Delta Debugging

Carlo Curino, Alessandro Giusti
curino@elet.polimi.it, lale@leet.it

ABSTRACT

Il debugging è una delle attività più onerose nel processo di sviluppo del software; in particolare il compito più arduo ed imprevedibile in termini di tempo è quello di isolare la sorgente del problema. Il Delta Debugging è una tecnica innovativa, sistematica ed automatica, che fornisce una solida base teorica per affrontare proprio questo compito; le applicazioni sperimentate finora riguardano l'input di programmi, il codice, lo stato e lo scheduling: ciascuna ha all'attivo importanti successi su scenari reali. Benchè l'intervento del programmatore sia comunque necessario, questa nuova tecnica permette in molti casi una sensibile riduzione dello sforzo dedicato al Debugging.

1. INTRODUZIONE

Presenteremo in questa sezione le motivazioni che hanno spinto allo sviluppo di questa tecnica, con riferimento ai passi fondamentali di un generico processo di debugging.

1.1 Il debugging e lo strumento automatico

Il debugging è definito come "un processo metodico per trovare e ridurre il numero di *bug*, o *difetti*, in un programma..."[1]. Tuttavia, solo una ridotta minoranza di programmatori sarebbe d'accordo nel definire "metodico" il processo di debugging; abitualmente si tratta di un mix tra arte e improvvisazione, in cui il fattore tempo è un'incognita non definita fin quasi al termine del processo.

Una possibile formalizzazione del debugging consiste nel dividere l'attività nelle seguenti fasi:

1. riconoscere l'esistenza del problema;
2. isolarne la sorgente;
3. identificare il difetto nel codice che causa il problema;
4. determinare una soluzione;
5. applicare la soluzione.

[1]Wikipedia.com, Giugno 2005

Il *testing* è una disciplina che affronta il primo di questi passi: un problema non si può risolvere se non si sa che esiste. D'altra parte, questa è anche un'attività ben formalizzata, e che può contare su un potente arsenale di strumenti (quasi) automatici; il testing automatizzato si avvale comunque del programmatore, che sfruttando la propria conoscenza del software, scrive le *funzioni di test* che restituiscono in modo formale un fallimento (il che implica presenza di difetti), o un successo (che non implica nulla, visto che i difetti possono rimanere latenti). A questo punto il testing automatizzato si preoccupa semplicemente di utilizzare questo potente strumento per allertare il programmatore non appena una funzione di test evidenzia la presenza di un difetto.

Le restanti attività, nella pratica odierna, sono spesso lasciate alla pura improvvisazione, al più aiutata in modo indiretto da strumenti automatici. Il nome (promettente) di *Debugger*, infatti, indica oggi un software più o meno sofisticato, il cui compito è di *coadiuvare* il programmatore nella pratica del debugging. Sfortunatamente chiunque abbia usato un *debugger* si sarà accorto che l'aiuto ottenuto, seppur utile è spesso piuttosto limitato. Un *debugger* permette di analizzare (ed eventualmente modificare) lo stato di un programma durante la sua esecuzione, che può essere interrotta, ripresa, e manipolata a piacimento; talvolta offre anche funzioni per studiare nel dettaglio il contenuto di strutture dati articolate e complesse; ma rimane pur sempre l'equivalente di un cacciavite per un meccanico che deve riparare un motore: indispensabile, ma null'altro che uno strumento passivo.

Statistiche recenti sui costi di debugging hanno mostrato che *isolare la sorgente* di un problema è in molti casi l'attività più dispendiosa in termini di tempo: si calcola che possa raggiungere il 90% dell'intero processo di debugging. Nel debugging di software, agire contemporaneamente su un corpus di centinaia di migliaia di righe di codice, di righe di input, o di locazioni di memoria rende spesso improba la ricerca del *bug*. L'isolamento del problema permette di circoscrivere l'attenzione ad un più limitato sottoinsieme del sistema, rendendo così il problema più trattabile.

A seconda della situazione, questa operazione può essere affrontata in modo più o meno analitico da parte del programmatore. Ad un estremo, meditare sul problema verificatosi

e sugli eventuali output, insieme a una conoscenza approfondita della struttura logica del programma, può portare a una specifica ipotesi sulla sorgente del problema, che può poi essere velocemente verificata in modo sperimentale. D'altra parte, una scarsa conoscenza del software, o una eccessiva complessità dello stesso può imporre al programmatore un approccio puramente sperimentale: una serie di proved-errori più o meno guidate dall'analisi, effettuando varie modifiche e tentativi che prima o poi portano alla diagnosi. Su questo punto si concentra lo sforzo di miglioramento del team che ha concepito la metodologia del *Delta Debugging* (DD).

Il passo successivo merita un chiarimento; la sorgente di un problema non coincide necessariamente con il codice "difettato": la sorgente (entità comunque generica, su cui approfondiremo in seguito), trovata al punto precedente potrebbe essere infatti una sezione di codice perfettamente corretto, che però crea una situazione inconsistente a causa di problemi che già esistevano ma erano latenti; potrebbe essere anche un input che soddisfa determinate condizioni, o un insieme di bindings variabile-valore; ciò che il programmatore deve correggere è però il *difetto nel codice del programma*, che non coincide necessariamente con la sorgente del problema (per esempio un particolare input): tra queste due entità sussiste perciò una relazione, in molti casi molto evidente, in altri meno; molto spesso comunque conoscere la sorgente del problema è un valido aiuto per identificare il difetto nel codice.

Sarà proprio questo difetto che deve essere rimosso dopo aver individuato una possibile correzione; correzione peraltro non sempre banale ed immediata, dal momento che può essere necessario rivedere interi algoritmi: non è scontato infatti che un *difetto* nel codice debba essere un errore semplice e localizzato.

Come anticipato, il livello di automazione di questo procedimento è, a valle della fase di testing, praticamente nullo; tutto è nelle mani del programmatore, che, guidato dall'istinto e dall'esperienza, procede nelle fasi di *isolamento* della sorgente del problema, *identificazione* e *correzione* del difetto (*bug*).

E' ovvio che qualsiasi miglioramento nell'ambito del debugging è in generale considerato con grande interesse; e benchè la possibilità di sostituire il programmatore in un procedimento così creativo sia fuori discussione, il fatto che alcuni procedimenti possano essere automatizzati grazie ad un approccio automatico e puramente sperimentale (e non analitico) è di grande rilevanza.

1.2 Un caso reale: genesi del Delta Debugging

Luglio 1998: sulla mailinglist del progetto DDD, un un tool grafico per il debugging, apparve una notizia allarmante: il software non si integrava con la più recente release 4.17 di GDB (il debugger del progetto GNU), mentre funzionava perfettamente con tutte le versioni fino alla 4.16.

