsPacker.get_buffer())
882         except:
883             print 'TODO:U+FFFDDurmiendo :-( - Registrando Probe'
884         self.itsProbeId.__next__()
885         return theProbeId
886
887     def __registerThread__(self, aThreadSysId):
888         theThreadId = self.itsThreadId.__get__()
889         self.__addThread__(theThreadId,aThreadSysId)
890         self.itsPacker.reset()

```

```

890         self.itsPacker.pack_int(self.itsEvents['register'])
891         self.itsPacker.pack_int(self.itsObjects['thread'])
892         self.itsPacker.pack_int(theThreadId)
893         self.itsPacker.pack_int(aThreadSysId)
894         if self.FLAG_DEBUGG:
895             print self.itsEvents['register'],
896             print self.itsObjects['thread'],
897             print theThreadId,
898             print aThreadSysId
899             raw_input()
900         try:
901             self.itsSocket.sendall(self.itsPacker.get_buffer())
902         except:
903             print 'TODO: %U+FFFF Durmiendo :-( - Registrando Thread
904             ,
905             self.itsThreadId.__next__()
906             return theThreadId
907
908     def __trace__(self, aFrame, aEvent, aArg):
909         if aFrame.f_back == None:
910             sys.settrace(None)
911             return
912         theCode = aFrame.f_code
913         if theCode.co_name == '<module>':
914             return
915         theLocals = aFrame.f_locals
916         theGlobals = aFrame.f_globals
917         theDepth = self.itsCurrentDepth = self.__depthFrame__(
918             aFrame)
919         self.__markTimestampFrame__(aFrame)
920         theThreadId = self.__getThreadId__(thread.get_ident())
921         if aEvent == "call":
922             if re.search(self.itsMethodPattern, theCode.co_name)
923             :
924                 if not theCode.co_name == '__init__':
925                     return
926                 theParentTimestampFrame = self.
927                 __getTimestampParentFrame__(aFrame)
928                 if theCode.co_name == '__init__':
929                     #FIXME: cambio experimental, revizar!!!!!!
930                     """
931                     id = self.itsId.__get__()
932                     if not hasattr(theLocals['self'], '__dict__'):
933                         return
934                     theLocals['self'].__dict__.update({'__pyTOD__':
935                         id})
936                     self.itsId.__next__()
937                     """
938                     #aca se sacan las bases de la clase la cual
939                     #se ha instanciado
940                     #TODO: encontrar una mejor forma de hacerlo
941                     #ineficiente!!..quizas interviniendo la llamada
942                     #de la super clase?
943                     if 'self' in theLocals:
944                         print type(theLocals['self']).__bases__
945                     #si self esta en theLocals estamos en un metodo
946                     if theLocals.has_key('self'):
947                         if not self.__inMethod__(theCode):
948                             theKey = type(theLocals['self']).__name__

```

```

944         theKey = hT.__getClassKey__(theKey)
945         if theKey == None:
946             return
947         if not hT.itsClass.has_key(theKey):
948             return
949         theClassId = hT.itsClass[theKey].__getId__
950             ()
951         if not hT.itsClass[theKey].method.has_key(
952             theCode.co_name):
953             return
954         theMethodId = hT.itsClass[theKey].method[
955             theCode.co_name]
956         theArgs = self.__getArgs__(theCode)
957         self.__registerMethod__(theCode,theMethodId
958             ,theClassId,theArgs)
959         theCurrentTimestamp = aFrame.f_locals['
960             __timestampFrame__']
961         if theCode.co_name == '__init__':
962             Id = self.itsId.__get__()
963             if not hasattr(theLocals['self'],'__dict__'
964                 ):
965                 return
966             theLocals['self'].__dict__.update({'
967                 __pyTOD__':Id})
968             self.itsId.__next__()
969             self.__instantiation__(theCode,
970                 aFrame,
971                 theLocals['self'].
972                     __pyTOD__,
973                 theDepth,
974                 theCurrentTimestamp,
975                 theParentTimestampFrame
976                     ,
977                 theThreadId)
978         else:
979             self.__methodCall__(theCode,
980                 aFrame,
981                 theLocals['self'].
982                     __pyTOD__,
983                 theDepth,
984                 theCurrentTimestamp,
985                 theParentTimestampFrame
986                     ,
987                 theThreadId)
988     else:
989         #verificamos si es una funcion
990         if theGlobals.has_key(theCode.co_name):
991             if inspect.isfunction(theGlobals[theCode.
992                 co_name]):
993                 if not self.__inFunction__(theCode):
994                     self.__registerFunction__(theCode)
995             theCurrentTimestamp = aFrame.f_locals['
996                 __timestampFrame__']
997             self.__functionCall__(theCode,
998                 aFrame,
999                 theDepth,
1000                 theCurrentTimestamp,

```

