ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di laurea in Informatica

Debugging Reversibile

"Debugging involves backward reasoning"

Relatore:
Prof. Ivan Lanese

Presentata da:
Daniel Scerpa

Sessione
Marzo

Anno accademico
2017-2018

Indice

Introduzione

Il debug rappresenta un costo importante nel processo di sviluppo del software. Uno studio, del National Institute of Standards and Technology degli Stati Uniti, ha stabilito che gli errori software hanno un costo enorme e che gli sviluppatori spendono circa l'80% dei costi di sviluppo per identificare e correggere i bug.[1]

Un bug, noto anche come baco, non è altro che un errore presente all'interno di un programma, e può causare differenti problematiche che possono essere più o meno gravi a seconda delle situazioni.
Nei casi peggiori, i bug possono rendere i computer vulnerabili ed esporli a possibili attacchi informatici[2].

Più nello specifico, un bug in un programma può essere visto come un guasto, che ha come conseguenza il malfunzionamento del software.
Per capire e individuare le origini dei bug, è così necessario, analizzare il codice sorgente che viene scritto dai programmatori.

In inglese si utilizza l'espressione "buggy", che utilizzerò nei prossimi capitoli, per segnalare tutti quei programmi che sono caratterizzati da un numero elevato di bug e compromettono le funzionalità del programma[3].

Gli effetti dei bug all'interno dei programmi possono essere tanti: a volte sono quasi inesistenti o comunque non sono ben visibili perché incidono in maniera poco significativa sulle funzionalità del software, così da restare nascosti per molto tempo, destando pochi sospetti al programmatore; Altre volte invece possono essere molto pericolosi e ben visibili, soprattutto se determinano un blocco del software o, in casi critici, un crash, da cui deriva una mancata disponibilità del servizio che si desidera.

Vi sono quindi vari tipi di bug, alcuni di questi possono potenzialmente mettere a repentaglio la sicurezza dei dispositivi poiché concedono, per esempio, privilegi e accessi a utenti che invece non ne dovrebbero usufruire.
A prescindere dalla loro gravità, complessità e natura, è comunque necessario per lo sviluppatore, trovare e correggere i bug per disporre di un software il più possibile sicuro.

Come detto, alcuni bug possono essere ben visibili, ma non sempre è facile identificare la causa che li ha generati così come la posizione in cui sono sorti.

Il motivo principale per cui i bug sono difficili da rintracciare, da "Empirical study of debugging stories", è che risulta difficile stabilire l'arco temporale e lo spazio in cui si verificano.

Per favorire i programmatori nella ricerca dei bug, vengono utilizzati degli strumenti software appositi: i debugger.

I Debugger non sono altro che programmi, che consento di analizzare se un programma è sintatticamente corretto, in modo tale da effettuare una ricerca del bug in modo più veloce e accurato possibile. Grazie ai Debugger si ha la possibilità di scovare errori o malfunzionamenti all'interno del programma sfruttando funzioni specifiche per il debugging: l'attività che consiste proprio nell'individuazione della porzione di software affetta da bug.

Generalmente il tipo di debugger più utilizzato è il debugger a runtime che consente la ricerca dei bug tramite funzionalità standard come Breackpoint, ControlPoint e viste da Watch, che consentono al programmatore di analizzare più accuratamente determinate parti di codice per identificare gli errori con una scansione in avanti del codice.

La maggior parte dei debugger fornisce infatti, un‘assistenza limitata all’esecuzione in avanti del codice per la navigazione temporale, per cui i programmatori devono spesso ricorrere alla simulazione dell'esecuzione del programma mentalmente per cercare di immaginare i flussi di istruzioni che vengono eseguiti.

Per capire al meglio i limiti che un Debugger classico può avere, vediamo come viene gestito il bug illustrato nel seguente esempio:

```
let one = 1 in
let two = 2 in
let k = port in
thread {send k one} end; // T1
thread {send k two} end; // T2
thread let x = {receive k} in skip end end //t 3
end end end
```

Utilizzando il linguaggio μOz, che vedremo melgio nei prossimi capitoli, è possibile inviare tramite il comando “send” dei valori ad una rispettiva porta e, tramite un thread in stato di “receive”, ricevere il valore mandato.

In questo breve esempio di codice, ci sono due thread (T1 e T2) che effettuano una “send” su una porta e il thread(T3) è in attesa di ricevere il valore k dalla porta. Il programmatore si aspetterebbe di ricevere il valore 1 in quanto viene eseguita prima l’istruzione di send del thread T1 rispetto a quella T2. Questo però non è corretto, in quanto il thread T2 potrebbe venire eseguito più velocemente e, di conseguienza, il valore ricevuto potrebbe essere 2.
Questo potrebbe portare ad un errore software molto difficile da scovare oltre che pericoloso per il sistema se non gestito correttamente.

Con un debugger a Runtime, il programmatore può cercare il bug inserendo dei breackpoint all’interno dei thread e sperare di scovare il bug facendo dei tentativi, magari entrando in Watch per osservare come vengono modificate le variabili durante l’esecuzione del programma.
Assumendo che questi thread siano collegati a loro volta ad altri thread, la ricerca del bug richiederebbe molto tempo.

Nei prossimi capitoli, dopo aver introdotto i debugger reversibili, analizzeremo lo stesso esempio affrontandolo con la “mentalità” di un debugger reversibile per mostrare le potenzialità della reversibilità.

1. I Debugger Reversibili

1.1. Cosa sono i Debugger Reversibili?

Offrendo una navigazione temporale limitata all'esecuzione in avanti del codice, come descritto nella sezione precedente, per aiutare i programmatori a risparmiare risorse e tempo, sono stati introdotti dei nuovi programmi che consentono di navigare all'interno del codice: entrano cosi in gioco i debugger reversibili.

Essi consentono agli sviluppatori di registrare le attività del programma in esecuzione, per poi riavvolgere e riprodurre tali istruzioni, compresi eventuali errori, per ispezionare lo stato del programma.
È una sorta di viaggio nel tempo all'interno del codice in cui vengono eseguite tutte le istruzioni (come gli accessi in memoria, le chiamate al sistema operativo, le dichiarazioni di variabili) per poi, tramite opportuni comandi, replicare al contrario il programma, consentendo allo sviluppatore, di migliorare drasticamente la propria integrazione nell'esecuzione del programma, agevolandolo a navigare avanti e indietro (forward and backward) fra la cronologia delle istruzioni per trovare più facilmente l'origine dell'errore.

Ripercorrere questo flusso di informazioni in programmi concorrenti, ove il debugger lo consentisse, o in programmi molto strutturati può richiedere un consumo non indifferente di memoria.

Per migliorare le prestazioni, sta quindi allo sviluppatore, cercare di ridurre ad un sottoinsieme il più piccolo possibile il quantitativo di informazioni necessarie per identificare ed eventualmente correggere il bug del flusso esaminato, sfruttando le funzionalità del debugger stesso.

1.2. La nascita dei Debugger Reversibili

Per migliorare l'efficienza del processo di debug, nel corso degli anni, è cresciuta sempre di più l'idea di mettere, a disposizione del programmatore, un debugger che consentisse non solo, di fare un passo avanti nel codice, ma anche la possibilità di ripercorrerre all'indietro l'esecuzione del programma per ridurre i costi e i tempi nel sviluppare software.

Questi debugger reversibili, conosciuti anche con il nome di debugger temporali, o TTD (che utilizzerò come sinonimo di debugger reversibile), non sono in realtà una nuova invenzione. Questi debugger infatti sono nati nel mondo accademico verso la fine degli anni '60, ma sono stati sviluppati solo successivamente a partire dalla fine degli anni 2000.

I debugger reversibili sono stati infatti conosciuti soprattutto negli ultimi 40 anni grazie alle funzionalità aggiuntive di cui dispongono, come la possibilità di ripercorrere il codice tracciando l'esecuzione di un programma, hanno ottenuto solo recentemente una più ampia considerazione[4,5]. Negli ultimi anni infatti, l'interesse per questi debugger è tornato particolarmente a crescere, stimolato anche da un discorso di forte ispirazione tenuto da Bret Victor [24] sul miglioramento dello stato della programmazione e sulla riduzione delle risorse che comportano tali debugger.

Queste funzionalità infatti consentono di risalire e rintracciare più velocemente gli errori all'interno di un programma, e sono così diventate una caratteristica desiderata all'interno dei debugger dei programmatori.

Risalire le istruzioni è molto utile a tutti i programmatori, in particolare queste funzionalità trovano particolare apprezzamento per:

1) Sistemi di debugging in cui l'output della traccia viene salvata e può essere interrogata sotto il controllo del programmatore;
2) applicazioni di intelligenza artificiale in cui si cerca di dimostrare un determinato risultato.

1.3. Debugger Reversibili: perché usarli?

Il vantaggio principale di questi debugger rispetto a quelli classici è che, dato un bug, è possibile ripercorrere il flusso del codice per identificare con uno sforzo minore il bug in questione.
Per errori semplici, grazie ai compilatori classici, il bug può essere catturato semplicemente scorrendo ed esaminando il codice riga per riga sfruttando le funzionalità base dei debugger a runtime.

Tuttavia, questa strategia è molto più difficile da applicare nel caso di bug, la cui riproduzione risulti essere più complessa, in quanto uno sviluppatore può avere poche o nessuna informazione sul perché il bug sia stato generato.

I debugger reversibili sono la soluzione più utile per questo tipo di errori in quanto un programmatore può effettuare un'esecuzione del programma all'indietro, così come in avanti, al fine di raggiungere un punto di interesse, e trovare la causa principale del problema da due estremità del programma invece di una sola.

Il debugging a runtime è la pratica comune del debug in avanti. È la combinazione di punti di interruzione(BreackPoint), punti di controllo(ControlPoint) e altri elementi generici di debugger di runtime che fungono da indicatori per aiutare a trovare il bug nel software come per esempio le viste tramite Watch che consentono di visualizzare i valori che assumono le variabili a runtime.

Con questi debugger risulta così possibile eseguire il programma finché non raggiunge un punto di interruzione e analizzare eventuali problemi che si verificano in quella particolare sezione di codice[6].

In alcuni casi, questo approccio funziona perfettamente, ma il più delle volte, non si sa in quale sezione del codice è avvenuto l'errore: un grosso problema quando i programmi sono molto strutturati.

Utilizzando invece un debugger reversibile, per lo sviluppatore è sufficiente eseguire il proprio programma una sola volta per acquisire una registrazione completa dell'esecuzione del programma, che include qualsiasi problema software apparso durante il runtime nel corso della registrazione. Questa registrazione non solo contiene il bug, ma anche la sequenza degli eventi che hanno portato ad esso.

In un ambiente sequenziale, l'utilizzo di un debugger reversibile, è anche molto naturale in quanto i passaggi vengono semplicemente annullati nell'ordine inverso in cui sono stati eseguiti.