Per *isolare* la sorgente del problema, era necessario scoprire quale cambiamento nel codice di GDB interferiva con l'integrazione dei due programmi, ma le differenze tra le due release ammontavano ad almeno 178.000 righe di codice: verosimilmente solo una minima parte di queste erano responsabili del misfatto.

In questa situazione Andreas Zeller, leader del progetto DDD, concepì l'approccio del "Delta Debugging": una nuova procedura algoritmica – quindi automatizzabile – per assistere il programmatore nella pratica del debugging. In particolare, questo procedimento viene in aiuto nella fase di *isolamento*, che come evidenziato in precedenza costituisce spesso il sottoproblema più oneroso ed imprevedibile in termini di tempo.

1.3 I delta e le sorgenti di malfunzionamento

Sono molti gli scenari in cui ci si trova in una situazione simile a quella appena descritta: ogni qualvolta si hanno due configurazioni[2], di cui una manifesta un problema e l'altra no, si può sicuramente imputare il manifestarsi del problema alle differenze (i *delta*) tra le due. Questo non significa che l'errore nel programma risieda obbligatoriamente nel set di delta, ma solo che qualche porzione di questo set lo *renda evidente*. Quello che però ci interessa maggiormente ai fini del debugging è sapere quali tra i delta sono rilevanti perché si abbia proprio quel malfunzionalmento: la gran parte delle differenze tra le due configurazioni, infatti, saranno generalmente indipendenti e non correlate al manifestarsi del problema; sapendo ciò potremo ignorarle completamente durante le successive fasi del processo di debugging.

2. ASPETTI GENERALI

In questa sezione si esamineranno alcuni aspetti fondamentali del DD: si presenteranno gli ambiti a cui può essere applicato, e si richiameranno alcuni concetti fondamentali del testing che rivestono particolare importanza per questo approccio; in seguito si descriverà, prima in modo generale e poi in modo più formale e preciso, come opera l'algoritmo principale di isolamento; infine si analizzeranno gli aspetti relativi all'efficienza.

2.1 Ambiti applicativi

E' stato introdotto il concetto di differenze tra due configurazioni, la base su cui opera l'algoritmo di Delta Debugging (a cui ci riferiremo come *dd*); eccone una breve panoramica:

1. differenze nell'*input* del programma: se il medesimo programma mostra dei malfunzionamenti quando gli viene presentato uno specifico ingresso I_x, mentre funziona correttamente con un secondo input I_v; consideriamo tali differenze come i delta su cui lavorerà l'algoritmo che verrà a breve presentato.

2. differenze nel *codice* del programma: se uguali condizioni di esecuzione danno origine a malfunzionamenti in una specifica revisione del programma C_x, mentre non creano problemi a una differente versione C_v; consideriamo le differenze di codice tra le due revisioni del programma come i delta dell'algoritmo *dd*.

[2]Il concetto di configurazione è qui utilizzato per indicare in senso generale uno qualsiasi degli elementi sui quali, come vedremo, il DD può operare: input del programma, codice sorgente, scheduling di thread, stati del programma

3. differenze nello *scheduling* del programma: se lo stesso programma multithreaded, con medesimo input, presenta dei malfunzionamenti solo con alcune soluzioni di scheduling, non con altre; considereremo le differenze dei punti di thread switch come i delta dell'algoritmo *dd*.

Oltre a queste applicazioni, il DD può essere anche applicato alle differenze tra gli stati di un'istanza di esecuzione di un programma destinata a presentare problemi e una destinata a funzionare correttamente. Questo quarto approccio presenta alcune differenze rispetto a quelli introdotti finora, e verrà presentato ed approfondito nella Sezione 6.

2.2 Risultati

Il DD evidenzia come risultato un sottoinsieme delle differenze presentate sopra, più piccolo possibile, strettamente correlato con il presentarsi o meno del malfunzionamento. In generale vengono ritornate due istanze di configurazione, più simili possibile, tali che in una si presenta il problema e nell'altra no. Le differenze (minime) tra queste due configurazioni solitamente evidenziano in modo preciso la sorgente del problema;

1. quando applicato all'input del programma, il risultato sono due diverse stringhe di input, che presentano differenze il più limitate possibili (se possibile un solo carattere).
2. applicato al codice del programma, si ottengono due versioni del codice, con minime differenze tra loro (se possibile una sola istruzione).
3. quando applicato allo scheduling del programma, l'algoritmo ritorna due soluzioni di scheduling il più simili possibile (uno specifico thread switch anticipato o posticipato di un solo yield point[3]).

Una variante dell'algoritmo di DD, che denoteremo con *ddmin*, è stata sviluppata nel contesto dell'applicazione agli input di un programma, il cui risultato sarà in generale una stringa di input 1-minima, e non una coppia di stringhe a minima distanza.

2.3 Framework di testing

Verranno ora richiamati alcuni fondamenti della disciplina del testing che sono utilizzati in modo massiccio nelle procedure di DD.

Un test è una procedura che verifica un programma o una sua parte, fornendo uno specifico input, e verificando che l'output (in senso lato) sia quello previsto.

Lo standard POSIX 1003.3 per i framework di testing prevede l'utilizzo di una serie di possibili risultati:

0 PASS

1 FAIL

2 UNRESOLVED

3 NOTINUSE

4 UNSUPPORTED

5 UNTESTED

6 UNINITIATED

7 NORESULT

Nell'ambito del DD, si considerano solo tre di questi: "*Pass*", "*Fail*", e "*unresolved*".

1. *Pass* viene utilizzato quando il test ha successo: il programma si è comportato nel modo atteso, e non si è evidenziato alcun problema.
2. *Fail* viene ritornato se si manifesta il problema sotto esame.
3. *unresolved* indica che si è verificato un problema differente, per cui non è chiaro se il problema oggetto di studio si sarebbe verificato o meno. Ad esempio il problema che si sta cercando di risolvere è un errore di calcolo ed il problema che si è verificato in questo test è un crash del programma.

Uno dei pochi prerequisiti necessari per l'applicazione dell'algoritmo *dd* è la capacità, al termine di una esecuzione di un caso di test, di collocarne l'esito in una delle tre categorie presentate sopra.

Risulta chiaro come il discriminare gli esiti *Fail* da quelli *unresolved* dipenda in modo diretto da come il problema è stato definito, e da quali sono i "sintomi" che permettono di riconoscere se il problema verificatosi è proprio quello che si sta studiando o meno.

