INVESTIGACIÓN DESTACADA

Regreso al Futuro:
Depuración Omnisciente

La depuración es una actividad tediosa y costosa que requiere una comprensión profunda del comportamiento dinámico de los programas. Un depurador omnisciente, debido a que permite navegar en forma transparente en el historial de ejecución de los programas, facilita la localización de la causa raíz de los errores. ¿Por qué, entonces, no todos tenemos un depurador omnisciente en nuestro entorno de desarrollo favorito? Por cierto, para hacer práctica la depuración omnisciente es necesario superar varios desafíos, pero ¿son acaso estos una barrera definitiva para su adopción? Este artículo describe TOD, nuestro depurador omnisciente escalable para Java. TOD se integra en el ambiente de desarrollo Eclipse y contribuye a hacer práctica la depuración omnisciente.

Guillaume Pothier
Estudiante de Doctorado en Ciencias mención Computación, DCC, Universidad de Chile, bajo la supervisión del profesor Éric Tanter. Ingeniero en Ciencia de la Computación, École des Mines de Nantes, Francia.
gpothier@dcc.uchile.cl

Éric Tanter
Profesor Asistente, DCC, Universidad de Chile. Ph.D. en Computer Science, Universidad de Nantes y Universidad de Chile. Lidera el Laboratorio PLEIAD (Programming Languages and Environments for Intelligent, Adaptable, and Distributed Systems).
etanter@dcc.uchile.cl

INTRODUCCIÓN

La depuración representa una parte importante del costo del desarrollo de software. Un estudio del NIST (National Institute of Standards and Technology) de 2002 muestra que los errores de software tienen un costo enorme sobre la economía de EE.UU. [1] y menciona que *"los desarrolladores ya gastan aproximadamente 80 por ciento de los costos en identificar y corregir defectos"*. En un estudio empírico de "hazañas" de depuración,

Marc Eisenstadt determinó que la principal causa de la dificultad de encontrar los errores es la distancia temporal y espacial entre la *causa raíz* y el *síntoma* del error [2]; una vez que un error está precisamente localizado, arreglarlo es a menudo trivial. Desafortunadamente, la mayoría de los depuradores en uso hoy en día otorgan una ayuda muy limitada con respecto a la navegación temporal; los programadores deben con frecuencia simular mentalmente la ejecución de sus programas.

Los depuradores omniscientes mejoran de sobremanera esa situación, permitiendo a los programadores navegar fácilmente hacia adelante y atrás en el historial de ejecución de un programa, así como encontrar de inmediato la causa raíz de los errores, gracias a *vínculos causales* que pueden ser atravesados hacia atrás en el tiempo [3]. Por lo tanto, un depurador omnisciente puede tener un gran impacto sobre la eficiencia del proceso de desarrollo.

La depuración omnisciente no es, desde luego, una idea nueva: el primer depurador omnisciente, EXDAMS [4], fue creado en 1969. Sin embargo, a pesar de las numerosas propuestas que se han hecho desde entonces, los depuradores omniscientes todavía no forman parte del típico ambiente de desarrollo. ¿Serán acaso los desafíos de la depuración omnisciente una barrera definitiva para su adopción?

LA DEPURACIÓN OMNISCIENTE EN POCAS PALABRAS

Enfoques tradicionales de depuración

Existen dos enfoques tradicionales para la depuración: basada en registros, o *logs*, y basada en puntos de quiebre, o *breakpoints* (Figura 1). El primer enfoque consiste en insertar instrucciones de registro en el código fuente para producir una bitácora ad-hoc durante la ejecución del programa. Esa técnica revela efectivamente el historial de ejecución del programa, pero tiene serios inconvenientes: requiere modificaciones engorrosas, extendidas y anticipadas del código fuente, y no es escalable cuando las bitácoras deben ser analizadas manualmente.

El segundo enfoque consiste en correr el programa con una herramienta de depuración dedicada, que da al programador la posibilidad de detener la ejecución en determinados *breakpoints*, inspeccionar el contenido de la memoria, y seguir la ejecución paso a paso. Desafortunadamente, cuando la ejecución está detenida, la información acerca de estados y actividades anteriores del programa está limitada a la que está accesible desde la pila de activaciones. Por lo tanto, los desarrolladores que usan depuradores basados en breakpoints deben con frecuencia volver a ejecutar el programa entero con distintos breakpoints para progresivamente acercarse a la causa del error.