Il concetto di reversibilità è stato così interpretato e implementato in modi differenti, a seconda dell'approccio che si decide di utilizzare per rintracciare i passi che portano al bug.[7]

Riprendendo l'esempio visto in precedenza nel capitolo introduttivo, è facile notare il vantaggio di questi debugger:

```
let one = 1 in
let two = 2 in
let k = port in
thread {send k one} end; // T1
thread {send k two} end; // T2
thread let x = {receive k} in skip end end //t 3
end end end
```

Con un Debugger Reversibile, è possibile accedere direttamente al thread che ha inviato la varibaile tramite l'esecuzione all'indietro del codice, raggiungendo immediatamente il bug riscontrato. Come vedremo nei prossimi capitoli infatti, oltre a particolari comandi presenti in alcuni Debugger, sarà possibile visualizzare tutte le eccezioni che vengono catturate e avere l'elecondo di esecuzione dei thread aggiornati.

1.4. Determinstc Non-Deterministic e Casual-Consistent Reversibility

Per introdurre e analizzare in modo corretto i debugger che ho analizzato nei prossimi capitoli, vediamo i possibili approcci che possono essere utilizzati da questi debugger per gestire la concorrenza. Per farlo vediamo un semplice esempio presente nell'articolo "Casual-consistent reversible debugging" [7]:

"Supponiamo che ci siano dei passeggeri che devono prendere posto su un treno. Alcuni di loro hanno prenotato un posto, altri invece no. Quelli senza prenotazione scelgono casualmente un posto libero. Quelli con prenotazione prendono il posto assegnato a meno che non scoprano che sia stato occupato da qualcun altro, in tal caso scelgono un posto libero".

Quello che non dovrebbe mai accadere è che un passeggero X con una prenotazione si trovi in piedi senza posto a sedere. Se ciò accadesse, seguendo gli approcci in letteratura, si può risolvere il problema in tre diversi approcci, ovvero:

- **Non-deterministic replay debugging** [8]: mandare tutti fuori dal treno e avviare nuovamente l'algoritmo di seduta. Ma questa volta i passeggeri possono scegliere i posti in diversi ordini, quindi il problema potrebbe non verificarsi, oppure un altro passeggero potrebbe rimanere senza il posto di cui ha diritto.

- **Deterministic replay/reverse-execute debugging** [9]: iniziare a chiedere alle persone di alzarsi esattamente in ordine inverso rispetto a come hanno occupato i posti a sedere. Rischiando di far alzare molte persone innocenti dai loro posti prima di trovare colui che occupa davvero il posto del passeggero X.

- **Casual-Consistent Reversibility**[7,10]: Annullare la seduta del "predecessore causale" del passeggero X: colui che si è seduto. Se, a sua volta, avesse un'altra riserva, allora si annullerebbe la seduta del suo predecessore causale e così via.
  In questo modo, è possibile trovare un posto per il passeggero con una prenotazione annullando un numero limitato di azioni di seduta.

2. Principali Debugger Reversibili

2.1. Introduzione ai Debugger: Funzionalità e Caratteristiche

Dopo aver analizzato le principali caratteristiche e le diverse implementazioni che contraddistinguono un debugger reversibile da un debugger classico, in questo capitolo vedremo alcuni debugger reversibili che sono stati introdotti a partire dagli anni 2000 in avanti.

Oltre ad offrire le funzioni dei debugger classici come BreackPoint, Control Point e l'esecuzione limitata in avanti del codice, tutti i debugger avranno, alla base, delle funzionalità aggiuntive per rendere possibile l'esecuzione all'indietro del codice e poter parlare così, di reversibilità.
Alcuni di questi debugger vedremo che offriranno una GUI (Graphical User Interface) con cui lo sviluppatore potrà, tramite le viste dedicate, visualizzare lo stato del programma con informazioni aggiornate in tempo reale sulle variabili, su eventuali thread e sulle eccezioni del programma.

Per ciascuno di questi debugger, verranno mostrate le funzionalità disponibili, i differenti metodi di implementazione che adottano e informazioni più specifiche come il linguaggio supportato, il tipo di licenza e l'eventuale presenza di supporto.

Vedremo vari tipi di Debugger, dai tool accademici, passando dai prototipi, per poi arrivare ai prodotti commerciali che hanno reso disponibile il reverse debugging ai programmatori, cambiando notevolmente l'approccio al debugging.

I debugger sono elencati in ordine cronologico per data di rilascio e sono i seguenti:

- UndoDB
- GDB
- ODB
- TOD
- Chronon
- CaReDeb
- Expositor
- Elm TTD
- RevDeBug
- Time Travel Debugger for Windows
- CauDEr

2.1.1. UndoDB

UndoDB è un Debugger Reversibile per C/C++ disponibile per Linux e Android che funziona su qualsiasi codice compilato in modalità utente, su x86 e ARM [8].

Dispone di tutte le funzionalità dei debugger moderni (come lo scripting, conditional breakpoints e watchpoints) ma è inoltre integrato con il significato di Debugger Reversibile così da poter godere di tutti i vantaggi di un Time Travel Debug.

L'esecuzione inversa avviene tramite punti di controllo(checkpoint/replay): ottiene periodicamente un checkpoint della memoria del processo, e utilizza una tecnica di riproduzione per ricostruire lo stato del programma tra i checkpoint.

Questo meccanismo è particolarmente conveniente in quanto produce un sovraccarico a runtime relativamente basso, ma non consente la navigazione causale.

Il motore di UndoDB registra solo i dati non-deterministici, che sono sufficienti per ricreare l'intera memoria del debugger. I risultati delle operazioni non deterministiche vengono registrati in un registro eventi (event log), che viene archiviato nella memoria del processo dell'applicazione.

Per fare questo, esegue una traduzione binaria JIT (Just-In-Time) del codice macchina mentre viene eseguito, in modo che tutte le fonti di non-deterministiche possano essere catturate e registrate correttamente.

UndoDB può essere utilizzato come debugger indipendente o può essere integrato con uno specifico debugger, come per esempio GDB che analizzerò nel capitolo successivo.

Dispone di una libreria che può essere integrata nei programmi che consente di registrare in produzione, o in suite di test, il programma per successive analisi offline.

UndoDB funziona in modo univoco con quasi tutti i file eseguibili (compresi quelli che utilizzano thread multipli, IO asincroni, memoria condivisa, istruzioni non deterministiche), su quasi tutti i kernel Linux.
Le sue prestazioni lo rendono pratico per carichi di lavoro molto impegnativi: per utilizzi con carichi di lavoro fino a 100 GB, raggiunge tempi di esecuzione di giorni/settimane.

UndoDB è un tool commerciale attualmente supportato e leader tra i Debugger Reversibili. È possibile ottenere una licenza gratuita di prova per 30 giorni.
Per uso professionale è necessario acquistare invece una licenza, mentre è possibile scaricare gratuitamente UndoDB per uso non professionale.
Dispone di una vasta documentazione direttamente online sul sito principale (https://undo.io/docs/) [8].

2.1.2 GDB

A partire dalla versione 7.0 di settembre 2009, GDB include il supporto per implementare le funzionalità dei debugger reversibili, consentendo di eseguire il programma indietro nel tempo, ripristinando così, uno stato dell'esecuzione precedente.

Il motore di esecuzione di UndoDB appena analizzato nel capitolo precedente, si integra perfettamente ed espande le funzionalità di GDB.

Cronologicamente, essendo GDB reversibile solo dalla versione 7.0, ho inserito prima UndoDB anche se, in realtà, quest'ultimo altro non è che un'estensione delle funzionalità base di GDB che, senza considerare la reversibilità, sarebbe cronologicamente nato prima di UndoDB.

Gli sviluppatori possono infatti utilizzare UndoDB ovunque utilizzino GDB, direttamente da riga di comando o tramite IDE come, per esempio, Eclipse ottenendo un rallentamento inferiore rispetto alla registrazione di GDB che è di oltre 50.000x rispetto alla 2x di Undo.

A differenza di UndoDB, GDB gestisce la richiesta per le System Call (chiamate al sistema operativo) tramite l'arresto della destinazione, per consentire l'accesso deterministico alla memoria del target.

Il debug inverso è supportato solo per alcuni tipi di target GDB, tra cui, alcuni target remoti, inclusi i simulatori Simics, SID e UndoDB.

Con il debugger GDB, invece di avviare il programma dall'inizio e ripetere l'intera sessione di debug, è possibile impostare un breackpoint in un punto precedente del programma e tramite il comando"reverse-continue", fare in modo che il programma torni indietro fino a raggiungere lo stato precedente, da cui si potrà procedere, se necessario, di nuovo in avanti.

In alternativa, è possibile eseguire "reverse-step" e "reverse-next" per eseguire un'istruzione di programma alla volta (come il normale "step" e "next" dei debugger a runtime).

Utilizzando uno dei target GDB che supporta il debugging reversibile, saranno disponibili alcuni comandi principali che consentiranno di sfruttare varie funzionalità, ovvero[11]:

- **Reverse-continue**: Esegue il programma all'indietro finché non raggiunge un evento di arresto (ad esempio un punto di interruzione, un punto di controllo o un'eccezione).

- **Reverse-step**: Esegue il programma all'indietro fino all'inizio della linea precedentemente eseguita.

- **Reverse-stepi**: esegue il programma all'indietro esattamente di un'istruzione.

- **Reverse-next**: consente di processare le chiamate di funzione al contrario.

- **Reverse-nexti**: esegue all'indietro un'istruzione, non accetta che l'istruzione sia un return da una chiamata di funzione, nel qual caso l'intera funzione verrà eseguita al contrario.