```

989                                     theParentTimestampFrame
990                                     ,
991                                     theThreadId)
992
993     return self.__trace__
994 elif aEvent == "line":
995     if re.search(self.itsMethodPattern,theCode.co_name)
996     :
997         if not theCode.co_name == '__init__':
998             return
999
1000     theParentTimestampFrame = self.
1001         __getTimestampFrame__(aFrame)
1002     theObject = self.__getObject__(theCode)
1003     if theObject == None:
1004         return
1005     theLnotab = theObject.__getLnotab__()
1006     if theLnotab.has_key(aFrame.f_lasti):
1007         theBytecodeLocals = self.__getpartcode__(
1008             theCode,theLnotab[aFrame.f_lasti])
1009         self.__register__(theObject,theBytecodeLocals)
1010         self.__localWrite__(theCode,
1011             theBytecodeLocals,
1012             theLocals,
1013             theObject,
1014             aFrame.f_lasti,
1015             aFrame.f_lineno,
1016             theDepth,
1017             theParentTimestampFrame,
1018             theThreadId)
1019
1020     return self.__trace__
1021 elif aEvent == "return":
1022     if re.search(self.itsMethodPattern,theCode.co_name)
1023     :
1024         if not theCode.co_name == '__init__':
1025             return
1026
1027     theParentTimestampFrame = self.
1028         __getTimestampFrame__(aFrame)
1029     if theLocals.has_key('__init__'):
1030         if not self.__inClass__(theCode):
1031             #registramos la definicion de la clase
1032             theLocals.update(
1033                 {'__setattr__':Descriptor.__dict__[
1034                     '__setattr__']})
1035             theLocals.update({'__metaclass__':
1036                 MetaDescriptor})
1037             self.__registerClass__(
1038                 theCode,
1039                 theLocals,
1040                 theParentTimestampFrame
1041                 ,
1042                 theDepth,
1043                 aFrame.f_lineno)
1044
1045     else:
1046         theObject = self.__getObject__(theCode)
1047         if theObject == None:
1048             return
1049         if self.FLAG_THROWN == True:
1050             self.FLAG_THROWN = False
1051         return
1052     theLnotab = theObject.__getLnotab__()

```

```

1039         if theLnotab.has_key(aFrame.f_lasti):
1040             theBytecodeLocals = self.__getpartcode__(
1041                                     theCode,
1042                                     theLnotab[
1043                                         aFrame.
1044                                             f_lasti
1045                                             ]
1046                                         )
1047             self.__register__(theObject,
1048                               theBytecodeLocals)
1049             self.__localWrite__(theCode,
1050                                 theBytecodeLocals,
1051                                 theLocals,
1052                                 theObject,
1053                                 aFrame.f_lasti,
1054                                 aFrame.f_lineno,
1055                                 theDepth,
1056                                 theParentTimestampFrame
1057                                 ,
1058                                 theThreadId)
1059             self.__behaviorExit__(aFrame,
1060                                   aArg,
1061                                   theDepth,
1062                                   theParentTimestampFrame,
1063                                   theThreadId,
1064                                   False)
1065         elif aEvent == "exception":
1066             theParentTimestampFrame = self.
1067                 __getTimestampFrame__(aFrame)
1068             if type(aArg[1]) is tuple:
1069                 theArgument = aArg[1][1]
1070             else:
1071                 theArgument = aArg[1]
1072             self.__registerException__(aFrame,
1073                                       theArgument,
1074                                       theDepth,
1075                                       theParentTimestampFrame
1076                                       ,
1077                                       theThreadId)
1078             for theTuple in dis.findlinestarts(theCode):
1079                 if aFrame.f_lineno in theTuple:
1080                     theIndex = theTuple[0]
1081                     theOp = ord(theCode.co_code[theIndex-3])
1082                     theInstruction = dis.opname[theOp]
1083                     if theInstruction == 'SETUP_EXCEPT':
1084                         return self.__trace__
1085             self.__behaviorExit__(aFrame,
1086                                   theArgument,
1087                                   theDepth,
1088                                   theParentTimestampFrame,
1089                                   theThreadId,
1090                                   True)
1091         self.FLAG_THROWN = True
1092
1093     hT = hunterTrace(
1094         generatorId(),
1095         generatorId(),
1096         generatorId(),
1097         xdrlib.Packer(),
1098         '127.0.0.1',

```