La depuración omnisciente

Un depurador omnisciente registra en forma automática el historial completo, o huella de ejecución, del programa depurado, y permite al usuario explorarlo libremente (Figura 1). Este enfoque combina las ventajas de la depuración basada en registros (la información sobre la actividad pasada no se pierde) y las de la depuración basada en breakpoints (navegación interactiva, ejecución paso a paso, inspección de la pila de activación completa). Los depuradores omniscientes simulan la ejecución paso a paso hacia adelante y *atrás*, haciendo

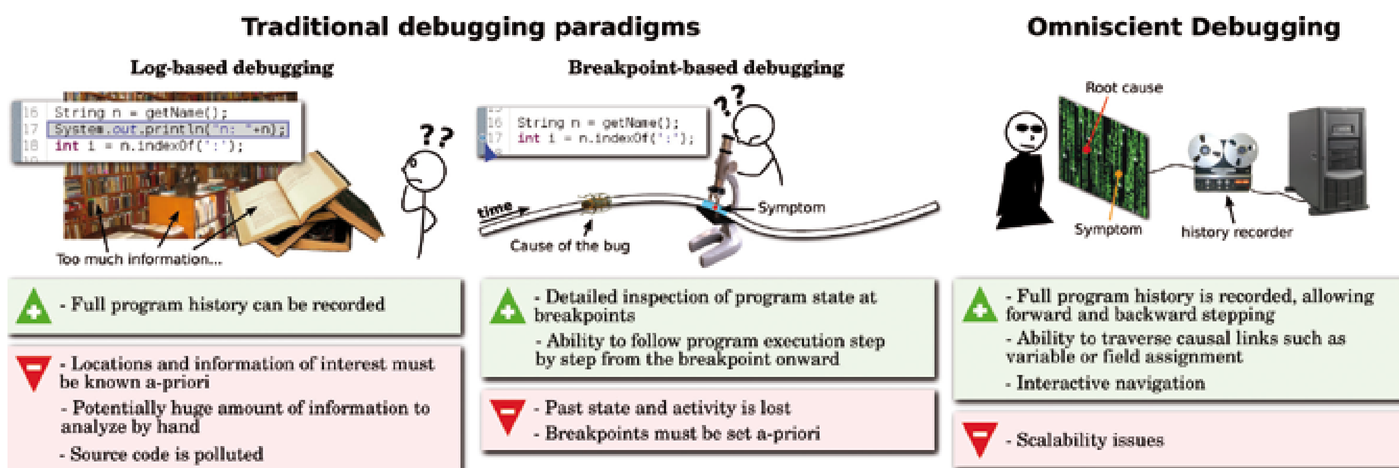


Fig. 1 Enfoques de depuración.

innecesario volver a ejecutar el programa varias veces para encontrar la causa raíz de un error. Y sobre todo, permiten atravesar vínculos causales, respondiendo instantáneamente a preguntas como *"¿Cuándo adquirió la variable x el valor null?"*, o *"¿cuál era el estado del objeto o cuándo fue pasado como argumento al método foo?"*

Desafíos

A pesar de sus claras ventajas sobre los enfoques tradicionales, la depuración omnisciente todavía no es considerada realista a causa de importantes problemas de escalabilidad:

1. La captura de una huella de ejecución causa una sobrecarga sobre el programa depurado, reduciendo su eficiencia.
2. Las huellas de ejecución llegan rápidamente a ser muy pesadas, por lo tanto requieren un sistema de almacenamiento rápido y escalable.
3. Las consultas sobre una huella de ejecución posiblemente muy pesada deben ser procesadas con suficiente prisa para poder garantizar la rapidez de las interacciones con el usuario.
4. Cualquier sea el peso de la huella de ejecución, el desarrollador debe siempre poder localizar rápidamente puntos de interés, y establecer relaciones significativas entre distintos puntos.

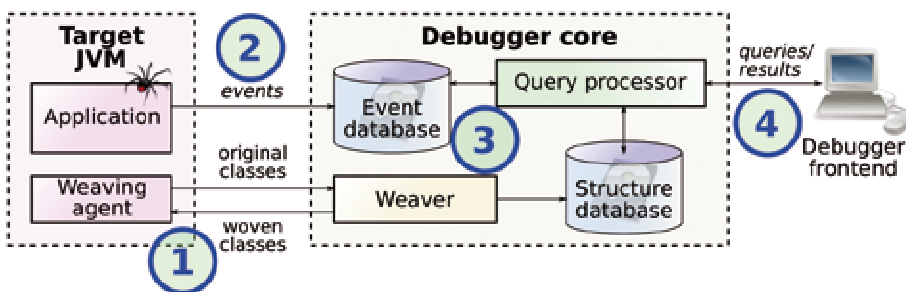


Fig. 2 Cómo funciona TOD: arquitectura y operación.