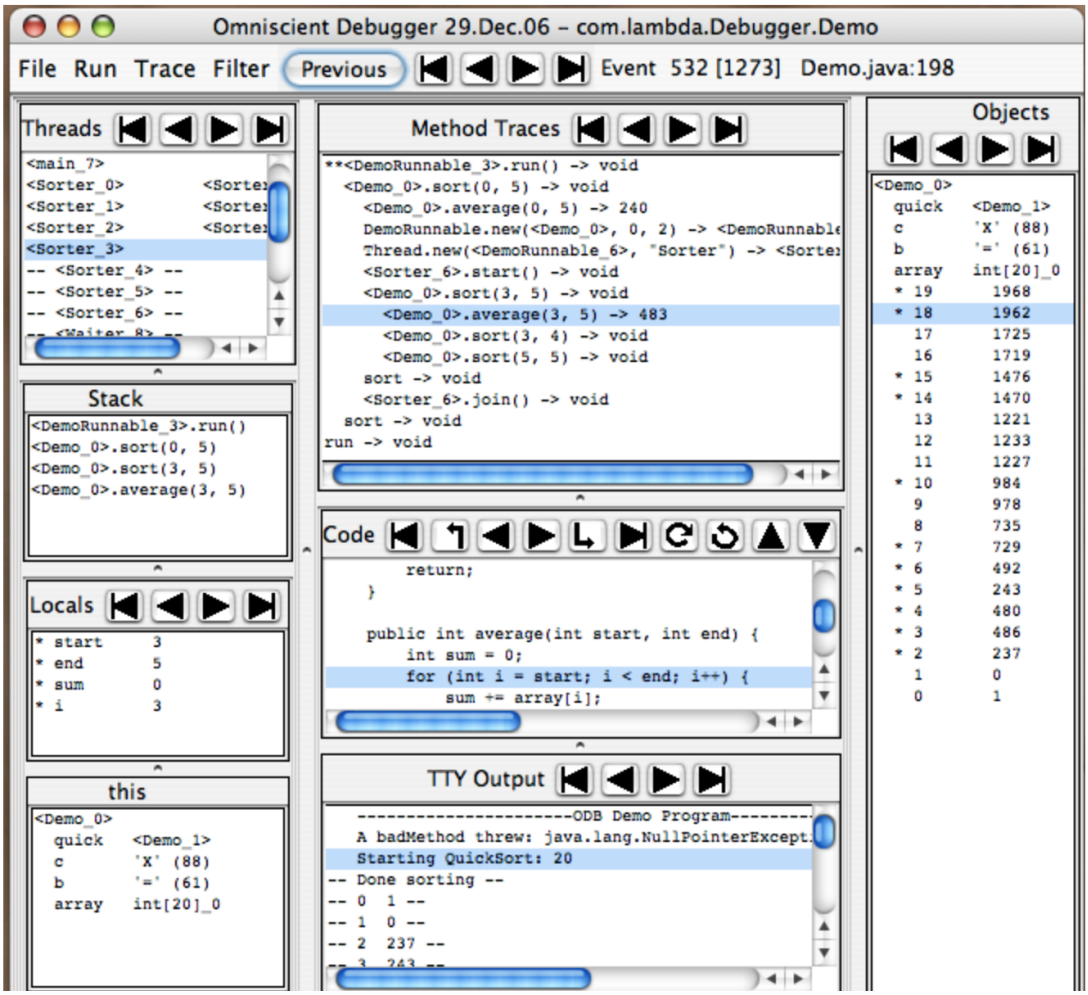
- **Reverse-finish**: come il comando "finish", "reverse-finish" esegue la funzione corrente al contrario, finché l'esecuzione non raggiunge la funzione di chiamata per poi arrestarsi.

GDB è un tool free software con licenza GPL (GNU General Public License). È attualmente supportato e offre una vasta documentazione online (https://sourceware.org/gdb/documentation)[12].

2.1.3 ODB

ODB è invece uno dei primi Debugger Reversibili per Java.
Ottiene le tracce di esecuzione organizzando le classi mentre vengono caricate dalla JVM (Java Virtual Machine) memorizzando i dati acquisiti della traccia all'interno della JVM di destinazione.

ODB registra infatti tutti gli eventi del programma in ordine preciso e li assegna tramite Timestamp (una sequenza di caratteri che rappresetanno una data e/o un orario). Ogni variabile ha un valore noto ad ogni Timestamp, in modo da avere nella finestra del debugger, i valori sempre aggiornati per mostrare i dati corretti ogni volta che si ripristina il debugger in uno stato precedente.



(ODB debugger window [13])

La finestra principale del debugger (riportata nell'immagine sopra) appare quando il programma chiama la exit () o quando viene premuto il pulsante "Stop Recording" nella finestra di controllo. Il programmatore potrà così navigare indietro nel tempo per esaminare oggetti, variabili e chiamate ai metodi[13].

Ciò significa che è possibile vedere quali valori hanno causato eccezioni inattese consentendo di capire da dove provengono questi valori, chi li ha impostati e perché.
A differenza dei due debugger precedentemente analizzati, offre una navigazione Casual Consistent: senza la necessità di punti di interruzione è infatti possibile saltare direttamente nei thread coinvolti e nelle variabili che hanno portato alla generazione del bug.

Una caratteristica notevole di ODB, essendo uno dei primi Debugger Reversibili rilasciati per Java, è la possibilità di riprendere l'esecuzione in qualsiasi momento con uno stato modificato e proseguire con le istruzioni che seguono senza la necessità di dover ricompilare il programma.

Altra funzionalità interessante che offre ODB è la possibilità di filtrare le informazioni che si vuole memorizzare durante la registrazione. È infatti possibile escludere le classi che sono attendibili come per esempio le classi JCF(Java Collection Framework), le classi di libreria e, forse tra le opzioni più importanti, le proprie classi ritenute ben testate.

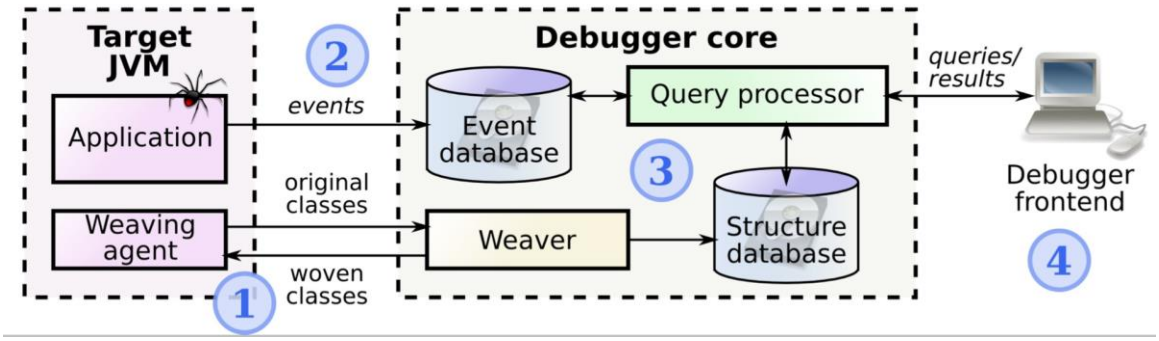
Il supporto a questo debugger non risulta attualmente attivo se non tramite un contatto diretto tramite la mail dello sviluppatore.
Dispone di una pagina web (http://omniscientdebugger.github.io) dove sono presenti gli ultimi aggiornamenti e il link alla pagina Github del software (https://github.com/OmniscientDebugger/LewisOmniscientDebugger/releases/tag/0.1.5).

2.1.4 TOD

TOD (Debugger Trace-Oriented) è un prototipo di Debugger Reversibile che si basa sulle logiche di ODB. Come quest'ultimo è infatti un Debugger Reversibile per Java integrato nell'ambiente di sviluppo Eclipse ed estende ODB tramite funzionalità disponibili in interfaccia molto potenti.

Supporta come ODB la navigazione temporale tramite la scansione, in avanti e indietro, del codice e supporta la navigazione Casual-Consistent tramite un collegamento diretto con le variabili consentendo di visualizzare accanto ad esse il valore che assumono.

Questo fa si che l'utente possa saltare direttamente all'evento che ha assegnato alla variabile il suo valore corrente.



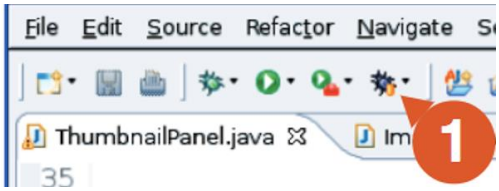
Per comprendere l'implementazione e le funzionalità di TOD, vediamo ora quattro fasi principali che svolge[14], raffigurate nell'immagine sopra elencata (Le immagini utilizzate in questo capitolo sono prese dal sito ufficiale di TOD [16] e dal paper [14]):

1) Una prima fase di "Strumentazione" in cui, quando la Macchina virtuale Java (JVM) sta per caricare una classe, l'agent invia al Weaver il suo bytecode che inserisce il codice di generazione degli eventi nella classe per poi rinviare l'informazione alla JVM.

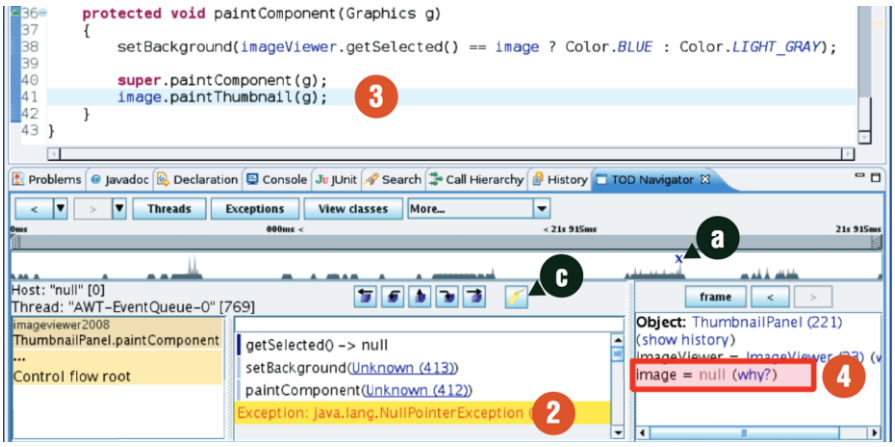
2) Una seconda fase di “emissione degli eventi” in cui mentre il programma viene eseguito, tutti gli eventi che vengono generati vengono inviati al database degli eventi e viene così costruito un trace di esecuzione.

3) Nella terza fase il Database memorizza gli eventi inserendo “Rate” e indici molto alti per consentire alle query un elaborazione dei dati meno pesante. Mentre il Database degli eventi indicizza e archivia queste tracce, il database di struttura memorizza le informazioni statiche sul programma di debug come classi e metodi.

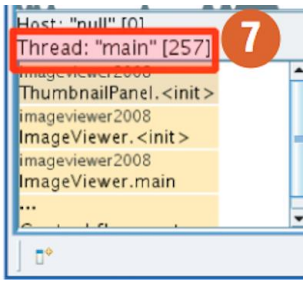
4) Infine lo sviluppatore può “interrogare e navigare” le informazioni acquisite, interrogando il database e navigando all’interno della traccia di esecuzione utilizzando il front-end debugger integrato nell’IDE di Eclipse.



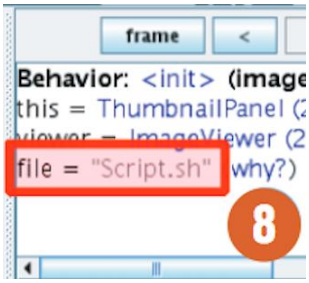
Vediamo ora un semplice esempio, presente nel paper di introduzione [14], in cui si mostra come sia possibile trovare un bug all’interno del codice con TOD. Avviando il programma di debug (1) è possibile individuare nella traccia di esecuzione l’evento che ha generato l’eccezione.
Una volta individuato, nella vista principale del flusso di controllo (2) viene automaticamente evidenziata la riga di codice associata (3).