```

1091         8058)
1092
1093     class MetaDescriptor(type):
1094         def __setattr__(self, aName, aValue):
1095             import sys
1096             theFrame = sys._getframe()
1097             theCode = theFrame.f_back.f_code
1098             theCurrentLasti = theFrame.f_back.f_lasti
1099             theCurrentDepth = hT.itsCurrentDepth
1100             theCurrentTimestamp = hT.__convertTimestamp__(time.time
1101                 ())
1102             theParentTimestamp = hT.__getTimestampParentFrame__(
1103                 theFrame)
1104             theThreadId = hT.__getThreadId__(thread.get_ident())
1105             theKey = self.__name__
1106             theKey = hT.__getClassKey__(theKey)
1107             if theKey == None:
1108                 return
1109             theObject = hT.itsClass[theKey]
1110             theObjectId = theObject.__getId__()
1111             theBehaviorId = hT.__getObjectId__(theCode)
1112             sys.settrace(None)
1113             theObject.__addStaticField__({aName:-1})
1114             Id = theObject.staticField[aName]
1115             if not hT.itsProbe.has_key((theCurrentLasti,
1116                 theBehaviorId)):
1117                 theProbeId = hT.__registerProbe__(theCurrentLasti,
1118                     theBehaviorId,
1119                     theFrame.f_lineno
1120                 )
1121             else:
1122                 theProbeId = hT.itsProbe[(theCurrentLasti,
1123                     theBehaviorId)]
1124             #preguntar si el valor de U+FFFF registrado y si es un string
1125             if type(aValue) == types.StringType:
1126                 if not id(aValue) in hT.itsRegisterObjects:
1127                     hT.__registerObject__(aValue,
1128                         theCurrentTimestamp)
1129                     hT.itsPacker.reset()
1130                     hT.itsPacker.pack_int(hT.itsEvents['set'])
1131                     hT.itsPacker.pack_int(hT.itsObjects['
1132                         classAttribute'])
1133                     hT.itsPacker.pack_int(Id)
1134                     theDataType = hT.__getDataType__(aValue)
1135                     hT.itsPacker.pack_int(theDataType)
1136                     hT.itsPacker.pack_int(id(aValue))
1137                     hT.itsPacker.pack_int(theProbeId)
1138                     hT.itsPacker.pack_hyper(theParentTimestamp)
1139                     hT.itsPacker.pack_int(theCurrentDepth)
1140                     hT.itsPacker.pack_hyper(theCurrentTimestamp)
1141                     hT.itsPacker.pack_int(theThreadId)
1142                     super(MetaDescriptor, self).__setattr__(aName,
1143                         aValue)
1144                 if hT.FLAG_DEBUGG:
1145                     print hT.itsEvents['set'],
1146                     print hT.itsObjects['classAttribute'],
1147                     print Id,
1148                     print theDataType,

```

```

1141         print id(aValue),
1142         print theProbeId,
1143         print theParentTimestamp,
1144         print theCurrentDepth,
1145         print theCurrentTimestamp,
1146         print theThreadId
1147         raw_input()
1148     try:
1149         hT.itsSocket.sendall(hT.itsPacker.
1150                               get_buffer())
1151         pass
1152     except:
1153         print 'TODO: U+FFFD durmiendo :-(', theCode.
1154             co_name
1155         sys.settrace(hT.__trace__)
1156 else:
1157     hT.itsPacker.reset()
1158     hT.itsPacker.pack_int(hT.itsEvents['set'])
1159     hT.itsPacker.pack_int(hT.itsObjects['classAttribute
1160                             '])
1161     hT.itsPacker.pack_int(Id)
1162     theDataType = hT.__getDataType__(aValue)
1163     hT.itsPacker.pack_int(theDataType)
1164     thePackValue = hT.__packValue__(theDataType, aValue
1165                                     )
1166     hT.itsPacker.pack_int(theProbeId)
1167     hT.itsPacker.pack_hyper(theParentTimestamp)
1168     hT.itsPacker.pack_int(theCurrentDepth)
1169     hT.itsPacker.pack_hyper(theCurrentTimestamp)
1170     hT.itsPacker.pack_int(theThreadId)
1171     super(MetaDescriptor, self).__setattr__(aName,
1172                                              aValue)
1173 if hT.FLAG_DEBUGG:
1174     print hT.itsEvents['set'],
1175     print hT.itsObjects['classAttribute'],
1176     print Id,
1177     print theDataType,
1178     print thePackValue,
1179     print theProbeId,
1180     print aParentTimestamp,
1181     print theCurrentDepth,
1182     print theCurrentTimestamp,
1183     print theThreadId
1184     raw_input()
1185 try:
1186     hT.itsSocket.sendall(hT.itsPacker.get_buffer())
1187 except:
1188     print 'TODO: U+FFFD durmiendo :-(', theCode.
1189         co_name
1190
1191 class Descriptor(object):
1192
1193     def __setattr__(self, aName, aValue):
1194         import sys
1195         theFrame = sys._getframe()
1196         theCode = theFrame.f_back.f_code
1197         theCurrentLasti = theFrame.f_back.f_lasti

```

```

1193         theCurrentDepth = hT.__getDepthFrame__(theFrame.f_back)
1194             + 1
1195         theCurrentTimestamp = hT.__convertTimestamp__(time.time
1196             ())
1197         theParentTimestamp = hT.__getTimestampParentFrame__(
1198             theFrame)
1199         theThreadId = hT.__getThreadId__(thread.get_ident())
1200         theKey = type(self).__name__
1201         theKey = hT.__getClassKey__(theKey)
1202         if theKey == None:
1203             return
1204         theObject = hT.itsClass[theKey]
1205         theObjectId = theObject.__getId__()
1206         theBehaviorId = hT.__getObjectId__(theCode)
1207         sys.settrace(None)
1208         theObject.__addAttribute__(aName, theObjectId)
1209         Id = theObject.attributes[aName]
1210         if not hT.itsProbe.has_key((theCurrentLasti,
1211             theBehaviorId)):
1212             theProbeId = hT.__registerProbe__(theCurrentLasti,
1213                 theBehaviorId,
1214                 theFrame.f_lineno
1215             )
1216         else:
1217             theProbeId = hT.itsProbe[(theCurrentLasti,
1218                 theBehaviorId)]
1219         if type(aValue) == types.StringType:
1220             if not id(aValue) in hT.itsRegisterObjects:
1221                 hT.__registerObject__(aValue,
1222                     theCurrentTimestamp)
1223                 hT.itsPacker.pack_int(hT.itsEvents['set'])
1224                 hT.itsPacker.pack_int(hT.itsObjects['attribute'
1225                     ])
1226                 hT.itsPacker.pack_int(Id)
1227                 hT.itsPacker.pack_int(self.__pyTOD__)
1228                 theDataType = hT.__getDataType__(aValue)
1229                 hT.itsPacker.pack_int(theDataType)
1230                 hT.itsPacker.pack_int(id(aValue))
1231                 hT.itsPacker.pack_int(theProbeId)
1232                 hT.itsPacker.pack_hyper(theParentTimestamp)
1233                 hT.itsPacker.pack_int(theCurrentDepth)
1234                 hT.itsPacker.pack_hyper(theCurrentTimestamp)
1235                 hT.itsPacker.pack_int(theThreadId)
1236                 object.__setattr__(self, aName, aValue)
1237                 if hT.FLAG_DEBUGG:
1238                     print hT.itsEvents['set'],
1239                     print hT.itsObjects['attribute'],
1240                     print Id,
1241                     print theBehaviorId,
1242                     print theDataType,
1243                     print id(aValue),
1244                     print theProbeId,
1245                     print theParentTimestamp,
1246                     print theCurrentDepth,
1247                     print theCurrentTimestamp,
1248                     print theThreadId
1249                 raw_input()
1250             try:

```

```

1243         hT.itsSocket.sendall(hT.itsPacker.
1244                               get_buffer())
1245     except:
1246         print 'TODAY U+FFFD durmiendo :-(', theCode.
1247             co_name
1248         sys.settrace(hT.__trace__)
1249     else:
1250         hT.itsPacker.reset()
1251         hT.itsPacker.pack_int(hT.itsEvents['set'])
1252         hT.itsPacker.pack_int(hT.itsObjects['attribute'])
1253         hT.itsPacker.pack_int(Id)
1254         hT.itsPacker.pack_int(self.__pyTOD__)
1255         theDataType = hT.__getDataType__(aValue)
1256         hT.itsPacker.pack_int(theDataType)
1257         thePackValue = hT.__packValue__(theDataType, aValue
1258                                         )
1259         hT.itsPacker.pack_int(theProbeId)
1260         hT.itsPacker.pack_hyper(theParentTimestamp)
1261         hT.itsPacker.pack_int(theCurrentDepth)
1262         hT.itsPacker.pack_hyper(theCurrentTimestamp)
1263         hT.itsPacker.pack_int(theThreadId)
1264         object.__setattr__(self, aName, aValue)
1265     if hT.FLAG_DEBUGG:
1266         print hT.itsEvents['set'],
1267         print hT.itsObjects['attribute'],
1268         print Id,
1269         print theBehaviorId,
1270         print theDataType,
1271         print thePackValue,
1272         print theProbeId,
1273         print theParentTimestamp,
1274         print theCurrentDepth,
1275         print theCurrentTimestamp,
1276         print theThreadId
1277     raw_input()
1278     try:
1279         hT.itsSocket.sendall(hT.itsPacker.get_buffer())
1280     except:
1281         print 'TODAY U+FFFD durmiendo :-(', theCode.
1282             co_name
1283         sys.settrace(hT.__trace__)
1284
1285 if th:
1286     settrace(hT.__trace__)
1287 sys.settrace(hT.__trace__)

```