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 desarrollo de software, considerándolo en términos de tiempo y costo. Desafortunadamente los depuradores en muchos ambientes de desarrollo sólo entregan una asistencia mínima, esto contribuye a que la depuración sea algo tedioso y una tarea que consume mucho tiempo.

Existen dos métodos tradicionales para hacer depuración: log-based (basado en mensaje) y break point-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 pero, (a) requiere tremendas modificaciones del código fuente, y (b) esta técnica no es escalar, debido a que el análisis manual de grandes huellas es complicado. El segundo método consiste en correr el programa bajo un depurador dedicado cuando el programador permita pausar la ejecución en un punto determinado, examinando el contenido de memoria, y pudiendo continuar la ejecución paso a paso. A pesar de lo anterior, breakpoint-based debugging es limitado: cuando la ejecución es pausada, la información sobre el estado y actividad anterior del programa es limitada por la introspección[1] de la pila de llamada actual.

Los depuradores omniscientes, también conocidos como atrás-en-el-tiempo o después-de-la-muerte, superan todas estas características. 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. Este acercamiento combina las ventajas de la depuración basada en log — la actividad pasada no se pierde nunca — y también la de depuración basada en break point – 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 preguntras / respuestas que podrían de una u otra forma exigir un esfuerzo significante 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 nombradas poseen una característica única y es la de identificar la causa inicial de los bugs[2]

Existen varias implementaciones de los depuradores omniscientes, pero basaremos nuestro caso de estudio en TOD[3], 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
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 quien realizará depuración omnisciente a scripts escritos en un lenguaje de tipado dinámico llama Python, basándose en el estudio, análisis y utilización de TOD.

[1]Capacidad de un programa de razonar sobre su propia estructura

[2]Defecto de software

[3]Trace-Oriented debugger implementado en el lenguaje de programación Java(lenguaje de tipado estático)

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El proceso de depuración de software cada vez se hace 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 defectos de software. Si el programador no cuenta con un ambiente adecuado para poder inspeccionar su software y corregir ciertas anomalías el costo y tiempo del desarrollo de software aumenta dramáticamente [Paper: A study of the effect of imperfect debugging on software development cost] En muchos casos los softwares desarrollados no son expuestos a pruebas exhaustivas para conocer su verdadero comportamiento en diversos ambientes, debido a lo costoso de esto, lo cual conlleva a tener comportamientos inesperados en el tiempo de producción de estos.

El proceso anterior se define como la manera de identificar y eliminar errores dentro de un programa de 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 software evite que los programadores tengan que introducir lineas adicionales en sus programas para intentar saber el comportamiento de ciertos componentes de su software. Además es importante que este ambiente logre el concepto de inmediatez [Paper: Debugging and the experience of immediacy] que consiste específicamente en conectar al programador con el ambiente de depuración el cual lo haga sentir más cercano

al programado depurado, para hacer más gráfico el concepto de inmediatez se puede imaginar a un conductor de carreras Fórmula uno, el concepto de inmediates es que cuando el gire su manubrio a la izquierda la dirección del automóvil inmediatamente vaya en la dirección deseada, ¿se imagina si esto no sucediera? el conductor no se sentiría que sus manos están conectadas con su automovil eliminando la sensación de que automovil y corredor son uno solo.

Es de esta forma como la industria del software ha planteado diversas soluciones las cuales mitigan el pesado proceso de depuración de software. Las soluciones mayormente utilizadas son los depuradores basados en logs y basados en puntos de quiebre, relegando de forma increible 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 le basta con los depuradores break-point based y log-based?, ¿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?.

Enfocándonos en la solución de depuración omnisciente, no basta con sólo almacenar la historia del programada depurado, si no que se debe contar con un almacen de datos adecuado que sea escalable y eficiente para guardar esta enorme cantidad de información. Además de esto un depurado omnisciente debe implementar una interfaz que permita el concepto de inmediates, permitiendo conectar al programador con la herramienta de depuración evitando con esto que el programador se desconecte del problema actual.

Es por lo anterior que se ha seleccionado como caso de estudio TOD (Trace-oriented Debugger) [Paper: indicar el trabajo publicado en OOPSLA 2007] un depurador omnisciente con almacenamiento escalable. Su interfaz gráfica implementa el concepto de inmediatez y el overhead que produce es menor que otras implementaciones existentes sobre este concepto.

De este modo este trabajo de titulo se centra en la implementación de un prototipo experimental el cual realice depuración omnisciente a scripts escrito 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.[Extraído del estándar 729 del IEEE[2].

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 el cual se realizan las tareas que comúnmente se conocen como programación; que consiste, escencialmente, en llevar a código fuente, en el lenguaje de programación elejido, 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 insume la mayor parte del trabajo de desarrollo del software; sin embargo, esto puede ser relativo (y generalmente aplicable a sistemas de pequeño porte) ya que las etapas previas son cruciales, críticas y pueden llevar bastante más tiempo. Se suele hacer estimaciones de un 30 % del tiempo total insumido en la programación, pero esta cifra no es consistente ya que depende en gran medida de las características del sistema, su criticidad y el lenguaje de programación elegido. En tanto menor es el nivel del lenguaje mayor

será el tiempo de programación requerido, así por ejemplo se tardaría más tiempo en codificar un algoritmo en Assembly 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 elegido:

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 alto nivel. 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 la computadora. 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 el 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. Pero una gran ventaja de la modalidad intérprete puro, es que el 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 elejido 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 resultado 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 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".

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 gastaron semanas en intentos infructuosos para encontrar el error en un 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 programs ha sido llamado "depuración".

Pero, de acuerdo a Edsgar 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 hacía 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 están estrechamentes 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 depurar un problema, pero la dificultad de la depuración de software varía enormemente referente al lenguaje de programación utilizado y las herramientas disponibles, tal como los depuradores. Los depuradores son herramientas de software las cuales habilitan al programar para 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 termino depurador puede ser además relacionado con la persona quien hace la depuración.

Generalmente, los lenguajes de alto nivel, como Java, hacen la depuración fácil, porque ellos tienen características tales como manejo de excepción que hacen real fuentes de comportamiento herratico más facil de localizar. En los lenguajes de bajo nivel tal como C o assembler, los bugs pueden causar silenciosos problemas tal como corrupción de memoria, y esto frecuentemente dificil ver donde el problema principal ha ocurrido. En estos casos, las herramientas de depuradores de memoria pueden ser requeridos.

En determinadas situaciones, herramientas de software de propósito general que son lenguajes especificos pueden ser muy utiles. Estas son llamadas herramientas de analisis estatico. 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 estás herramientas raramente son detectados por el compilador o interprete, por lo tanto ellos no son correctores sintacticos, si no más bien correctores semanticos.

2.1. Herramientas

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

Cuando un programa lanza un error, el depurador muestra la posición en el código original, esto si es que es un depurador a nivel de código fuente, comunmente vistos en ambientes de desarrollos integrados. Un lanzamiento de error sucede cuando el programa no puede 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 le 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 expectadores de la ejecución de este.

La importancia de un buen depurador no puede ser exagerado. Incluso, la existencia y calidad de una herramienta para un lenguaje dado y su plataforma puede incluso ser un factor de decisión en su uso, incluso si otro lenguaje/plataforma es más apropiado. Aunque, es importante también notar que el programa puede (y incluso hacer) comportarse de forma diferente cuando corre bajo un depurador, debido a los inevitables cambios que provoca la presencia de un depurador. Como resultado, incluso con buenas herramientas de depuración, este es a menudo muy difícil localizar problemas en tiempo de ejecución en sistemas multi-hilos o en sistemas distribuidos.

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 especifico del usuario y el uso de la historia puede hacer que sea dificil reproducir el problema.

Después de que el bug es reproducido, la entrada del programa necesita ser simplificada para hacer más fácil el proceso. Por ejemplo, un bug en un compilador puede hacer que este caiga cuando este parseando un programa fuente muy largo. Tal simplificación puede ser realizada manualmente, utilizando una enfoque "divide y vencerás". El programador intenterá de sacar algunas partes del caso original de prueba y verificar si el problema aún existe. Cuando depuramos 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 depurado 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 impremen el valor de las variables en determinados puntos del programa en ejecución.

La depuración remota es el proceso de depuración de un programa el cual está corriendo 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 este.

2.3. Objetivo general

El objetivo general de este trabajo de título 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 programar 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 instrumente*[1] *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 el cual capturará todos los sucesos ocurridos en el programa objetivo escrito en Python, para luego enviarlos a la base de datos de TOD.

2. *Analizar la compatibilidad del modelo actual de huella de ejecución*[2] *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

[1] agregar instrucciones al código para que realice determinadas tareas

[2] Secuencia ordenada de eventos heterogéneos

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 creada en pyTOD hacía 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 una interfaz gráfica para el depurador pyTOD.*

De ser necesario se construirá una interfaz gráfica en la cual el usuario pueda interactuar con el programa depurado. Para esto se debe evaluar si la actual interfaz gráfica de TOD sirve y es compatible. De ser compatible realizando modificaciones se tomará esta opción como valida y se reutilizará este componente.

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

Como se planteo anteriormente el principio de inmediatez es fundamental en un ambiente de depuración eficiente, por lo tanto se cree que no basta con tener una interfaz gráfica independiente del ambiente de desarrollo del programador si no que se encuentra 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 desarrallo el ide Eclipse. 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.

6. *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 de interés que la herramienta desarrollada, pyTOD sea de utilidad verdadera y muestre con resultados reales que es una aproximación a lo que podría ser un ambiente de depuración real para asistir efectivamente a los programadores que utilizan el lenguaje de programación Python. En este punto se realizarán pruebas con programas de softwares reales y que se encuentren disponibles en la red bajo la categoría de software open source.

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.

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 objetos[3] dentro del programa depurado se necesita marcarlos a cada uno de ellos con un identificador, pero lamentablemente en Python no se puede y la única manera de realizarlo es modificando la maquina virtual, situación que queda fuera del alcance de este trabajo de titulo.

[3]referente 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

En esta etapa principalmente se enfocó a la búsquedas de papers relacionados con el tema de depuración en general.

3.2. Análisis del Marco teórico

Básicamente se fue avanzando en forma iterativa y por la técnica ensayo y error. El objetivo central era lograr construir un capturador de huella y en la construcción de este se fueron corrigiendo temas de compatibilidad con TOD y cosas adicionales de que hacer

3.3. Análisis de TOD

TOD es un depurador omnisciente el que tiene como principal característica el ser escalable en el sentido de almacenamiento de la huella de ejecución. Para nuestro trabajo debemos analizar las siguientes partes de TOD:

- Event Database
- Structure Database
- Debugger frontend

3.4. Modelamiento y diseño de pyTOD

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

3.5. Implementación de pyTOD

Esta sección comprende toda el desarrollo práctico del presente trabajo de título, en la cual se implementará el capturador de huellas utilizando el paradigma de orientación de objetos. Se utilizarán recursos internos del lenguaje de programación Python. Es importante señalar que el uso de la función settrace perteneciente al módulo sys se torna en un elemento central en este desarrollo. Además se inspeccionará el bytecode de Python para ciertas operaciones.

3.6. Evaluación de pyTOD en casos de pruebas

Se tomarán aplicaciones open source que hayan sido implementadas en el lenguaje de programación Python. Se analizará su bugtracker y de ahí se tomarán algunos bugs (abiertos o cerrados) y se verá como es el comportamiento de pyTOD en estos casos de pruebas.

Capítulo 4

Marco teórico

4.1. Introducción

En la practica, un programador inevitablemente gasta 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 leer 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 o indeseado.

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 bug. 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 tentarle 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. Esto de no ser posible es deseable que pase a alguien más este problema para que lo pueda estudiar y 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 indentificarlo. Esto es particularmente importante cuando se está trabajando con un reporte debug 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 oscuros, como cuando el programa genera una salida la cual es ligeramente incorrecta.

Muchos reportes de bug recibidos desde los usuarios son de la forma "Yo hice esto y esto otro, y luego algo fue mal". Antes de hacer cualquier cosa, el programador debe encontrar que es lo que fue mal – esto es, el programador debe identificar el bug determinando el comportamiento del programa el cual fue inesperado o indeseado. Cualquier intento de corregir el bug antes de entender que es lo que fue mal es generalmente tiempo desperciado.

Identificar un reporte de bug hecho por un usuario, típicamente requiere obtener la respuesta a estas dos preguntas: "¿Qué hizo el programa?ƨ "¿Qué esperaba que hiciera el programa?". El objetivo es determinar precisamente el comportamiento del programa el cual fue inesperado o indeseable.

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 çaracterística 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 verdaderos bugs, y requieren de mucho trabajo.

4.1.2. Replicar el bug

El primer paso para solucionar un bug es replicarlo. Esto significa recrear el comportamiento indeseado bajo condiciones controladas. El objetivo es encontrar

un conjunto de especificaciones precisas de los pasos los cuales demuestran el bug.

En muchos casos esto es sencillo. El programador ejecuta el programador con una esistirse a la tentación, graba el bug. determinada información de entrada, o presiona un botón en particular o una ventana de dialogo y el bug ocurre. En otros casos, replicar 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, replicación puede ser casi imposible.

4.1.2.1. El programador nunca debe saltar este paso

Los programadores son tentados algunas veces de 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 correción. La correción 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 el cual puede suceder muchas veces. En un programa complejo, es frecuentemente fácil encontrar algo que arreglar. 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, si está bien o mal. Una solución incorrecta lidera futuros problemas. En promedio, saltarse el paso de replicación derrocha más tiempo que ahorralo.

4.1.2.2. Situaciones de dificil 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 facilmente 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 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 desarrolladiquiqueo. 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, confirmando 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 puede replicarlo el mismo en su propio sistema, y el programador no sabe 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 tratar de hacer cualquier presunción sobre como el programa está ejecutándose – el usuario puede estar haciendo algo que el programador nunca se 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 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 muy dificil 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. Entender el bug

Una vez que el programador es capaz de replicar el bug, debe entender 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 de la estructura general. 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.

Comunmente, 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 lineas 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 comunmente 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 el, 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 las cuales el programador necesita considerar: el código que hace que el comportamiento incorrecto sea visible y el código que en realidad es incorrecto. Es muy común que estas sean las mismas piezas de código. Sin embargo, es bastante común que estos estén en diferentes partes del programa. Un ejemplo típico de esto es cuando existe un error en un parte del programa que causa un acceso a memoria restringida el cual provoca una visión erronea 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 erroneo es en realidad la parte incorrecta del programa.

Por lo general, el programador tiene que encontrar el código que produce el comportamiento erroneo. Conociendo el comportamiento erroneo, y sabiendo como el código fuente fue organizado, a menudo esto lo dirige rapidamente 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 linea 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 la localización del comportamiento erróneo en el código fuente, y 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 (problema 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, hilos).

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 intrucciones 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 el programador 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 erroneo 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 adivinar 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. Encontrar el error

Ahora que el programador ha encontrado el código que causa el comportamiento erroneo, es necesario identificar el error en la codificación real. A menudo son el mismo código - es decir, la codificación erronea 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 erroneos, o en el momento equivocado. Un error en la codificación de otros lugares puede causar una estructura de datos que espera valores erroneos. 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 son 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 debe 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 erroneo del programa.

4.1.4. Corregir el bug

El último paso en el proceso de depuración es, por supuesto, eliminar el bug. Este punto se cree que no es importante detallarlo y sólo se mencionarán un par de puntos interesantes.

- 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 de 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 erroneo. 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 este no vuelva a manifestarse.

A partir de lo anterior, es interesante observar las técnicas que el mercado de software a 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 registro en un código es una baja tecnología para hacer depuración. También puede ser la única manera ya que los depuradores no siempre están disponibles o se pueden aplicar. Normalmente este es el caso de múltiples aplicaciones y aplicaciones distribuidas.

La experiencia indica que la acción de registrar cierto comportamiento es un componente importante dentro del ciclo de desarrollo. Estos ofrecen varias ventajas. Proporciona contexto precisa 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 luego ser estudiados en otro momento. Además de su uso en el ciclo de desarrollo, un buen manejo de las intrucciones de registro de comportamiente pueden ser vistas como una herramienta de auditoría.

Como Brian W. Kernighan y Rob Pike exponen en su libro "La Práctica de Programación":

Como opción personal, tendemos a no utilizar depuradores más allá de obtener un pila de traza o el valor de una variable o dos. 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 duramente y añadir instrucciones de salida que verifiquen por si mismos puntos criticos de este. Al hacer clic 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 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 directacamente en una consola. En general

esto se hace nivel de código fuente y lo que se imprime siempre está dirigido para que un humano lo comprenda.

4.2.1.3. Ventajas / Desventajas

La técnica de mensajes tiene sus inconvenientes. Puede retardar una solicitud que se le realiza al programa. Si los mensajes son demasiado extensos estos pueden causar ceguera de desplazamiento sobre el código fuente original. Es importante señalar que esta técnica no escalable.

4.2.1.4. Herramientas

- log4j

4.2.2. Depuración basada en puntos de quiebre

4.2.2.1. Motivación

Un punto de quiebre, en desarrollo de software, es una detención intencional o pausa en un lugar del programa, 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 bien.

4.2.2.2. Proceso de depuración

En la practica, un punto de quiebre consiste en una o más condiciones que determinan cuando la ejecución de un programa debe ser interrumpido. La forma más común de un punto de quiebre es uno donde la ejecución del programa es interrumpido antes de ejecutar la instrucción especificado 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 especifica en el area de memoria. Esto es comúnmente llamado información de punto de quiebre o un punto de observación.

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

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

4.2.2.3. Ventajas / Desventajas

El depurador al momento de encontrar el punto de quiebre, no tiene en sus registros la pila de llamadas que se producieron anteriormente perdiendo toda la información anterior al punto de quiebre.

4.2.2.4. Herramientas

Hay que seleccionarlas bien, aún en estudio

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 herrado, no es la pregunta correcta. La pregunta que debieran tener que responder es: *¿Qué información ayudaría más al programador?*.

De estudios informales se ha revelado que varios cientos de programadores, aproximadamente 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. 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.

Existen consideraciones para programas que superan los 10 millones de eventos.

4.2.3.1.2. Es interesante la depuración omnisciente Existen varias razones para hacer un esfuerzo y estudiar la depuración omnisciente:

- La primera y más famosa, es fácil depurar cuando puedes ir hacia atrás. La pregunta más común que los programadores se hacen es "¿Quién definió esta variable?".
- Este elimina los terribles problemas con depuradores 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 .ºhh, fuí demasiado lejos". Se eliminan los problemas no deterministicos.
- Este entrega al programador una lista única del programa siendo capaz de ver las huellas de llamadas a métodos.
- Toda la información es serializada. Puede ser analizada remotamente.

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 para explicar al lector 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 debemos grabar cada cambio de cada objeto o variable. Necesitamos 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 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 ¿ésto 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 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 linea 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

Se debe tener una capacidad de almacenamiento bastante grande, un cluster de preferencia.

4.2.3.4. Herramientas

Algunas de las herramientas que existen en el mercada actualmente son:

- EXDAMS
- ODB
- ZSTEP
- TOD

4.3. Lenguajes de Programación

Esta sección es importante ya que señala las diferencias entre el lenguaje de programación en la cual está implementado TOD y entre el lenguaje de programación que se intentará portar.

4.3.1. Python

4.3.1.1. ¿Qué es Python?

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90. 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).

4.3.1.2. 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 (lenguajes compilados).

La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y más 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 a un pseudo código máquina intermedio llamado bytecode la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

4.3.1.3. 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.

4.3.1.4. 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 tenemos una variable que contiene un texto (variable de

tipo cadena o string) no podremos tratarla 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.

4.3.2. Python v/s Java

Java es un lenguaje 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 varibles con sus tipos antes de usarlas.

Existen muchas más diferencias entre estos dos lenguajes pero para el caso de estudio sólo nos interesa mostrar esta diferencia como la principal.

Capítulo 5

Implementación de depuradores omniscientes

A lo largo de la historia de las ciencias de la computación han existido muchos esfuerzos por implementar depuradores omniscientes es por esto que encontramos importante conocer las distintas perspectivas de diferentes equipos de implementación. A continuación mostramos, dividida por etapas, las distintas implementaciones existentes al día de hoy.

5.1. Implementaciones históricas

En esta sección consideraremos dos implementaciones importantes:

- EXDAMS - Extendable Debugging And Monitoring System
- ZSTEP - para prolog al parecer

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 aliviado de los detalles de programación requeridos en el nivel de lenguajes ensamblador. Esta expectativa ha sido en gran parte cumplida. Muchos sistemas son ahora construídos en lenguajes de alto nivel.

Sin embargo, la habilidad de depurar programas tiene avances estos son pequeños en relación al incremento en el uso de estos lenguajes de alto nivel. Como Evans and Darlay indican: *Nosotros encontramos que, hablando en términos ge-*

nerales, 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 alguno lenguaje de alto nivel. Sin embargo, las facilidades de la depuración en linea para los lenguajes de alto nivel son en general menos bien desarrolladas y menos ampliamente usadas (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 linea de los lenguajes en ensamblador, más que diseños de facilidades totalmente nuevas para los lenguajes de alto nivel. Ellos no han creado ni formatos graficos en los cuales se presente información acerca de la depuración, no han entregado una manera rasonable con la cual los usuarios puedan especificar el proceso requerido en cualquier depuradora de información disponible.

EXDAMS, es un intento por quebrar este impase entregando un ambiente simple en el cual los usuarios pueden facilmente añadir nuevas funcionalidades a un depurador en linea sin tene que modificar mayormete 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 linea diseñado para facilitar la experimentación con nuevas herramientas de ayuda que cumplan estas mismas características y para proveer alternarnancia de forma flexible entre estas ayudas como el tiempo de ejecución es controlado en cualquiera de los sentidos adelante o atrás.

5.1.1.3. Características

Es un poderozo depurador y monitor a nivel de código fuente para lenguajes de computación. Sus facilidades son de dos tipos: estatico, el cual hace referencia a un punto específico en tiempo de ejecución; y pelicula, el cual cambia en tiempo de ejecución y puede ser visto en tiempo de ejecución sin importar la dirección atrás o adelante (ej.: 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 sucedio 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 algunos propuestas, pero sin aplicarlas, para facilidades de depuración y monitoreo en linea.
- Como facilidades extendibles las cuales los nuevos depuradores y monitores pueden añadir facilmente 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 está siendo depurado y/o monitoreado, si no 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 exhaustiva debe incluir facilidades poderosas en los espacios de información y control y 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 invariante en relación el 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 De otra manera, 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 ultimo valor de n de un conjunto de variables, la instrucción actual y la subrutina, y el 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 rapidamente 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 continuando 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 como 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 el cual el análisis inverso fue solicitado. Cada nodo consiste en una instrucción de asignación a nivel de codigo 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 en el nodo que está conectado con esos nodos. Esos nodos tienen la misma forma como el original y están conectados para todos los valors no constantes utilizados en una instrucción de asignación en particular produciendo su valor. De esta manera, la siguiente figura muestra en analisis 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 usado en la asignación de A fue 8 y 107, respectivamente, y fueron producidas mediante las instrucciones de asignación B̈=R-1;ÿ C̈=A+E;ïespectivamente. Cada uno de los otros nodos se explican de la misma manera.

5.1.1.6. Arquitectura

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

5.1.1.6.1. Análisis del programa La primer fase analiza el programa fuente del usuario como este 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 através 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 codigo 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 vayamos en las direcciones de 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 afectadas 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 recoleción de historia en tiempo de ejecución.

5.1.1.6.2. Compilación El procesador estándar del lenguaje fuente compila el programa fuente, como actualizado durante el analisis del programa.

5.1.1.6.3. recoleción de historia en tiempo de ejecución La vesión compilada del programa actualizado es ejecutada con un conjunto rutines en tiempo de ejecución que este llama. Estas rutinas reunen 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 simbolica y el modelo, es suficiente para recrear el comportamiento del programa en cualquiera de las direcciones (atrás o adelante) en el tiempo de ejecución.

5.1.1.6.4. depuración en tiempo de ejecución con navegación a através de la historia del programa. Esta fase contiene a los asistentes para depuración y monitoreo los cuales presentan la información historica 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 dos implementaciones importantes:

- ODB - Omniscient Debugging
- Chronicle - para prolog al parecer

5.2.1. ODB

ODB es una implementación Java la cual recolecta información mediante la instrumentación del byte code del programa objetivo, al momento que éste es cargado. Este utiliza un mecanismo simple de alto nivel para ordenar los eventos de diferentes hilos. Este 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:

Presentación de la información ¿Como el programador puede obtener desde el depurador el estado del programa que le interese?

Navegación ¿Cómo el programador reconoce el estado?

5.2.1.1. La naturaleza de los bug bajo ODB

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

5.2.1.1.1. La serpiente en el pasto Si el programa imprime *la respuesta es 41* en vez de *42*, entonces nosotros tenemos un manejador sobre el bug. Nosotros vemos la serpiente en el pasto y podemos agarrar su cola. Ahora si el programa falla al producir la respuesta correcta, entonces nosotros no tenemos un manejador directo del bug. Esta es la serpiente en el pasto que no podemos ver.

Para programas que encajan y nosotros podemos ver la serpiente en el pasto, entonces el ODB es absolutamente maravilloso. Es posible comenzar sobre la salida erronea e ir hacia atrás, encontrando el valor inapropiado en cada punto, y luego seguir a este valor hasta su causa. (Si tu tienes una cola de serpiente y tu la tiras lo suficientemente fuerte, tu obtendrás su cabeza.) No es inusual iniciar el depurador, seleccionado la salida erronea, y navegar por el código fuente en 60 segundos. Con absoluta confianza.

Los depuradores basado en punto de quiebre sufren del problema "La largatija en el pasto". Incluso cuando ellos pueden ver la largatija y tirar de su cola, la lagartija quebrará su cola y se irá. Luego ellos vuelven al problema "La lagartija perdida en el pasto".

Con ODB, existe una versión de este problema. Cuando la producción incorrecta es causada por la falta de un determinado objeto cuando esta siendo asignado, entonces nosotros tenemos que responder la pregunta *¿Quién pudo haber echo esto?*, la cual es una pregunta mucho más dificil que *¿Quién hizo esto?* (Esta es el lagarto sin pata en el problema del pasto). Nosotros pensamos que tenemos una serpiente, pero despues de tirar tu cola por un momento, esta la quiebra de cualquier forma.

Para problemas relativamente pequeños (digamos 10.000 de eventos), esto es fácil y basta con solo examinar por completo las huellas de asignación de los métodos. Para problemas más grandes, la "serpiente perdida en el pasto", se tornan más problemáticos. Nosotros tenemos 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 (poner referencia!)

5.2.1.1.2. Tamaño Otra importante dimensión es la del 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 encajan dentro de este espacio. Un buen porcentaje no. Existe un número de opciones para atacar este problema. Nosotros podemos:

- 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. Esto de lo contrario es malo porque no se elimina basura, si no que solo eventos antiguos que pudieran ser importantes. Esto además es malo porque permite el programa correr lo bastante para que el rendimiento se convierta en un tema importante. Este 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 evento del tiempo cuando la grabación fue apagada, luego nosotros volvemos el 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 muy normal para un gran porcentaje de un programa consista en bien conocido, seguro código fuente. (Existen cualquier metodo recomputable, o metodos que nosotros no queremos ver el interior de ellos). Mediante solicitud que estos metodos no sea instrumentados, muchos de eventos no interesantes son eliminados. ODB permite al programar 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 (ej.: Cuando presiono este botón el programa se cae), luego 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. Control automático de la manera que nosotros iremos a examinar un gran conjunto de eventos por un patrón particular, el cual será prendido / apagado 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 indice de linea (20 bits) y un indice de tipo (4 bits). Un evento para cambiar una instancia de una varible 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 una pequeño montón 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 el ODB quiere saber cual valor de una variable fue en el tiempo 102, este justo agarra el valor que está más cerca al valor previo. La lista de historia para variables locales y argumentos cuelgan sobre la Linea de huella.

Siempre que ocurra un evento, este bloquea por completo el debugger, una nueva marca de tiempo es generada y registrada dentro de la lista, y estructuras asociadas son construidas. Los valres de retorno y excepciones son agregadas dentro de la apropieda linea de huella como ellas fueran generadas.

La inseción en el código es muy simple. El byte code fuente es examinado y un código instrumentado es insertado antes de cada asignación y cerca de cada llamada de método.

Una inserción tipica 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 eran las lineas 289 y 291, asignando un valor a la variable local *ie*. La instrumentación crea un evento que registra que sobre la linea 64 de Micro.java(#292), la variable local ie(#295), de quien su Lista de historia puede ser encontrada sobre la linea 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 omnisciente para Java

5.3.1. Contribución

La contribución de este trabajo es mostrar que el debugging omnisciente puede ser realizado realisticamente a través del uso de diferentes técnicas aumentando la eficiencia, escalabilidad y usabilidad. Lo planteado es validado por TOD, un debugger portable orientado a la huella para Java integrado dentro de ECLIPSE [5]. 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 y rápido y consultas sobre eventos, cuando se apalancan el manejador de restricciones natural de la ejecución de 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, gracias 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 del debugging omnisciente

Presentamos las características principales de un degugger omnisciente comparado con los debuggers tradicionales y el perfil de los desafíos de escalabilidad del debugging omnisciente.

5.3.2.1. Características de un debugger omnisciente

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

- paso (pasando)
- estado de reconstitución
- reconstitución del control del flujo
- encontrar la principal causa.

El último es únicamente en los debuggers omnisciente, mientras que los otros son características de los debuggers típicos.

En debuggers basado 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 comportamiento llama a las declaraciones sin estar esperando dentro del llamado al comportamiento, mientras el step into espera a la primera (comienzo) de la llamada al comportamiento. Reconstitución del estado consiste en ir entregando al programador un objeto de estado inspector cuando el programa objetivo está esperando. Reconstitución del control de flujo permite obtener una vista en la llamada de la pila actual del programa, con ligadura a variables y objetos. OD's extiende estas tres características con completa libertad con respecto 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 de entrega en el tiempo, y pueden libremente navegar por el árbol de control.

Finalmente, una de las características más usadas de OD's es su habilidad para encontrar "donde" y "en que contexto" a un campo o variable en particular le fue entregado un cierto valor. Ciertamente, frecuentemente los bugs se manifiestan mucho después de suceder su causa raíz. Para instanciar, tratando de referirnos a NULL referencia obtenida desde un campo que entrega una causa de caída, cuando es el síntoma del bug. La información que el programador necesita es

donde el campo fue definido con valor NULL. Con debugger basado en punto de quiebre, siempre si la ejecución es pausada justo antes del imperfecto referido, la causa raíz del bug se pudo haber perdido. Ejemplo: porque el código que causó no está en la llamada nunca más.

5.3.2.2. Desafío de escalabilidad

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

Eventos deben ser grabados rápidamente, preferentemente en tiempo real, como que (a) debugging 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 (ejemplo debugging Eclipse).

El debugger 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 debugger omnisciente debe estar alineado con un número esperado de eventos dentro del uso total de la huella; con GHZ CPU's, cientos de millones de eventos pueden ser generados en solamente pocos minutos de ejecución.

Fig. 1 Arquitectura de auto nivel de TOD

Consultas dentro la huella de ejecución debe 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, habilitando una rápida identificación de los bug's.

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

5.3.3. Acerca de TOD

TOD es un debugger 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 debugger 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 alrededor de dos ideas centrales: desacoplar el núcleo del debugger desde el programa objetivo en ejecución y ser portable. Esto es hecho sobre un árbol de componentes (Fig.1): el objetivo (JVM) en el cual el programa debugged corre y emite eventos, el núcleo del debugger que implementa las principales funcionalidades de TOD, y la interfaz del debugger a través del cual el usuario mutuamente 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 debuggers omniscientes [8,13,15] está precisamente dirigido a algunos de los desafíos discutidos en la sección 2.2.: almacenando eventos en el espacio de dirección del objetivo de la aplicación 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 se incrementa por el uso de una base de datos que es compensado por una mejor escalabilidad y sin generar intrusión. Como un lado efecto, la habilidad de serializar la huella de ejecución permite para debugging post-morten, quien abre un interesante perspectiva para las compañías de software dispuesta a ofrecer software con alta calidad de soporte: observando el costo de almacenamiento, una huella en ejecución navegable es una información lejos mucha más relevante para reportar un bug que un texto descriptivo apropiado.

Tabla 1. Eventos y sus atributos

La cabecera de las filas son tipos de eventos y la cabecera de las columnas son posible 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 de comportamiento, 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 debugger, donde ellos son grabados e in3.1 dexados 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 levante 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 (ver en sección 4 y 5). El núcleo del debugger contiene otra base de datos, la base de datos de estructura, quién contiene información estática sobre la aplicación objetivo. En particular conserva la huella el identificador de 32 bits en enteros que son asignados hacia eventos estructurales del programa objetivo(por ejemplo: clases, métodos y campos(celdas)). Las consultas son presentadas por el usuario confiando entre el evento y la base de datos estructura.

5.3.3.2. Representación y emisión eventos

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

Modelo de evento y huella Un evento es una estructura caracterizada por un número de atributos elegidos entre el conjunto A=a0,...,ak. Observemos e.aj el valor del atributo aj del evento e. Para cada j perteneciente [0...k], dejamos Dj como el dominio de aj, por ejemplo el conjunto de todos los valores distintos que pueden ser tomados por aj. para cualquier evento dentro de la huella. Una huella del evento T=<e1,...,en> es una secuencia ordenada de n eventos heterogéneos.

El atributo a0 corresponde al timestamp del evento; este es caracterizado por el hecho que (a) todos los eventos tienen un valor para a0, (b) allí existe un completo orden sobre D0 y (c) los eventos en T están ordenados por sus valores de a0. Tabla 1 muestra cuales eventos concretamente son capturados y cuáles son sus atributos.

Emisión de los eventos El núcleo debugger de TOD captura los eventos emitidos por la aplicación objetivo (Fig. 1). Existen tres vías en las cuales los eventos pueden ser emitidos: puertos especializados de la huella del hardwa-

re [7], maquina virtual o interpretador de instrumentación [14], y aplicación de instrumentación [8, 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 JVM que está alojada en la aplicación objetivo está configurada para usar el agente JVMTI[1] . El agente intercepta los eventos cargados por la clase y reemplaza las definiciones originales por la versión instrumentada. Instrumentación es realizada por el tejedor en el núcleo del debugger: el agente envía el bytecode original al núcleo, el tejedor implementa la clase y almacena estructuradamente la información en la base de datos de estructuras, y la clase modificada es enviada nuevamente a la JVM objetivo donde es eventualmente cargada (Fig. 1). 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 en el JDK.

Instrumentación es realizando usando la librería de bytecode ASM [3]: el código de captura del evento es añadida antes y/o después de un patrón especifico 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 construidos a partir de sus atributos, serializados en un formato binario personalizado, y envía a través de un socket a la base de datos de eventos.

Sincronización no ambigua del evento Aunque los timestamps del evento son obtenidos a través del servicio de precisión en nano segundo de Java, es potencialmente carente de exactitud haciendo posible que varios eventos del mismo hilo tengan el mismo valor timestamps. Como esto es incompatible con el esquema de indexación usado por TOD (sect. 4), cambiamos 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 del mismo timestamp. Cuando comparamos el timestamps de los eventos de diferentes

[1]Interfaz de herramienta Máquina virtual de Java, parte de la plataforma de Java 5.

hilos, usamos el timestamps original para preservar el orden de los hilos internosinter-thread.

Alcance de la captura de huella El esquema de instrumentación describe sobre es selectivo, eso es, es posible de proveer filtros definidos para los usuarios que limite el número de eventos emitidos. Esta característica es descrita en la sección 7.

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 exceptions son pasadas por valor.

5.3.3.3. Bajo nivel de consultas: cursores y contadores

Todas las características presentadas en la sección 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, cuales introducimos a continuación. Ambos basados en el filtrado de eventos de la huella de acuerdo a algunas condiciones de sus atributos. Condiciones puede 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.

La actual posición del cursor está representado por la linea negra entre los eventos cuatro y cinco. Los Eventos que juntan el predicado del cursor están en gris. llamamos sucesivamente a next() retornando los eventos 5,6,11 and 14; llamando posNext(11) la posición del cursor entre los eventos 10 y 11; llamando posPrev(11) las posiciones estarán entre los eventos 11 y 12.

Fig. 2 Navegación entre la unión de los eventos y el predicado del cursor 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 cuyo timestamp 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 2. operaciones de los cursores

cursores Definimos cursor(Q) como un iterador sobre los eventos que reúne la condición Q (Fig. 2). Los cursores tienen una posición actual que está situada entre dos eventos consecutivos (o hacia el principio o el termino de la huella). Un cursor soporta un número de operaciones de navegación, como las mostradas en la tabla 2.

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 regresa un arreglo de s enteros tales que s[i] = — {e pertenece a T: dentro(e, t1 + i*delta t) $\hat{Q}$(e)} — donde dentro(e,t) ¡= ¿e.ts ¿= t ê.ts ¡t + delta t. Cada ranura 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

Ahora explicaremos como los cursores y los contadores son combinados algoritmicamente para implementar el alto nivel características descritas en la sección 2.1. La sección 4 discute la agresiva optimización de la base de datos habilitado por sólo estar usando las consultas basadas en los filtros.

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 instancia, paso adelante dentro es definido como sigue:

c ¡– cursor(thread = ev.thread) c.posPrev(ev); ev ¡– c.next()

Paso adelante sobre cambia la condición del cursor a: thread = ev.thread $\hat{d}$epth = ev.depth. Paso atrás es simétrico al paso hacía atrás.

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 ¡–cursor(kind = FW $\hat{f}$id = f $\hat{t}$arget = o) c.posPrev(t); v ¡– c.prev().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 ¡– cursor(thread = e.thread ĝdepth = e.depth + 1) c.posPrev(e.ts); cflow = ¡¿repeat ev = c.next(); cflow ¡– cflow union ¡ev¿until ev.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 arriba: en lugar de obtener el valor del evento de la celda escrita que asignó el valor al campo, el evento por si mismo se hace el actual, dando acceso para el contexto en eso 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 (Fig. 3). El navegador de usuario entre diferentes vistas utiliza las ampliamente conocidas metáforas de los navegadores web (hyper vínculos, botón hacia atrás). Las vistas disponibles son: inspector de objetos, control de flujo, y mural. La 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. La vista de control de flujo muestra una constitución del flujo de control y permite las operaciones de caminar así como encontrar la raíz de la causa por 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 [9] como una “reducción de la representación de la información del espacio de un entero que ajusta enteramente dentro de una ventana mostrada ”. Las características de los eventos murals de TOD, 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 instancia, TOD puede mostrar la densidad del evento por cada uno de los hilos para el conjunto de ejecución de la aplicación objetivo (Fig. 4), o la densidad de llamadas a métodos de un objeto en particular. Las densidades son obtenidas a través de los contadores (sección 3.3), 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 murals; cuando el nivel de acercamiento permite distinguir los eventos individuales del usuario puede seleccionar un evento y ver el contexto dentro.

Botón (A) lanza el programa con el trace recordingrecordando 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 linea 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 why? Siguiente eso (next to it).

Fig. 3 Caminando con TOD en Eclipse

de una vista de paso. El mural de hilos tiene distintas aplicaciones, por ejemplo: entender la interoperación entre los hilos, o spotting dead y las livelocks.

5.3.4. Soporte para base de datos de alto nivel

Ahora describimos y analizamos el esquema de indexación de TOD, aun cuando permite la ejecución de consultas eficientes mientras estén rápidas enough bien disponible un alto rendimiento de grabar. En la sección 5 se muestra como nuestra solución es favorable para la paralelización, y en la sección 6 reportes sobre el actual medidas de rendimiento.

La necesidad para desarrollar una cimiento de base de datos especializada para TOD fue motivado por el pobre rendimiento de el uso profundo de la administración de sistemas de base de datos para nuestros propósitos: PostgreSQL y Oracle sólo soportan almacenar eventos a una taza de 50 y 500 eventos por segundo respectivamente, mientras nuestros registros la taza está en el orden de los cientos de millones de eventos por segundo [19]. Nuestro alto rendimiento especializado en los cimientos de la base de datos apalancan la siguiente especificaciones para el stream del evento de

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

una huella de ejecución: (a) el stream del evento es de sólo lectura, (b) los eventos al llegar son ordenados a partir del timestamp y (c) las consultas son limitadas por el filtrado.

5.3.4.1. Indexación jerárquica de los eventos

TOD adoptó un esquema de indexación jerárquica que permite traer el evento juntando un predicado en timestamp ordenar sin siempre realmente accesando a los mismos eventos, reduciendo el proceso de costo de la consulta.

Índices en los valores de los atributos Usando la notación definida en la sección 3.2. nosotros definimos el índice de T sobre aj como una función Ij: Dj –¿(Do,N)* para j pertenece [1..k] so que Ij mapas alguna posibilidad valor v de aj una secuencia del indice entrada del formulario (ts,i). Una entrada aparece en el indice Ij(v) si y sólo si ei.aj = v êi.a0 = ts, donde ei es el i-esimo evento de T. Adicionalmente, las entradas son ordenadas por ts. Esos indice pueden ser usados directamente para recuperar todos los eventos que junten una consulta simple de la forma attribute = value; condiciones de composición son discutidas en la sección 4.3.

Jerarquía de indexación por timestamp Debido a que las consultas en TOD no consisten sólo en encontrar eventos juntados finding matching, pero también also cuando se encuentran los eventos juntados ocurre esto, después o antes de un punto particular en el tiempo, los índices son ordenado por sus valores de ts puede ser posible presentarlos en una búsqueda binaria para encontrar el timestamp deseado.

Esto es siempre mucho más eficiente para extender el índice de estructura en una manera jerárquica (Fig. 5). Cada índice jerárquico para el valor v del atributo aj contiene un número de niveles; El índice Ij(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 creado un índice en el nivel 1: ts es tomado desde la primera entrada (ts,i) de la reciente pagina llena, y pid es un puntero a esa página. Las entradas del Nivel 1 están acumuladas en turno en una página; cuando una página del nivel es llenada, una entrada 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.

5.3.4.2. Costo de la creación de un índice

Los experimentos 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 han determinado que el tamaño de una página óptima es P = 4096 bytes, por lo tanto el Nivel 0 las paginas de índices contienen 256 entradas, en los niveles superiores las páginas contienen 341 entradas y las paginas 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 $\hat{5}$ 4x10 $\hat{1}$2 entradas) Asumiendo que la actual página de eventos puede residir kept 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 ese índice no separado sobre a0). Los experimentos indican que sobre el promedio k = 10. Dado ciertos eventos que llegan en orden con respecto a a0, actualizando un índice sólo pensando en adherir una entrada en el final de la página en el Nivel 0, y en el final de las paginas de nivel superior sólo cuando un página de nivel inferior es llenada. La entrada/salida y costos de memoria de esta operación son los siguiente:

- Si la página de índices actual para cada nivel puede ser residida en memoria, un costo de E/S es realizado sólo cuando una pagina está llena. El número promedio de páginas accedidas por el evento entrante es:

- Si sólo la pagina de índices está en el nivel 0 puede ser residida en memoria, cuando una página es llenada debe ser escrita, El nivel de la página leída es el 1, actualizado y leído tras el disco. Analizando, el nivel más alto, tiene aproximadamente 0.13 accesos por evento.

- Si la pagina no puede ser residida en memoria, actualizar siempre implicará las tres operaciones acerca, entregando 20 accesos por evento.

El aumento de memoria dedicado al buffering de página es por lo tanto lo más importante: Hay una diferencia de 400 directorios entre la situaciones extremas arriba.

Ahora estimamos el número total de índices, sumatoria de j=1 hasta k sobre — Dj —, en orden de determinar cuántas páginas de buffer de memoria se necesita para permanecer en el caso donde por lo menos la página actual del nivel 0 de los ajustes de cada índices en memoria. Los atributos de los eventos pueden ser divididos en dos categorías. El dominio de los atributos estáticos (por ejemplo: behavior id id de comportamiento, field id id de la celda, type id id de tipo,ubicación, etc.) depende solamente de la estructura del programa, no del tamaño de la huella. En nuestras pruebas con la captura de las huellas en Eclipse, observamos que ellos 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*10 $\hat{6}$ = 4GB de espacio es necesario para buffer, el cual es alcanzable por el actual mid-end systems.

5.3.4.3. Costo de recuperación del evento

Ahora presentamos los algoritmos que permiten recuperar eventos buscando un predicado arbitrario en la linea de tiempo con respecto al tamaño de los índices involucrados. Los algoritmos son recuperados en orden respecto a su timestamp; la recuperación al revés tiene el mismo costo.

condiciones simples Para una condición simple de la forma aj = C donde C es una constante, podemos recuperar eventos buscados ordenados por timestamp, simplemente obteniendo las entradas (ts,i) desde Ij(C). Si el evento actual es requerido (por ejemplo: por los cursores), estos son directamente recuperados desde la huella como ei; si no (por ejemplo: para los contadores) el evento no necesita ser accedido. En cualquier caso, todas las entradas pueden ser recuperadas en la linea de tiempo, como el índice se explora simplemente una vez.

condiciones conjuntivas Para una conjunción booleana de una condición simple de la forma aj1 = C1 ^... âjm = Cm, nosotros usamos una variante del algoritmo sort merge join[2], extensamente utilizado en la administración de los sistemas de bases de datos, para identificar los eventos buscados sin accesarlos (Algoritmo 1): obtenemos el Ij1 (Cl) para cada una de las condiciones simples, y por cada uno mantenemos un puntero a la entrada actual (tsl,il). Entonces iteramos: en cada paso verificamos si todos los il son iguales, en este caso añadimos cualquier resultado a las entradas actuales: El echo que todos ellos point del mismo evento significa que el evento coincide con todas las condiciones. Entonces avanzamos el puntero de índice cuya entrada actual tenga el mínimo valor para ts. Como cada índice es revisado una sola vez y hay ninguna jerarquizado ciclo, merge join corre en tiempo lineal con respecto a la suma de los tamaños de los índices considerados.

5.3.5. Medidas de rendimiento

We now report on a first set of benchmarks evaluating different aspects of TOD: the distributed database, both in terms of recording of events and evaluation of queries, and the event emission overhead imposed on the debugged application.

5.3.5.1. Desempeño de la base de datos

To evaluate the performance of our distributed databases for TOD we conducted several measurements regarding recording and query performance under various setups. We captured a large execution trace of an Eclipse session where the user performed a simple sequence of tasks: opening an existing Java source file, editing it, using autocompletion, creating a new class and editing it. The captured trace comprises around 516 million events and weighs in at 20GB. We then imported this trace into the TOD database deployed on a dedicated 16-nodes cluster.6 In this experiment up to 10 nodes were available to be used as database nodes, and 1 as dispatcher and query aggregator. Each node is an Intel Itanium at 1.6GHz with 2GB of RAM and 7GB available local hard disk space, running Linux kernel 2.6.9 and the BEA JRockit 1.5.0 06 JVM. The nodes are connected by a 1Gbps ethernet link. Unfortunately this is not an ideal setup for TOD. We stated in Sect. 4.2 that at least 4GB of RAM should be available for page buffers so that enough space is available to keep the current level-0 index page of each

index in memory. In this setup the amount of RAM available at each node permits only 700MB of page buffer7 . In order to obtain significant results in spite of this limitation we had to artificially reduce the number of indexes. The number of distinct object ids was limited to 150,000 (using a modulo operation) so that the total number of indexes was around 185,000. Another limiting factor was the available disk space at each node. Even when all 10 nodes were used we could not import more than 15.5GB of the trace. We imported as much of the trace as possible in each configuration. The first benchmark measures the time taken to import the execution trace into the database; the second benchmark measures the rate at which individual events matching an arbitrary condition can be retrieved using a cursor, and the third benchmark measures the time taken to compute event counts of those same events for the whole duration of the trace. Recording. We first determined the maximum throughput of the dispatcher by shutting down event processing by the database nodes: the dispatcher can handle at most 200,000 events per second, independently of the number of nodes (Fig. 7). Actual recording results show that a single database node is able to handle around 50,000 events/s, and that with 10 nodes we get close to the dispatcher limit with 170,000 events/s (Fig. 8). The throughput increases as more nodes are used, though not quite linearly (Fig. 9). The comparison is however biased by the fact that with fewer nodes less events were imported, because of the disk space limit mentioned above. We conjecture that the throughput obtained with fewer nodes would have been lower if we had been able to import the full trace. Cursor queries. We measured the execution speed of two type of stepping-related cursor queries: seek and step. Both are based on a randomly-generated com- pound condition that matches events of a certain thread and call depth. The seek queries request a cursor with such a condition, position it at a randomly chosen timestamp within the span of the execution trace and fetch the next 6 We realized this experiment in two steps (capture and import) because of pecu- liarities of our experimental setup. With the right hardware and software it can be realized in a single step. 7 The remaining of the 1500MB heap we allocated to the JVM is occupied by the skeleton of each index structure, which exists even if none of its pages is buffered. This is an implementation issue rather than a fundamental design flaw.

5.3.6. Trabajando con huellas parciales

Aunque TOD es diseñado para soportar grandes huella de ejecución, no siempre es practicable grabar cada evento: El tiempo de ejecución mayor de una captura de evento es importante (Sección 6.2), como también lo es los requerimientos de almacenamiento. La idea de las huellas parciales es que nosotros podemos levantar el echo que durante el desarrollo de una pieza de software, algunos componentes son verificados, por ejemplo: maduro y bien probado, y eso puede ser no necesario para 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 el debugging 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 TODO en eclipse. Este ejemplo es bastante representativo de un componente de desarrollo existente, probado, frameworks o plugin arquitecturas. Aquí, nosotros estamos fuertemente interesados en dos tipos de bugs: aquellos que son internos del plugin y aquellos que relacionados con la interacción entre el plugin y la plataforma. En el primer caso, nosotros no necesitamos capturar eventos que ocurren dentro de la plataforma ECLIPSE porque eso es considerado como probado. En el segundo caso, nosotros tenemos que grabar eventos que ocurren dentro de la plataforma, pero no necesariamente todos: puede que sea bastante los eventos que se deban grabar de la Java Tooling(JDT), o solamente alguna parte de ella, por instanciar la interfaz de usuario.

Figura 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 podemos ver que por apropiado que sea el alcance de la captura de la huella, hay sobre cinco ordenes de magnitud de diferentes en el número de eventos emitidos, y que los aumentos en el exceso del tiempo de ejecución puede ser sobre 20 tiempos, realza grandemente la aplicabilidad de TOD.

5.3.6.2. Tratando con información incompleta

El inconveniente de ignorar algunos eventos es que la capura 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 sistematicos, 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 codigo 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 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 contro pudo ser perdida: El método Collections.sort no está instrumentado pero llama al método de comparación de la clase instrumentada Comp durante su ejecución

Fig. 11. Reportando una información de flujo de control potencialmente incompleta.

Capítulo 6

pyTOD

6.1. Arquitectura de pyTOD

6.2. El Modelo y Diseño de pyTOD

6.3. Implementación de pyTOD

6.3.1. Protocolo Comunicación de pyTOD

6.3.2. Estructuras y Clases de pyTOD

6.3.3. Código Principal de pyTOD

6.3.4. Interfases Principales de pyTOD

6.4. Aspectos Básicos de Funcionamiento de pyTOD

6.5. Medidas de rendimiento de pyTOD

Capítulo 7

Evaluación de pyTOD en casos de pruebas

7.1. Selección de casos de uso

7.2. Resultados obtenidos con pyTOD

Capítulo 8

Conclusiones

Capítulo 9

Trabajo futuro

Bibliografía

[1] http://docs.python.org/lib/bytecodes.html

Apéndices

Apéndice A

Código fuente de pyTOD