L’oggetto “ThumbnailPanel” risulta essere ‘null’ (4): in qeusto punto del codice ha generato l’eccezione. Cliccando su “why?” (4) si viene proiettati direttamente nella riga del codice in cui il valore viene impostato e sull’evento che l’ha causato.



Nell’esempio l’azione si verifica in un thread diverso rispetto a quello che ha poi lanciato l’eccezione (7).



Esaminando lo stato del programma emerge che è avvenuto un tentativo di creare un file .sh (8) che però è fallito.

TOD, grazie alla sua interfaccia e alle funzionalità di cui dispone, ha proiettato il programmatore direttamente sul bug. Senza le funzionalità di un TTD, con un normale debugger a runtime, sarebbe stato molto più lungo e complicato trovare l’errore ed intuire il thread responsabile della generazione dell’eccezione.

TOD può arrivare a registrare una quantità enorme di eventi, è quindi fondamentale, per aiutare gli utenti a non perdersi durante la navigazione nella traccia di esecuzione, di poter aggiungere segnalibri a eventi e oggetti[15].

L’evento attualmente selezionato nella vista principale, ad esempio, appare anche nella timeline, in modo che gli utenti possano trovare immediatamente la loro strada nella traccia dell'esecuzione relativa ai punti di riferimento conosciuti.

Questo è particolarmente utile quando si utilizza il collegamento “why?" (8), che può portare a eventi accaduti nel passato presenti in diversi contesti.

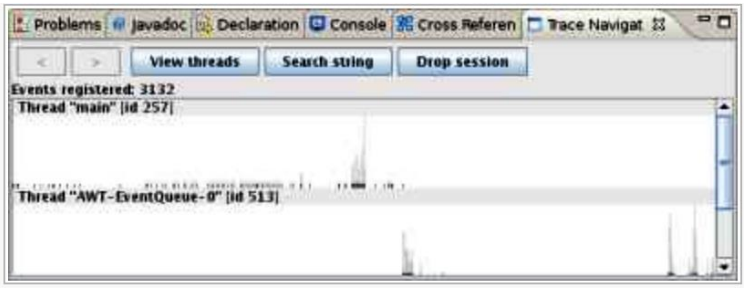
Il problema di sovraccarico di informazioni può essere ridotto dagli utenti più esperti semplicemente riducendo i dati, utilizzando delle tracce parziali.

Gli sviluppatori ottengono tracce parziali in TOD utilizzando sia lo scope statico che dinamico: lo scope statico consiste nella selezione di quali classi devono o meno generare eventi.

L'ambito dinamico consiste nell'abilitare o disabilitare l'acquisizione di tracce in fase di runtime, utilizzando una semplice API direttamente nel file di debug o utilizzando un pulsante che funge da interruttore nel front-end del debugger.

L'ambito dinamico è particolarmente utile quando si verifica un errore dopo un lungo periodo di esecuzione o in specifiche condizioni dinamiche.
Ad esempio, in un'applicazione Web, potrebbe essere interessante limitare l'acquisizione di tracce all'elaborazione di un flusso di controllo di una particolare richiesta HTTP.

TOD dispone inltre di una vista in cui è possibile vedere ed esaminare i thread presenti nel programma.
Nella pagina viene mostrata anche una panoramica dell'attività del programma: ogni banda orizzontale rappresenta l'evoluzione del flusso del programma.



È possibile effettuare il download di TOD direttamente dal sito web (http://pleiad.cl/tod) dove è anche disponibile una documentazione abbastanza dettagliata sul Debugger(https://pleiad.cl/tod/documentation/index.html).

2.1.5 Chronon

Anche Chronon è un Debug Reversibile per Java che si basa sulla registrazione delle tracce, che vengono salvate sul disco in un file.

Questo debugger rispetto ai precedenti, offre una funzionalità di condivisione avanzata che consente di trasferire la registrazione su altre macchine e di condividerla tra i membri del team.

I membri del team possono così utilizzare il file registrato per riprodurre l'intera esecuzione del programma sui loro desktop e trovare più velocemente la causa del problema riscontrato.

Il Chronon Time Traveling Debugger legge i dati dal file di registrazione, come se li leggesse da un database, per riprodurre istantaneamente lo stato del programma in qualsiasi momento: i file di registrazione sono tutto ciò che serve per riprodurre l'esecuzione del programma.

A differenza dei debugger precedenti, questa implementazione consente alle registrazioni Chronon di essere completamente indipendenti dal sistema e lo mantiene sicuro, in quanto non c'è nulla che possa accidentalmente eseguire ed effettuare cambiamenti, ad esempio, al filesystem.

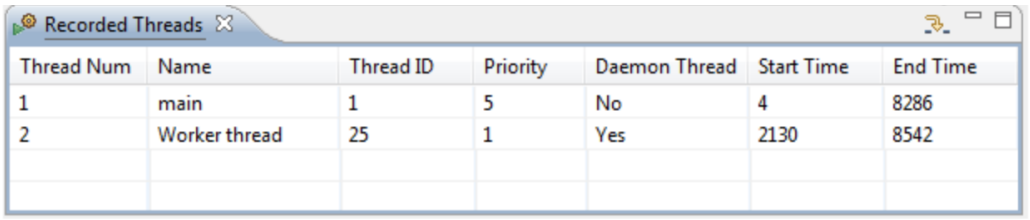
Grazie a queste caratteristiche chronon dispone di alcune importanti caratteristiche[17]:

- **Piattaforma indipendente** : ad esempio, è possibile effettuare una registrazione da una macchina Linux a 64 bit e riprodurla su una macchina Windows a 32 bit;
- **Ambiente indipendente**: Non è necessario disporre del database, della coda di messaggi o di altri componenti dell'ambiente durante la riproduzione delle registrazioni di Chronon. I file di registrazione sono l’unica cosa necessaria;

- **Hardware indipendente**: Anche le registrazioni sono indipendenti dall'hardware. Pertanto, se si dispone di un server di grandi dimensioni con 16 GB di RAM e una piccola finestra di sviluppo con 4 GB di RAM, è comunque possibile riprodurre la registrazione;
- **Nessuna modifica del codice sorgente**: Non è necessario applicare una singola modifica al codice sorgente per utilizzare Chronon. Basta infatti aggiungere solamente alcuni argomenti extra al lancio del programma;
- **Non-Deterministic**: registra tutti gli input e gli eventi non deterministici che interessano il server delle applicazioni mentre è in esecuzione, creando un singolo file di registrazione;

Altra funzionalità di Chronon è che questo Time Travel Debugger è pensato per consentire il debug di programmi di lunga durata (giorni, ma anche mesi) e mette a disposizione tutta le funzionalità standard di un debugger reversibile per consentire di analizzare facilmente il codice. Offre la possibilità di filtrare per thread i flussi registrati, e mette a disposizione del programmatore delle viste per navigare tra i metodi, eccezioni e segnalibri salvati in precedenza.

Chronon supporta la gestione dei thread con la possibilità di scorrere la lista dei thread registratati e navigare all'interno di essi tramite la tabella dedicata mostrata in figura:



Thread Num	Name	Thread ID	Priority	Daemon Thread	Start Time	End Time
1	main	1	5	No	4	8286
2	Worker thread	25	1	Yes	2130	8542

Oltre a informazioni base come il nome del thread o l'id associato è possibile sapere anche l'arco di vita di un thread e consente di effettuare "jump" all'interno del thread che si seleziona.

Chronon implementa un Id univoco per ogni thread, nel "Thread Num" viene riportato questo id del thread che viene assegnato durante la registrazione del programma.

Selezionando un thread è così possibile, cliccando il bottone “jump”, di saltare direttamente al punto in cui il thread inizia a eseguire il codice.

Chronon è un Tool commerciale che offre la possibilità di ottenere una licenza gratuita del Chronon Time Traveling Debugger a colore che soddisfino, come requisito, l’esistenza di un progetto Open Source con una comunità attiva. Consentendo l’invio di registrazione di Chronon al repository di bug solo agli utenti, si ha il vantaggio di poter inviare le registrazione dei bug direttamente sulla repository per mostrare il bug identificato senza dover spiegare o indicare l’errore rilevato.

La richiesta per questa licenza può essere effettuata direttamente dal sito ufficiale, dove si può trovare anche il download alla licenza da sviluppatore professionale.

Anche Chronon dispone di documentazione online e supporto diretto sulla pagina web ed è attualmente supportato(http://chrononsystems.com/solutions/support).

2.1.6 CaReDeb

CaReDeb è un uno dei primi prototipi di debugger che unisce le funzionalità di un debugger reversibile con la metodologia di reversibilità Casual-Consistent[7].

Questo debugger Reversibile Casaual-Consistent è scritto in java e si basa, per implementare le primitive, sul linguaggio μOz.

Il linguaggio μOz è higher-order e si basa per gestire la concorrenza sui thread e dispone di una comunicazione asincrona tramite porte.

La sintassi di μOz è riportata in Figura:

$S ::=$		Statements
	skip	Empty statement
|	$S_1\ S_2$	Sequential composition
|	**let** $x = v$ **in** S **end**	Variable declaration
|	**if** x **then** S_1 **else** S_2 **end**	Conditional statement
|	**thread** S **end**	Thread creation
|	**let** $x = c$ **in** S **end**	Procedure declaration
|	$\{\ x\ x_1 \ldots x_n\ \}$	Procedure call
|	**let** $x =$ `NewPort` **in** S **end**	Port creation
|	$\{$ `Send` $x\ y\ \}$	Send on a port
|	**let** $x = \{$ `Receive` $y\ \}$ **in** S **end**	Receive from a port
$v ::=$	**true** | **false** | 0 | 1 ...	Simple values
$c ::=$	**proc** $\{\ x_1 \ldots x_n\ \}\ S$ **end**	Procedure

I valori riconosciuti in μOz sono booleani, numeri naturali, porte e procedure (di comunicazione). Le variabili sono immutabili, cioè sono variabili di sola lettura inizializzate al momento della loro dichiarazione. La comunicazione è asincrona e viene realizzata mediante azioni di invio e ricezione su una porta, a cui è associata una coda FIFO. Dichiarazione delle variabili, dichiarazione delle procedure, creazione e ricezione delle porte sono leganti.

Oltre alle funzionalità principali dei debugger reversibili viste nei precedenti debugger, per implementare l'approccio casuale coerente, CaReDeb dispone di particolari comandi specifici, in particolare dispone di una suite di comandi e funzionalità che consentono di annullare l'ultima azione che ha portato ad un comportamento imprevisto[7]:

- **Valore errato in una variabile**: se l'ID di una variabile ha un valore imprevisto, Rollvariable id del comando, consente al programmatore di andare nello stato appena prima della creazione dell'id della variabile associata;
- **Valore errato in un elemento della coda**: se un elemento della coda associato alla porta ID ha un valore imprevisto, rollsend id del comando n consente al programma di annullare le ultime n mandate a questa porta. Se n non è specificato, l'ultima mandata viene annullata;
- **Thread bloccato su una ricezione**: se un thread è bloccato su una ricezione su una coda vuota, è possibile che il messaggio desiderato sia stato letto da un altro thread. Questo può essere verificato osservando la cronologia della coda, che contiene i messaggi che erano precedentemente nella coda. In questo caso, il comando rollreceive ID n consente al programmatore di annullare le ultime n ricezioni della porta ID. Se n non è specificato, l'ultima ricezione viene annullata;
- **Thread imprevisto**: se viene trovato un thread inatteso t, il comando rollthread t consente al programmatore di annullare la creazione del thread.