ARQUITECTURA DE TOD

TOD [5] es un depurador omnisciente para Java integrado en el ambiente de desarrollo Eclipse. La figura 2 delinea sus principios de operación:

1. **Instrumentación:** cuando una clase está a punto de ser cargada por la JVM, el agente envía su bytecode al tejedor (*weaver*), que inserta código de generación de eventos en la clase, y luego devuelve la versión modificada a la JVM.
2. **Generación de eventos:** A medida que el programa instrumentado se ejecuta, se generan eventos y se envían a la base de datos. La secuencia de eventos generados constituye la huella de ejecución.
3. **Almacenamiento e indexación:** La base de datos altamente especializada almacena los eventos a un ritmo muy alto, y al mismo tiempo los indexa para permitir un procesamiento rápido de las consultas. Además, una base de datos estructural almacena información sobre la estructura estática del programa depurado, tal como sus clases y métodos.
4. **Consultas y navegación:** El desarrollador navega en la huella de ejecución a través de la interfaz del depurador, que está integrada en el ambiente Eclipse.

Aprovechando las características muy especializadas de las huellas de ejecución (los eventos llegan casi ordenados temporalmente y nunca son modificados una vez registrados), más el hecho de que todas las acciones de navegación requeridas por un depurador omnisciente pueden ser calculadas usando simples consultas de filtrado, pudimos diseñar un sistema particularmente escalable: la base de datos de eventos se puede paralelizar, y en nuestros experimentos usando un cluster de 10 máquinas, pudo resistir por largos ratos un flujo de entrada de un medio millón de eventos por segundo.[1] Sin embargo, el mismo hecho de capturar la huella de ejecución de un programa tiene un impacto significativo sobre éste: se puede observar una pérdida de eficiencia de hasta 80 veces en el peor caso (es decir, un programa totalmente instrumentado y haciendo uso intensivo de CPU), aunque es posible reducir netamente ese impacto excluyendo partes del programa del proceso de instrumentación (por ejemplo, las clases del JDK).

En nuestra propia experiencia, una configuración de una sola máquina es suficiente para huellas relativamente pequeñas (más o menos 10 millones de eventos) y una configuración con dos máquinas (es decir, con una máquina dedicada a la base de datos además de la máquina de desarrollo) puede aguantar cómodamente huellas de 150 millones de eventos, lo que resulta suficiente para depurar, por ejemplo, la base de datos de eventos de TOD. Para huellas más grandes, organizaciones que se lo pueden permitir, se beneficiarían de una configuración más poderosa.

Depurando con TOD

TOD permite navegar temporalmente, con la ejecución paso a paso simulada hacia adelante y atrás, y hace posible navegar causalmente gracias a un vínculo desplegado al lado del valor de las variables inspeccionadas, lo que permite saltar directamente al evento que asignó a una variable su valor actual. A continuación

[1] El lector interesado se puede referir a nuestro trabajo previo para obtener más detalles [5].

describimos una sesión de depuración haciendo uso de esta funcionalidad (Figura 3a).

Después de correr el programa erróneo con el botón de arranque de TOD (1), podemos fácilmente localizar el evento correspondiente a la excepción en la huella de ejecución. Una vez que este evento se encuentra seleccionado en la vista principal de flujo de control (2), la línea de código respectiva resalta automáticamente (3). En ese momento nos damos cuenta de que el campo thumbnail del objeto ThumbnailPanel actual tiene un valor nulo (4), lo que causó la excepción. Haciendo clic en el vínculo *why?* (4) llegamos inmediatamente no sólo a la línea de código fuente donde se asigna el valor al campo (5), sino que también al evento preciso que causó esa asignación en particular (6). Nótese que la asignación ocurrió en un hilo de ejecución distinto en que ocurrió la excepción (7). Al inspeccionar el estado del programa en el momento de la asignación, uno se da cuenta que éste intentó crear una miniatura de un archivo .sh, lo cual falló.

En este ejemplo simple, TOD permitió saltar en algunos pocos pasos directamente desde el síntoma del error (la excepción) a su causa (el manejo erróneo de archivos que no son imágenes).

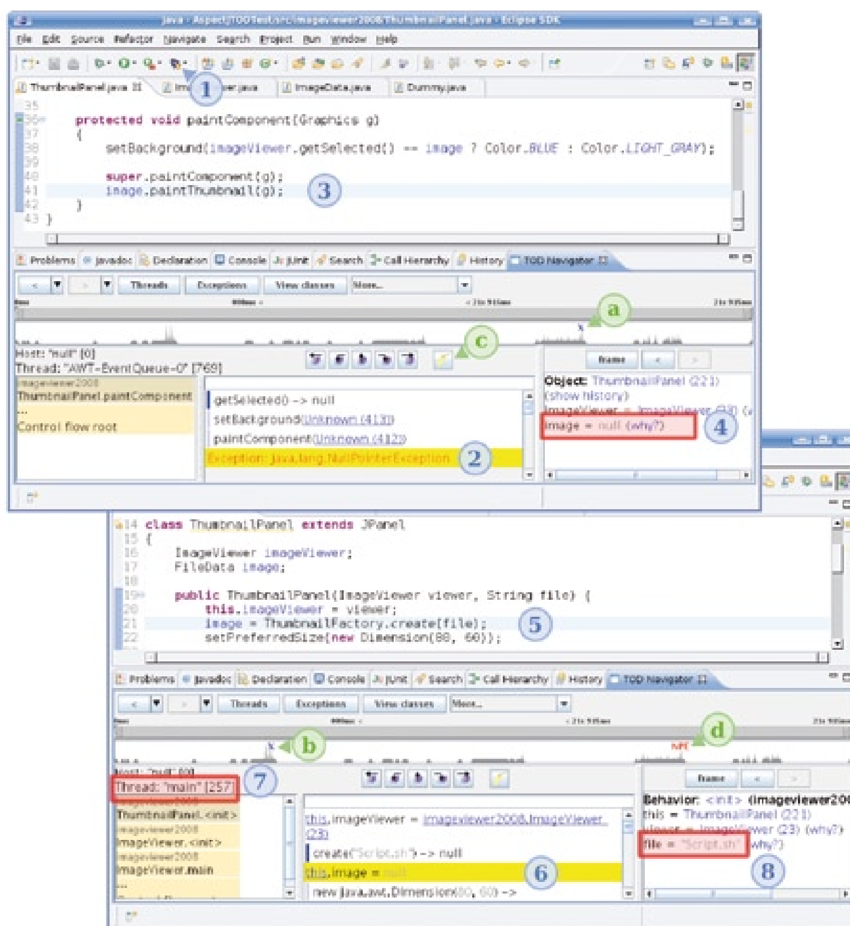
La misma búsqueda con un depurador basado en breakpoints hubiera sido más tediosa porque hay potencialmente muchos lugares en el programa donde el campo thumbnail es asignado, aparte del constructor, y hubiera sido necesario examinar paso a paso la ejecución de código sobre instancias válidas de ThumbnailPanel.

Un ejemplo de juguete como éste no muestra todo el potencial de TOD. Sería demasiado largo exponer aquí en detalles, pero podemos mencionar que hemos logrado usar TOD para solucionar rápidamente problemas difíciles, como errores en la base de datos de TOD, así como para entender problemas que encontramos en nuestro uso de programas altamente complejos como el compilador de AspectJ abc[2].

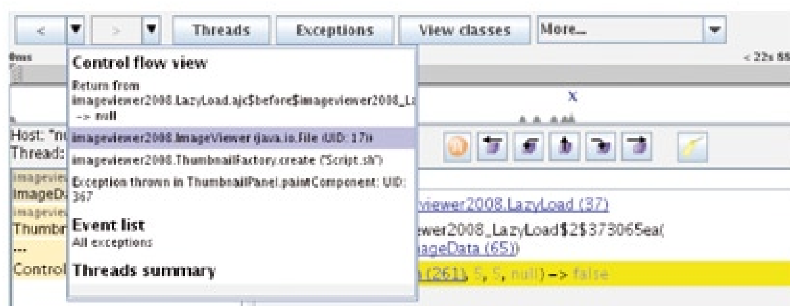
Usando marcadores para no perderse

Dada la enorme cantidad de eventos registrados por TOD, es de suma importancia ayudar al usuario a no perderse navegando en las huellas de ejecución. Para evitar eso, TOD deja al usuario marcar eventos y objetos, y permite acceder con rapidez a ubicaciones previamente visitadas.

Los eventos marcados están desplegados en una línea de tiempo arriba de la vista principal de TOD. El evento seleccionado en la vista principal también está indicado en la línea de tiempo, de tal manera que



(a) Buscando la causa raíz de un error de tipo NullPointerException usando el vínculo *why?*



(b) Historial de navegación.

Fig. 3 Interfaz de usuario de TOD.

[2] http://abc.comlab.ox.ac.uk/introduction

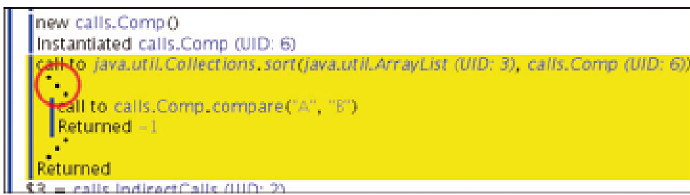


Fig. 4 TOD muestra la información que falta en una reconstitución de flujo de control.

el usuario pueda ubicarse rápidamente en relación a hitos conocidos. Esto, en particular es útil al usar el vínculo *why?* antes descrito, ya que éste puede causar saltos a eventos ocurridos lejos en el pasado, en contextos totalmente distintos.

Los globos verdes en la Figura 3a ilustran el proceso de marcación de eventos. La posición del evento actual está siempre marcada en la línea de tiempo (a, b). Cuando el usuario piensa que el evento actual es un hito importante, o un punto de partida para explorar varios recorridos de la huella de ejecución, puede marcarlo pulsando el botón de marcado (c), que también da la posibilidad de elegir un nombre y color para el evento (d).

Los objetos también se pueden marcar: es posible asignar un nombre y/o color a un objeto en particular, de tal manera que siempre aparecerá con esas características cuando esté desplegado.

Esto hace posible, por ejemplo, marcar un objeto involucrado en un error de tal manera que usos previos de este objeto sean destacados durante la navegación.

Además de los marcadores, la interfaz de usuario provee botones para navegar hacia delante y atrás, muy parecidos a los de un navegador web (Figura 3b), que permiten acceder rápidamente a todo el historial de navegación.

Soporte para huellas parciales

Si bien TOD está diseñado para aguantar huellas de ejecución muy grandes, no siempre es práctico registrar absolutamente todos los eventos: el impacto causado por la captura de huella sobre el programa depurado es considerable, así como lo son los requisitos de almacenamiento.

Para aliviar esta situación, uno puede aprovechar el hecho de que sólo algunas partes del programa son interesantes, capturando *huellas parciales* [5].

Con TOD, las huellas parciales se obtienen usando una mezcla de scoping *estático y dinámico*. El scoping estático consiste en seleccionar las clases que deben o no generar eventos. El scoping dinámico consiste en activar o desactivar la captura en tiempo de ejecución, ya sea con una simple API que se puede usar directamente desde el programa depurado, o con un botón en la interfaz del depurador. El scoping dinámico es particularmente útil cuando un error se produce sólo después de un largo tiempo de ejecución, o bajo condiciones dinámicas particulares. Por ejemplo, en una aplicación web, puede resultar interesante limitar la captura de huella al procesamiento de una consulta HTTP en particular.

El inconveniente de las huellas parciales es que son, justamente, parciales. Por lo tanto, algunas partes del historial del programa depurado no se pueden reconstituir. Para permitir al desarrollador razonar de forma correcta sobre la información disponible, TOD hace sistemáticamente *explícita* la información faltante. Por ejemplo, en la Figura 4, los puntitos indican que la información de flujo de control es incompleta: entre la ejecución de `sort` y la de `compare`, ocurrieron operaciones no registradas porque el método `Collections.sort` del JDK estaba fuera del scope estático.

En la práctica, los beneficios de las huellas parciales superan sus inconvenientes. En nuestra experiencia, combinar scoping estático y dinámico ha sido indispensable para depurar programas que hacen uso intensivo de la CPU por largos tiempos, tal como la base de datos de TOD.

ESTADO ACTUAL DEL ARTE

La mayoría de los ambientes de desarrollo modernos proveen un depurador basado en breakpoints. Todos tienen más o menos las mismas capacidades: breakpoints normales o condicionales, ejecución paso a paso hacia adelante, inspección del marco de activación actual y de los objetos alcanzables desde él. Pero también existen algunos depuradores omniscientes disponibles hoy. Los describimos brevemente a continuación.

Depuradores omniscientes para Java

ODB [3] es uno de los primeros depuradores omniscientes para Java. Al igual que TOD, obtiene huellas de ejecución instrumentando las clases a medida que están cargadas por la JVM; sin embargo, ODB almacena la huella de ejecución dentro de la JVM depurada, lo que causa algunos problemas: el tamaño de la huella está limitado por la memoria heap disponible, y referencias a objetos que ya no están en uso están conservadas, impidiendo la recolección de basura. Un aspecto único de ODB es su capacidad para reanudar la ejecución del programa desde cualquier punto en el tiempo con un estado alterado.