Questo è un nodo critico dell'intera metodologia, almeno nel caso in cui la failure sia rappresentata da un crash del programma; il team di Andreas Zeller propone una soluzione pragmatica ma efficace: si confrontano gli *Stack trace* a seguito di un crash; tali metodi euristici sono implementati anche in AskIgor[4] [4]. Benchè siano possibili meccanismi più precisi ed evoluti, questo sistema sembra offrire risultati accettabili.

Nel caso più generale il difetto non si manifesta necessariamente tramite un *crash*, sinora semplicisticamente usato come evidente esempio di errore. Qualunque comportamento del programma che differisca da quello atteso può essere considerato un errore. Sarà necessario definire un criterio molto preciso affinchè il fallimento del test sia in stretta relazione con il manifestarsi del problema considerato *e non di un altro*. Se ad esempio il test di una funzione fallisce quando viene restituito un valore errato (ovvero, l'errore che stiamo ora considerando si manifesta in questo modo), nel caso in

[3]Punto stabile all'interno del codice, tipicamente l'ingresso di una chiamata di funzione, dove il thread switch avviene in modo sicuro.

[4]Servizio di debugging remoto offerto dal team del DD raggiungibile presso: www.askigor.com

cui la funzione non termini a causa di un'eccezione *non potremmo sapere* se il nostro problema si sarebbe manifestato o meno – il test deve ritornare *unresolved*; nello stesso scenario, nel caso in cui la funzione ritorni `null` (quando non dovrebbe), non c'è dubbio che un problema si sia verificato: non abbiamo però informazioni sul fatto che si sia verificato proprio il *nostro* problema, di conseguenza anche in questo caso il test deve ritornare *unresolved*.

2.4 Funzionamento generale dell'algoritmo

Presentati i risultati del DD, il funzionamento e il significato delle funzioni di test, presentiamo ora l'algoritmo generale *dd*. L'algoritmo che presenteremo può essere utilizzato senza modifiche per tutte le applicazioni del DD a cui abbiamo fatto riferimento, la presentazione sarà quindi volutamente generale.

L'input dell'algoritmo sono due configurazioni, una delle quali funziona correttamente ($Conf_v$) mentre l'altra determina uno specifico problema ($Conf_x$).

Il problema che l'algoritmo risolve è l'isolamento delle differenze tra le due configurazioni che, se applicate o rimosse, determinano il presentarsi o meno dell'errore in esame.

La strategia di base è semplice ed itera su questi 3 punti:

1. Genera un caso di test a partire dalle configurazioni $Conf_v$ e $Conf_x$, combinandone i delta.
2. Esegue il programma con questa configurazione.
3. In base all'esito modifica il test scartando o tenendo parte dei delta applicati, secondo i criteri che saranno presentati nel seguito.

2.4.1 L'idea iniziale

Sembra a questo punto possibile applicare un procedimento ispirato alla ricerca binaria: partendo dal $Conf_v$, si aggiunge la metà delle differenze, ottenendo una nuova configurazione; si verifica poi per mezzo del test se questa nuova configurazione presenta o meno il problema in esame:

1. in caso affermativo, abbiamo determinato che la sorgente dello stesso è da ricercarsi nelle differenze che abbiamo appena aggiunto, e possiamo in tal modo dimezzare lo spazio di ricerca per la prossima iterazione;
2. in caso negativo, se la nuova configurazione passasse il test, abbiamo determinato che possiamo escludere che le differenze appena aggiunte siano rilevanti per il presentarsi del problema.

Da un algoritmo siffatto, potremmo aspettarci una complessità logaritmica e un tempo di esecuzione estremamente prevedibile. Risulta però abbastanza evidente come la realtà non sia così rosea: non abbiamo infatti tenuto conto della possibilità di risultati *unresolved*. Sono state fatte inoltre in modo implicito una serie di ipotesi, come la non-interattività del programma e il suo determinismo, che possono sembrare molto forti; fortunatamente, quest'ultima limitazione non crea problemi poichè può essere facilmente aggirata.

2.4.2 Le configurazioni "unresolved"

Ciò che differenzia l'algoritmo generale *dd* da una semplice ricerca binaria è la gestione del risultato "*unresolved*". La versione definitiva dell'algoritmo è guidata dai seguenti principi:

1. Un risultato *unresolved* non porta alcuna informazione riguardo al manifestarsi o meno dell'errore in una data configurazione; in tale iterazione non si ha di conseguenza alcun progresso: le configurazioni *unresolved* sono pertanto da evitare, per quanto possibile.
2. se una configurazione è *unresolved* è perché si è verificato un errore diverso da quello sotto esame: per uscire dall'*empasse*, si possono adottare due strategie, che sostanzialmente cambiano la granularità delle differenze applicate:
   (a) diminuire il numero di cambiamenti applicati a $Conf_v$: questo rende più probabile ottenere una configurazione che passa il test;
   (b) aumentare il numero di cambiamenti applicati a $Conf_v$: questo rende più probabile ottenere una configurazione che fallisce il test, ma non è Unresolved.

2.5 L'algoritmo *dd*: formalizzazione

Riportiamo in Figura 1 la formalizzazione dell'algoritmo come presentata da Andreas Zeller in [7].

La Figura 1 evidenzia come l'algoritmo *dd* sia definito come algoritmo ricorsivo. La formalizzazione mostra inoltre come procedendo in modo simile ad una binary search il set di differenze tra $Conf_v$ e $Conf_x$ vengano ridotte ad ogni passo. Tuttavia quando applicando un set di modifiche il risultato è "*unresolved*" viene aumentata la granularità e il passo ripetuto.

2.6 Considerazioni generali di efficienza

La terminazione dell'algoritmo è garantita: si andrà a aumentare la granularità sino a considerare singolarmente ogni delta[5]. Questo porta il caso pessimo dell'algoritmo ad eseguire un numero di test pari a:

$$| Conf_x |^2 +3 | Conf_x |$$

dove $| Conf_x |$ indica la cardinalità di $Conf_x$ in termini di delta.

Tuttavia, questo valore si riferisce ad un caso *particolarmente* sfortunato: si tratterebbe infatti di un caso in cui ogni test risulti "*unresolved*" fino all'ultimo, che evidenzia il delta rilevante per il manifestarsi dell'errore. Nel caso ottimo la complessità dell'algoritmo è

$$2 \log_2 | Conf_x |$$

. Gli autori sostengono che nelle applicazioni reali da loro testate, la complessità dell'algoritmo si assesta molto più verso il logaritmo del caso ottimo che non verso il polinomio del caso pessimo. Si potrebbe considerare il caso pessimo

[5]L'algoritmo è *n-minimo*, ossia non considera l'insieme delle parti dei delta; arriva al più a considerare gruppi di n delta per volta, dove $n = 1$ nell'implementazione corrente.

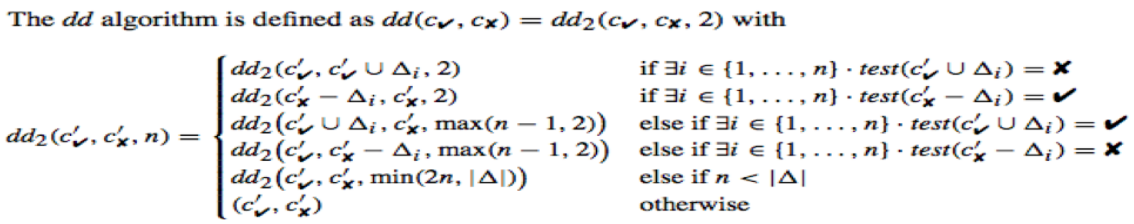

Figure 1: Formalizzazione dell'algoritmo *dd*.

non poi così grave, si tratta comunque di un polinomio di secondo grado. Tuttavia, come vedremo quando parleremo di thread scheduling come applicazione del DD, la dimensione dell'input $| Conf_x |$ può essere di miliardi di delta (quasi 4 miliardi nell'esempio riportato), per cui anche un polinomio di secondo grado rende l'uso pratico dell'algoritmo particolarmente problematico.

3. APPLICAZIONE ALL'INPUT DEL PROGRAMMA

Vedremo ora come il DD sia stato applicato con successo al problema dell'isolamento della porzione *Failure Inducing* dell'input di un programma. Per questa specifica applicazione è stata realizzata una variante dell'algoritmo generale *dd*. Questa variante, a cui ci riferiremo come *ddmin*, produce come risultato non una coppia di casi di test a minima distanza, bensì un singolo input *1-minimo* [6], ovvero un input dal quale rimosso un qualsiasi delta l'errore non si manifesta. Questa variante, computazionalmente meno efficiente, restituisce al programmatore un risultato molto più utilizzabile in questo contesto.

Ipotizziamo di avere un programma che ricevendo un determinato input si comporti in modo scorretto, andando in crash o assumendo un generico comportamento inatteso. Questo input conterrà al suo interno una porzione *Failure Inducing*, spesso molto limitata rispetto alla dimensione complessiva dell'input. Il primo compito del programmatore che si occupa di un debugging di questo tipo sarà proprio la riduzione dell'input al minimo *Failure Inducing*, per poter così più facilmente capire in quale porzione di programma si annidi l'errore.

Un esempio concreto presentato in [7] è quello di un bug del browser *Mozilla* [3]. Cercando di stampare una determinata pagina HTML (circa 900 righe di codice) il browser andava in crash (segmentation fault). Applicando l'algoritmo *ddmin*, la variante di minimizzazione, dopo 57 test completamente automatici l'input è stato ridotto al solo tag `<SELECT>`. Il debugging a questo punto è risultato facile: la porzione di programma di gestione della stampa del tag `<SELECT>` è stata presa in esame e il problema risolto: il DD ha offerto un efficace metodo per "zoomare sul problema, che è stato infine individuato e risolto dal programmatore.

[6]Non è in generale detto che questo sia un minimo globale e nemmeno locale, potrebbero esistere infatti combinazioni di rimozione di più delta che portano ad un caso ancor più piccolo ma sempre *Failure Inducing*. Si parla infatti di 1-minimalità quando la rimozioni di gruppi di al più un delta non permettono di ridurre ulteriormente l'input.

Per applicare il DD all'input di un programma avremo in generale bisogno di:

1. un input che manda il programma in stato di errore: I_x
2. un input per cui il programma esegue correttamente il compito applicativo: I_v, che in generale può essere un input nullo (se il programma accetta come corretto un input nullo) come nell'esempio citato relativo a Mozilla.
3. la possibilità di eseguire ripetutamente il programma con input generati dalla combinazione dell'input I_x e dell'input I_v.
4. la capacità di distinguere tra "*Fail*", "*Pass*", "*unresolved*".

Come già introdotto esistono due sostanziali varianti dell'algoritmo: le Figure 2 e 3 mostrano l'ultima porzione di esecuzione relativa all'esempio di Mozilla riportato precedentemente per le due varianti dell'algoritmo.

1. *ddmin*: come mostrato in Figura 2, partendo da un input I_x l'algoritmo permette di ridurlo alla sua parte più piccola che genera comunque l'errore; questo viene effettuato rimuovendo parti dell'input sempre diverse, e raffinando sempre di più le modifiche man mano che i risultati dei test evidenziano come la parte *Failure Inducing*sia ancora presente nell'input modificato (il test fallisce), o sia stata rimossa (il test ha successo). Il procedimento è computazionalmente meno efficiente dell'algoritmo di semplificazione ma restituisce un input 1-minimo più facilmente utilizzabile dal programmatore. Questa variante è stata sinora applicata solo al problema degli input di un programma.
2. *dd* l'algoritmo generale del DD, mostrato in Figura 3, partendo dall'input I_x e dall'input I_v procede aggiungendo delta ad I_v (che è in generale l'input nullo in partenza) e rimuovendone da I_x, ricercando la coppia di input che differiscano per un solo delta (un singolo carattere nell'esempio precedente). Questo risultato sarà in generale meno intuitivo per il programmatore ma costituisce la base ideale per applicare il DD allo stato del programma come verrà presentato nella Sezione 6.

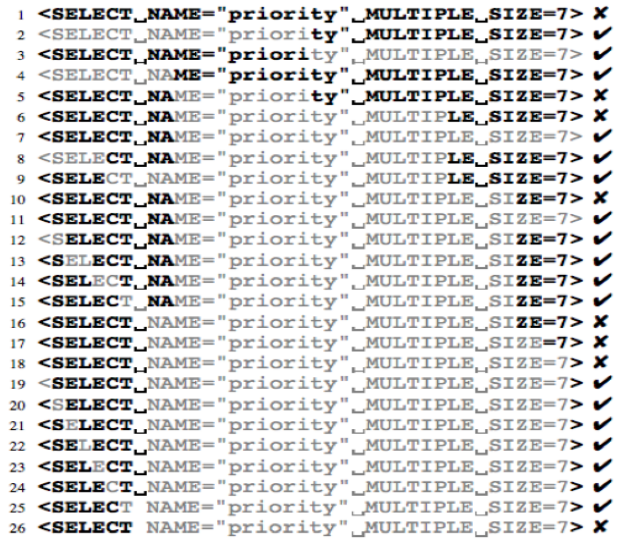

Figure 2: Esecuzione degli ultimi 27 passi dell'algoritmo *ddmin* applicato ad una pagina HTML che manda in crash Mozilla. L'algoritmo *ddmin* isola il tag $< SELECT >$ come responsabile dell'errore.

4. APPLICAZIONE AL CODICE DEL PROGRAMMA

In [5] l'autore mostra l'applicazione dell'algoritmo *dd* a diverse revisioni di uno stesso programma. Procederemo nella presentazione in modo più rapido poiché ci si può sostanzialmente rifare a quanto detto nella sezione relativa all'applicazione dell'algoritmo *dd* all'input di un programma.

I delta che andremo a considerare saranno qui le linee di codice che sono state cambiate tra due revisioni di un programma.

Il problema che consideriamo è quello in cui la versione più vecchia di un programma esegue correttamente un compito, mentre la versione successiva a parità di condizioni presenta un errore.

La supposizione che gli autori fanno è la seguente: "l'errore si anniderà nel set di modifiche apportate. Questo non è vero in senso letterale, ma le differenze tra le due revisioni sono sicuramente la causa del *manifestarsi* del problema. Tuttavia l'output dell'algoritmo *dd* è l'input del programmatore che deve risolvere il problema: quindi, anche ipotizzando che l'errore si annidi nella porzione di codice non modificata, l'algoritmo, che isola quali delta palesano il problema, offre al programmatore informazioni su quale porzione della parte comune possa contenere l'errore (in generale quella che più strettamente collabora con i delta responsabili del manifestarsi dell'errore). L'algoritmo procederà come per gli input di un programma, applicando solo parzialmente i delta aggiunti nella versione più recente, eseguendo test automatici e verificandone l'esito. Molti test saranno "*unresolved*", poiché, inserendo o togliendo delta in modo puramente sperimentale e "cieco, si otterranno spesso programmi che non compilano o che vanno in crash immediatamente dopo l'avvio. In ogni caso il processo è automatico e non richiede

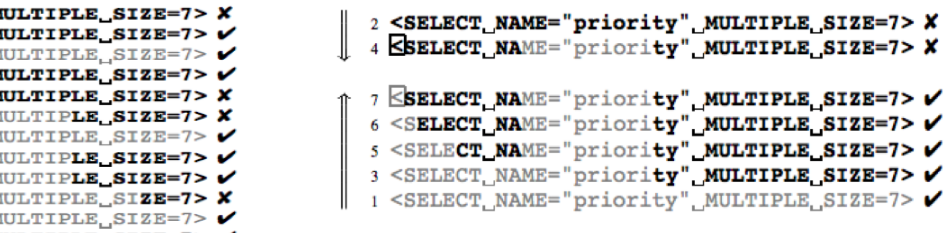

Figure 3: Esecuzione degli ultimi 7 passi dell'algoritmo *ddmin* applicato ad una pagina HTML che manda in crash Mozilla. L'algoritmo restituisce una coppia di input, uno processato correttamente e uno no, che differiscono di un solo delta, il carattere "<".

ore-uomo. L'esito è l'isolamento di un subset di modifiche che se applicate generano l'errore. Il programmatore ha anche in questo caso ottenuto un effetto di zoom sulla porzione di programma che è responsabile dell'errore.

5. APPLICAZIONE ALLO SCHEDULING DI PROGRAMMI MULTITHREADED

In programmi multithreaded, il debugging è un task ancora più complesso: la natura non deterministica degli errori, dovuta al non determinismo del thread scheduling rende il compito del programmatore davvero arduo. In generale è difficile identificare e ricreare le condizioni di errore; questo rende il debugging un compito a volte quasi impossibile, ed estremamente *time-consuming*.

Il DD si applica a questo problema [1] sfruttando *Dejavu*, un framework di esecuzione che elimina il problema del non-determinismo. Questo tool, sviluppato da IBM, consente l'esecuzione di un programma multithread utilizzando uno specifico scheduling, che può essere registrato da una run qualsiasi o realizzato artificialmente.

A questo punto possiamo considerare due diversi schedule, uno per il quale il programma esegue correttamente ed uno per cui si presenta un errore. I delta in questa applicazione sono gli spostamenti in avanti o indietro del momento in cui avviene il thread switch. Non si considera un asse temporale vero e proprio ma una serie di *yield point*, ovvero i punti in cui è sicuro fare un thread switch.

I delta sono quindi definiti come lo spostamento in avanti o indietro di un determinato thread switch nella misura di un singolo yield point. Si può quindi applicare il classico algoritmo che andrà ad evidenziare quali sono i delta che influiscono. In conclusione noteremo come l'ordine di esecuzione di alcune istruzioni sia rilevante per il mostrarsi dell'errore. Il programmatore potrà quindi concentrarsi solo su queste righe di codice per trovare l'errore.

E' importante sottolineare come per questa applicazione le considerazioni sull'efficienza dell'algoritmo siano particolarmente rilevanti. Infatti anche per un programma semplice i delta da considerare possono essere nell'ordine di miliardi di differenze[7]. Fortunatamente come già introdotto l'algorit-

[7]Il caso di studio riportato nell'articolo [1] mostrava quasi quattro miliardi di differenze tra i due schedule considerati.

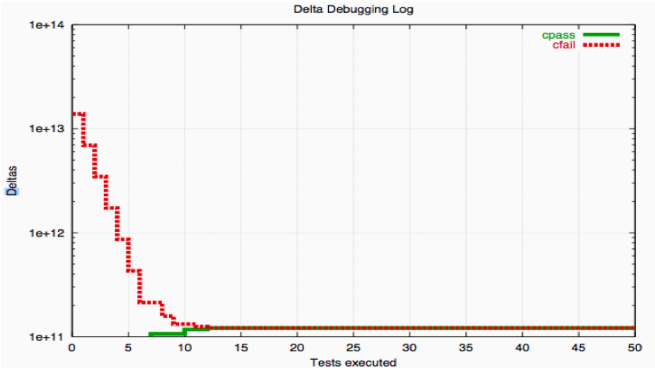

Figure 4: Algoritmo *dd* applicato al problema dello schedule di programmi multithread: il grafico mostra come il numero di test necessari per isolare il problema sia ridotto anche a fronte di un input enorme.

mo si assesta su una complessità prossima al logaritmo della cardinalità dell'input, come mostrato in Figura 4.

6. APPLICAZIONE ALLO STATO DEL PROGRAMMA

Finora sono state presentate tre applicazioni del DD, che agiscono su entità decisamente eterogenee (input, codice, scheduling), ma ottengono risultati degni di nota applicando sempre lo stesso schema di azioni: l'output dell'algoritmo permette di isolare la sorgente del problema all'interno dell'entità considerata. Questa quarta applicazione [6] del DD si rivolge, senza sostanziali novità nel procedimento, a qualcosa di più complesso ed articolato: ma, proseguendo sulla stessa strada, si otterranno risultati importanti, che superano l'isolamento della sorgente del problema, per arrivare ad affrontare una questione fondamentale per il debugging: la localizzazione dei difetti nel codice.

6.1 Lo stato di un programma

Un programma in esecuzione è caratterizzato da uno stato che evolve; l'esecuzione di istruzioni causa *transizioni* di stato; lo stato stesso determina (ad esempio tramite le istruzioni condizionali) l'evoluzione futura del programma.

6.1.1 Il problema della rappresentazione

In prima approssimazione si potrebbe immaginare lo stato come l'insieme delle variabili e dei loro valori; ci sono però una serie di particolarità da tenere in considerazione:

1. i tipi strutturati e gli array devono essere rappresentati nella loro interezza;
2. i puntatori (o le references di Java) devono mantenere il loro significato, e non ridursi ad un mero indirizzo di memoria;

Peraltro, avendo ormai una certa esperienza con il funzionamento dell'algoritmo *dd*, possiamo già intuire che sullo stato del programma andranno svolte diverse operazioni non solo di lettura ma anche di modifica: ci troveremo infatti a dover "mischiare" stati diversi tipicamente relativi ad un'esecuzione che ha successo ed una che va in failure.

A questo punto risulta ovvio come una rappresentazione dello stato lineare e semplicistica quale una lista di bindings variabile-valore non sia una strada percorribile. La soluzione a questo problema è, per fortuna, già nota in letteratura: i *Memory Graphs*.

6.1.2 Grafi di memoria

L'idea centrale è rappresentare lo stato tramite un grafo.

Al di là delle giustificazioni teoriche e pratiche, che presenteremo a breve, balzano subito all'occhio i vantaggi di una rappresentazione di questo tipo: la teoria dei grafi infatti fornisce solidi fondamenti teorici, oltre che una serie di algoritmi (e relative implementazioni) efficienti e già disponibili per risolvere i problemi che andremo a porci; appare ora decisamente più fattibile la prospettiva di cercare differenze tra stati, ed effettuare modifiche progressive in modo ben definito e ripetibile. D'altra parte questa soluzione è anche piuttosto naturale:

1. i nodi del grafo rappresentano
   (a) variabili in senso stretto
   (b) membri di strutture o array
2. gli archi rappresentano
   (a) operazioni di accesso a membri di strutture
   (b) operazioni di dereferenziazione di puntatori

6.1.3 Problemi aperti

Si potrebbe infine obiettare che in generale, dello stato di un programma, in senso lato, fanno parte anche entità esterne alla sua propria area di memoria, come ad esempio il contenuto delle memorie di massa, la situazione del sistema operativo o della rete... Tutti attori che non sono rappresentati in un grafo di memoria.

Tuttavia, nello stesso modo in cui il DD in generale dà per assunti l'assenza di interattività e assoluto determinismo del programma, e mostra come ciò non leda la generalità dei risultati, possiamo ora ignorare queste "terze parti", a patto di gestirle opportunamente. Esse, se necessario, saranno rappresentate esplicitamente all'interno del programma (e del suo stato).

6.2 Stati e Delta Debugging

Rimanendo nell'ambito delle ipotesi di determinismo e non interattività possiamo garantire che:

1. lasciando evolvere due istanze del programma a partire da stati identici, le esecuzioni saranno identiche;
2. di conseguenza, se partendo da uno stato specifico il programma continua la sua esecuzione fino a fallire (passare) il test, *tutte* le esecuzioni che partono da quello stato arriveranno a fallire (passare) il test.

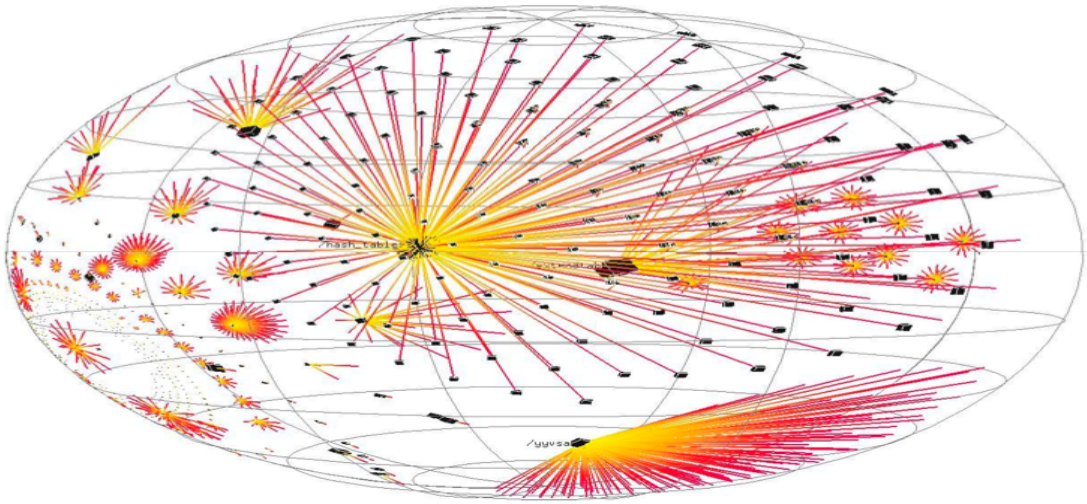

Figure 5: Un esempio di grafo di memoria.

Diventa ora naturale considerare, al pari di quanto avevamo fatto in precedenza con input, codice e scheduling, stati che portano a fallimento del test (*failing*, S_x) e stati che portano a passarlo (*passing*, S_v).

Dati uno stato *passing* e uno *failing*, S_v e S_x, applichiamo l'algoritmo *dd* alle loro differenze, isoleremo così quel sottoinsieme dello stato che è *failure inducing*: questa volta, sarà un insieme (più piccolo possibile) di variabili (o puntatori, references...) modificando il valore delle quali si determina o meno il fallimento del test.

Ad esempio, se le differenze iniziali tra S_v e S_x ammontano a qualche centinaio di variabili[8] (si noti che lo stato nel suo complesso può invece essere assai più grande), l'algoritmo proverà a creare un nuovo stato prendendo i valori di una parte di tali variabili (in questo esempio possono essere dell'ordine di 50 variabili) da S_v e il resto da S_x. Continuerà poi l'esecuzione del programma partendo dallo stato appena creato, per verificare se si ha un successo, un fallimento, o un risultato *unresolved*.

Seguendo i dettami dell'algoritmo generale *dd*, nelle successive iterazioni in modo analogo verranno creati e classificati come *passing*, *failing* o *unresolved* nuovi stati, fino a giungere all'isolamento del sottoinsieme delle differenze *Failure Inducing*; verranno ritornati due stati molto simili, uno *passing* e uno *failing*: le loro differenze sono *Failure Inducing*.

Chiaramente, nel tentativo di ridurre al minimo sin dall'inizio le differenze tra i due stati, considereremo come ingressi due input, uno *Failure Inducing* ed uno che esegue correttamente, il più vicini possibili tra loro, di modo che i due stati iniziali differiscano il meno possibile almeno all'inizio dell'esecuzione. Saranno perfetti a questo scopo i due input restituiti dall'applicazione dell'algoritmo generale *dd* agli ingressi di un programma (Vedi Sezione 3), dove come già introdotto l'output dell'algoritmo sarà una coppia di input del programma, uno *Failure Inducing* e uno no, tali che differiscano tra loro di un solo delta, ove possibile.

Sapere che la modifica di alcune (poche) variabili (o puntatori, etc...) nello stato di un programma destinato a passare il test porta al fallimento del test stesso può essere un'informazione molto preziosa per il debugging.

6.2.1 *I risultati Unresolved*

Nel DD applicato agli stati, non è difficile immaginare come si originino risultati "*unresolved*": quello che si fa è prendere parti di stato di un'esecuzione e "trapiantarli" senza alcun criterio semantico nello stato di una diversa esecuzione, effettuando operazioni verosimilmente inconsistenti e pericolose; possiamo aspettarci molti *core dump*, *segmentation fault*, e altre conseguenze fatali: tutti risultati "*unresolved*".

A tale proposito, le sperimentazioni effettuate hanno mostrato una proporzione di risultati "*unresolved*"comunque relativamente limitata rispetto alle aspettative: la spiegazione proposta è che l'esecuzione della parte rimanente di programma agisca da *filtro* sulle modifiche effettuate allo stato, ovvero, si possa verificare una sorta di *attenuazione* delle differenze, per cui le conseguenze sul programma vengono ridotte, "ammorbidendo" inconsistenze potenzialmente fatali. Naturalmente, è anche possibile una *amplificazione* della modifica applicata allo stato: si verifica allora che l'inconsistenza dilaga in breve tempo, portando subito a un crash.

6.3 Una novità fondamentale: una catena di cause-effetti

Il fatto di utilizzare un *memory graph* come rappresentazione dello stato non introduce novità sostanziali nel procedimento di DD: e nulla comunque che ci possa portare oltre i risultati ottenuti con gli approcci presentati finora.

La vera novità di questo quarto approccio sta nel fatto che l'operazione del DD viene svolta in *un preciso momento* del-

[8]Per semplicità di esposizione non prendiamo qui in considerazione puntatori, strutture, array ed altro, ai quali lo stesso procedimento si applica in modo naturale.

l'esecuzione del programma (o meglio, delle due esecuzioni parallele);

Come abbiamo sottolineato in precedenza, lo stato del programma durante l'esecuzione è in continua *evoluzione*: applicheremo quindi il procedimento di DD in un istante preciso di tale evoluzione.

Risulta evidente come sia possibile applicare l'algoritmo di DD in diversi momenti dell'esecuzione ottenendo informazioni utili: sarà sufficiente eseguire parallelamente le due istanze *failing* e *passing* e interromperne diverse volte l'esecuzione in istanti corrispondenti (tipicamente all'ingresso di funzioni) per evidenziare le differenze tra lo stato *failing* e quello *passing*, su cui applicheremo il DD.

Quello che si ottiene è una catena di cause-effetti; la conoscenza del sottoinsieme dello stato *Failure Inducing* viene estesa nella dimensione temporale: abbiamo quindi un quadro di come evolve il set di variabili *Failure Inducing* durante l'esecuzione del programma.

Non c'è dubbio che conoscere questa catena abbia un'importanza fondamentale per comprendere più facilmente e in modo più approfondito la dinamica che porta al fallimento del test.

Il DD applicato allo stato del programma, come è stato descritto finora, è implementato nel web service AskIgor [4]. L'utilizzo è semplice, basta inviare un eseguibile per Linux compilato con i simboli di debugging, e fornire due invocazioni (possibilmente simili), una sola delle quali manifesta il problema.

Sul server viene effettuato il procedimento di DD, e all'utente viene fornito un elegante *report* html con la catena di cause-effetti rilevata, e la possibilità di focalizzare la ricerca su una parte di essa.

6.4 La localizzazione dei difetti nel codice

I risultati ottenuti finora, benchè estremamente rilevanti, non esauriscono le potenzialità di quest'ultima applicazione del DD: la catena di cause-effetti trovata, infatti, può essere utilizzata anche per localizzare i difetti nel codice. Nell'ambito della rudimentale strutturazione a "fasi" del procedimento di debugging (Vedi Sezione 1), stiamo quindi lasciando il problema di isolamento, per passare al passo successivo.

Per capire come si possa fare questo salto utilizzando i risultati del DD applicato agli stati sono necessarie poche premesse.

6.4.1 Rompere la catena

Per evitare il verificarsi del fallimento, è necessario rompere la catena di cause-effetti. Il grosso problema da risolvere è determinare *dove* questa vada interrotta.

Agli occhi di un sistema automatico che non ha una comprensione ad alto livello dei meccanismi e della semantica del programma, tutti i punti si equivalgono. Il programmatore vuole però agire sulla catena dove l'esecuzione inizia a "naufragare"; cercheremo ora di definire meglio questo concetto.

6.4.2 Stati intended e stati faulty

E' evidente come subito dopo l'inizio del programma, gli stati dell'istanza "Passing" e dell'istanza "Failing" presentino delle differenze – altrimenti, sotto le ipotesi definite in precedenza, non potrebbero avere evoluzioni differenti; E' anche evidente come alcune di queste differenze siano *Failure Inducing*. Bisogna però ammettere che queste differenze *Failure Inducing*, non sono di per sè problematiche. Banalmente se un programma di sorting fallisce ogni volta che è presente uno 0 nel suo input, la presenza di uno 0 nella memoria del programma che contiene i valori da ordinare sarà senza dubbio una differenza *Failure Inducing*. Questo non implica però che lo stato in questione abbia delle inconsistenze interne, o che possieda delle caratteristiche di cui il programmatore si possa sorprendere.

Al contrario, subito prima del fallimento del test, il programma si sta comportando in modo "sbagliato": di riflesso il suo stato presenterà delle caratteristiche inconsistenti o comunque scorrette.

Definiamo quindi i primi come stati "*intended*", i secondi, minati da inconsistenze interne, stati "*faulty*".

6.4.3 I difetti nel codice e le infezioni

Il cuore della questione: la catena di cause-effetti che porta al fallimento del test va rotta nel momento in cui uno stato *intended* evolve in *faulty*. L'istruzione che causa la transizione dello stato da *intended* a *faulty* sarà in qualche modo strettamente correlata con il difetto nel codice.

Si utilizza anche una terminologia molto suggestiva: il difetto nel codice origina *un'infezione* nello stato; l'infezione di solito si propaga durante l'esecuzione del programma, fino a portare, alla fine, al fallimento del test; il compito del programmatore è proprio quello di individuare l'origine dell'infezione.

6.4.4 Ricadute pratiche

La catena di cause-effetti messa in evidenza dal procedimento di DD applicato allo stato offre un valido aiuto nell'identificare l'origine dell'infezione: il programmatore, sfruttando la sua conoscenza dello stato a livello semantico, individua facilmente quali differenze *Failure Inducing* sono innocue, e quali sono indice di uno stato infetto; aumenta poi la granularità dell'analisi (con un procedimento che, ancora una volta, può ricondursi a una ricerca binaria) fino ad identificare con sufficiente precisione il codice origine del problema. L'identificazione del codice difettato è a questo punto un'operazione piuttosto facile.

Tuttavia, la capacità di distinguere stati intended da stati faulty è propria dell'operatore umano, poiché, come abbiamo già avuto modo di sottolineare, richiede conoscenze ad alto livello sul significato delle variabili, dei puntatori... Esistono approcci al debugging basati su specifiche esplicite di consistenza dello stato, che quindi sono in grado di identificare in modo automatico gli stati infetti. In generale però, i risultati presentati a ICSE05 [2] hanno mostrato come si possa in

molti casi limitare la ricerca del codice difettato a poche aree sospette, senza bisogno di alcuna informazione aggiuntiva, e in modo completamente automatico.

6.4.5 Cause transitions

Questo interessante risultato è reso possibile dalle cosiddette "Cause Transitions". Si definisce Cause Transition una transizione di stato che modifica il sottoinsieme *Failure Inducing* dello stato. In termini pratici, ciò si traduce nell'avere alcune variabili (o puntatori) che sono *Failure Inducing* in un determinato stato, mentre, dopo l'esecuzione di un'istruzione, le variabili (o puntatori) *Failure Inducing* sono cambiate.

Identificare le Cause Transitions è estremamente facile e viene fatto in modo efficiente, sistematico, e soprattutto automatico: visto che il sottoinsieme *Failure Inducing* dello stato si ottiene dall'algoritmo *dd*, è sufficiente applicare una ricerca binaria che opera sui successivi stati nei diversi momenti dell'evoluzione del programma.

E' comunque importante sottolineare che una Cause Transition non ha nulla di intrinsecamente "sbagliato": nella maggior parte dei programmi, sia di piccole che di grandi dimensioni, si hanno un certo numero di Cause Transition fisiologiche.

Tuttavia, ed è abbastanza intuitivo, un difetto nel codice generalmente è strettamente correlato all'istruzione che origina una cause transition: è infatti molto probabile che a seguito dell'esecuzione di un'istruzione scorretta (il bug) una parte dello stato inizi ad essere *Failure Inducing* (dando origine, appunto, a una Cause Transition).

7. CONCLUSIONI

Il Delta Debugging è un procedimento formale e automatico che assiste il programmatore nella fase di *isolamento* della sorgente del problema. Gli ambiti a cui si può applicare sono diversi: isolamento e minimizzazione dell'input *Failure Inducing*, isolamento dei cambiamenti *Failure Inducing* nel codice di un programma, identificazione di soluzioni di scheduling *Failure Inducing* per programmi multithreaded. Inoltre applicazioni molto promettenti e di più ampia portata sono rese possibili dall'applicazione dell'algoritmo *dd* allo stato dei programmi.

I risultati ottenuti finora, non limitati a situazioni "di laboratorio" ma applicati a programmi reali e complessi, hanno mostrato che questo algoritmo ha un grande potenziale in ciascuna delle applicazioni presentate.

Essendo un approccio black-box puramente sperimentale, l'efficienza e la velocità della diagnosi dipendono fortemente dai requisiti temporali delle esecuzioni del programma sotto esame, mentre sono indipendenti dalla *complessità del codice*, e scalano di solito molto bene con la dimensione dell'entità da minimizzare (input, codice, soluzione di scheduling, stato). Dal punto di vista pratico, questo significa che può essere molto più facile e rapido isolare l'input *Failure Inducing* in un file di diversi MByte elaborato da un programma assai complesso che conta di decine migliaia di righe di codice, ma che passa o fallisce il test in pochi secondi, piuttosto che operare su un banale programma di poche linee di codice, con un input breve, per il quale però si richiedano diverse ore perché si determini il risultato del test. La complessità del Delta Debugging non è stimabile a priori con precisione, perché dipende fortemente dai molti fattori che determinano l'incidenza di risultati "*unresolved*".

Di conseguenza, non ci si può aspettare una "silver bullet" per l'isolamento delle sorgenti dei problemi: in alcuni casi un soluzione di gran lunga più rapida è procedere manualmente e in modo più mirato, sfruttando la conoscenza della struttura logica del programma come, ad oggi, solo un operatore umano può fare, tuttavia risulta evidente che per alcune applicazioni i risultati ottenuti siano estremamente promettenti.

Allo stato attuale le implementazioni del DD hanno un livello qualitativo notevole (anche perché sono incarnazioni di un algoritmo ben specificato e quindi hanno basi solide), pur non mancando ampi margini di miglioramento. Tuttavia, ciò che è carente è l'*utilizzabilità pratica* di queste soluzioni: nella maggior parte dei casi, infatti, l'utilizzo del DD, pur vantaggioso e utile, non porta vantaggi tali da giustificare un elaborato procedimento per applicarlo. E' quindi auspicabile che le procedure vengano strettamente integrate negli ambienti di sviluppo, analogamente a quanto già avviene, ad esempio, per Eclipse e JUnit; peraltro, i prototipi di *plugin* per l'isolamento e la minimizzazione dell'input *Failure Inducing* (gli unici al momento disponibili), evidenziano chiaramente tutti i vantaggi di tale integrazione: le release di nuovi *plugin* previste per l'estate fanno ben sperare per il prossimo futuro.

8. REFERENCES

[1] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220, New York, NY, USA, 2002. ACM Press.

[2] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, NY, USA, 2005. ACM Press.

[3] Mozilla development team; http://www.mozilla.org/. Mozilla project.

[4] Zeller. Askigor debugging web service; http://www.askigor.com/.

[5] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[6] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press.

[7] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. volume 28, pages 183–200, Piscataway, NJ, USA, 2002. IEEE Press.