(Comandi CaReDeb [7] **)**

control	**forth (f) t**	(forward execution of one step of thread `t`)
	run	(runs the program)
	rollvariable (rv) id	(causal-consistent undo of the creation of variable id)
	rollsend (rs) id n	(causal-consistent undo of last n send to port `id`)
	rollreceive (rr) id n	(causal-consistent undo of last n receive from port `id`)
	rollthread (rt) t	(causal-consistent undo of the creation of thread `t`)
	roll (r) t n	(causal-consistent undo of n steps of thread `t`)
	back (b) t	(backward execution of one step of thread `t` (if possible))
explore	**list (l)**	(displays all the available threads)
	store (s)	(displays all the ids contained in the store)
	print (p) id	(shows the state of a thread, channel, or variable)
	history (h) id	(shows thread/channel computational history)

Tutti questi comandi sono causal consistent cioè, annullano tutte le azioni che dipendono dall'azione di destinazione, mentre non annullano le azioni concorrenti.

Ad esempio, annullare l'invio di un valore richiede di annullare la ricezione dello stesso valore, se eseguita e non ancora annullata.
Allo stesso modo, annullare la creazione di un thread richiede di annullare tutte le azioni eseguite dal thread creato.

Questo è fondamentale per garantire la coerenza causale: da un lato, questo permette di tornare ad uno stato passato che potrebbe essere raggiunto da un'esecuzione diretta, dall'altra parte viene annullato il numero minimo di azioni che sono necessarie per raggiungere questo scopo.

CaReDeb è uno dei primi prototipi di Debugger Reversibile Casaual-Consistent scritto in java, che si basa, per implementare le primitive, sul linguaggio μOz.
È possibile effettuare il download direttamente dalla pagina web (http://www.cs.unibo.it/caredeb/deb.html).

2.1.7 Expositor

Expositor è un time-travel debugger per Python e rappresenta un ambiente di debug che combina lo scripting e il debug Time-travel per consentire ai programmatori di automatizzare complesse attività di debug[22].

Expositor richiede UndoDB, e offre prestazioni migliori con una versione recente di GDB.
Anche questo debug si basa sui Trace di esecuzione: una sequenza indicizzata nel tempo di istantanee dello stato del programma.
A differenza dei precedenti time-trave debugger cerca però di minimizzare il numero di accessi ad esso.

I programmatori possono manipolare le tracce inoltre, Expositor include le funzionalità di un debugging Scriptable: una tecnica molto potente per il test in cui i programmatori possono scrivere script per eseguire complesse attività di debug. Vediamo un semplice esempio che descrive questa funzionalitàè[18]:

Supponiamo di avere un bug che coinvolge i valori di una struttura dati.
Con Expositor possiamo eseguire il debug del problema scrivendo direttamente uno script che mantiene una struttura di dati “shadow” che implementa lo stesso insieme ma tramite una struttura più semplice di quella precedentemente implementata, per esempio, con una lista[18].

Eseguendo il programma buggy, lo script tiene traccia delle chiamate del programma da inserire e rimuovere, interrompendo l'esecuzione quando i contenuti della struttura dei dati “shadow” non corrispondono a quelli del buggy, aiutando ad individuare l’errore senza ricorrere a particolari fasi di debug.

Si potrebbe ovviamente utilizzare la stessa strategia di debug modificando il programma stesso (ad esempio inserendo istruzioni di stampa), ciò implicherebbe però la ricompilazione del programma, e questo potrebbe richiedere molto tempo per i programmi di grandi dimensioni, rallentando così notevolmente il tasso di verifica dell'ipotesi.

Gli script Expositor trattano il Trace di esecuzione di un programma come una lista (potenzialmente infinita) immutabile di istantanee dello stato del programma con annotazioni temporali.

Gli script possono creare o combinare tracce utilizzando le operazioni comuni della lista: le Tracce possono infatti essere filtrate, mappate, divise e unite per creare proiezioni più leggere dell'intera esecuzione del programma.

In questo senso, Expositor è particolarmente adatto per controllare le proprietà temporali di un'esecuzione e per scrivere nuovi script che analizzano le tracce calcolate da script precedenti.

Expositor estende infatti l'ambiente Python di GDB e utilizza il time-travel backend di UndoDB per GDB, gli utenti possono cosi passare agevolmente da script in esecuzione e interagire direttamente con un'esecuzione tramite GDB.

L'idea chiave per rendere efficiente Expositor è di sfruttare l'implementazione “Lazy” delle tracce: invocare il debugger time-travel è costoso e questa implementazione aiuta a ridurre al minimo il numero di chiamate.

Ad esempio, supponiamo di utilizzare il “combinatore di breakpoint” di Expositor per creare una traccia “TraceMalloc” contenente solo le chiamate Malloc dell'esecuzione del programma[22].

Se chiediamo il primo elemento di "TraceMalloc" prima del tempo t-esimo (per esempio con t=42 , perché c'è un output del programma sospetto); quello che Expositor farà è di dirigere il debugger time-travel sul tempo t-esimo (42) e lo eseguirà all'indietro fino a raggiungere la chiamata, catturando lo stato risultante nel Trace della struttura dei dati. Il promemoria della traccia, dopo il tempo 42 e prima della chiamata Malloc, non viene calcolato.

Queste strutture di dati devono essere anch'esse lazy per non compromettere la traccia: se avessimo calcolato l'insieme appena citato per rispondere a una query di appartenenza al tempo t-esimo, avremmo dovuto eseguire il debugger time-travel dall'inizio fino a t, considerando tutte le chiamate Malloc, anche se solo la chiamata più recente era sufficiente per soddisfare la query.

Expositor per consentire questa interessante funzionalità fornisce una nuova struttura di dati: l'edit hash array mapped trie (EditHAMT), che fornisce costruzione lazy e query per insiemi, mappe, multiset e multimaps[22].

Expositor non risulta al momento supportato ed è, all'ultima versione rilasciata, utilizzabile solo per il debug di applicazioni Linux 32-bit. È possibile esegueire Expositor seguendo la guida sul sito (https://bitbucket.org/khooyp/expositor)

2.1.8 Elm TTD

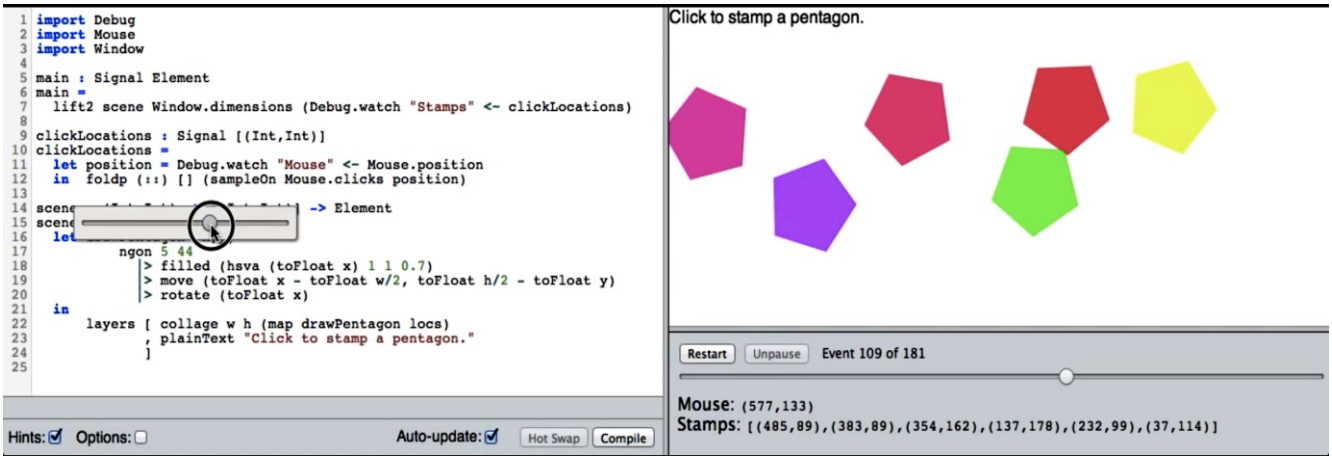
Ispirato dalla discussioni come “Inventing on Principle” di Bret Victor[19], Laszlo Pandy ha implementato un debugger reversibile open source.
Elm TTD consente infatti di visualizzare e interagire con il programma permettendo di vedere come gli oggetti, e di conseguenza il programma, cambiano nel tempo.

Elm è un linguaggio di programmazione funzionale per la realizzazione di interfacce utente grafiche (GUI) basate su browser web. La programmazione funzionale, così come l'immutabilità degli oggetti, rende un TTD semplice da implementare per ELM. Gli effetti collaterali sono modellati esplicitamente, questo significa che "Per eseguire un effetto collaterale, si deve prima creare una struttura dati che rappresenta ciò che si vuole realizzare. Quindi si fornisce la struttura dati al sistema di runtime del linguaggio per eseguire effettivamente l'effetto collaterale.”[20]
Pertanto, in ELM per riprodurre eventi senza effetti collaterali, è sufficiente che a runtime venga richiesto di non eseguire alcun effetto collaterale.