Java Whyline [6] deja el usuario seleccionar preguntas sobre por qué cierto comportamiento *ocurrió* o *no ocurrió*. Estas preguntas están generadas en forma automática usando una combinación de análisis estático y dinámico, y pueden abarcar no solamente el estado interno del programa (por ejemplo, "¿por qué la variable x tiene el valor y?"), sino también su salida textual y gráfica, hasta el nivel de píxeles individuales. A pesar de que Whyline puede

aguantar huellas relativamente grandes (por ejemplo, 35 millones de eventos), su escalabilidad es limitada por el hecho de que el análisis se hace en memoria.

JIVE [7] es un *ambiente de visualización* interactivo para programas Java. Provee diagramas de secuencia parecidos a los de UML, pero extendidos con información acerca del llamado de método actual. El nivel de detalle de los diagramas puede ser reducido para poder desplegar más información en pantalla, pero no es claro que este mecanismo pueda escalar a más de unos centenares de elementos. JIVE soporta ejecución paso a paso hacia adelante y atrás, pero no navegación causal rápida. La huella de ejecución es capturada con JPDA, la interfaz de depuración de la JVM, y procesada en memoria, lo que limita la escalabilidad del sistema.

Otras plataformas

Lisp: En el 1984, ZStep [8] proveía un simulador de ejecución paso a paso para Lisp: permitía ir paso a paso hacia adelante y atrás, así como ver el resultado de la evaluación de las expresiones en paralelo al código fuente correspondiente. Su seguidor, ZStep95 [9], agregó la posibilidad de relacionar las salidas gráficas del programa al evento que la causó, y también proveía controles similares a una grabadora de cinta para facilitar la navegación. Sin embargo estos sistemas no manejaban efectos de borde, vínculos causales (excepto para las salidas gráficas), o problemas de escalabilidad.

Nativos:

TimeMachine, de Green Hills Software[3], es un depurador omnisciente para sistemas embedidos (PowerPC, ARM y arquitecturas similares). En algunas plataformas, una sonda de hardware permite capturar huellas de ejecución sin tener ningún impacto sobre el programa depurado; para las otras plataformas se usa la tradicional instrumentación de código. Además de las funcionalidades habituales de los depuradores omniscientes, TimeMachine se puede usar como herramienta de profiling.

UndoDB, de Undo Ltd[4], es un depurador omnisciente para programas Linux x86 nativos. Al contrario de la mayoría de las otras herramientas presentadas aquí, es basado en un mecanismo de instantáneo/reejecución: se obtiene periódicamente un *instantáneo* (o *snapshot*) de la memoria del proceso, y se usa una técnica de reejecución para reconstituir el estado del programa entre los instantáneos. El uso de este mecanismo resulta en un impacto reducido sobre el programa depurado, pero no permite navegación causal.

	Platform	*Mechanism*	*Storage media*	*History size*	*Runtime overhead*	*Partial traces*	*Causal nav.*	*High-level overviews*	*IDE integration*
TimeMachine	Embedded	?	RAM/ Probe	1e9	Soft.: ? Hard.: none	?	?	✔	Part of Green Hill's MULTI IDE
UndoDB	Linux	Checkpoint replay	RAM	Not applicable	7x	Not applicable	✘	✘	Wrapper for gdb
Chronicle	Linux	Event log	Disk	1e9	300x	?	✔	✘	Plugin for Eclipse CDT
[Lienhard]	Squeak	Event log	RAM	1e5	6x	Events on unreachable objects are discarded	✘	✘	Integrates into the platform
Unstuck	Squeak	Event log	RAM	1e5	250x	Lexical scoping	✘	✘	Integrates into the platform
Whyline	Java	Event log	Disk	1e7	252x/20x	Lexical scoping	✔	✘	No
JIVE	Java	Event log	RAM	?	?	Lexical scoping	✘	✔	Plugin for Eclipse JDT
ODB	Java	Event log	RAM	1e6	95x/37x	Lexical scoping	✔	✘	Limited Eclipse integration
TOD	Java	Event log	Disk	1e9	83x/28x	Lexical & dynamic scoping Missing info explicit in GUI	✔	✔	Plugin for Eclipse JDT/AJDT

Fig. 5

La columna *History size* indica el orden de magnitud de la cantidad de eventos que se pueden razonablemente almacenar y procesar. *Runtime overhead* da una idea del impacto sobre el programa depurado. Para los sistemas en los cuales hicimos nuestros propios experimentos, aparecen dos cifras (X/Y) donde X es el impacto en el peor caso (es decir, un programa enteramente instrumentado que usa intensivamente la CPU), e Y corresponde a una situación más típica (en nuestro caso, una ejecución del limpiador de código HTML jTidy). Para los demás sistemas, la única cifra es la indicada por el propio autor del sistema. En la columna *Partial traces, lexical scoping* significa que es posible seleccionar las clases o paquetes que se instrumentan, y *dynamic scoping* significa que se puede activar o desactivar la captura en tiempo de ejecución. *Causal navigation* indica si el depurador permite navegar directamente al evento que asignó su valor actual a una variable. *High-level overviews* indica si el sistema puede proveer vistas resumidas del programa depurado.

[3] http://www.ghs.com/products/timemachine.html

[4] http://undo-software.com/

Chronicle[5] es un depurador omnisciente open-source para programas Linux x86 nativos. Su arquitectura es similar a la de TOD: los binarios son instrumentados de tal manera que envían la huella de ejecución a una base de datos externa, almacenada en disco. Una característica clave de Chronicle es la compresión e indexación agresiva de los eventos, lo que permite registrar huellas muy grandes y procesar consultas eficientemente.

Smalltalk:

Unstuck [10] es un depurador omnisciente para Smalltalk, muy similar a ODB en su arquitectura y operación. Lienhard *et al.* [11] proponen otro depurador omnisciente para Smalltalk que trata el problema de escalabilidad usando huellas parciales, pero en una manera muy distinta a la de TOD. Ellos postulan que la información acerca de objetos que ya no están alcanzables en un cierto momento (es decir, objetos que pueden ser recolectados por el recolector de basura) puede ser descartada. Si bien descartar esa información mejora mucho la eficiencia del sistema, pensamos que la causa raíz de un error puede haber ocurrido en el contexto de objetos que han sido descartados mucho antes de que los síntomas del error se manifiesten, lo que vuelve este enfoque inoperativo en algunos casos.

Resumen

La figura 5 resume las características de las herramientas antes descritas. A continuación mencionamos las características de TOD que, en nuestra opinión, hacen de él una alternativa competitiva:

- La base de datos escalable permite registrar y consultar eventos en forma rápida. Además, puede ser distribuida sobre un cluster de máquinas para aumentar aún más su eficiencia. Eso hace de TOD el depurador omnisciente más escalable para la plataforma Java.
- El soporte para huellas parciales aumenta de sobremanera la aplicabilidad de TOD porque ofrece una manera expresiva de especificar una captura de huella selectiva, y porque reporta de manera adecuada información faltante. Pensamos que TOD provee el soporte más extensivo para huellas parciales.
- La interactividad de la interfaz, obtenida gracias al procesamiento rápido de consultas, permite navegar en forma interactiva en huellas de ejecución muy grandes.
- Las metáforas de interacción especializadas, como el vínculo *why?*, los marcadores y las líneas de tiempo, permiten una navegación y comprensión de programas eficiente.
- La integración con Eclipse permite integrar fácilmente TOD en el proceso de desarrollo.[6] Por otro lado, algunas de las funcionalidades otorgadas por diversos sistemas hacen falta en TOD:
- Whyline permite formular preguntas negativas como *"¿por qué el método foo no se ejecutó?"*; ese tipo de preguntas es recurrente en el proceso de depuración.
- Whyline permite relacionar las *salidas textuales y gráficas* del programa con el evento que las causó. El soporte para salidas textuales en TOD está considerado, pero el soporte para las salidas gráficas requeriría un trabajo considerable.
- TOD tiene un *impacto sobre el programa depurado* similar al de ODB y de Whyline, y a la vez provee una escalabilidad mucho mayor. Sin embargo, sistemas como UndoDB y el de Lienhard tienen un impacto mucho menor, mientras TimeMachine puede no tener ningún impacto al usar sondas de hardware. Nos esforzamos por reducir el impacto de TOD usando análisis estáticos para limitar la cantidad de información redundante capturada.
- JIVE provee *visualizaciones del grafo de objetos,* lo que puede ser muy útil para la comprensión de programas. Aunque TOD soporta formateadores personalizados, estos son solamente textuales.

Pleiad

Programming Languages and Environments for Intelligent, Adaptable and Distributed systems

PERSPECTIVAS

Mientras la depuración omnisciente parece estar atrayendo poco a poco más la atención de los industriales, nuevos desafíos están apareciendo con la emergencia de nuevos lenguajes y paradigmas de programación. Ya podemos identificar tres áreas mayores en las cuales el desarrollo de sistemas de depuración omnisciente prácticos es crucial:

- Los lenguajes dinámicos (como Python, Ruby...) se están haciendo cada vez

[5] http://code.google.com/p/chronicle-recorder/
[6] La integración con NetBeans e IntelliJ está en curso.

más populares. Mejorar el soporte de depuración ayudaría a aliviar, al menos hasta cierto grado, la ausencia de verificación de tipos estática.

- Desarrollar sistemas concurrentes y distribuidos es notoriamente difícil, en particular porque errores pueden ser difíciles de reproducir. Ser capaz de registrar automáticamente historiales de ejecución para luego navegar en ellos es entonces de gran importancia.
- Porque agrega más lugares donde ocurre el enlace tardío, la Programación Orientada a Aspectos (AOP) [12] hace más difícil para los programadores la reconstrucción mental del flujo de ejecución de los programas. Se requieren herramientas de desarrollo adecuadas, y en particular depuradores, para soportar AOP.

Ya estamos explorando como la depuración omnisciente puede proveer un soporte adecuado para AOP [13]. También estamos desarrollando una versión de TOD para Python. La programación concurrente y distribuida también está en nuestra agenda de investigación.

Dado lo eficiente que es la depuración omnisciente para la comprensión de programas, mejora de sobremanera el proceso de desarrollo de software. Es entonces sumamente importante dedicar esfuerzos para hacerla práctica y aplicable a la mayor cantidad de situaciones posibles, resolviendo los distintos desafíos para su adopción. BITS

Disponibilidad. TOD está disponible en http://pleiad.dcc.uchile.cl/tod/

REFERENCES

[1] National Institute of Standards and Technologies, "Software errors cost U.S. economy $59.5 billion annually," June 2002. http://www.nist.gov/public affairs/ releases/n02-10.htm.

[2] M. Eisenstadt, "My hairiest bug war stories," *Commun. ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[3] B. Lewis, "Debugging backwards in time," in *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)* (M. Ronsse and K. D. Bosschere, eds.), (Ghent, Belgium), 2003.

[4] R. M. Balzer, "EXDAMS– extendable debugging and monitoring," in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 567–580, 1969.

[5] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, (Montreal, Canada), pp. 535–552, Oct. 2007. , 42(10).

[6] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in ICSE 2008: *Proceedings of the International Conference on Software Engineering*, 2008.

[7] P. Gestwicki and B. Jayaraman, "Methodology and architecture of JIVE," in SoftVis '05: *Proceedings of the 2005 ACM symposium on Software visualization*, (New York, NY, USA), pp. 95–104, ACM, 2005.

[8] H. Lieberman, "Steps toward better debugging tools for lisp," in LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, (New York, NY, USA), pp. 247–255, ACM, 1984.

[9] H. Lieberman and C. Fry, "ZStep 95: A reversible, animated source code stepper," in *Software Visualization — Programming as a Multimedia Experience* (J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, eds.), (Cambridge, MA-London), pp. 277–292, The MIT Press, 1998.

[10] C. Hofer, M. Denker, and S. Ducasse, "Implementing a backward-in-time debugger," in *Proceedings of NODe'06*, vol. P-88, pp. 17–32, Lecture Notes in Informatics, 2006.

[11] A. Lienhard, T. G˜A R rba, and O.N˜ ierstrasz, "Practical object-oriented back-in-time debugging," in *Proceedings of ECOOP'08: European Conference on Object-Oriented Programming (to appear)*, 2008.

[12] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming," *Communications ACM*, vol. 44, Oct. 2001.

[13] G. Pothier and É. Tanter, "Extending omniscient debugging to support aspect-oriented programming," in *Proceedings of the 23rd ACM Symposium on Applied Computing (SAC 2008)*, vol. 1, (Fortaleza, Cear´a, Brazil), pp. 266–270, Mar. 2008.

Nota y Agradecimientos. Este artículo es una traducción y leve adaptación de nuestro artículo "Back to the Future: Omniscient Debugging" aceptado para publicación en el journal IEEE Software.

Agradecemos a Greg Law de Undo Ltd. por proveer información técnica sobre UndoDB, así como a Alexandre Bergel, Johan Fabry, Adrian Lienhard, Olivier Motelet y los revisores anónimos de IEEE Software por sus valiosos comentarios.