I dati immutabili garantiscono poi che i dati non vengano sovrascritti. Infine, ELM si basa su FRP significa che gli eventi sono già tracciati nel tempo tramite "segnali". "Ciò rende la gestione del replay una questione di registrazione degli eventi in arrivo nel programma e di eliminazione degli eventi in uscita. Fintanto che nessuno agisce sugli eventi in uscita, non ci saranno effetti collaterali indesiderati." [20] Forse, l'aspetto più impressionante del debugger di ELM è il supporto per hotswapping.
L’hotswapping è l’azione di sostituire il codice mentre un programma è in esecuzione. La combinazione dei due consente all'utente di riavvolgere il tempo, modificare i valori e riprodurre l'esecuzione senza riavviare l’esecuzione

Ho analizzato una serie di esempi per capire le funzionalità di questo debugger reversibile e sull’utilità che ha nel rintracciare i bug tramite l’interfaccia grafica di cui dispone, (gli esempi sono presenti nel sito ufficiale online, con le rispettive immagini[20]):

Il primo esempio mostra le basi del debugger. Utilizzando un semplice programma di timbratura viene mostarto, visivamente, in cosa consiste la possibilità di viaggiare nel tempo e come influiscono eventuali modifiche sul codice sugli oggetti esistenti:



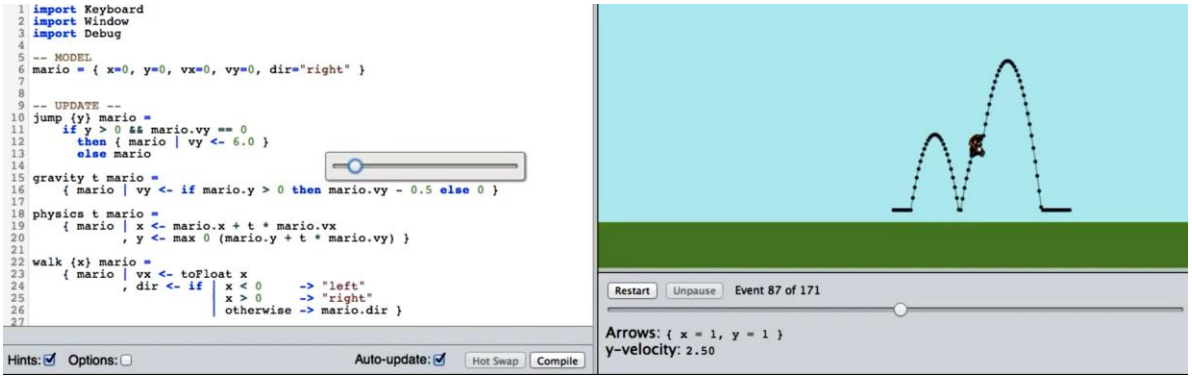
Tutte le istruzioni per stampare gli oggetti in figura con lo stampatore sono registrate in modo da poter mettere in pausa, riavvolgere e riprodurre una sequenza di eventi.

È possibile modificare il codice in qualsiasi momento per vedere come cambiamo direttamente gli oggetti a runtime.

Questo rende facile vedere come cambiano i francobolli nel tempo ma non solo, ad esempio può essere necessario, per essere più precisi, modificare le coordinate di stampa o sapere dove un francobollo è stato posizionato: come si può allora sapere dove si trova esattamente tale francobollo?

Nella figura mostrata precedentemente, sotto il cursore del tempo si possono notare i valori che mostrano la posizione esatta a runtime di ciascun timbro e la posizione del cursore del mouse offrendo all'utente un feedback a runtime immediato.
Questo consente di correggere eventuali errore in maniera molto rapida senza la necessità di breackpoint o variabile in Watch per verificare le coordinate degli oggetti, e allo stesso tempo, consente di posizionare i nuovi oggetti in modo preciso molto velocemente.

Nel prossimo esempio, leggermente più complicato, ho analizzato come un bug, apparentemente più complicato, può essere risolto velocemente con queste funzionalità.



Il bug: nel codice è presente qualche istruzione errata che permette a Mario di effettuare un salto che risulta essere il doppio in altezza di quanto stabilito. Elm TTD consente di scorrere la traccia temporale per identificare esattamente l'arco temporale in cui si verifica il bug e determinare esattamente in quale posizione di codice si verifica il problema.

Questa traccia è fondamentale per visualizzare la posizione del bug e avere un filo diretto con le variabili e funzioni che hanno causato l'errore. Per rintracciare il bug del salto doppio, è stato sufficiente vedere come il cambiamento di Mario nel tempo andava ad impattare con le variabili del programma.

Elm TTD offre questa funzionalità per cui, un elemento visivo è contrassegnato da una stringa che funge da ID univoco in tutto il programma, consentendo al debugger di tracciare l'oggetto, nell'esempio Mario, nel tempo. È facile immaginare di attivare e disattivare la traccia per elementi specifici o di avere un IDE in grado di aggiungere tag di traccia senza realmente modificare esplicitamente il codice sorgente.

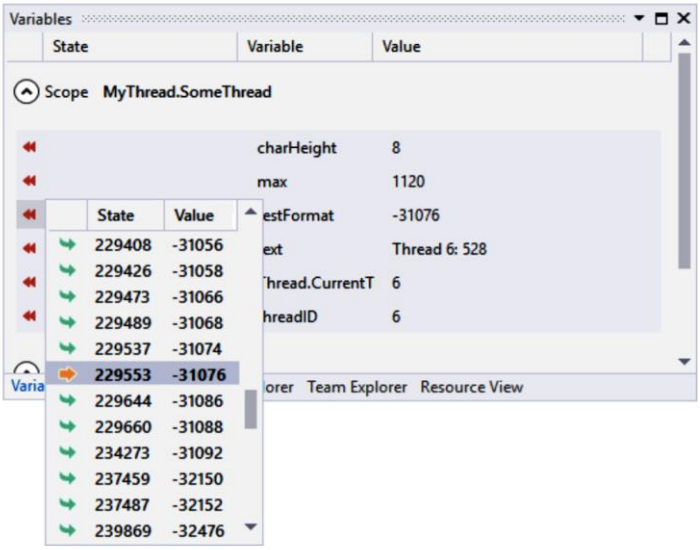
Elm TTD offre così un valido strumento di debugger grafico, è attualmente supportato e conta più di 200 contributi nell'ultimo anno. È possibile scaricarlo direttamente dal sito online e pur non disponendo di una documentazione dettagliata come i debugger visti in precedenza dispone di numerose risorse sulla comunity GitHub, "Official Organization for developing Elm's compilare" (https://github.com/elm-lang).

2.1.9 RevDeBug

RevDeBug è un Debug Reversibile per .Net e C# che registra ciò che avviene durante l'esecuzione di un'applicazione.

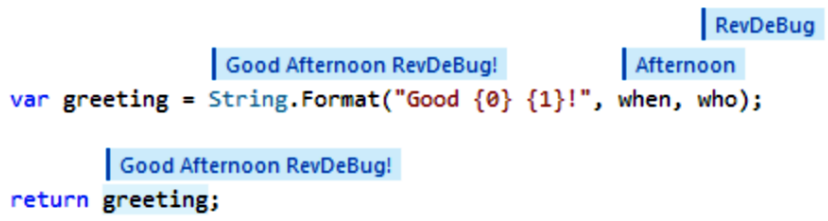
Le principali funzionalità di questo Debugger sono[21]:

- **Registra la cronologia delle variabili locali:** registra la cronologia di tutti i valori delle variabili locali. Se si vuole sapere dove è stato impostato o utilizzato il valore di una particolare variabile è sufficiente saltare istantaneamente a quel particolare momento con l'uso dei comandi associati.



(GUI RevDebug,[Immagini capitolo 21])

- **Valori delle variabili visibili:** Mentre si è in fase di debugging, la vista del codice viene "decorata" con i valori di variabili e valori di ritorno dei metodi eseguiti senza la necessità di dover compiere particolari azioni come per esempio lo spostamento nel Watch delle variabili per vedere il valore che assumono

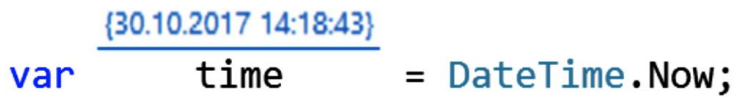


- **Dove e quando sono stati eseguiti i metodi**: Con RevDeBug è possibile impostare i punti di interruzione di debug per navigare rapidamente verso tutte le occorrenze in esecuzione che sono state registrate in particolari punti nel codice.

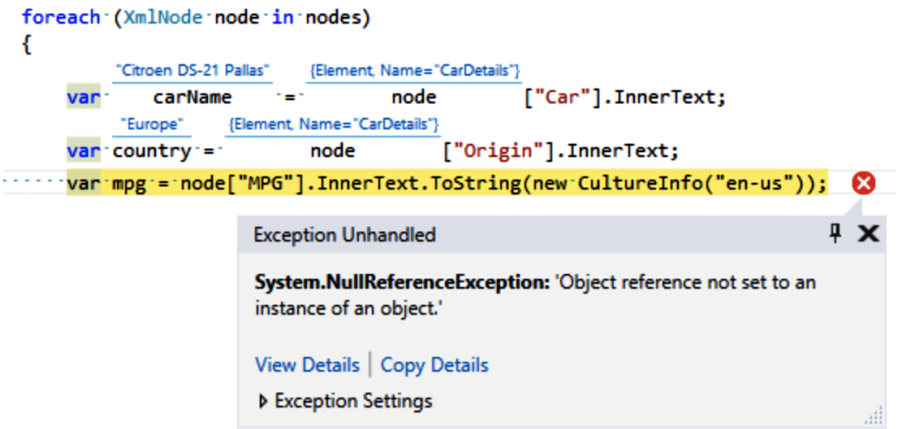
- **Vista di eccezioni gestite e non gestite**: RevDeBug traccia tutte le eccezioni gestite e non gestite, generate dal flusso del programma. Questo permette di risalire il codice a partire dal momento in cui sono state lanciate o catturate le eccezioni aiutando lo sviluppatore a comprendere il vero flusso che ha portato alla generazione del bug.

Il team di RevDebug ha infine rilasciato un nuovo programma per migliorare il comfort del debugging, integrabile con RevDebug, che prende il nome di Prompter.

Prompter permette di vedere i valori nel codice durante il debugging consentendo cosi di evitare copie, a volte scomode, nel Watch.



Prompter offre poi altre funzionalità tra le quali, la possibilità di vedere all'interno del codice il motivo che ha portato alla generazione di un'eccezione, con le variabili tutte visibili grazie alla funzionalità mostrata precedentemente.



RevDeBug è un debugger commerciale che richiede l'acquisto di una licenza con la possibilità di ottenere la versione Trial di 30 giorni.
Dispone di documentazione sul sito online ed è attualmente supportato (https://www.revdebug.com/doc)[21].

2.1.10 Time Travel Debugging for Windows

Il 25 Settembre 2017, Microsoft ha annunciato il rilascio del proprio Time Travel Debugging (TTD) per l'ultima versione di WinDbg per Windows.
Questa versione è una preview di TTD e rappresenta la prima volta in cui un debugger di windows consente di scorrere, con le funzionalità di un time travel debugger, il codice del programma[23].

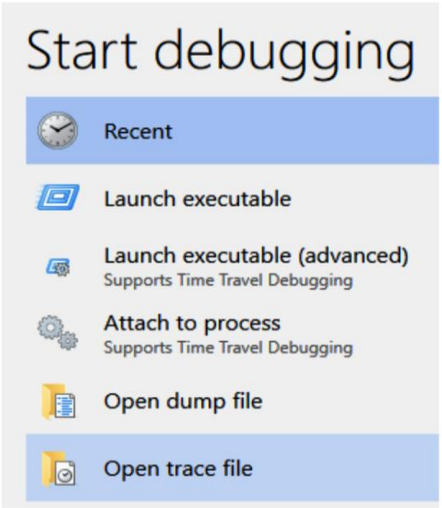
Windows Time Travel Debugging (TTD) è stato per molti anni uno strumento di debug per gli sviluppatori di software e ingegneri all'interno di Microsoft.

TTD è disponibile per Windows 10 installando WinDbg e si basa su tre funzionalità principali[23]:

1) **Record**: registra il processo sulla macchine per poter riprodurre il bug. Viene creato un file di Trace (.RUN) che contiene tutte le informazioni per riprodurre il bug;

2) **Replay**: aprendo il file di Trace in WinDbg si può riprodurre l'esecuzione del codice in avanti e indietro tutte le volte necessarie fino a quando non si comprende la causa del bug;

3) **Analyze**: Eseguendo le query e i comandi è possibile identificare i problemi e avere pieno accesso alla memoria per capire cosa accade nel codice.

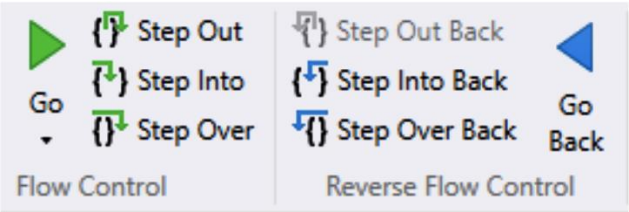


Nel file di Trace (.RUN) viene memorizzato il codice della registrazione del codice in esecuzione.

Una volta interrotta la registrazione, viene creato un file indice (.IDX) per consentire un accesso più rapido alle informazioni del Trace.
Questi file possono arrivare a raggiungere dimensioni pari al doppio del file di trace.
WinDbg TTD non limita le dimensioni del file di

Trace e può quindi arrivare a occupare molto spazio sul disco.

Sta quindi all'utente controllare che sia disponibile uno spazio libero adeguato. (Gigabyte per pochi minuti di registrazione)
Una volta che viene creata la traccia, viene caricata e indicizzata automaticamente per una riproduzione e una ricerca in memoria più veloce.



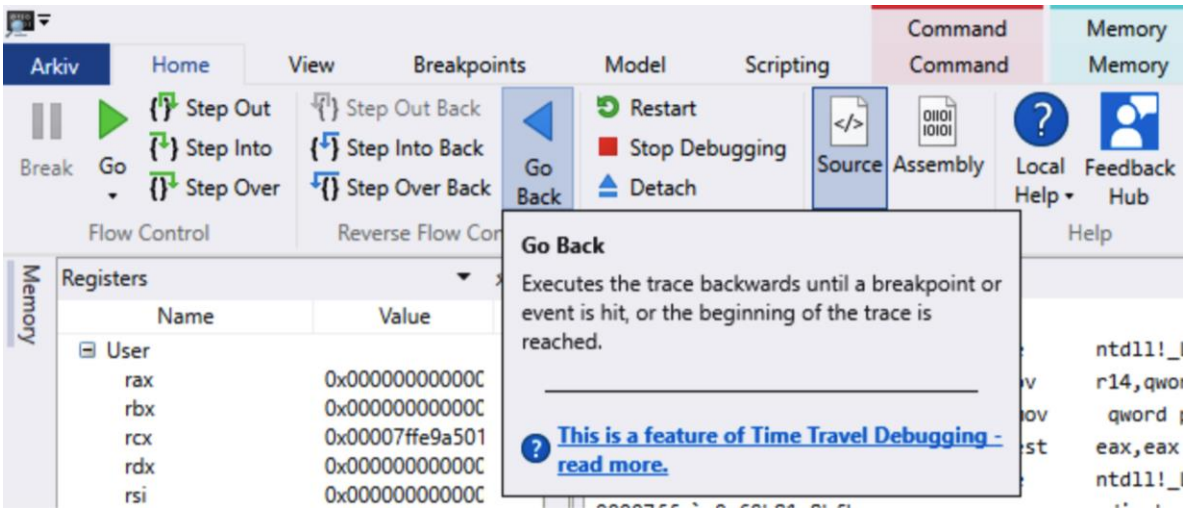
Semplicemente utilizzando i pulsanti e comandi dalla barra multifunzione di WinDbg si può poi scorrere avanti e indietro il codice.

L'implementazione si basa quindi sulla riproduzione di record: il replay viene eseguito utilizzando lo stesso motore binario di traduzione della registrazione, a cui viene assegnato il codice da eseguire e i valori iniziali del registro.

Quando vengono letti valori imprevisti, essi vengono catturati dal file di Trace: il file di Trace include effettivamente una copia del programma binario target.

Il debugger inverso si rende conto di aver creato uno stato corrente incoerente del programma. Dai documenti TTD[24]:

> "TTD funziona eseguendo un emulatore all'interno del debugger, che esegue le istruzioni del processo di debug per replicare lo stato di quel processo in ogni posizione della registrazione. I deragliamenti accadono quando questo emulatore osserva una sorta di discrepanza tra lo stato risultante e le informazioni trovate nel file di traccia."



WinDbg ha un concetto di identificatore di "posizione" in due parti che consiste in un numero di "evento di sequenziamento" e un conteggio "di passaggio". Tali posizioni sono mantenute separatamente per ogni thread in un programma. La registrazione viene quindi sincronizzata sugli eventi di sequenziamento.

WinDbg consente infine di condividere una registrazione con altri utenti, senza che sia installato il programma originale. Questo viene fatto includendo fondamentalmente il codice del programma e lo stato iniziale nel file di Trace, dai documenti TTD[24]:

> “I file di Trace TTD possono essere condivisi con altri utenti copiando il file .RUN. Questo può essere utile per avere un collaboratore che ti aiuta a capire il problema. Non è necessario installare l'app in crash o eseguire altre impostazioni correlate per tentare di riprodurre il problema. É sufficiente caricare il file di Trace ed eseguire il debug dell'applicazione come se fosse installata sul proprio PC”.

L’interfaccia utente (GUI) dell’applicazione è molto chiara ed intuitiva. La GUI è ottimizzata per spiegare nel modo più chiaro possibile le funzionalità degli oggetti presenti. I pulsanti come nell’esempio sopra “Go Back” dispongono di suggerimenti e indicano in link alla documentazione online, al fine di semplificare l’accesso alla funzione.

Altra funzionalità molto intelligente della GUI è la possibilità di accedere direttamente al database che contiene la traccia del programma che può essere direttamente interrogata con query LINQ.

Il nuovo Debugger di windows dopo l'aggiornamento offre, dopo anni di uso interno, anche agli sviluppatori esterni il proprio debug reversibile. È un tool con licenza commerciale che dispone di una vasta documentazione online, direttamente sul sito Microsoft:
(https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview).

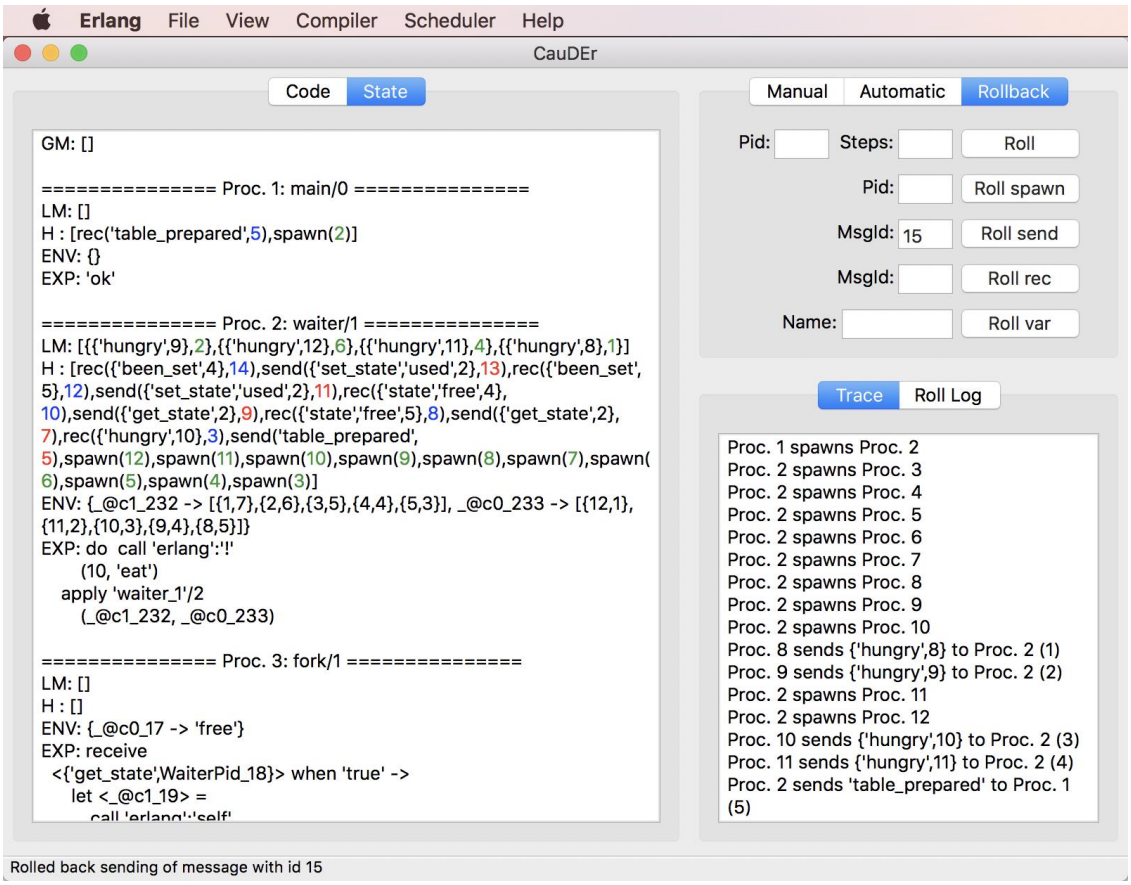
2.1.11 CauDEr

CauDEr è un nuovo prototipo di debugger reversibile, che come CaReDeb, implementa una reversibilità di tipo Casual-Consistent per un sottoinsieme di programmi Erlang[10].

Lo strumento si basa su alcuni sviluppi recenti sulla semantica Casual-Consinstent reversibile di Erlang (linguaggio di programmazione funzionale concorrente e distribuito) ed introduce una nuova semantica di rollback utile per le funzionalità del debugging reversibile.

Tramite questa semantica, a differenza delle implementazioni precedenti, è infatti possibile eseguire un programma all'indietro fino all'invio di un particolare messaggio, alla creazione di un determinato processo o all'introduzione di un'associazione di alcune variabili.

CauDEr è integrato con un'interfaccia grafica per facilitare l'interazione degli utenti con il debugger reversibile riportata in figura (Immagina presente nel papaer [10])



Quando viene avviato, dopo aver aperto un file sorgente di Ergang, il file selezionato viene tradotto in Core Erlang e il codice risultante viene visualizzato nella scheda “Code”.
L’utente può scegliere una delle funzioni dal modulo e scrivere gli argomenti con cui vuole valutare la funzione.

Quando l'utente preme il pulsante START, viene visualizzato nella scheda “State” uno stato iniziale del sistema, con una casella postale globale vuota e un singolo processo che esegue l'applicazione della funzione specificata.

CauDEr consente all'utente di esplorare le possibili esecuzioni del programma sia in avanti che all'indietro, in base a tre diverse modalità (presenti nella sezione in alto a destra nella figura riportata sopra)[10]:

- **Modalità Manuale**: l’utente può selezionare un processo (o un identificatore del messaggio) e sono disponibili i pulsanti corrispondenti alle riduzioni abilitate in avanti e all'indietro per il processo (o messaggio) scelto.

- **Modalità automatica**: è possibile decidere la direzione (in avanti o all'indietro) e il numero di passi da eseguire. I passi effettivi sono selezionati da uno scheduler appropriato. Al momento dispone di due scheduler (random), uno dei quali dà la priorità ai processi che hanno la pianificazione dei messaggi, mentre l'altra ha una distribuzione uniforme. La modalità automatica include anche un pulsante “Normalize”, che esegue tutte le azioni abilitate associate alla schedulazione dei messaggi.

- **Modalità Rollback**: implementa gli operatori di rollback, che consento di effettuare:
    - un passo indietro;
    - una derivazione all'indietro fino all'invio di un messaggio etichettato con un simbolo identificativo;
    - una derivazione all'indietro fino alla consegna di un messaggio etichettato con un simbolo identificativo ;
    - una derivazione all'indietro fino alla ricezione di un messaggio etichettato con un simbolo identificativo;
    - una derivazione all'indietro fino alla deposizione delle processo con pid p;
    - una derivazione all'indietro fino alla creazione del processo annotato;
    - una derivazione all'indietro fino all'introduzione della variabile X;

Durante l'esplorazione dell'esecuzione, sono presenti due schede che vengono aggiornate per fornire informazioni sul sistema e sulla sua esecuzione. La scheda "State" descrive il sistema corrente, compresa la casella postale globale (GM), e, per ogni processo, i seguenti componenti: la casella postale locale (LM), la cronologia (H), l'ambiente (ENV) e l'espressione in valutazione (EXP). Gli identificatori dei messaggi sono evidenziati a colori. Questa scheda può essere configurata per nascondere qualsiasi componente della rappresentazione del processo.
La scheda "Trace" fornisce invece una descrizione linearizzata delle azioni simultanee eseguite nel sistema, ovvero invio e ricezione di messaggi e generazione di processi. Questo ha lo scopo di dare un quadro globale dell'evoluzione del sistema, per evidenziare anomalie che potrebbero essere causate da bug.

È disponibile un'ulteriore scheda, "Roll Log", che viene aggiornata in caso di rollback, mostrando quali azioni sono state effettivamente annullate su una richiesta di rollback. Questa scheda consente di comprendere le dipendenze causali del processo di destinazione della richiesta di rollback, evidenziando frequentemente dipendenze indesiderate o mancanti direttamente causate da bug.

La versione di rilascio (v1.0) di CauDEr è scritta in Erlang ed è disponibile pubblicamente direttamente sul sito (https://github.com/mistupv/cauder) con licenza MIT.
L'unico requisito per creare l'applicazione è avere Erlang/OTP installato e compilato con wxWidgets.
Il repository include anche alcuni documenti e alcuni esempi per testare facilmente l’applicazione.

3. Conclusioni

3.1. Riepilogo sui Debugger: Tabella Riepilogativa

Nel corso della mia analisi ho potuto analizzari vari Debugger Reversibli che, pur avendo alla base delle funzionalità simili, hanno implementazioni che possono varirare notevolemnte. Ci sono infatti diverse tipologie di Debugger Reversibili:
I debugger che funzionano registrando lo stato del programma sottoposto a debug dopo ogni istruzione e quindi, tramite ricostruzione dello stato dei log, on demand; e Debugger Reversibili che si basano sul replay, registrando i risultati delle chiamate di sistema che il programma effettua, creando dei checkpoint intermedi, in modo che il debugger possa poi ricostruire un determinato stato richiesto del programma iniziando da un checkpoint e ripetendo il programma con le chiamate di sistema registrate.

Questi Debugger con funzionalità base a volte simili sono implementati seguendo logiche e criteri differenti, per conlcudere al meglio questo elaborato ho ritenuto opportuno riassumemere all'interno di due tabelle le informazioni e caratteristiche principali dei debugger analizzati.

Nella prima tabella che riporto di seguito sono mostrati i debugger che ho analizzato durante la redazione della tesi in cui vengono mostrate le caratteristiche principali e le funzionalità che offrono:

	Platform/(Lenguage)	Storage	Casual Nav.	IDE Integration	Concurrency
UndoDB	 Linux & Android	Ram & Disk	✖	Eclipse, CLion, DDD	N-D
GDB	Unix-like & Microsoft (C)	Ram & Disk	✖	Eclipse, DDD etc	Det
ODB	 (Java)	Ram	✔	Limited Eclipse Integration	CCN
TOD	 (Java)	Disk	✔	Eclipse	CCN
Chronon	 (Java)	Disk	✖	Eclipse	N-D
CaReDeb	 (µOz)	Ram	✔	Command-line Tool	CCN
Expositor	 (Python)	Disk	✖	UndoDB/GDB IDE	?
Elm TTD	 (Elm)	Ram	✖	Soon	N-D
RevDeBug	 (.Net & C#)	Disk	✖	Microsoft Visual Studio	?
WinDbg TDD	 (.Net)	Disk	✖	WindDbg	?
CauDEr	 (Erlang)	Ram	✔	GUI	CCN

La colonna “Concurrency” indica per ogni debugger quale tipo di implementazione adotta per gestire la concorrenza tra le diverse implementazioni viste nel capitolo 1.4 (N-D: Non-Deterministic, DET: Deterministic, CCN: Casual-Consistent Navigation)

Nella seconda tabella riporto invece le informazioni dei debugger riguardo al supporto (attualmente supportato o non più supportato), al tipo di tool (Accademico, prototipo, commerciale..ecc) e al tipo di licenza adottato:

	Support	Tool	Software Licenses
UndoDB	✔	Commercial	Fixed/Floating License
GDB	✔	Free Software	GNU GPL
ODB	✖	Free Software	GNU GPL
TOD	✖	Prototype	Open Source
Chronon	✔	Commercial	Commercial license
CaReDeb	✔	Prototype	GNU GPLv3
Expositor	✖	Accademico (University of Meriland)	ISC
Elm TTD	✔	Open Source	BSD3
RevDeBug	✔	Commercial	Commercial license
WinDbg TDD	✔	Commercial	Commercial license
CauDEr	✔	Prototype	MIT

4. Ringraziamenti

Il primo ringraziamento va al prof. Ivan Lanese per la disponibilità mostrata nei miei confronti e per il tempo che mi ha dedicato per svolgere la tesi.

Ringrazio la mia famiglia che mi ha supportato in questi lunghi anni e mi ha permesso di frequentare questo corso.
Un grazie immenso alla mia ragazza, Elisa per il costante sostengo e supporto nei momenti più difficili...finalmente ce l'ho fatta!!!

Un grazie enorme ai miei amici Vadesi, che da sempre mi supportano e con cui condivido, fin da piccolo, una grande amicizia.

Ringrazio poi tutte le persone che ho avuto modo di conoscere in questi lunghi anni universitari. Per primi i miei compagni di corso Alessandro, Costanza, Daniele, Domenico, Federico, Matteo e Lorenzo, a loro vanno i miei più sentiti ringraziamenti per l'aiuto, le risate e i bei momenti trascorsi insieme che non dimenticherò mai.
Infine ringrazio Daniele, anzi Dani c, (il nome Daniele è sempre stato un problema ahah) per la sua infinita bontà, dedizione e passione che ha mostrato in questi anni sopportandomi sempre.

5. Riferimenti Bibliografici

[1] *National Institute of Standards and Technologies, "Software Errors Cost U.S. Economy $59.5 Billion Annually," June 2002; www.nist.gov/public_affairs/releases/n02-10.htm.*

[2]*Weiser, M. (1982). Programmers use slices when debugging. Communications of the ACM, 25(7), 446-452.*

[3]*Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005, September). SOBER: statistical model-based bug localization. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 5, pp. 286-295). ACM.*

[4]*Zelkowitz, M.V.: Reversible execution. Commun. ACM 16(9), 566 (1973)*

[5]*Grishman, R.: The debugging system AIDS. In: AFIPS 1970 (Spring), pp. 59–64. ACM (1970)*

[6]*Engblom, J. (2012, September). A review of reverse debugging. In System, Software, SoC and Silicon Debug Conference (S4D), 2012 (pp. 1-6). IEEE.*

[7] *Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: FASE. LNCS, vol. 8411, pp. 370–384. Springer (2014)*

[8] *Undo Software. Commercial reversible debugger, http://undo-software.com/*

[9] *Chronon Systems. Commercial reversible debugger, http://chrononsystems.com/*

[10]*Lanese, I., Nishida, N., Palacios, A., & Vidal, G. CauDEr: A Causal-Consistent Reversible Debugger for Erlang⋆.*

[11]*GDB reversible debugger https://www.gnu.org/software/gdb/news/reversible.html*

[12] *GDB documentation https://sourceware.org/gdb/documentation/*

[13]*Bil Lewis. "Debugging Backwards in Time". In: (Oct. 2003). url: http : // arxiv.org/abs/cs/0310016.*

[14]*Pothier, Guillaume, and Éric Tanter. "Back to the future: Omniscient debugging." IEEE software 26.6 (2009).*

[15]*Pothier, G., & Tanter, É. (2008, March). Extending omniscient debugging to support aspect-oriented programming. In Proceedings of the 2008 ACM symposium on Applied computing (pp. 266-270). ACM.*

[16] *TOD reversible debugger https://pleiad.cl/tod/screenshots.html*

[17] *Chronon Time Travel Debugging http://chrononsystems.com/products/chronon-time-travelling-debugger*

[18]*Khoo, Y. P., Foster, J. S., & Hicks, M. (2013, May). Expositor: scriptable time-travel debugging with first-class traces. In Proceedings of the 2013 International Conference on Software Engineering (pp. 352-361). IEEE Press.*

[19]*Victor, Bret. "Inventing on principle, 2012." Accessed on 11.04 (2013).*

[20]*Elm Debugger. url: http://debug.elm-lang.org/*

[21] *RevDeBug Reversible Debugger https://www.revdebug.com*

[22] *Microsoft Research Talks 2012(https://www.microsoft.com/en-us/research/video/expositor-scriptable-time-travel-debugging-with-first-class-traces/)*

[23]*https://blogs.msdn.microsoft.com/windbg/2017/09/25/time-travel-debugging-in-windbg-preview/*

[24]*https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview*