

Università degli studi di Pisa
Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Tesi di laurea

**“Progetto e sviluppo di uno strumento
grafico integrato per la gestione
di tracce di esecuzioni concorrenti”**

Relatori

Prof. Cosimo Antonio Prete

Prof.ssa Gigliola Vaglini

Candidato

Daniele Baldi

Anno accademico 2002-2003

Indice

Introduzione	v
Capitolo 1 Tecniche di analisi di programmi concorrenti	1
1.1 <i>La concorrenza</i>	<i>1</i>
1.2 <i>Comprendere il comportamento dei programmi concorrenti.....</i>	<i>3</i>
1.3 <i>Analisi delle tracce</i>	<i>4</i>
1.4 <i>Visualizzazione delle tracce.....</i>	<i>5</i>
1.4.1 <i>I diagrammi spazio-tempo (STD)</i>	<i>6</i>
1.5 <i>Tecniche avanzate d'analisi.....</i>	<i>7</i>
1.6 <i>Conclusioni</i>	<i>8</i>
Capitolo 2 Uno strumento di visualizzazione: analisi dei requisiti	9
2.1 <i>Obiettivi generali</i>	<i>9</i>
2.2 <i>Caratteristiche dello strumento di visualizzazione.....</i>	<i>10</i>
2.2.1 <i>Analisi requisiti utente</i>	<i>10</i>
2.2.2 <i>Specifica requisiti utente.....</i>	<i>11</i>
2.2.3 <i>Specifica requisiti software.....</i>	<i>14</i>
Capitolo 3 Eventi astratti, clustering ed event pattern.....	18
3.1 <i>Tecniche d'astrazione</i>	<i>18</i>
3.2 <i>Clustering.....</i>	<i>21</i>
3.2.1 <i>Rappresentare eventi astratti.....</i>	<i>21</i>
3.3 <i>Event Pattern</i>	<i>22</i>
3.4 <i>Conclusioni</i>	<i>24</i>
Capitolo 4 Architettura dello strumento integrato	25
4.1 <i>Identificazione dei moduli dello strumento.....</i>	<i>25</i>
4.2 <i>Il sottosistema ViewTraces</i>	<i>27</i>

4.2.1 Il modulo Core	28
4.2.2 Algoritmi utilizzati.....	29
Pattern Matching.....	29
4.2.3 Il modulo Gui.....	33
4.3 Il sottosistema debugger	34
4.3.1 I moduli controller e gui	34
4.4 Conclusioni	37
Capitolo 5 Utilizzo dello strumento.....	38
5.1 Installazione dello strumento.....	38
5.2 Visualizzazione delle tracce d'esecuzione	43
5.3 Funzioni dello strumento	47
5.3.1 Operazioni effettuabili sui componenti.....	48
5.3.2 Operazioni effettuabili sugli eventi.....	49
5.3.3 Operazioni effettuabili su una selezione	51
5.3.4 Operazioni effettuabili su una sessione	52
5.3.5 Funzioni extra supportate dalla modalità stand alone.....	54
File	54
5.4 Conclusioni	56
Capitolo 6 Caso d'uso dello strumento	57
6.1 Tracce di eventi Java	57
6.2 Esempio.....	59
6.2.1 AdvancedMailbox.....	60
Analisi di AdvancedMailbox ad un livello d'astrazione più alto	62
6.3 Conclusioni	65
Conclusioni	66
Bibliografia.....	68

Indice delle figure

<i>Figura 1: Confronto tra computazione concorrente e sequenziale.....</i>	<i>3</i>
<i>Figura 2: Fasi dell'analisi off-line.....</i>	<i>5</i>
<i>Figura 3: Diagramma STD.....</i>	<i>7</i>
<i>Figura 4: Communication Pattern</i>	<i>8</i>
<i>Figura 5: Use Case Programmatore Gestione Visualizzazione Informazioni</i>	<i>12</i>
<i>Figura 6: Use Case Programmatore Analisi Informazioni</i>	<i>13</i>
<i>Figura 7: Use Case Ambiente di Sviluppo.....</i>	<i>14</i>
<i>Figura 8: Evento Astratto non Convesso.....</i>	<i>20</i>
<i>Figura 9: Evento Astratto Convesso.....</i>	<i>20</i>
<i>Figura 10: Clustering.....</i>	<i>22</i>
<i>Figura 11: Pattern Connesso</i>	<i>23</i>
<i>Figura 12: Pattern non Connesso</i>	<i>23</i>
<i>Figura 13: Sottosistemi che compongono lo strumento</i>	<i>27</i>
<i>Figura 14: Moduli del sottosistema ViewTraces.....</i>	<i>27</i>
<i>Figura 15: Classi del modulo Core</i>	<i>28</i>
<i>Figura 16: Pattern visto come sequenza di stringhe comunicanti</i>	<i>30</i>
<i>Figura 17: Pattern matched</i>	<i>32</i>
<i>Figura 18: Pattern non-matched</i>	<i>32</i>
<i>Figura 19: Classi del Modulo DrawArea.....</i>	<i>33</i>
<i>Figura 20: Classi del modulo controller</i>	<i>35</i>
<i>Figura 21: File che rappresentano le tracce.....</i>	<i>36</i>
<i>Figura 22: Collegamento tra debugger e ViewTraces</i>	<i>37</i>
<i>Figura 23: Albero delle directory d'installazione</i>	<i>39</i>
<i>Figura 24: Procedura d'installazione del modulo.....</i>	<i>40</i>
<i>Figura 25: Menu build prima dell'installazione del modulo, dopo l'installazione del modulo debugger</i>	<i>41</i>
<i>Figura 26: Menu debug prima dell'installazione del modulo, dopo l'installazione del modulo debugger</i>	<i>42</i>
<i>Figura 27: Menu popup associato al file .traces.....</i>	<i>42</i>
<i>Figura 28: Finestra principale dello strumento stand alone</i>	<i>43</i>
<i>Figura 29: Visualizzazione delle tracce nello strumento stand alone.....</i>	<i>44</i>

<i>Figura 30: Esempio di segmenti selezionati.....</i>	<i>46</i>
<i>Figura 31: Visualizzazione tracce all'interno dell'IDE.....</i>	<i>47</i>
<i>Figura 32: Popup menu di un componente</i>	<i>48</i>
<i>Figura 33: Popup menu di un evento</i>	<i>49</i>
<i>Figura 34: Popup menu di una selezione</i>	<i>51</i>
<i>Figura 35: Popup menu associato ad una sessione</i>	<i>53</i>
<i>Figura 36: Barra dei menu e barra degli strumenti.....</i>	<i>54</i>
<i>Figura 37: Visualizzazione tracce di AdvancedMailbox.....</i>	<i>60</i>
<i>Figura 38: Numero di componenti</i>	<i>61</i>
<i>Figura 39: Numero totale di eventi primitivi.....</i>	<i>61</i>
<i>Figura 40: Creazione dei threads produttori e consumatori</i>	<i>62</i>
<i>Figura 41: Insieme di eventi primitivi che intendiamo astrarre</i>	<i>62</i>
<i>Figura 42: Pattern matched in AdvancedMailbox</i>	<i>63</i>
<i>Figura 43: Visualizzazione delle tracce dopo astrazione patterns.....</i>	<i>64</i>
<i>Figura 44: Numero di eventi dopo astrazione patterns.....</i>	<i>64</i>

Introduzione

Il testing ed il debugging sono due fasi del ciclo di vita di un sistema software che contribuiscono a più di metà del costo di sviluppo di un'applicazione. Il testing rappresenta il processo di esecuzione di un programma con input selezionati allo scopo di trovare errori; il debugging è il processo di individuazione e correzione degli errori scoperti durante il testing. Per il testing di programmi sequenziali sono state sviluppate varie tecniche e metodologie, oltre alla messa a punto di vari strumenti automatici; il testing di programmi concorrenti (paralleli e distribuiti) ha messo in evidenza nuovi problemi e nuove difficoltà che non possono essere affrontate semplicemente con le tecniche sviluppate per applicazioni sequenziali. Infatti le applicazioni concorrenti presentano comportamenti assai più complessi dovuti soprattutto ai meccanismi di comunicazione e sincronizzazione fra i vari processi. Le modalità con cui gli errori relativi a tali meccanismi possono innescare o meno la catena che produce un eventuale guasto, possiedono caratteristiche profondamente distinte da quelle legate agli errori tipici della programmazione sequenziale. E' noto infatti che un passaggio chiave per lo studio del comportamento di un'applicazione concorrente è la conoscenza e la comprensione della sequenza di eventi di sincronizzazione che essa può produrre (analisi statica) o che essa ha prodotto durante una particolare esecuzione (analisi dinamica).

Nel corso degli anni sono stati sviluppati numerosi strumenti che hanno cercato di dare una soluzione adeguata al problema e di alcuni di essi verranno analizzati i pregi ed i difetti così da poter realizzare un'applicazione che dia le garanzie necessarie in termini di efficacia ed efficienza. Tradizionalmente, uno dei metodi più usati per lo studio del

comportamento di un'applicazione concorrente nelle fasi di testing e debugging, consiste nel produrre tracce d'esecuzione sottoposte successivamente ad analisi. Tali tracce contengono tipicamente informazioni riguardanti stati successivi dell'applicazione, che risultino essere significativi per l'analisi in corso; nel caso di applicazioni concorrenti, queste informazioni risultano essere necessarie all'individuazione degli eventi di sincronizzazione occorsi su ciascun componente sequenziale, e delle relazioni fondamentali che sussistono tra tali eventi. La tipologia di dati raccolti nelle tracce ed il modo in cui essi verranno utilizzati, dipendono fortemente dall'architettura dell'applicazione e dei sistemi che ne supportano l'esecuzione, così come dal livello a cui si sta operando. In un modello a memoria condivisa potranno essere rilevanti accessi in memoria, eventi relativi alle cache, eventi relativi a meccanismi semaforici o procedure in mutua esclusione, mentre per sistemi a scambio di messaggi verranno tipicamente registrati eventi di *send* e *receive*, operazioni relative alle connessioni o chiamate a procedura remota. Ma ad alto livello, gli eventi a cui si potrebbe essere interessati possono assumere caratteristiche diversissime per tipologia e complessità.

Il presente lavoro, che si propone come naturale prosecuzione di tesi svolte nell'ambito di un filone di ricerca in seno alla facoltà di Ingegneria degli Studi di Pisa, rivolge l'attenzione a come le tracce d'esecuzione possono essere utilizzate per un'analisi finalizzata alla comprensione del comportamento di un'applicazione; occorre considerare che la mole di dati presenti complessivamente nelle tracce di esecuzione può essere enorme, specie nel caso in cui si stia pensando ad un'analisi di basso livello o se l'applicazione presenta un elevato grado di concorrenza. Per questo motivo, e per le relazioni potenzialmente complesse esistenti tra i dati lasciati su tracce diverse, è difficilmente pensabile di poter leggere, correlare ed interpretare i dati prodotti senza l'ausilio di strumenti automatici. Uno strumento di reale utilità dovrebbe da un lato effettuare le analisi adeguate al problema in esame e dall'altro fornire all'utente un quadro sintetico ed espressivo dei risultati ottenuti. Un mezzo potente per il raggiungimento di tale obiettivo, consiste nel presentare i dati, o i risultati di una loro elaborazione, in forma grafica. Una presentazione grafica ben progettata può infatti trasportare istantaneamente diversi contenuti informativi tramite l'utilizzo di strutture,

di diversi stili (linee, simboli, colore) e, nei casi più evoluti, del movimento. Il compito di una visualizzazione non è comunque quello di produrre una bella immagine, ma di fornire rapidamente informazioni rilevanti, che l'utente possa mettere in relazione con ciò che egli sa dell'applicazione che sta studiando, senza in ogni caso impedire o ostacolare indagini più dettagliate, su porzioni ridotte dei dati a disposizione.

Lo scopo di questo lavoro è quindi fornire le citate funzioni in accordo ai principi di semplicità di utilizzo cercando di costruire uno strumento generale, che però offra la possibilità di personalizzazioni che lo rendano più specifico. Per esemplificare la trattazione ed ottenere uno strumento realmente utilizzabile, si prenderà in considerazione il linguaggio di programmazione Java (sia come linguaggio di implementazione che come linguaggio in esame dal punto di vista del testing) non solo per le sue indubbie caratteristiche di semplicità e diffusione nella comunità degli sviluppatori, ma soprattutto perché esso fornisce in maniera nativa il supporto per il *multithreading* ed i meccanismi di comunicazione tra *processi/thread* sia a memoria comune che a scambio di messaggi. L'obiettivo ultimo della tesi è quello di realizzare uno strumento non solo "accademicamente" utile, ma soprattutto utilizzabile nel mondo reale; per raggiungere questi scopi, lo strumento si proporrà sia come applicazione funzionante in modalità *stand alone* sia come estensione di uno dei più diffusi ambienti di sviluppo integrato presenti in commercio.

Il piano di lavoro è il seguente:

- Introduzione agli approcci utilizzati per l'analisi dei dati raccolti attraverso l'utilizzo di uno strumento grafico.
- Definizione dei requisiti necessari ad uno strumento di *traces view* perché possa supportare le funzioni richieste dal testing e dal debugging di applicazioni concorrenti.
- Progettazione ed implementazione dello strumento.
- Illustrazione del manuale d'uso
- Infine si mostrerà il funzionamento dello strumento con l'ausilio di un esempio pratico.

Ringraziamenti

*Un ringraziamento particolare ad Alessio Bechini
che ha offerto ampi spunti di riflessione sul problema
(e molte soluzioni) e senza il cui aiuto il lavoro
sarebbe stato molto più arduo.*

*GRAZIE di cuore a Patrizia e Elisa
che mi hanno dato una mano nei momenti di sconforto.*

*Dedico questo lavoro ai miei genitori
che in questi anni di studio
hanno saputo aiutarmi
ogni volta che mi sono rivolto a loro.*

Capitolo 1

Tecniche di analisi di programmi concorrenti

Questo capitolo prova ad offrire una breve descrizione sulle tecniche utilizzate per lo studio e la comprensione delle computazioni concorrenti. Dopo una breve descrizione riguardante il problema della concorrenza all'interno dei programmi, s'introducono le principali tecniche di analisi soffermando principalmente la nostra attenzione sulla fase di tracing e sulle tecniche di visualizzazione delle tracce registrate.

1.1 La concorrenza

Un aspetto che non si può tralasciare riguarda il tipo di programma che si sta analizzando: è diverso testare un programma sequenziale da uno concorrente. Il debugging e il testing di programmi sequenziali è stata un'area di ricerca attiva per lungo tempo che ha portato alla definizione di diverse teorie. Queste teorie sono fortemente connesse alle caratteristiche dei programmi sequenziali che non sono presenti nel software concorrente. Le tecniche di analisi per i programmi sequenziali si basano sull'ipotesi che, per ogni programma, ad ogni dato in ingresso corrisponde uno ed un solo output. Questo è dovuto al fatto che ogni istruzione è eseguita sequenzialmente e i cammini sono selezionati deterministicamente sulla base del valore assunto dai dati del programma. Tale ipotesi non è valida nei programmi concorrenti, dove l'esecuzione non deterministica e parallela di segmenti di codice implica che si

possono avere diversi output corretti ottenuti con lo stesso input. Ripetute esecuzioni di un programma sequenziale P con un dato input t ($P(t)$) seguiranno sempre lo stesso percorso di esecuzione producendo lo stesso output o . Se P contiene un errore f e $P(t)$ esegue f terminando in fallimento, ogni esecuzione di P fallirà. Questo concetto può essere generalizzato dicendo che: dato un programma P , se eseguendo P con tutti gli elementi di un insieme di test T si ottiene un insieme di fallimenti F , ogni volta che viene eseguito P con ogni elemento di T si otterrà lo stesso insieme di fallimenti. Queste congetture sono fondamentali per il test di programmi sequenziali che si basa sulla possibilità di riprodurre i fallimenti osservati. Le cose cambiano quando si analizza un programma concorrente, perché non è garantito che diverse esecuzioni del programma dato lo stesso input seguano lo stesso percorso di esecuzione dato che il controllo del flusso durante la sua esecuzione è determinato non solo dal valore delle variabili in un preciso punto, ma anche dai fattori esterni che non sono sotto il controllo dell'utente o del programmatore. Questo comportamento implica che, se l'esecuzione di $P(t)$ termina in un fallimento, una seconda esecuzione di $P(t)$ potrebbe anche non produrre lo stesso fallimento. Per valutare il risultato di un'esecuzione, non è sufficiente conoscere la coppia $\langle input, output \rangle$, ma occorre anche la sequenza di istruzioni seguita durante l'esecuzione. Quindi, la correttezza non è data da una relazione funzionale tra input e output, ma dal comportamento del programma. Un comportamento è denotato da un input, un output e una sequenza di esecuzione (vedi Figura 1).

La concorrenza è basata sia sulle proprietà dei linguaggi di programmazione che offrono un supporto diretto come Java [1] che usando il supporto di librerie ad hoc, come i P-threads [4].

	Sequenziale	Concorrente
Correttezza	Output Atteso	Output Atteso + Sincronizzazioni attese
Testing	Esplorazione spazio di input	Esplorazione spazio di input
Misura	Output	Output Atteso + Sincronizzazioni avvenute
Comportamento	Deterministico	Non Deterministico

Figura 1: Confronto tra computazione concorrente e sequenziale

1.2 Comprendere il comportamento dei programmi concorrenti

La *comprensione di un programma* consiste nel processo di estrazione di informazioni sul disegno concettuale dal codice sorgente. Questo processo è oggi molto importante, perché non solo permette di ridurre il costo di modifica e manutenzione delle applicazioni in commercio, ma anche perché consente di costruire dei legami tra i vecchi sistemi software e le successive versioni.

Le persone che di solito si trovano di fronte alla necessità di analizzare il comportamento di un programma hanno due alternative: studiare il codice sorgente (**analisi statica**) o far girare l'applicazione per osservare quello che fa (**analisi dinamica**). La prima tecnica è anche conosciuta come *reverse engineering* e attraverso essa si cerca di ottenere una descrizione ad alto livello del programma partendo dal

codice sorgente. Purtroppo le informazioni ottenibili mediante un'analisi statica non sono in grado di spiegare del tutto il comportamento di una computazione concorrente, perché questo dipende, come già detto in precedenza, dalla triade *<input, output, Syn_sequence>*, dove per *Syn_sequence* s'intende la sequenza di eventi di sincronizzazione tra processi seguita dal programma durante una sua esecuzione. Proprio per questo motivo, per avere una piena comprensione delle applicazioni distribuite e di programmi che fanno uso di un elevato grado di parallelismo, si ricorre alla sopra citata analisi dinamica, che consiste nel registrare gli eventi di sincronizzazioni tra processi che permettono di estrarre il maggior numero d'informazioni per lo studio in corso. Tutte queste informazioni vengono memorizzate in file di log chiamati "tracce"; questa fase di registrazione prende il nome di fase di *tracing*.

Tali tracce possono essere prodotte sfruttando meccanismi eventualmente messi a disposizione dall'ambiente di esecuzione, che consentono tipicamente di intercettare eventi di basso livello (eventi hardware, di sistema operativo, di scambio dati su rete, ecc.), oppure instrumentando opportunamente l'applicazione in modo che essa stessa lasci traccia degli eventi d'interesse. Quest'ultima metodologia è chiaramente indirizzata ad eventi non intercettabili dal supporto a tempo di esecuzione o ad eventi di un certo livello d'astrazione, relativi, cioè, a stati visibili esclusivamente dall'interno dell'applicazione. L'instrumentazione offre evidentemente una grande flessibilità di controllo sui tipi di eventi da registrare e sulle informazioni ad essi associate, in vista del tipo di analisi da effettuare sulle tracce.

1.3 Analisi delle tracce

Dopo aver raccolto tutte le informazioni del caso durante una esecuzione, è necessario analizzarle per controllare se si sono verificati funzionamenti anomali [15]. Per questa fase ci sono due approcci:

- *Run-time-analysis* (on-line analysis). In questo caso l'analisi viene effettuata mentre il programma è in esecuzione. È questo il caso degli algoritmi per la rilevazione delle condizioni di deadlock, o condizioni di "data-race".

- *Post-mortem-analysis* (off-line analysis). Tutte le operazioni sulle tracce vengono effettuate quando l'applicazione è terminata, con l'ausilio di strumenti appositi per l'ispezione delle tracce.

In questo lavoro siamo più interessati ad un'analisi off-line dei dati raccolti; però occorre considerare che la mole di informazioni raccolte durante la fase di *tracing* può essere enorme, specie nel caso in cui si stia pensando ad un'analisi a basso livello o se l'applicazione presenta un elevato grado di concorrenza. Per questo motivo, e per le relazioni potenzialmente complesse esistenti tra i dati lasciati su tracce diverse, è difficilmente pensabile di poter leggere, correlare ed interpretare i dati prodotti senza l'ausilio di strumenti automatici.

1.4 Visualizzazione delle tracce

Un mezzo potente per poter fornire all'utente un quadro sintetico dei risultati ottenuti consiste sicuramente nel presentare i dati sotto forma grafica. Una presentazione grafica ben progettata può infatti trasportare istantaneamente diversi contenuti informativi tramite l'utilizzo di strutture, di diversi stili e, nei casi più evoluti, del movimento. Lo scopo di una visualizzazione non è comunque quello di produrre una bella immagine, ma di fornire rapidamente informazioni rilevanti, che l'utente possa mettere in relazione con ciò che egli sa dell'applicazione che sta studiando, senza in ogni caso impedire o ostacolare indagini più dettagliate, su porzioni ridotte dei dati a disposizione.

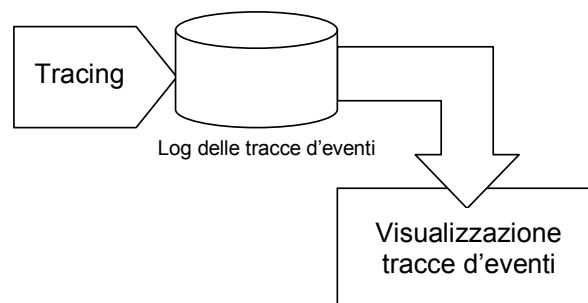


Figura 2: Fasi dell'analisi off-line

1.4.1 I diagrammi spazio-tempo (STD)

I diagrammi spazio tempo sono probabilmente il tipo di visualizzazione più usato per illustrare le relazioni di precedenza che sussistono tra gli eventi di una computazione concorrente. Nel nostro caso non siamo interessati ad informazioni sulla collocazione temporale (tempo fisico) di ciascun evento, per cui faremo uso di diagrammi spazio tempo discreti, definiti STD. Si può pensare ad un STD come ad una matrice grafica, o una griglia, su cui giacciono i *simboli* rappresentanti gli eventi d'interesse. Una riga corrisponde ad un *componente*, dove per componente, nella nostra esposizione, intendiamo sia i processi/thread che le entità dedicate alla gestione delle risorse condivise (semafori, monitor). Una colonna corrisponderà invece ad una posizione disponibile per un evento (su ciascuna riga). I punti seguenti completano la caratterizzazione di un STD:

- Dati due eventi e_1 ed e_2 tali che $e_1 \rightarrow e_2$, dove con questa notazione s'intende che l'evento e_1 precede casualmente l'evento e_2 secondo quando definito da Lamport [13], la posizione del simbolo relativo al primo evento sarà minore di quella del simbolo relativo al secondo evento.
- Due eventi appartenenti ad uno stesso insieme sincrono hanno la stessa posizione.
- Per rappresentare le comunicazioni sincrone tra gli eventi si utilizzano dei segmenti verticali, mentre per quelle asincrone si utilizzano dei segmenti obliqui.

Da quanto detto dovrebbe risultare evidente la differenza tra un STD e un diagramma cartesiano. Spostandosi da una linea ad un'altra ci si sposta nello "spazio" dell'applicazione, che ovviamente non corrisponde ad uno spazio fisico, o comunque non ne vuole essere la rappresentazione. Ma differenze ancor più profonde riguardano lo spostamento nel "tempo". Questo perché un STD vuole essere una rappresentazione di un ordinamento parziale, mentre il tempo fisico implica un ordinamento totale degli eventi. Scorrendo lungo una linea relativa ad un componente ci si sposta in un "tempo" non legato ad una misurazione fisica, ma determinato esclusivamente dall'ordinamento degli eventi su tale componente (Figura 3).

Vogliamo inoltre evidenziare che in generale, per quanto detto da Lamport sul tempo logico, non è vero che se un simbolo ha una posizione minore di un altro, allora sussiste una relazione di precedenza tra gli eventi corrispondenti.

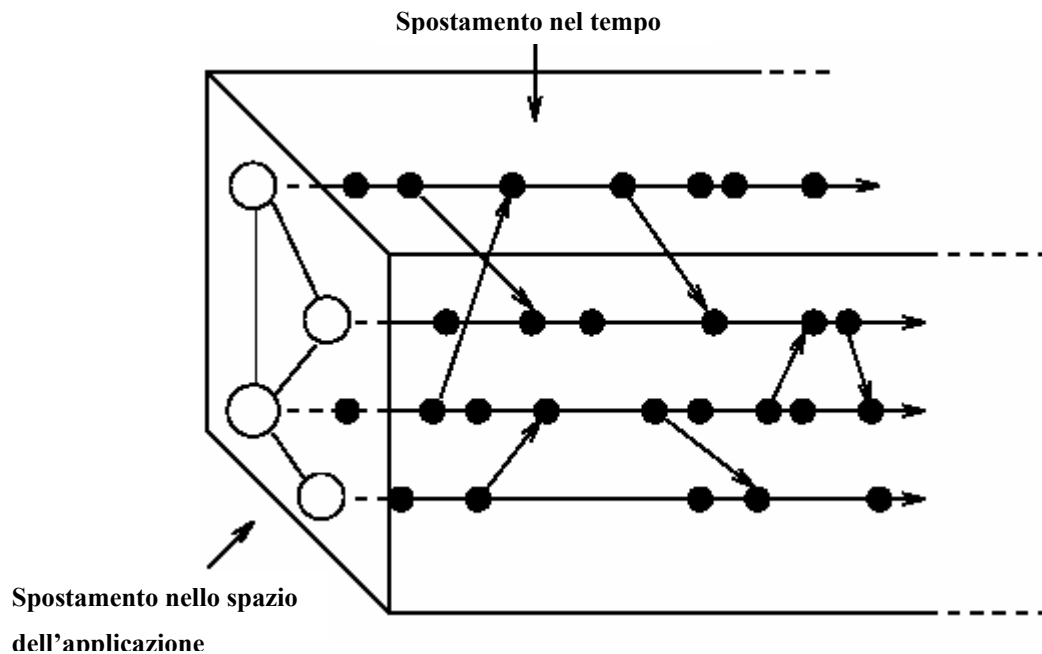


Figura 3: Diagramma STD

1.5 Tecniche avanzate d'analisi

Le caratteristiche discusse in questo paragrafo integrano le tecniche d'analisi descritte precedentemente. Infatti molto spesso, il numero di processi/thread che compongono un'applicazione avente un elevato grado di concorrenza, raggiunge la centinaia o addirittura la migliaia, complicando notevolmente la visualizzazione dell'STD. Proprio per questo al fine di ottenere una migliore comprensione dell'applicazione, è utile introdurre tecniche che permettano l'astrazione di eventi che formano “un'unità logica di lavoro” in un unico evento. Tale tecnica viene utilizzata anche per permettere al programmatore di analizzare le sincronizzazioni che avvengono ad un livello più alto. Nel prosieguo della tesi ci riferiremo a queste unità di lavoro con il termine “communication patterns” [20]. Noi cercheremo questi communication patterns all'interno delle tracce d'esecuzione e astrarremo gli eventi che formano un pattern.

Tale problema di ricerca viene di solito identificato con il termine *pattern matching*; la maggior parte delle tecniche di pattern matching si concentra su programmi sequenziali andando ad agire principalmente sul codice sorgente. Noi invece utilizzeremo un algoritmo di pattern matching che va a ricercare i pattern direttamente all'interno delle tracce d'esecuzioni concorrenti prodotte durante l'analisi dinamica. Prima di poter astrarre il pattern trovato in un unico evento devono essere rispettati un certo numero di vincoli che verranno introdotti in seguito quando daremo una descrizione più dettagliata dell'algoritmo. Un esempio di communication pattern più volte ripetuto all'interno delle tracce è mostrato in Figura 4.

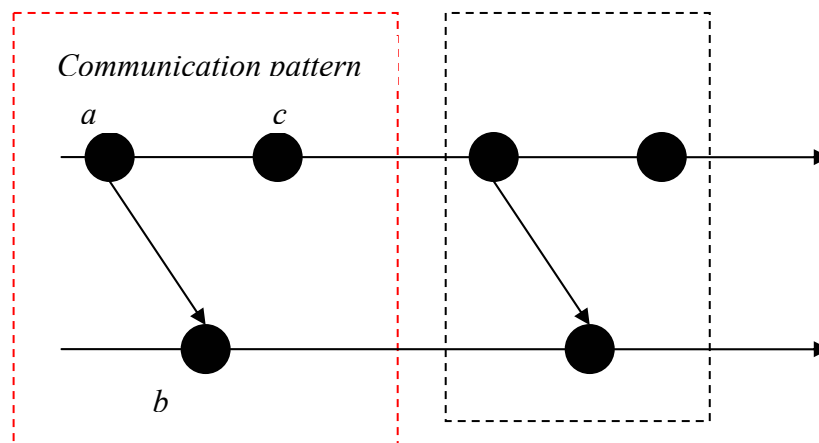


Figura 4: Communication Pattern

1.6 Conclusioni

Sinora si è introdotto in termini generali il problema dell'analisi di applicazioni concorrenti. Nei capitoli successivi vengono osservati i requisiti che deve avere uno strumento di software che permetta la visualizzazione delle tracce ed sviluppato il progetto dello strumento. Dello strumento di visualizzazione se ne fornisce inoltre l'implementazione stand alone e come modulo aggiuntivo di un ambiente di programmazione integrato.

Capitolo 2

Uno strumento di visualizzazione: analisi dei requisiti

Dopo aver trattato in generale il problema dell'analisi di applicazioni concorrenti, in questo capitolo viene affrontata la questione della definizione delle caratteristiche di uno strumento grafico che permetta una gestione automatizzata delle tracce di eventi di esecuzioni concorrenti prodotte durante la fase di tracing.

2.1 Obiettivi generali

Abbiamo visto nel capitolo precedente che per riuscire a comprendere ed analizzare a pieno un programma concorrente è necessario:

- Raccogliere la sequenza di sincronizzazioni (*Syn_sequence*) tra processi accadute durante le varie esecuzioni (fase di tracing)
- Rappresentare le sequenze ottenute al fine di valutarne la validità attraverso un opportuno strumento grafico di gestione di tracce d'esecuzione.

In precedenti lavori di tesi svolti presso il nostro dipartimento è stato sviluppato uno strumento in grado di instrumentare opportunamente un'applicazione Java, in modo da ottenere delle tracce di esecuzione contenenti gli eventi relativi ad alcuni tra i più importanti costrutti di sincronizzazione e comunicazione presenti in tale linguaggio.

Nel seguito si farà riferimento all'analisi di applicazioni Java concorrenti ma la soluzione proposta sarà del tutto generica e potrà essere adattata a qualunque altro linguaggio di programmazione.

Per cui l'idea è stata quella di progettare innanzitutto una piattaforma generale, avente come dati d'ingresso tali tracce, che consentisse di visualizzare, con un piccolo sforzo, i diagrammi spazio-tempo (descritti in dettaglio nel precedente capitolo) relativi ad una qualsiasi computazione concorrente, di cui fossero disponibili le informazioni sufficienti a ricostruire l'ordinamento parziale degli eventi. Dopo di che, a partire da tali diagrammi, consentire alcune tipologie di analisi del tutto generali, riguardanti le relazioni di causalità esistenti tra diverse regioni di una computazione.

2.2 Caratteristiche dello strumento di visualizzazione

Avendo a disposizione i file di *log* relativi alle tracce di esecuzioni concorrenti, l'unica operazione necessaria alla loro visualizzazione è l'indicazione della cartella del filesystem in cui risiedono le tali file. Si è scelto di rendere disponibili le funzionalità offerte dallo strumento sia attraverso un'interfaccia grafica funzionante in modalità *stand alone* sia come add-in di un ambiente di sviluppo integrato in modo tale che possa sfruttarne le funzionalità già disponibili.

2.2.1 Analisi requisiti utente

Dopo questa breve introduzione al problema cerchiamo di evidenziare quali sono i casi relativi all'uso dell'interfaccia grafica. I diagrammi UML mostrati sono stati creati con Microsoft Visio Version 2002. Si assumono per dati i casi d'uso associati all'installazione dello strumento.

Gli utenti principali del nostro software sono sicuramente il *programmatore* e l'*ambiente di sviluppo* integrato a cui viene agganciato lo strumento. Quest'ultimo si occupa della gestione delle finestre grafiche e fornisce i servizi base per il l'editing ed il debugging ed è da considerarsi a tutti gli effetti un utente in quanto aggancia i servizi offerti dallo strumento e li utilizza. Vediamo quali sono i casi d'uso di questa gerarchia di utenti.

Il *programmatore* può:

1. Visualizzare il maggior numero d'informazioni circa la computazione concorrente da studiare.
2. Configurare lo strumento.
3. Effettuare diverse tipologie di analisi della computazione concorrente visualizzata.

Inoltre i casi d'uso associati con l'*ambiente di sviluppo* sono:

4. Anche quest'ultimo deve configurare lo strumento secondo le sue esigenze e modificarne lo stato.

2.2.2 Specifica requisiti utente

In questo paragrafo specificheremo con un dettaglio maggiore i requisiti degli utenti del nostro sistema software indicando ciò di cui hanno bisogno per avere una comprensione della computazione concorrente che si sta studiando:

- 1.1 Il programmatore deve poter visualizzare la storia completa dell'esecuzione di un'applicazione.
 - 1.2 Il programmatore deve poter visualizzare i componenti dell'applicazione analizzata in modo facile e intuitivo.
 - 1.3 Il programmatore deve poter visualizzare gli eventi che compongono i vari componenti.
 - 1.4 Il programmatore deve poter visualizzare le relazioni di causalità che legano questi eventi.
 - 1.5 Il programmatore deve poter modificare la visualizzazione corrente in modo da renderla più comprensibile.
-
- 2.1 Il programmatore deve poter impostare in maniera facile e intuitiva la configurazione grafica dell'interfaccia scegliendo tra diversi stili messaggi a disposizione.

- 3.1 Il programmatore deve essere in grado di evidenziare il passato causale degli eventi selezionati.
- 3.2 Il programmatore deve essere in grado di evidenziare il futuro causale degli eventi selezionati.

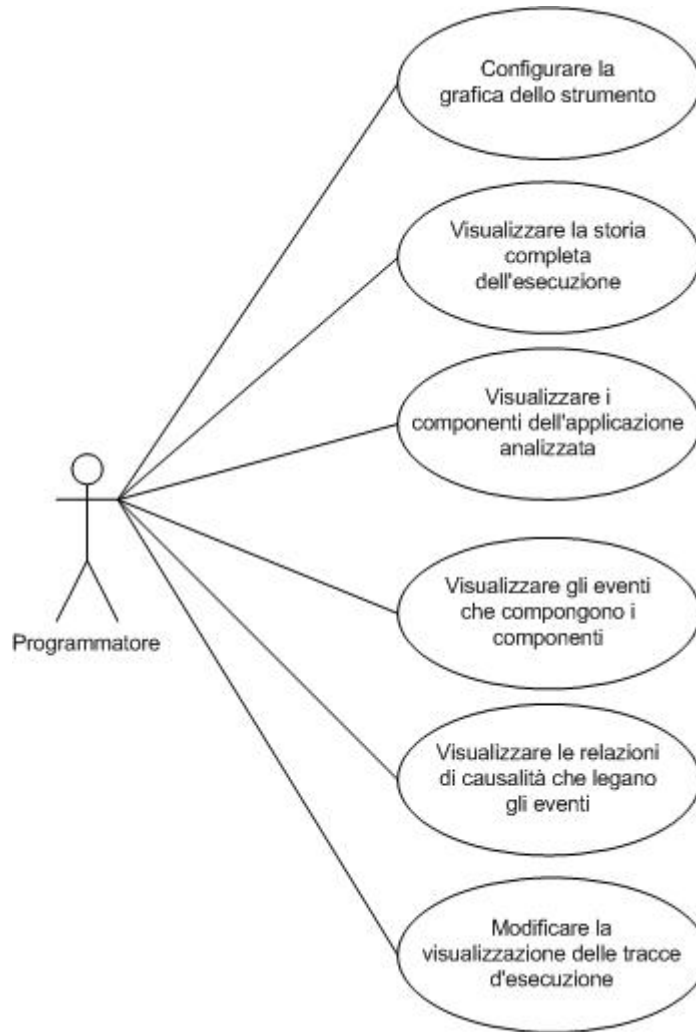


Figura 5: Use Case Programmatore Gestione Visualizzazione Informazioni

- 3.3 Il programmatore deve avere la possibilità di risalire al codice sorgente dell'evento selezionato.
- 3.4 Il programmatore deve poter inoltre etichettare gli eventi di suo interesse per poterli ritrovare più facilmente durante il processo di analisi.
- 3.5 Il programmatore deve poter conoscere informazioni più dettagliate circa gli eventi visualizzati.

- 3.6 Il programmatore deve avere la possibilità di astrarre un gruppo di eventi primitivi in un unico evento.
- 3.7 Il programmatore deve aver la possibilità di individuare dei *communication patterns* all'interno della visualizzazione.

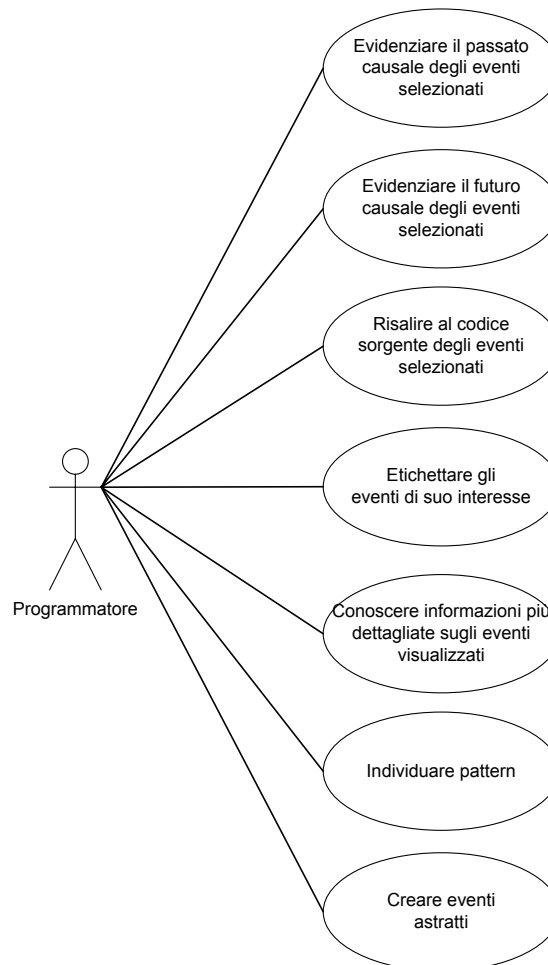


Figura 6: Use Case Programmatore Analisi Informazioni

- 4.1 Con questo caso d'uso s'intende che l'ambiente di sviluppo deve poter registrare i servizi offerti dallo strumento.
- 4.2 L'ambiente di sviluppo deve inoltre avere la possibilità di leggere e impostare lo stato dello strumento.

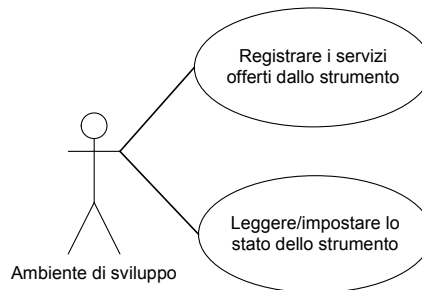


Figura 7: Use Case Ambiente di Sviluppo

2.2.3 Specifica requisiti software

Per il momento non vengono trattate le specifiche software dell'ambiente di sviluppo perché le modalità con cui i casi d'uso illustrati sopra sono sviluppati dipendono dal particolare ambiente scelto.

- 1.1.1 Il nostro strumento utilizza, per visualizzare le tracce di esecuzione, un diagramma STD discreto come già detto più volte in precedenza.
- 1.1.2 La vista delle tracce può essere più larga della finestra dello strumento, per cui si devono utilizzare delle scrollbar per mostrare la storia completa dell'esecuzione dell'applicazione.
- 1.1.3 Si possono utilizzare anche ulteriori scrollbar per comprimere o espandere il diagramma in modo da avere una vista scalata di esso.
- 1.1.4 Il nostro strumento offre anche la possibilità di stampare l'STD corrente.

- 1.2.1 I componenti della computazione sono rappresentati attraverso delle linee parallele e orizzontali.
- 1.2.2 Tali linee hanno colori diversi a seconda del tipo di componente.
- 1.2.3 Ad ogni componente è associato un nome che deve essere ben evidente all'estremità delle linee parallele.
- 1.2.4 Questo nome può essere cambiato a seconda delle esigenze del programmatore.

- 1.3.1 Ogni evento di sincronizzazione è rappresentato da un simbolo disposto sulla linea relativa al componente in cui è avvenuto.
- 1.3.2 I simboli sono di varie forme a seconda del tipo di evento.
- 1.3.3 I simboli sono disposti sulla linea corrispondente a ciascun componente, in modo consistente con l'ordinamento locale degli eventi.

- 1.4.1 Nel nostro strumento si considera che il tempo scorra da sinistra verso destra per cui il posizionamento degli eventi sui componenti ne indica la relazione temporale.
- 1.4.2 Gli eventi legati da una relazione di causalità, appartenenti a componenti diversi, sono uniti attraverso una freccia obliqua e chiaramente la posizione dell'evento attivo sarà minore di quella dell'evento passivo.
- 1.4.3 Gli eventi appartenenti ad un insieme sincrono occupano la medesima posizione e sono collegati tramite un segmento perpendicolare alla linea dei componenti.

- 1.5.1 Il nostro strumento deve porsi il problema di come ordinare i componenti, perché un ordinamento casuale può portare a visualizzare diagrammi inutilmente complicati. Infatti la confusione grafica deriva dal fatto che i segmenti che indicano gli insiemi sincroni e le relazioni asincrone, attraversano il diagramma in senso latitudinale e obliquo, intersecando potenzialmente linee relative a componenti che nulla hanno a che vedere con le relazioni rappresentate. Tali intersezioni vengono chiamati conflitti, per cui il software che stiamo sviluppando deve prevedere degli algoritmi di visualizzazione che minimizzino tali conflitti.
- 1.5.2 Se il diagramma così visualizzato non ci soddisfa ancora perché sono presenti sempre alcuni conflitti grafici, lo strumento offre la possibilità di scambiare la posizione dei componenti con la precedente o la successiva, rispettivamente, aggiustando automaticamente la posizione dei simboli.
- 1.5.3 Vi è inoltre la possibilità di spostare di una cella grafica verso destra i vari eventi e tutto il loro futuro causale; questa opzione può essere utile se non si è

soddisfatti dell'attuale disposizione dei simboli. Da notare che non esiste l'operazione inversa perché sarebbe assai complicata.

- 2.1.1 Attraverso un opportuno menù di opzioni si devono poter scegliere i simboli da associare con gli eventi.
- 2.1.2 Deve essere inoltre possibile cambiare i colori delle linee rappresentanti i componenti dell'applicazione sotto esame e i colori dei segmenti che uniscono i vari eventi.

- 3.1.1 Il programmatore deve selezionare l'evento di cui vuole conoscere il passato causale e fare un click sull'apposito tasto.
- 3.1.2 Dopodiché lo strumento evidenzierà in maniera opportuna gli eventi richiesti.

- 3.2.1 Il programmatore deve selezionare l'evento di cui vuole conoscere il futuro causale e fare un click sull'apposito tasto.
- 3.2.2 Dopodiché lo strumento evidenzierà in maniera opportuna gli eventi richiesti.

- 3.3.1 Il programmatore deve selezionare l'evento d'interesse e clickare sull'apposito tasto.
- 3.3.2 Lo strumento a questo punto, conoscendo il nome del file e il numero della linea associati all'evento, è in grado di aprire una finestra con il file caricato e la linea corretta selezionata.

- 3.4.1 Il programmatore deve selezionare l'evento d'interesse e clickare sull'apposito tasto.
- 3.4.2 Apparirà così una finestra di dialogo in cui è possibile associare un nome a tale evento.
- 3.4.3 Una volta assegnato, questo nome sarà registrato all'interno di una lista.
- 3.4.4 Dopodiché, in qualsiasi momento, selezionando tale nome dalla lista, lo strumento visualizzerà la regione dell'STD contenente tale evento, che apparirà selezionato.

- 3.5.1 Ogni qualvolta il programmatore passa con il mouse sopra un evento della visualizzazione il software utilizza una status bar per mostrare informazioni il cui significato è noto all'utente.

- 3.6.1 Il programmatore dapprima seleziona la regione che intende astrarre.
- 3.6.2 Se questa regione osserva certi requisiti, che verranno descritti nel prosieguo del lavoro, allora lo strumento crea una nuova visualizzazione contenente l'evento astratto.

- 3.7.1 In un primo momento l'utente programmatore seleziona il *communication pattern* di suo interesse.
- 3.7.2 Anche in questo caso lo strumento cerca tale pattern all'interno delle tracce di eventi solo se la selezione rispetta certi vincoli di cui si parlerà nei prossimi paragrafi; se la ricerca da esito positivo il programmatore può decidere se astrarre tutti i pattern trovati in maniera automatica così da ridurre notevolmente gli elementi visualizzati oppure astrarli manualmente uno ad uno.

Nei capitoli successivi si daranno delle basi teoriche per poter effettuare le operazioni di astrazione e pattern matching sopra introdotte, verrà descritta l'architettura software dello strumento grafico, definite alcune caratteristiche dell'implementazione e spiegato come utilizzarne le funzioni attraverso un esempio.

Capitolo 3

Eventi astratti, clustering ed event pattern

Il principale obiettivo di questo capitolo è quello di fornire le basi teoriche e di evidenziare le implicazioni pratiche per lo sviluppo di meccanismi automatici di astrazione. Dapprima definiremo le proprietà degli eventi astratti, poi introdurremo il concetto di insieme convesso e clustering, ed infine sfrutteremo le nozioni suddette per mettere in luce le caratteristiche principali della ricerca degli event pattern.

3.1 Tecniche d'astrazione

Nei paragrafi precedenti è emerso più volte il concetto di astrazione: con ciò noi intendiamo il raggruppamento di più eventi primitivi in un evento di alto livello di cui sia nascosta la struttura interna, creando quindi una visualizzazione astratta della computazione concorrente. Quindi si può avere interesse a prendere in considerazione le relazioni che legano gruppi di eventi non necessariamente mappabili in singoli costrutti del linguaggio sorgente. Gli eventi in questione potrebbero anche appartenere a più di un componente: si pensi, ad esempio, ad un insieme di eventi appartenenti ai sotto-processi istanziati su un certo nodo di un sistema distribuito.

Questo processo d'astrazione non deve introdurre precedenze spurie o nascoste, ma, idealmente, la relazione di precedenza sugli eventi astratti deve mantenere le stesse caratteristiche di quella sugli eventi primitivi; cioè l'ordinamento parziale deve essere

ancora rispettato. Comunque questo argomento risulta essere piuttosto complesso, e dipende fortemente dalla definizione di precedenza tra eventi astratti; ci sono due modi ovvi per definire tale relazione [21]:

- Si dice che un evento astratto A precede un evento astratto B se ogni evento primitivo di A precede ogni evento primitivo di B .
- Si dice che un evento astratto A precede un evento astratto B se almeno un evento primitivo di A precede un evento primitivo di B .

La prima definizione pone alcuni problemi, perché, nonostante la sua intuitività, da essa si deduce che un evento primitivo che preceda solo alcuni elementi di un evento astratto, risulta concorrente con esso. Ciò va contro l'idea non formalizzata, ma intrinseca nel concetto informale di concorrenza, che in una computazione due eventi concorrenti possano avvenire in qualsiasi ordine. Per cui nel prosieguo del lavoro adotteremo la seconda definizione come nozione di precedenza tra eventi astratti.

Noi tratteremo solo una classe di eventi astratti, gli *eventi convessi*, per i quali è mantenuto un ordinamento parziale con gli eventi primitivi. In seguito parleremo indifferentemente di evento o insieme convesso. Questo insieme di eventi è stato analizzato approfonditamente in precedenti lavori di tesi [32], per cui noi ci limitiamo a riportarne la definizione:

Definizione (Evento convesso): un insieme A di eventi è *convesso* se e soltanto se

$$\forall a, b \in A: a \rightarrow c \rightarrow b \Rightarrow c \in A.$$

La convessità è un requisito importante per gli eventi astratti perché permette di considerare quest'ultimi, al degli eventi primitivi, come delle unità di esecuzione. Ossia un evento convesso, come uno primitivo, rappresenta un insieme di operazioni che possono essere eseguite senza alcuna attesa di eventi di altri componenti, una volta, ovviamente, che siano state eseguite le operazioni rappresentate dagli eventi che lo precedono casualmente.

Consideriamo la figura 8: gli eventi a e b appartengono ad un evento astratto candidato A . Possiamo osservare che l'evento A , in accordo a quanto detto nella precedente definizione, non rispetta i requisiti di convessità in quanto l'evento a precede c che a sua volta precede b ; lo stesso vale anche per l'evento d , quindi per poter considerare A come un evento astratto convesso questo avrebbe dovuto contenere sia c che d (Figura 9).

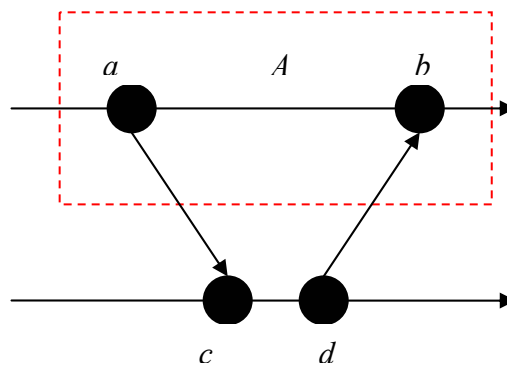


Figura 8: Evento Astratto non Convesso

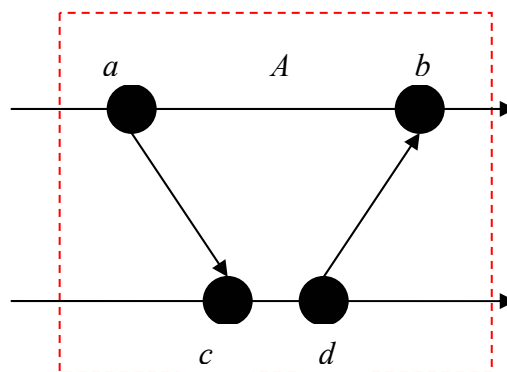


Figura 9: Evento Astratto Convesso

3.2 Clustering

Abbiamo detto che la funzione degli eventi astratti vuole essere quella di costruire delle rappresentazioni di alto livello di una computazione concorrente. Tale operazione consiste nel valutare le relazioni che intercorrono tra insiemi di eventi ma piuttosto nel rappresentare tali insiemi come eventi singoli.

Nelle tesi precedenti [32] è stata formalizzata anche una procedura di astrazione di tali insiemi convessi, definendo il processo attraverso il quale sia possibile costruirne una rappresentazione astratta di livello qualsiasi. Tale procedura, che permette la costruzione di una nuova rappresentazione grafica della computazione concorrente sotto analisi creando un unico evento astratto a partire da più eventi primitivi, è stata definita *clustering*.

Il presente lavoro non intende fornire di nuovo le basi teoriche per poter effettuare l'operazione di *clustering*, ma si limita a riportarne le caratteristiche principali per poterne sfruttare le potenzialità secondo quanto descritto nei precedenti paragrafi.

3.2.1 Rappresentare eventi astratti

Per poter ottenere una rappresentazione comprensibile degli eventi astratti devono essere rispettati i seguenti principi:

- La loro visualizzazione deve rispettare i vincoli di precedenza.
- Si deve minimizzare lo spazio occupato da tali eventi.
- Infine devono essere indicati i componenti aventi eventi appartenenti all'evento astratto.

Detto questo, dato un insieme convesso A , si definisce *clustering* di tale insieme, la costruzione di una nuova rappresentazione della computazione caratterizzata dalle seguenti proprietà:

- Ogni evento non appartenente ad A rimane immutato ivi incluse le relazioni di precedenza diretta tra di essi.

- A viene rappresentato attraverso un insieme sincrono S avente eventi sincroni che appartengono agli stessi componenti degli eventi dell'insieme convesso.

Per spiegare meglio il tutto diamo una rappresentazione grafica (Figura 10) dell'operazione di *clustering* effettuata sull'evento astratto mostrato in Figura 9.

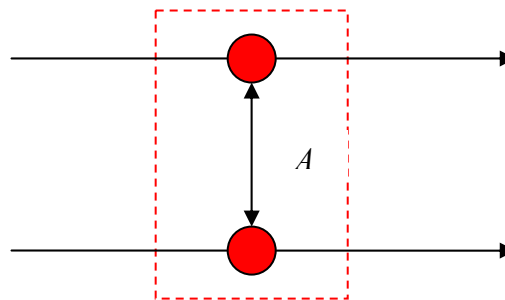


Figura 10: Clustering

Un insieme sincrono è l'ideale per identificare un evento astratto perché è visto come un evento che occorre nello stesso istante su più componenti.

3.3 Event Pattern

Una particolare tecnica d'astrazione è, come già detto nel Capitolo 1, quella che ci permette la ricerca all'interno delle tracce e l'astrazione dei *communication patterns*. Per quanto riguarda quest'ultimo punto le nozioni fondamentali sono già state introdotte nel precedente paragrafo, invece per l'operazione di ricerca, prima di descrivere l'algoritmo di pattern matching nel dettaglio nel prossimo capitolo, è utile introdurre il concetto di *pattern connesso*.

Secondo quanto definito Michiel F.H. Seuren [20], un communication pattern può definirsi *connesso* se tutti i componenti che lo compongono sono raggiungibili da ogni altro componente del pattern (Figura 11). Si può osservare che un pattern non connesso può essere formato da due o più pattern connessi non legati da relazioni di causalità

(Figura 12). Da notare che i componenti 1 e 2 della Figura 12 formano un pattern connesso così come i componenti 2 e 3.

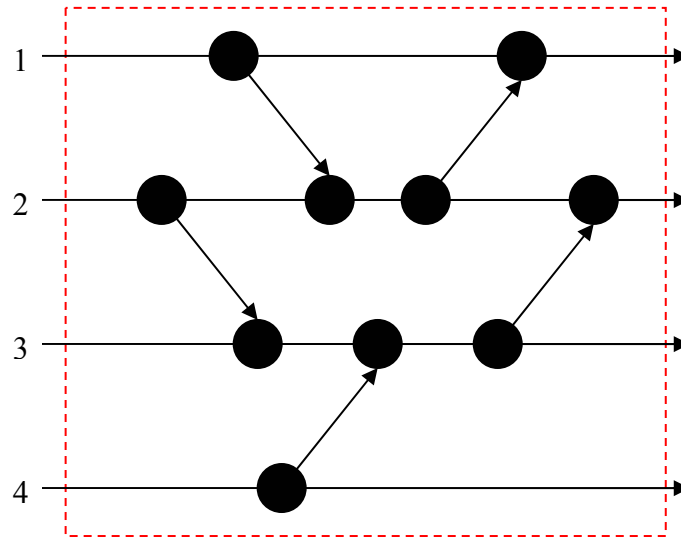


Figura 11: Pattern Connesso

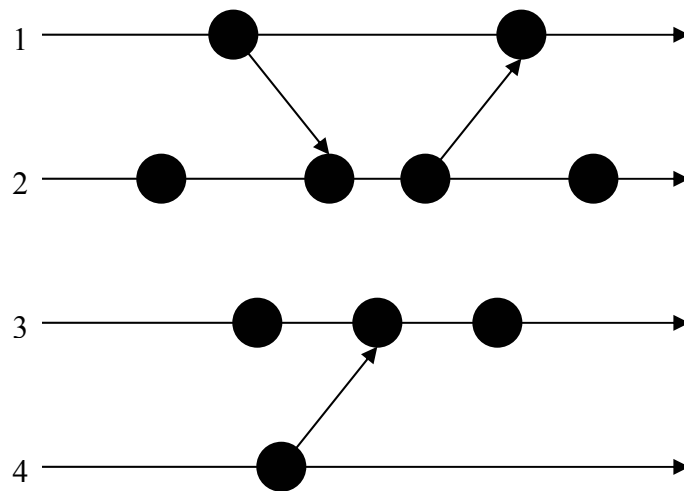


Figura 12: Pattern non Connesso

Abbiamo introdotto tutte queste nozioni perché il nostro strumento ha come requisito fondamentale il fatto che può ricercare e quindi astrarre esclusivamente communication pattern *connessi* e *convessi*.

3.4 Conclusioni

Questo capitolo si è concentrato sull'analisi delle tecniche utilizzate per l'astrazione di eventi primitivi. Nella prima parte si sono discusse le argomentazioni teoriche che stanno alla base di tali metodologie, definendo la relazione di precedenza per gli eventi astratti e introducendo la nozione di insieme convesso. Nell'ultima parte abbiamo definito il concetto di *clustering* ed è stata discussa una possibile rappresentazione per gli eventi astratti. Infine si sono visti a quali requisiti deve sottostare l'algoritmo di *pattern matching* che verrà discusso nel prossimo capitolo.

Capitolo 4

Architettura dello strumento integrato

Nei capitoli precedenti si sono illustrate le caratteristiche di uno strumento grafico per la gestione di tracce di esecuzioni concorrenti; in questo capitolo viene sviluppato il progetto concettuale dello strumento in base alle specifiche introdotte. Dapprima si illustrano le scelte architettoniche motivandole in base alle specifiche del sistema; quindi si illustrano i moduli che compongono i sottosistemi.

4.1 Identificazione dei moduli dello strumento

Per quanto esposto nell'analisi dei requisiti (Capitolo 2), il nostro strumento grafico deve avere sia un'interfaccia grafica propria che gli permetta il funzionamento come applicazione a se stante (*stand alone*), sia dei metodi di aggancio ad un opportuno ambiente di sviluppo integrato: nel nostro lavoro si è scelto di utilizzare l'ambiente Net Beans. Tale ambiente è al tempo stesso un IDE (Integrated Development Environment) ed una piattaforma di sviluppo (normalmente riferendosi alla piattaforma si suole chiamarla OpenAPI). La piattaforma è stata inizialmente progettata come un insieme di classi di servizio per transazioni basate su protocolli di rete, poi si è evoluta fornendo un ambiente aperto per la progettazione di interfacce legate al mondo della programmazione. In virtù del fatto che le API sono sviluppate con filosofia open source hanno riscosso molto successo e si è assistito al crescere del numero di IDE basate sulle OpenAPI. Una tra le più conosciute è l'ambiente SunOne Studio (ex Forte for Java) oltre naturalmente all'evoluzione dello specifico NetBeans IDE. Negli ultimi tempi

L'evoluzione delle API sta puntando ad aprirsi verso ambiti non strettamente legati alla produzione di software.

I vantaggi di questa piattaforma sono legati all'estrema potenza offerta dalle API fornite ed all'immensa disponibilità di moduli di espansione non solo per lo sviluppo, ma anche per l'analisi ed il testing di applicazioni scritte nei più diversi linguaggi di programmazione. La piattaforma e l'IDE sono scritti completamente in Java e la configurazione dei servizi sono descritti interamente da file XML. La struttura interna è essa stessa organizzata in moduli che offrono un'interfaccia di astrazione coerente e completa. Se a tutto questo si aggiunge che qualsiasi modulo sviluppato per uno qualsiasi degli IDE in circolazione è completamente compatibile con tutti gli altri, allora si comprende come "investire" tempo e risorse nello sviluppo dello strumento per questo ambiente sia un'ottima decisione.

Al fine di soddisfare le condizioni sopra esposte si utilizzano due sottosistemi distinti:

- Il sottosistema *ViewTraces* rappresenta la struttura portante del nostro applicativo: infatti è questo sottosistema che si occupa delle gestione e della visualizzazione delle tracce.
- Invece per far funzionare lo strumento grafico all'interno dell'ambiente di sviluppo ci si è avvalso di un sottosistema, *debugger*, già sviluppato nel precedente lavoro di tesi di S.Alfeo[31]. Il nostro compito è stato quello di aggiungere delle funzionalità a questo affinché potesse usufruire dei servizi offerti da *ViewTraces*.

Nel prosieguo del capitolo viene dettagliata la struttura dei suddetti sottosistemi attraverso un'analisi *top-down* completa al solo costo di identificare ad ogni passo quali siano le caratteristiche del servizio offerto e quali i servizi utilizzati.

La notazione utilizzata per descrivere l'architettura dello strumento grafico integrato è l'UML.

Tale analisi è articolata nelle seguenti fasi:

- identificazione dei moduli che formano i sottosistemi e del loro ruolo all'interno dell'applicazione.
- identificazione e analisi delle classi che costituiscono i moduli. In questa fase verranno utilizzati diagrammi UML di classe in cui vengono visualizzate le singole classi e i legami esistenti fra le classi dello stesso sottosistema.



Figura 13: Sottosistemi che compongono lo strumento

4.2 Il sottosistema ViewTraces

Applicando i principi di scomposizione strutturale esposti in precedenza in questo capitolo è possibile scomporre tale sottosistema come in Figura 14. Si identificano i seguenti moduli:

- *Core*: modulo si occupa di effettuare tutte le operazioni necessarie alla gestione delle informazioni contenute nei file di *log* prodotti durante una precedente fase di *tracing*.
- *Gui*: modulo d'interfaccia grafica che accede alla logica applicativa del modulo suddetto per presentare i dati all'utente.



Figura 14: Moduli del sottosistema ViewTraces

4.2.1 Il modulo Core

In Figura 15 sono rappresentate le classi più importanti di questo package. La struttura di classi utilizzata è un adattamento delle classi contenute in precedenti lavori di tesi: questo adattamento è stato fatto affinché lo strumento grafico riuscisse a gestire le tecniche avanzate di analisi descritte nel Capitolo 1. In questa sede si daranno indicazioni generali sui ruoli delle singole classi di questo package in funzione dell'uso che se ne farà. Per maggiori dettagli riguardo alla logica di funzionamento di queste classi ed alla loro implementazione si faccia riferimento a tali testi [32].

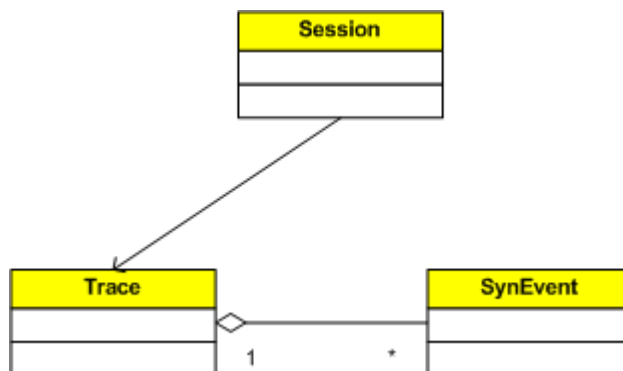


Figura 15: Classi del modulo Core

La classe *Session* contiene tutte le informazioni riguardanti le tracce d'esecuzioni concorrenti prodotte precedentemente. La struttura dati fondamentale in essa contenuta è una lista (classe *Vector* della libreria *java.util*) chiamata *traces*, di oggetti di classe *Trace*; quest'ultima non è altro che un insieme di oggetti *SynEvent*. Quindi, in definitiva, l'insieme degli eventi di una computazione concorrente è implementata come lista di tracce, che a loro volta sono liste di eventi.

La classe *Session* è astratta, non è quindi istanziabile direttamente. Essa contiene infatti un metodo astratto, la cui firma è:

```
public abstract Vector buildTraces().
```

Questo è l'unico metodo astratto di tale classe, e deve essere implementato da una sua sottoclasse. Esso restituisce una lista che viene assegnata a *traces* dall'unico costruttore della nostra classe, che a sua volta deve essere chiamato (istruzione *super*) dal costruttore della sottoclasse. Il costruttore di tali classi ha bisogno del nome della directory in cui si trovano le tracce memorizzata sotto forma di file di *log*.

Questo meccanismo consente di rendere il nostro strumento indipendente dal formato delle tracce originali. Infatti la classe *Session* non si occupa di come costruire le tracce di eventi nel formato interno, sarà il metodo *buildTraces* di una sua sottoclasse a operare la traduzione.

A questo punto è possibile comprendere in cosa consiste il lavoro di costruzione delle tracce che deve essere effettuato dal suddetto metodo. A partire dalle tracce originali, in qualsiasi formato esse siano, occorre creare gli opportuni oggetti *SynEvent*, e gli oggetti *Trace* che li contengono. Il punto fondamentale però è la costruzione di liste di partner passivi, attivi e sincroni di ciascun oggetto *SynEvent*. I criteri con cui ciò debba essere fatto dipendono ovviamente dagli eventi con cui si ha a che fare.

Per maggiori dettagli implementativi si rimanda al lavoro descritto in [32].

Affinché si potesse utilizzare questo meccanismo per tradurre le tracce, prodotte dal modulo sviluppato nel lavoro di tesi di Salvatore Alfeo [31], contenenti gli eventi relativi ad alcuni tra i più importanti costrutti di sincronizzazione e comunicazione del linguaggio Java, si è implementato una sottoclasse di *Session* chiamata *JavaSession*.

4.2.2 Algoritmi utilizzati

All'interno della classe *Session* sono stati implementati tutti i principali algoritmi; noi non ci soffermeremo su quelli ripresi dalle tesi precedenti [32], ma descriveremo con maggiore attenzione l'algoritmo di *Pattern Matching*.

Pattern Matching

L'algoritmo di pattern matching parte dal presupposto che un pattern consta di un certo numero di stringhe, sulle quali sono localizzati gli eventi, con dei possibili collegamenti

tra eventi che appartengono a stringhe diverse. Queste stringhe sono formate dalla concatenazione dei nomi/tipi degli eventi che appartengono allo stesso componente. Per cui cercare un pattern è equivalente alla ricerca di tutte le stringhe che lo compongono all'interno delle stringhe che formano i vari componenti, tenendo conto dei possibili collegamenti tra di esse (Figura 16).

Il primo passo del nostro algoritmo consiste nell'individuare la stringa del pattern avente il maggior numero di connessioni, siano N_c , all'interno del pattern stesso; ci riferiremo a tale stringa con il termine *search string*. Per ogni communication pattern abbiamo l'obbligo di avere solamente una sola *search string*; per questo motivo se troviamo più stringhe che hanno lo stesso numero di link si sceglie quella avente il maggior numero di caratteri ad ognuno dei quali è associato un evento.

In Figura 16 possiamo osservare che l'utente ha specificato un *communication pattern* composto da tre stringhe: risulta evidente che all'interno di questa struttura l'unica stringa a soddisfare entrambi i requisiti sopra citati è la Stringa 2, per cui questa viene etichettata come *search string*.

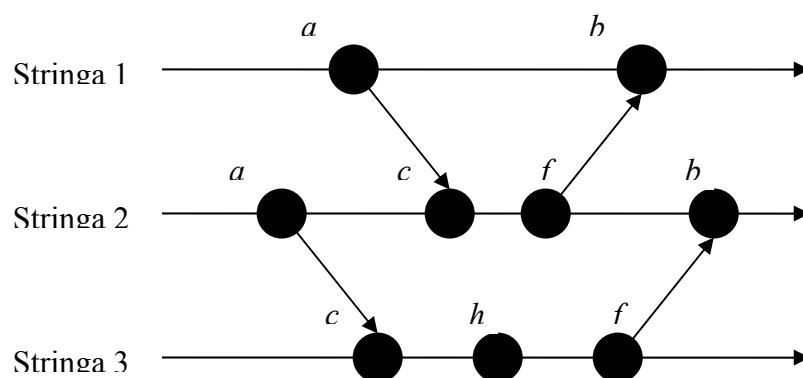


Figura 16: Pattern visto come sequenza di stringhe comunicanti

Una volta determinata tale stringa si riduce il numero di componenti all'interno dei quali cercare la *search string* a quelli aventi un grado di connettività maggiore o uguale a N_c ; a questo punto si applica un opportuno algoritmo di *String Matching*.

Noi abbiamo utilizzato l'algoritmo conosciuto in letteratura come *metodo ingenuo*: questo allinea il primo carattere della stringa S da ricercare con il primo carattere del testo T, confronta quindi i caratteri di S e di T da sinistra a destra finché o si arriva alla fine di S, nel qual caso viene segnalata un'occorrenza di S in T, oppure vengono incontrati due caratteri diversi (mismatch). In entrambi i casi S viene quindi spostato avanti di un posto e si ricominciano i confronti a partire dal primo carattere di S con il carattere del testo ad esso allineato. Il procedimento viene ripetuto finché l'ultimo carattere di S non sorpassi l'ultimo carattere di T. Abbiamo scelto questo metodo piuttosto semplice perché le stringhe e i testi da noi utilizzati non sono eccessivamente lunghi.

Infine si va a controllare la struttura interna di tutte le *search strings* risultanti dallo String Matching cercando di ricostruire il pattern definito inizialmente. Si può asserire di aver trovato il pattern se e solo se sono state "matched" tutte le stringhe che lo formano.

Per maggiore chiarezza scriviamo l'algoritmo in pseudo-codice; lo descriviamo a partire dalla ricerca delle *search strings* all'interno dei componenti aventi un numero di connessioni maggiore di N_c .

```
∀ componente avente grado maggiore di  $N_c$  {  
    applico String matching  
    if (trovo occorrenze) {  
        ∀ occorrenza trovata {  
            provo a ricostruire pattern  
            if (pattern trovato)  
                tengo traccia del pattern trovato  
        }  
    }  
    }  
    return patterns trovati
```

La Figura 17 ci da un esempio del ritrovamento del pattern definito precedentemente (Figura16), all'interno di tracce d'esecuzioni concorrenti costituite da quattro

componenti: l'area tratteggiata rappresenta il pattern cercato. Inoltre si può osservare che la *search string*, che è la seconda stringa del pattern, è recuperata all'interno del secondo componente. Viceversa la Figura 18 ci mostra il caso in cui il pattern non viene "matched" nonostante la *search string* sia riscontrata all'interno delle tracce. Questo avviene perché il quarto componente contiene una stringa diversa da quella definita all'interno del pattern a causa della presenza dell'evento *x*.

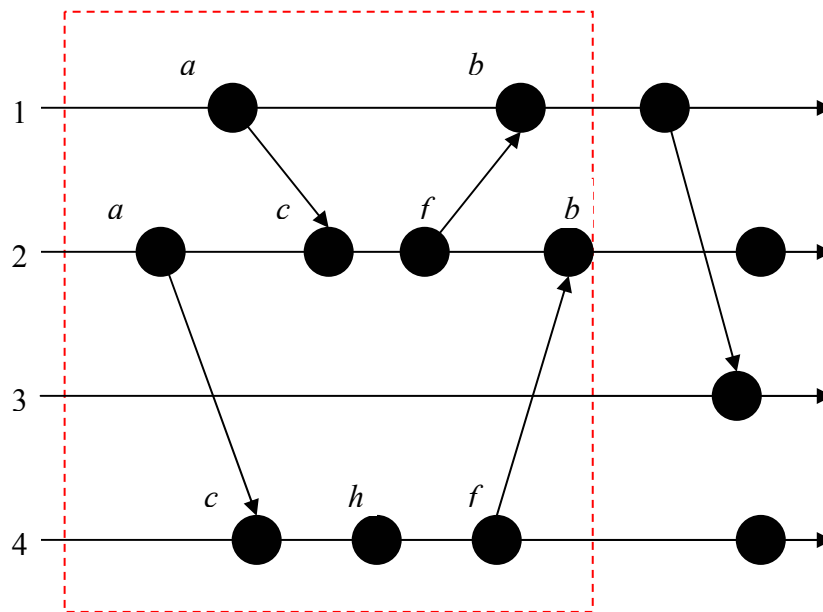


Figura 17: Pattern matched

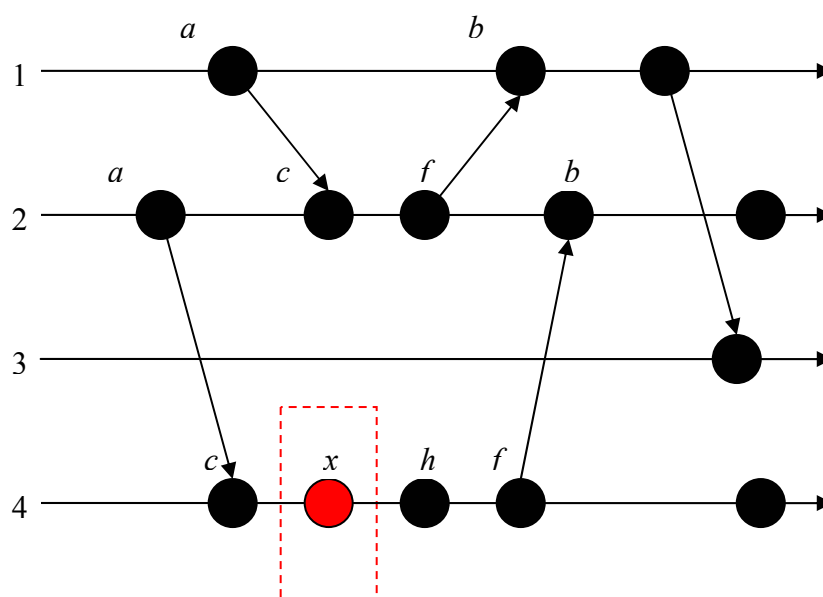


Figura 18: Pattern non-matched

4.2.3 Il modulo Gui

Questo modulo è stato sviluppato interamente facendo uso della libreria *Swing* che è parte integrante delle JavaTM Foundation Classes (JFC) [44]; le JFC raggruppano un insieme di caratteristiche utili all'utente per costruire delle GUIs.

Il package *Swing* è stato dapprima disponibile come “add-on” del JDK 1.1. Prima dell'introduzione di questo package, nelle piattaforme JDK 1.0 e 1.1, era la libreria *Abstract Window Toolkit (AWT)* che forniva gli strumenti necessari alla creazione di componenti per l'interfaccia grafica dell'utente (UI). Sebbene la piattaforma Java 2, quella utilizzata nel nostro lavoro, ancora supporta i componenti *AWT*, oggi si è fortemente incoraggiati ad utilizzare componenti *Swing* perché, non essendo questi legati alle caratteristiche della piattaforma su cui girano (infatti non utilizzano “codice nativo”), possono avere numerose funzionalità aggiuntive.

La classe principale di questo modulo è sicuramente *DrawArea*.

Questa classe è quella che si occupa della gestione grafica delle informazioni raccolte dalla classe *Session*; per poter accedere a tali dati viene istanziato un oggetto di tale classe all'interno di *DrawArea* (Figura 19).

Infatti è *DrawArea* che costruisce l'STD, disegnando le linee che rappresentano i componenti, associando vari simboli agli eventi e svolgendo tutti i compiti necessari per presentare la computazione concorrente secondo quanto specificato dai requisiti esposti nel Capitolo 2. Inoltre è sempre questa classe adibita al controllo delle interazioni con l'utente attraverso delle opportune classi *Manager*.

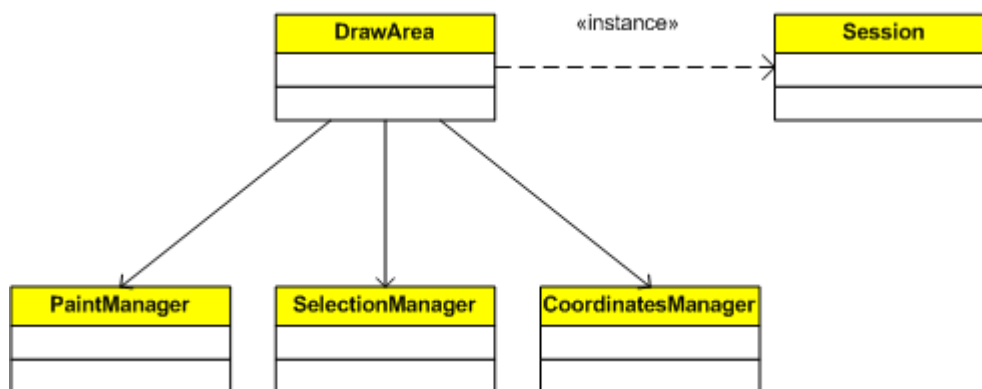


Figura 19: Classi del Modulo DrawArea

4.3 Il sottosistema debugger

Come già detto in precedenza questo sottosistema è stato sviluppato in un precedente lavoro di tesi[31] e i moduli in cui era stato suddiviso sono:

- *core*: modulo che si occupa di effettuare tutte le operazioni necessarie al tracing di esecuzioni di file sorgenti precedentemente instrumentati e di memorizzare i dati necessari alla visualizzazione.
- *controller*: modulo che fornisce la logica di aggancio tra il core dell'applicazione e l'interfaccia grafica.
- *gui*: modulo di interfaccia grafica che accede alla logica applicativa tramite il modulo *controller*.
- *cli*: modulo di interfaccia a linea di comando che accede direttamente alla logica applicativa senza passare per il modulo *controller*.

I moduli a cui noi abbiamo aggiunto delle funzionalità per permettergli di agganciare i servizi offerti da *ViewTraces* sono il modulo *controller* e il modulo *gui*, per cui questi sono gli unici che definiremo con un maggiore dettaglio descrivendone le classi da noi aggiunte.

4.3.1 I moduli controller e gui

Il modulo *controller* si occupa di interfacciare la logica applicativa contenuta nel modulo *core* con l'interfaccia grafica gestita dal modulo *gui*. L'insieme dei servizi esposti da questi moduli sono progettati per costituire un "NetBeans module", cioè un'estensione dell'ambiente NetBeans che supporta il servizio di tracing e di visualizzazione delle tracce. In questo paragrafo non verranno dettagliate le funzioni delle singole classi ma solo le linee guida che portano all'esposizione dei servizi verso l'interfaccia grafica.

In Figura 20 è rappresentato il diagramma di classe del modulo *controller*.

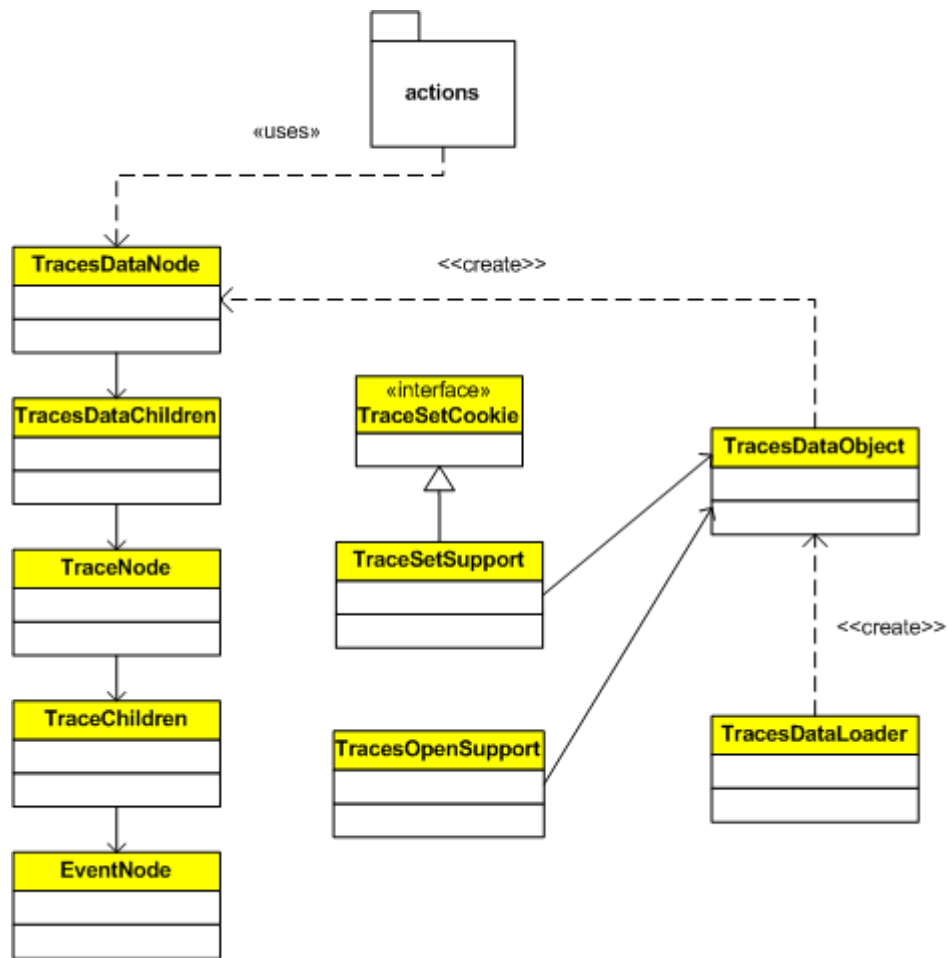


Figura 20: Classi del modulo controller

La suddetta struttura di classi è stata creata, secondo quanto specificato dalle OpenAPI [7], per permettere a NetBeans di riconoscere i files contenenti le tracce d'esecuzioni, creati in fase di *tracing*, e di mostrare il nodo che li rappresenta all'interno dell'Explorer (vedi Figura 21).

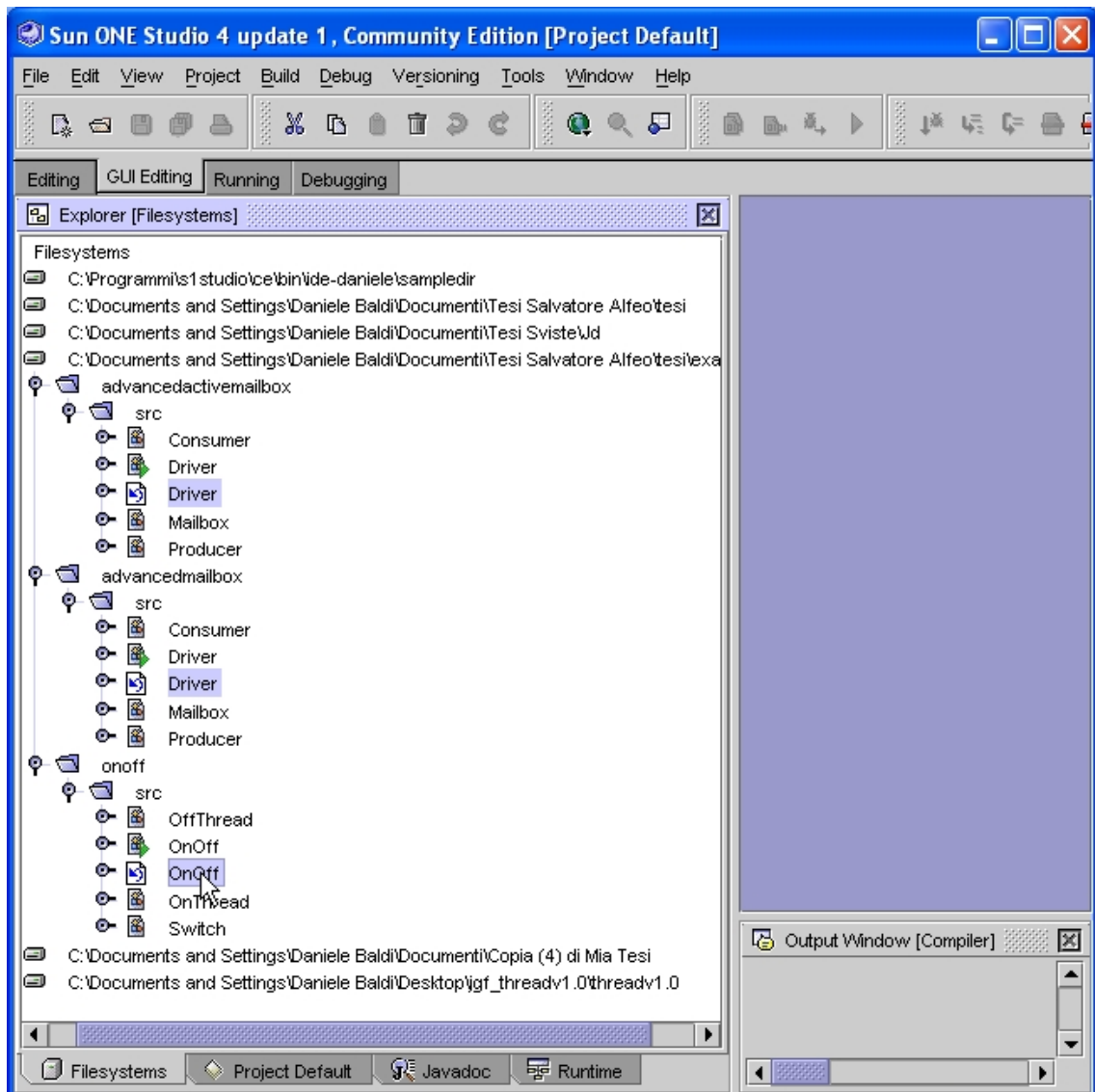


Figura 21: File che rappresentano le tracce

All'interno del package actions sono contenute le classi che rappresentano le azioni che possono essere intraprese dall'IDE una volta aggiunto il nostro modulo.

A quelle create nei precedenti lavori è stata aggiunta la classe *ViewTracesAction* che permette la visualizzazione delle tracce grazie alla creazione di un oggetto della classe *ViewTracesPanel* appartenente al modulo *gui*.

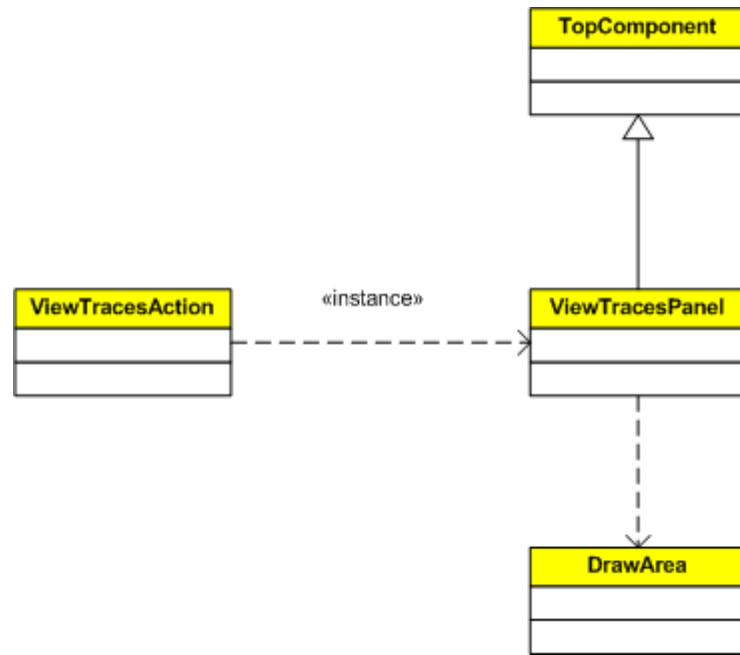


Figura 22: Collegamento tra debugger e ViewTraces

La classe *ViewTracesPanel* generalizza la classe *TopComponent* delle OpenAPI che rappresenta il componente grafico principale per la visualizzazione di finestre all'interno dell'IDE; inoltre *ViewTracesPanel* dipende fortemente da *DrawArea*, perché è questa classe che gli permette di agganciare i servizi offerti dal sottosistema *ViewTraces*, descritto in precedenza.

4.4 Conclusioni

In questo capitolo abbiamo visto come è organizzata l'architettura software dell'applicazione dal punto di vista delle classi Java che la compongono. Nel capitolo successivo, alla luce della struttura introdotta, verranno descritte tutte le funzionalità dello strumento.

Capitolo 5

Utilizzo dello strumento

Nei capitoli precedenti sono state illustrate le caratteristiche dello strumento grafico, ne è stata definita l'architettura software e sono stati evidenziati alcuni dettagli implementativi importanti. In questo capitolo viene illustrato come installare lo strumento, come configurarlo e quali sono le sue funzioni più importanti.

5.1 Installazione dello strumento

Come detto più volte lo strumento si propone sia come interfaccia grafica funzionante in modo indipendente dall'ambiente integrato, sia come modulo aggiuntivo di quest'ultimo, mantenendo però in entrambe le modalità le principali caratteristiche funzionali.

Per questo motivo di seguito esporremo due procedure d'installazione che permettono di utilizzare la nostra applicazione in entrambi i suddetti casi.

Nel primo caso l'installazione dei file del programma è determinata dalle regole con cui la Java Virtual Machine (JVM) ricerca i file relativi a ciascuna classe utilizzata in un programma Java. La classe in cui è contenuto il *main* dello strumento, chiamata *Viewer*, si trova all'interno del package *ViewTraces*. Le altre classi che compongono l'applicazione sono contenuti in tale package e in suoi sotto-package. L'albero delle directory in cui devono essere presenti i file sorgente è dato dalla struttura stessa dei

package, così come richiesto dalle regole sullo spazio dei nomi in Java. La directory *ViewTraces* deve essere resa nota alla JVM tramite una variabile d'ambiente di nome CLASSPATH. L'utilizzo di tale variabile varia comunque da piattaforma a piattaforma, per cui si rimanda alla documentazione del sistema Java utilizzato per i dettagli su tale aspetto [43].

L'albero dei package è mostrato in Figura 23.

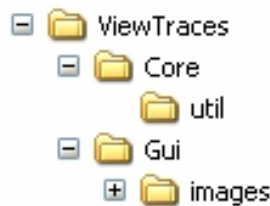


Figura 23: Albero delle directory d'installazione

All'interno della directory *Gui* deve essere presente la sottodirectory *images*, contenente le immagini per i pulsanti e i simboli dello strumento, così come la directory *util* deve essere contenuta in *Core*.

Una volta creato tale albero con i file sorgenti, e risolti i problemi relativi al CLASSPATH, si può compilare tutte le classi dello strumento.

Vista la portabilità di Java dovrebbe essere possibile creare direttamente l'albero di directory indicato contenente i file *.class* prodotti su un'altra macchina o su un'altra piattaforma. Noi, comunque, abbiamo lavorato con la JVM JSDK 1.4.0.03 della Sun, su macchine Windows.

Una volta compilato, possiamo lanciare lo strumento che parte senza creare alcuna sessione, cosa che si può fare dall'interfaccia grafica.

Nel secondo caso per poter agganciare i servizi forniti dallo strumento grafico all'ambiente di sviluppo, è stato reso disponibile un file di installazione denominato *debugger*. Tale file ha estensione *.nbm*: esso è uno speciale archivio di alto livello che contiene un modulo in formato *.jar* standard più tutte le risorse necessarie al funzionamento del modulo. Questo formato di file è il formato standard di installazione supportato dall'ambiente NetBeans di cui il modulo proposto è una estensione. Dal punto di vista del formato, esso non è altro che un semplice file compresso con allegate

alcune informazioni di installazione riguardanti la descrizione del modulo, la loro configurazione, l'indicazione delle posizioni in cui i file necessari devono essere copiati, più alcuni file "manifesto" che servono in fase di installazione a guidare l'installazione automatica. Per avviare l'installazione del modulo basta avviare il file *.nbm* corrispondente (tale file è associato di default all'installazione corrente dell'IDE) oppure lanciare l'opportuno *wizard* di installazione dall'interno dell'ambiente. Negli esempi che seguono si farà uso dell'ambiente SunOne Studio 4 update 1 basato sulle librerie NetBeans versione 3.3. Dal punto di vista del funzionamento del modulo e della loro installazione va bene qualsiasi IDE basato su tali librerie. Ovviamente da ambiente ad ambiente cambia il modo di installare i moduli e possono cambiare posizione e nomi di alcune impostazioni o voci di menu.

Per installare il modulo dall'interno dell'IDE basta selezionare la voce di menu *Update Center* contenuta nel menu *Tools* ed indicare di voler installare un nuovo modulo disponibile localmente. La procedura è del tutto automatica e non richiede alcun intervento da parte dell'utente (Figura 24).

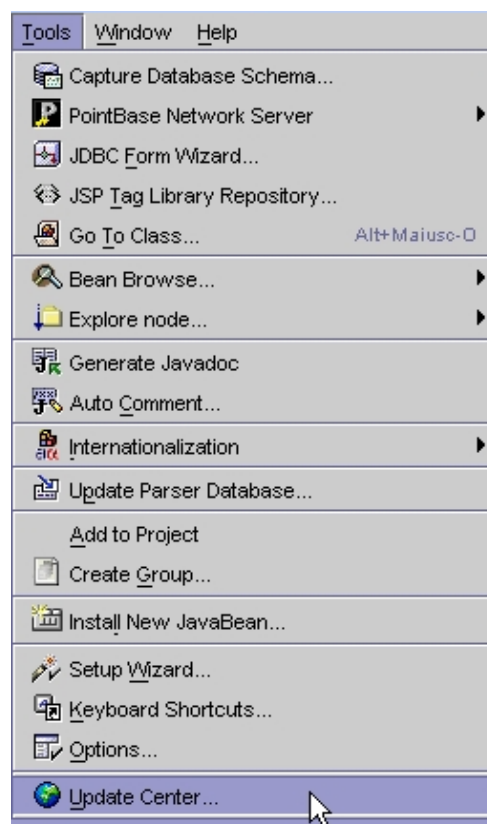


Figura 24: Procedura d'installazione del modulo

che attraverso il *popup* menu associato a tale file (Figura 27). A differenza di quanto esposto per la modalità *stand alone*, in questo caso l'interfaccia grafica dello strumento appare con la sessione, relativa al file selezionato, già caricata.

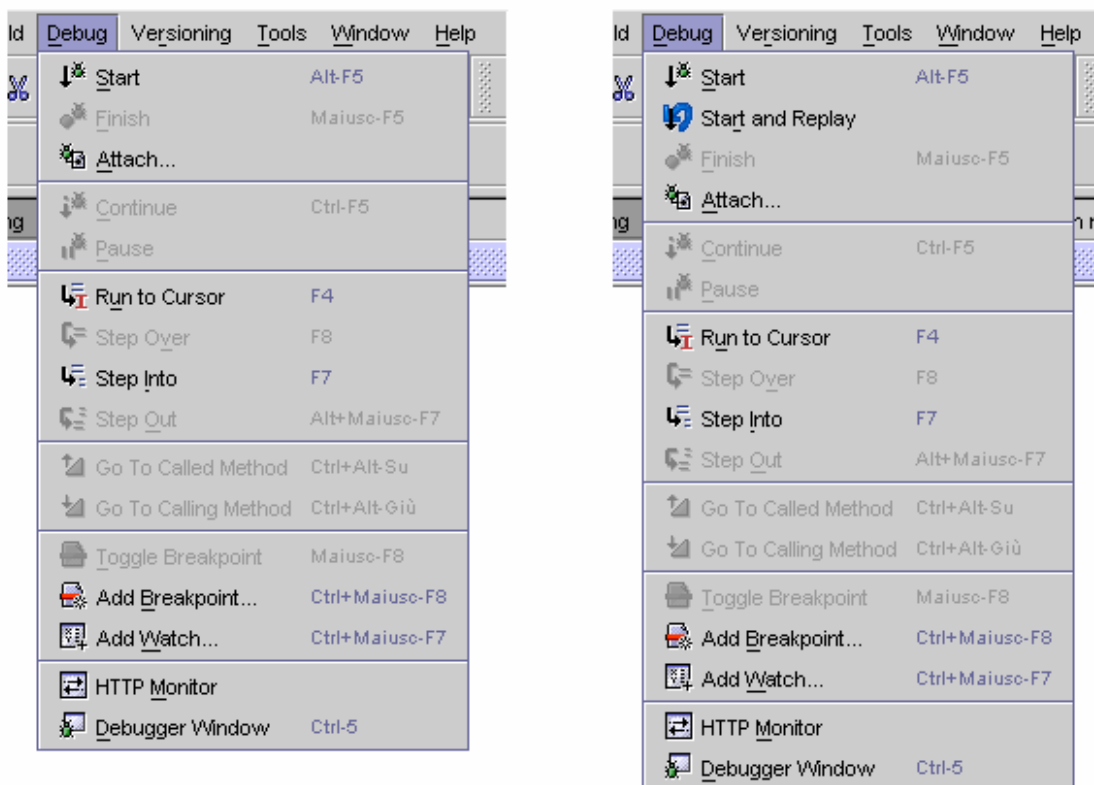


Figura 26: Menu debug prima dell'installazione del modulo, dopo l'installazione del modulo debugger

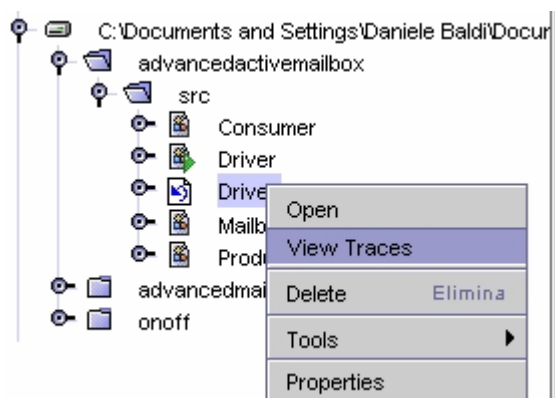


Figura 27: Menu popup associato al file .traces

5.2 Visualizzazione delle tracce d'esecuzione

In questo paragrafo, continuando a riferirci alla distinzione precedentemente esposta, mostreremo sia l'interfaccia grafica dello strumento *stand alone* sia come vengono visualizzate le tracce all'interno dell'IDE.

Nel primo caso la finestra principale dello strumento grafico, quando non è caricata alcuna sessione, ha il seguente aspetto della Figura 28.

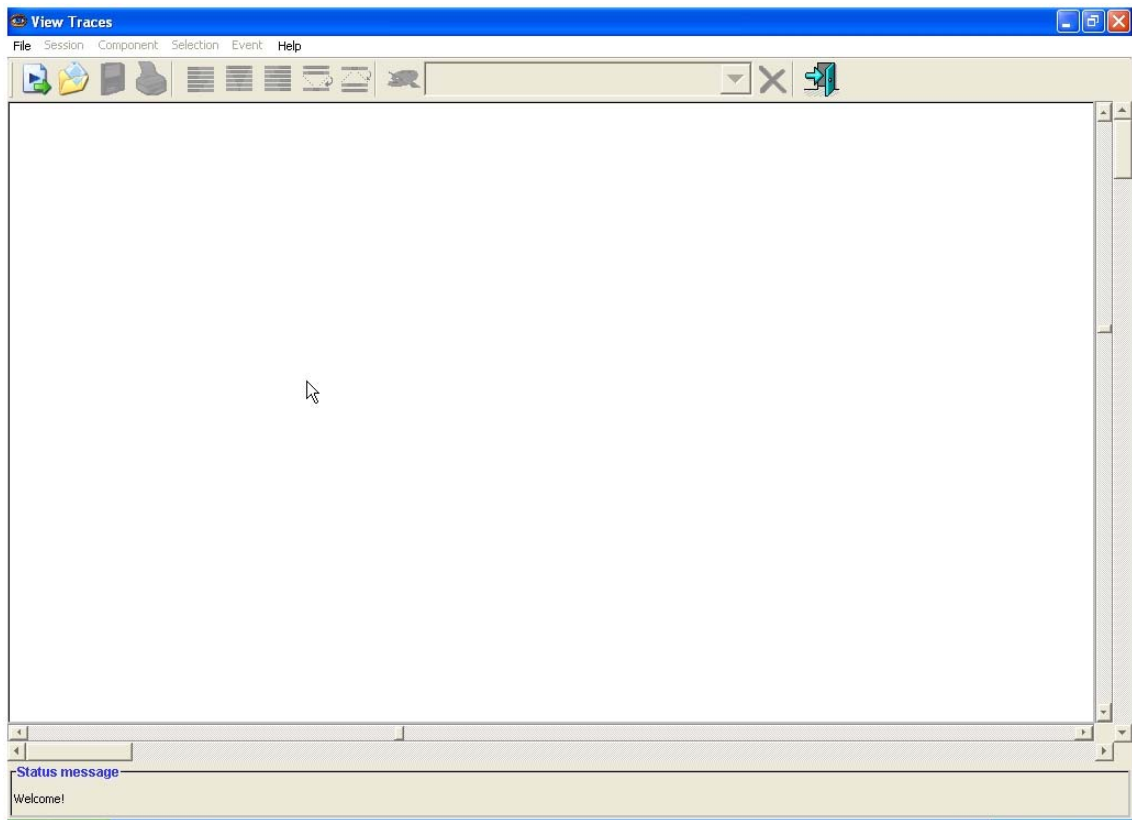


Figura 28: Finestra principale dello strumento stand alone

L'immagine è tratta da un sistema Windows, ma l'aspetto è simile su tutte le piattaforme. Oltre a elementi noti come la barra del titolo e i bottoni di iconizzazione, ripristino e chiusura, si notino la barra dei menu, la barra degli strumenti, e la barra di stato. Come è usuale, sulla barra degli strumenti sono replicati alcuni dei comandi che dovrebbero essere usati più frequentemente. La barra di stato, oltre che per un messaggio di benvenuto, viene utilizzata per notificare all'utente l'attività in cui è

impiegato lo strumento durante la costruzione di un STD, e per visualizzare informazioni sugli STD muovendo il mouse sui vari elementi.

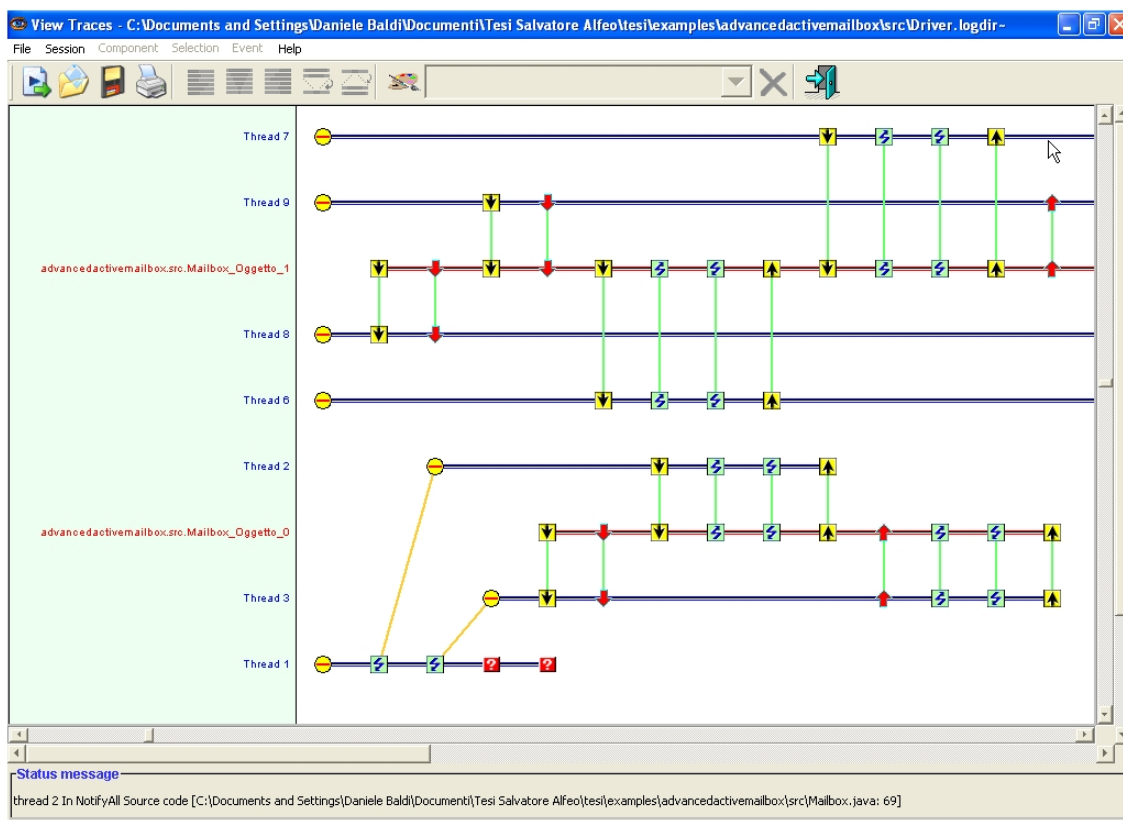


Figura 29: Visualizzazione delle tracce nello strumento stand alone

La figura 29 mostra una schermata con una sessione caricata. Le tracce, secondo quanto detto nelle specifiche, sono rappresentate da linee orizzontali, con il tempo che scorre verso destra. Tali linee hanno diversi colori. Il colore viene scelto in base al tipo di componente. Il tipo del componente dipende da un parametro che viene passato al costruttore di ciascun oggetto *Trace* nel metodo *buildTraces*. Tale parametro specifica se il componente è un processo (o thread), oppure se rappresenta un meccanismo di gestione di una variabile condivisa. I primi vengono rappresentati da linee blu, i secondi da linee rosse. Sulla barra verticale a sinistra dell'STD si trovano i nomi delle tracce.

Gli eventi sono rappresentati da simboli di varie forme. Il meccanismo con cui lo strumento sceglie i simboli è il seguente. Ogni *SynEvent* ha un campo intero che specifica il tipo di evento, inoltre in *Session* c'è un vettore che associa a ciascun tipo di

evento un simbolo; la creazione di tale vettore, così come l'assegnamento del tipo di evento, avvengono nel metodo *buildTraces*.

Ponendo il puntatore del mouse su un evento, sulla barra di stato appare una stringa. Vediamo da dove viene. Ogni *SynEvent* ha un campo di classe *Object*, chiamato *originalEvent*. Essendo di classe *Object*, può contenere qualsiasi oggetto Java. Tale oggetto viene passato al costruttore di un *SynEvent*. Ogni oggetto Java ha un metodo *toString* che restituisce una stringa. Il messaggio che appare nella barra di stato è il risultato di tale metodo del campo *originalEvent* dell'oggetto *SynEvent* relativo al simbolo su cui è il puntatore del mouse. Se a tale campo non è associato alcun oggetto, il messaggio lo ricorderà.

In Figura 29 si notino anche i segmenti verdi che indicano gli insiemi sincroni, e i segmenti obliqui di colore arancio che indicano relazioni asincrone.

E' importante sottolineare che sia i colori dei componenti, sia i colori dei segmenti e sia i simboli associati agli eventi possono essere modificati attraverso un opportuno menu Opzioni.

Le Scrollbar interne (vedi Figura 29) servono per modificare la dimensione assegnata a ciascuna cella della griglia grafica su cui sono disposti i simboli. Esse non modificano la dimensione dei simboli, ma consentono di visualizzare, in un'unica schermata, porzioni più o meno ampie della computazione. Le altre Scrollbar si attivano se il diagramma non è interamente contenuto nello schermo. Esse consentono di visualizzare diverse regioni del diagramma. Utilizzando i pulsanti di incremento la griglia si sposta di una cella, mentre cliccando sulle regioni della barra non occupate dal cursore ci si sposta di un numero di celle pari a quelle visibili sullo schermo (in ciascuna direzione).

Le operazioni di selezione si fanno esclusivamente con il mouse. Si può selezionare un segmento (o un singolo evento) su uno o più componenti. Non è possibile selezionare regioni non contigue su un singolo componente. Quando si clicca con il pulsante sinistro del mouse su un simbolo la cella apparirà in reverse, così come il nome del componente di appartenenza. Questa è una regola generale. Ogni volta che alcune zone del diagramma vengono mostrate in reverse, in seguito ad operazioni di selezione, sono in reverse anche i nomi dei componenti interessati.

Una volta selezionato un evento, cliccando su un altro simbolo (precedente o seguente) dello stesso componente, con il pulsante destro, la selezione verrà estesa dal primo al

secondo simbolo. Questo è il modo in cui si seleziona un segmento. Si può continuare ad usare il pulsante destro per estendere o contrarre il segmento selezionato. Mentre, cliccando con il pulsante sinistro, la selezione verrà spostata sul nuovo simbolo, indipendentemente da ciò che era precedentemente selezionato su quel componente.

Cliccando con il pulsante sinistro sul nome di un componente viene selezionato l'intero componente. Cliccando con il pulsante destro sul nome di un componente selezionato, lo si deselecta.

Una volta selezionato un segmento, si può selezionare un segmento su un altro componente, con le stesse modalità, e così via. I segmenti precedentemente selezionati, rimarranno tali. L'unico modo per spostare la selezione da un componente ad un altro è deselectare prima ogni componente selezionato.

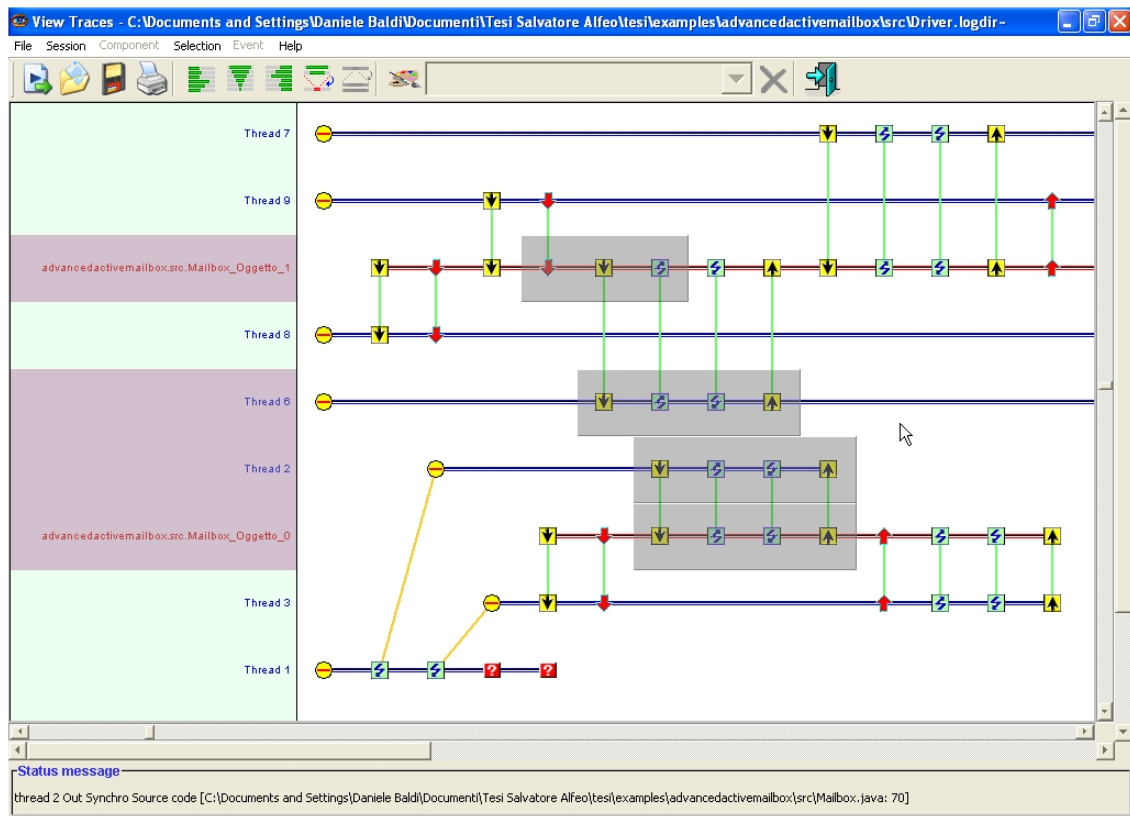


Figura 30: Esempio di segmenti selezionati

In figura 31 viene mostrato come appare la visualizzazione delle tracce all'interno dell'IDE.

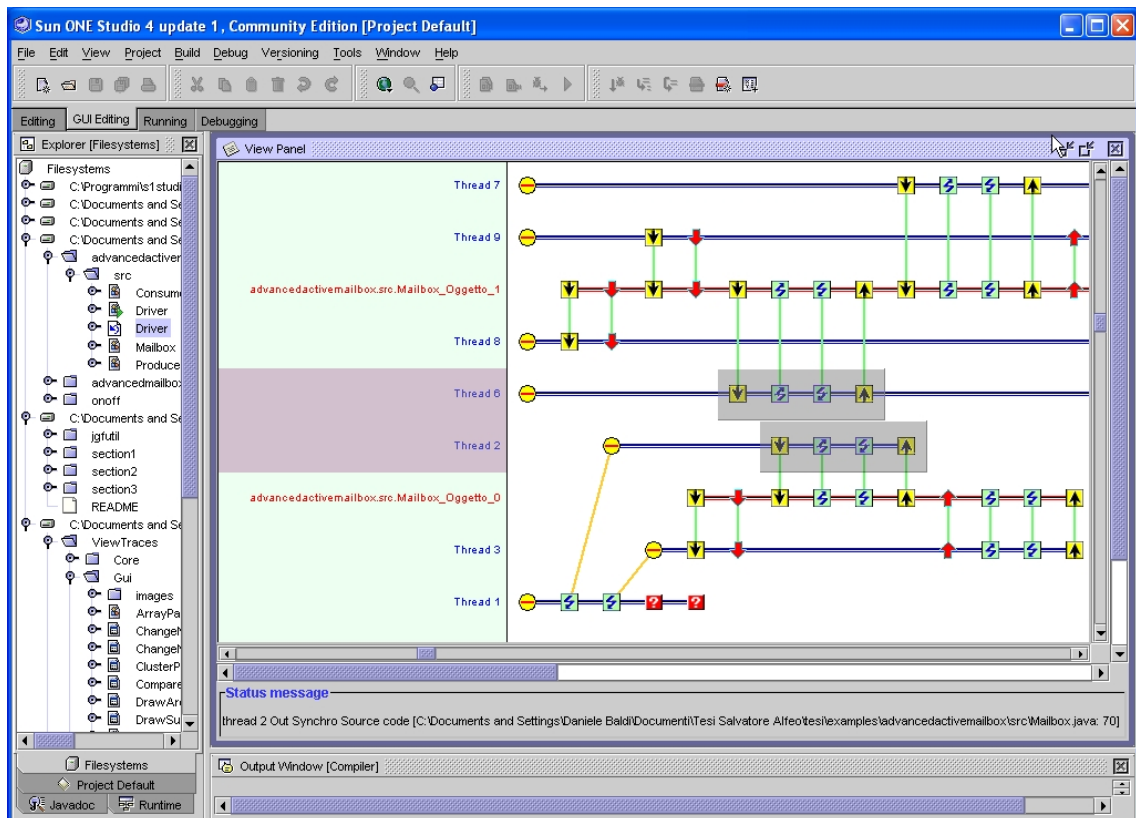


Figura 31: Visualizzazione tracce all'interno dell'IDE

Come si può notare le uniche differenze con quanto esposto in precedenza sono che all'interno di questo pannello di visualizzazione, definito *ViewPanel*, non sono più presenti né la barra dei menu né la barra degli strumenti. Questo è stato fatto perché le funzionalità di tali barre sono replicate all'interno dei popup menu per cui non si è ritenuto necessario inserirle; tali funzionalità saranno descritte nel prossimo paragrafo. Invece le modalità di selezione, il modo con cui vengono rappresentati gli eventi, i componenti e la gestione dei colori non variano da quanto detto per lo strumento stand alone per cui non ci inoltreremo nuovamente in tali argomentazioni.

5.3 Funzioni dello strumento

In questo paragrafo procederemo dapprima descrivendo le funzionalità a comune tra le due modalità d'uso dello strumento (stand alone e integrata nell'IDE), poi, facendo riferimento solo alla prima, descriveremo come può essere caricata e salvata una sessione e quali sono i servizi extra supportati da essa.

Da notare che nella prima parte della descrizione utilizzeremo tutti immagini contenenti raffigurazioni del *ViewPanel* presente all'interno dell'IDE.

5.3.1 Operazioni effettuabili sui componenti

Se non si è in modalità selezione, cioè se nessun nome di componente è in modalità reverse, cliccando con il pulsante destro del mouse su tale nome appare un menu di popup che permette di effettuare le seguenti operazioni:

- **Change name:** mostra una finestra di dialogo da cui è possibile cambiare il nome del componente.
- **Move up, Move down:** scambia la posizione della traccia con la precedente o la successiva, rispettivamente. Vengono aggiustate automaticamente le posizioni dei simboli.
- **Options:** mostra una finestra di dialogo che permette di cambiare il colore associato ai vari componenti.

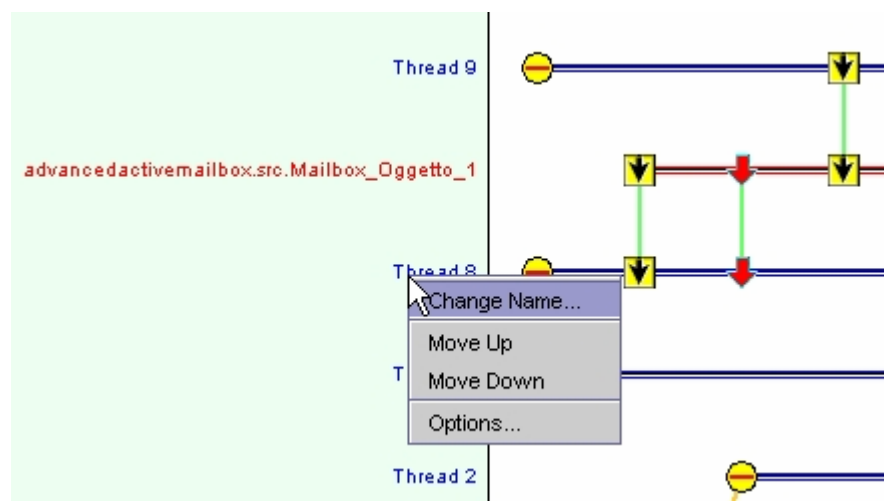


Figura 32: Popup menu di un componente

Queste stesse funzionalità sono presenti anche all'interno della voce *Component* della barra dei menu dello strumento *stand alone*; tuttavia ciascun comando di questo menu ha effetto solo se è selezionato uno ed un solo componente, per intero. Ricordiamo che

ciò può essere fatto cliccando con il pulsante di sinistra del mouse sul nome di un componente.

5.3.2 Operazioni effettuabili sugli eventi

Ciascun comando dei menu associati agli eventi ha effetto solo se è selezionato uno ed un solo evento. Anche in questo caso esiste sia un menu di popup, presente nella modalità stand alone e in quella integrata nell'IDE, che un menu *Event* presente nella barra dei menu quindi reso disponibile solo nel primo caso.

Vediamo le funzionalità offerte:

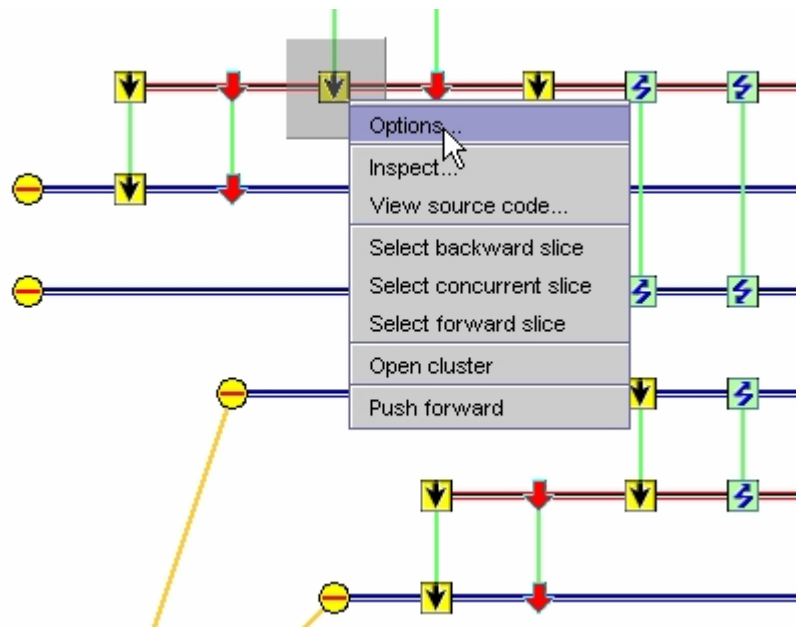


Figura 33: Popup menu di un evento

- **Options:** Questo menu è analogo a quello del menu Component, e permette di cambiare i simboli associati agli eventi.
- **Inspect:** questa è una funzione dello strumento interessante. Utilizzando le caratteristiche di riflessività di Java, viene presentata una finestra che descrive l'oggetto contenuto nel campo *originalEvent* dell'evento selezionato. Si ricorda che vengono visualizzati solo i campi pubblici di tale oggetto. Se il valore di tali campi è un tipo primitivo, o una stringa, questo compare nella finestra. Se

invece è un vettore, o un oggetto, accanto al nome del campo compare un nuovo pulsante *Inspect...* che apre un finestra analoga alla prima, e così via. Pur se limitata ad una presentazione di valori, tale funzione si presta a visualizzare informazioni il cui significato è noto all'utente. Esse potrebbero comprendere, ad esempio, informazioni speciali sull'evento, prelevate dalle tracce originali, o valori di variabili interne dell'applicazione che si sta studiando. Chiaramente, la costruzione degli oggetti visualizzati è a carico dell'utente. Ciò implica che la sottoclasse di *Session* implementata può diventare piuttosto complessa. Si ricordi inoltre che tutto ciò occupa memoria.

- **View Source Code:** ogni oggetto *SynEvent* può contenere una stringa ed un intero, che possono essere settati con un metodo opportuno nel metodo *buildTraces*, e che vengono interpretati da questo comando come il nome di un file ed un numero di linea. Se tali dati esistono, e se il file esiste sulla macchina, selezionando questo comando viene aperto il file opportuno nell'Editor di testo se siamo all'interno dell'IDE, oppure viene visualizzata una finestra di testo non editabile con il file caricato e la linea corretta selezionata se stiamo utilizzando lo strumento in modalità *stand alone*. Tale funzione è stata chiaramente concepita come un modo per vedere la linea di codice corrispondente ad un evento.
- **Select (Backward, Concurrent, Forward) Slice:** attraverso questo comando si seleziona il passato causale, il futuro causale e gli eventi concorrenti dell'evento selezionato.
- **Push Forward:** Spinge di una cella grafica verso destra l'evento selezionato e tutta la sua forward slice. Questo comando può essere utile se non si è soddisfatti dell'attuale disposizione dei simboli. Prima di utilizzare questo comando, si ricordi che non esiste un'operazione opposta, perché sarebbe, in generale, assai complessa.
- **Mark:** mostra una finestra di dialogo in cui è possibile associare un nome all'evento selezionato. Questo comando è presente solo all'interno dei menu dello strumento *stand alone* perché, una volta assegnato, tale nome apparirà nella lista presente sulla barra degli strumenti. Dopo di che, in qualsiasi

momento, selezionando tale nome dalla lista, lo strumento grafico visualizzerà la regione dell'STD contenente tale evento, che apparirà selezionato.

5.3.3 Operazioni effettuabili su una selezione

Ciascun comando dei menu associati ad una selezione ha effetto solo se esiste una selezione che non sia un singolo evento. Anche in questo caso esiste sia un menu di popup, presente nella modalità stand alone e in quella integrata nell'IDE, che un menu *Selection* presente nella barra dei menu quindi reso disponibile solo nel primo caso.

Vediamo le funzionalità offerte:

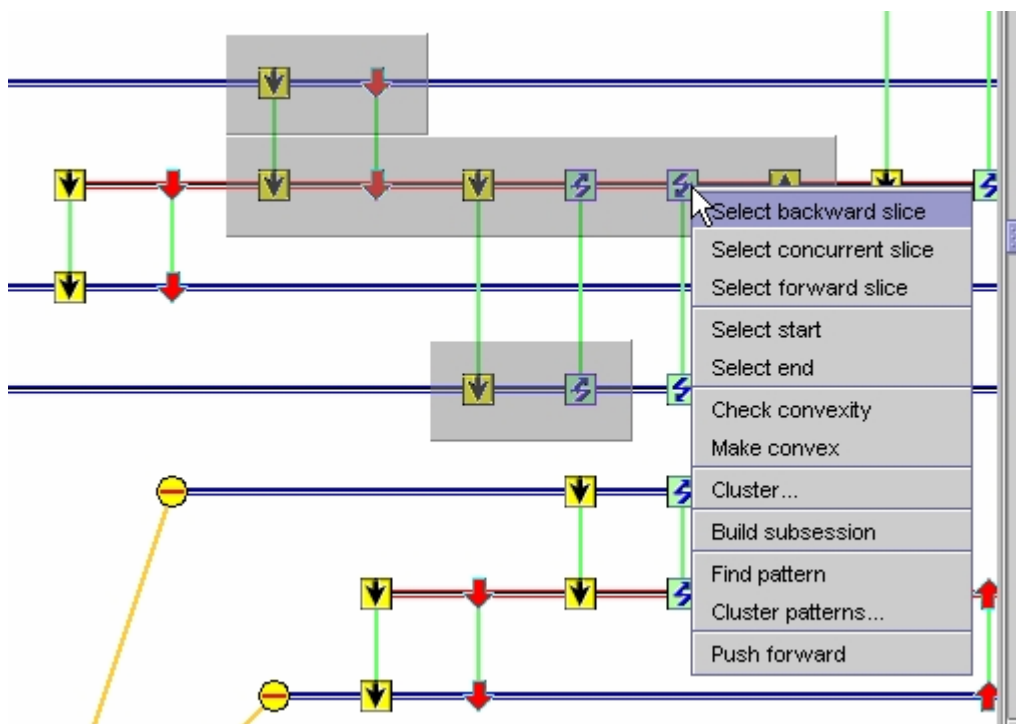


Figura 34: Popup menu di una selezione

- **Select (Backward, Forward, Concurrent) slice:** questo comando è del tutto simile al suo omonimo descritto nel precedente paragrafo.
- **Check Convexity:** controlla la convessità della selezione, mostrando una finestra con il risultato del controllo.
- **Make Convex:** seleziona la chiusura convessa della selezione attuale. La selezione precedente viene persa.

- **Select start, end:** seleziona il primo o l'ultimo evento in ordine temporale.
- **Cluster:** viene innanzitutto verificato se la selezione è convessa e, se non lo è, si dà la possibilità di renderla tale con l'ausilio del comando *Make Convex*. Il passo successivo è la clusterizzazione vera e propria. Lo strumento deve poi effettuare tutte le operazioni necessarie per la costruzione di una sessione. Con una operazione di clustering, la computazione originale viene persa.
- **Build subsession:** questo comando elimina dalla sessione tutti gli eventi esterni alla selezione, permettendo di concentrarsi su una regione limitata della computazione.
- **Find pattern:** viene innanzitutto verificato se la selezione rispetta i requisiti descritti in precedenza per poter effettuare l'operazione di ricerca di patterns. Il passo successivo è la ricerca vera e propria. Effettuata la ricerca, il risultato di tale operazione viene mostrato all'utente: infatti i pattern trovati vengono evidenziati con un opportuno colore.
- **Cluster patterns:** questo comando permette di effettuare l'operazione di clusterizzazione sia su tutti i pattern evidenziati che solamente su quello selezionato.
- **Push Forward:** questo comando è del tutto simile al suo omonimo descritto nel precedente paragrafo.
- **Mark:** questo comando è del tutto simile al suo omonimo descritto nel precedente paragrafo.

5.3.4 Operazioni effettuabili su una sessione

Anche in questo caso esiste sia un menu di popup, presente nella modalità stand alone e in quella integrata nell'IDE, che un menu *Session* presente nella barra dei menu quindi reso disponibile solo nel primo caso. Il menu di popup viene visualizzando cliccando con il tasto destro del mouse in un punto della visualizzazione dove non sono presenti componenti né eventi.

Vediamo le funzionalità offerte:

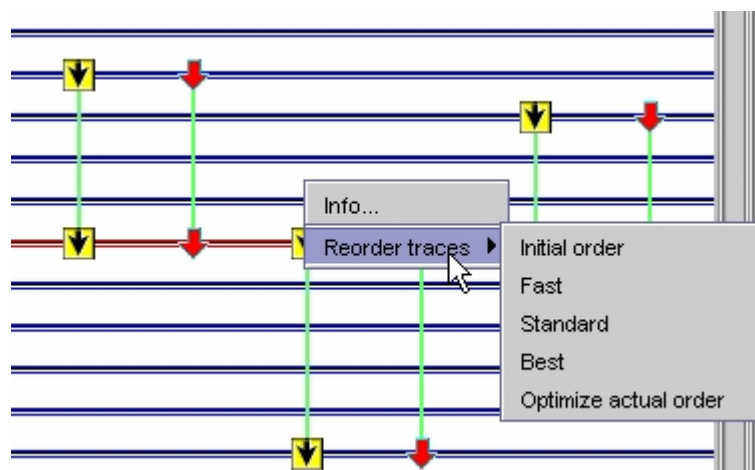


Figura 35: Popup menu associato ad una sessione

- **Info:** mostra una finestra con le informazioni generali sulla sessione: nome, numero di tracce, numero totale degli eventi. Per ogni traccia: nome, tipo, numero di eventi.
- **Reorder Traces:** è un sottomenu che fornisce varie opzioni per modificare l'ordine delle tracce su video. Selezionando uno di questi comandi lo strumento compie automaticamente gli aggiustamenti sulle posizioni dei simboli. Le opzioni sono le seguenti:
 - **Initial order:** dispone le tracce nello stesso ordine in cui l'oggetto *Session* le ha ricevute dal metodo *buildTraces*.
 - **Fast:** costruisce l'ordinamento seguendo l'algoritmo descritto nel lavoro di tesi [32], senza alcuna ottimizzazione.
 - **Standard:** è il metodo di ordinamento che lo strumento utilizza di default, quando mostra per la prima volta l'STD relativo alla sessione che costruisce. Si può desiderare riapplicare tale metodo se si sono spostate le tracce rispetto alla prima visualizzazione.
 - **Best:** costruisce l'ordinamento seguendo l'algoritmo descritto nel lavoro di tesi [32], con ottimizzazione.
 - **Optimize actual order:** cerca di applicare un algoritmo di ottimizzazione alla visualizzazione presente sullo schermo.

- **Change name:** mostra una finestra di dialogo da cui è possibile cambiare il nome della sessione. Tale comando è disponibile solo nella modalità stand alone.

5.3.5 Funzioni extra supportate dalla modalità stand alone

Abbiamo già esposto in precedenza che lo strumento grafico funzionante in modalità *stand alone* possiede una barra dei menu e una barra degli strumenti non replicate nel *View Panel* presente nell'IDE (Figura 36).

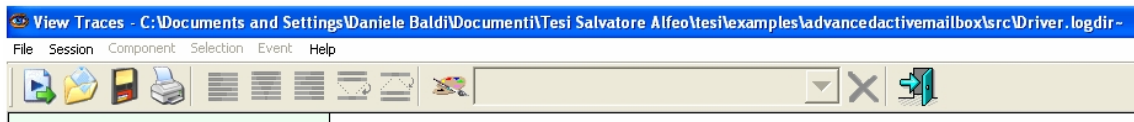


Figura 36: Barra dei menu e barra degli strumenti

Di seguito descriveremo i comandi di tali barre che non vengono ripetuti all'interno dei popup menu, e che quindi non sono stati descritti nell'elenco sopra esposto; tali comandi si trovano all'interno del menu *File* della barra dei menu. I comandi presenti all'interno del menu *File* sono tutti replicati sulla barra degli strumenti.

Vediamo i comandi del menu *File*:

File

- **New:** questo comando fa apparire una finestra di dialogo tramite la quale si può creare una nuova sessione; infatti come abbiamo già detto più volte la modalità stand alone, a differenza di quanto succede all'interno dell'IDE, parte senza alcuna sessione caricata. All'interno di tale finestra occorre inserire il nome della sottoclasse di *Session*, e l'argomento (una stringa) da passare al suo costruttore. Prima di utilizzare il nome di una sottoclasse esso va registrato. Tutto ciò che occorre fare è inserire il nome nella casella di testo *Register class* e premere il pulsante *Register*. Da questo istante il nome apparirà nella lista *Registered classes*, anche in esecuzioni successive dello strumento e per

scegliere il nome di una sottoclasse sarà sufficiente sceglierlo dalla lista. L'argomento da passare, va inserito nella casella di testo *Argument*. Se esso è il nome di una directory, ad esempio il nome della directory in cui sono presenti le tracce da tradurre, è possibile utilizzare il pulsante *Browse*, che attiverà la finestra di browsing tipica del sistema su cui si sta lavorando, si seleziona la directory voluta e si preme *OK*. Il nome della directory apparirà nella casella *Argument*. Una volta scelta la classe registrata e inserito l'argomento, si preme *OK* e lo strumento, dopo alcune elaborazioni, mostrerà l'STD relativo alla computazione. Tale comando ha un tasto di scelta rapida indicato nel menu stesso.

- **Save:** in Java è possibile salvare su disco qualsiasi oggetto appartenente ad una classe che implementi l'interfaccia *Serializable* (libreria *java.io*). Si veda una qualsiasi manuale Java [3] per maggiori dettagli sull'argomento. Tutte le classi che descrivono una sessione implementano tale interfaccia, per cui è possibile salvare su disco un'intera sessione, e caricarla in esecuzioni successive. Occorre però ricordare che con tale comando, lo strumento tenterà di salvare anche gli oggetti contenuti nel campo *originalEvent* di ciascun *SynEvent*. Tale operazione avrà successo solo se la classe di tali oggetti implementa l'interfaccia suddetta. Tale comando ha un tasto di scelta rapida indicato nel menu stesso.
- **Open:** con tale comando è possibile caricare una sessione precedentemente salvata. Tale operazione ha successo solo se nessuna delle classi che descrivono una sessione è stata modificata dal momento del salvataggio. Tale comando ha un tasto di scelta rapida indicato nel menu stesso.
- **Print current drawing:** stampa la parte visibile dell'STD corrente. Non si fa alcun controllo d'impaginazione, per cui la pagina stampata potrebbe non contenere tutto il disegno. Tale comando ha un tasto di scelta rapida indicato nel menu stesso.
- **Exit:** esce dal programma, chiedendo conferma, ma senza chiedere di salvare la sessione corrente. Tale comando ha un tasto di scelta rapida indicato nel menu stesso. Il comando *Exit* viene selezionato anche premendo il pulsante di chiusura della finestra principale.

5.4 Conclusioni

Questo capitolo ha avuto la funzione di Manuale d'Uso per lo strumento, descrivendo dapprima le procedure d'installazione sia della modalità stand alone che di quella integrata nell'IDE, poi mostrando quali sono le funzionalità offerte all'utente programmatore. Nel prossimo capitolo invece analizzeremo una particolare applicazione concorrente per evidenziare le caratteristiche principali dello strumento.

Capitolo 6

Caso d'uso dello strumento

Dopo aver descritto tutte le funzionalità dello strumento nel Capitolo precedente, qui si analizzerà l'utilizzo dello strumento per la visualizzazione di tracce d'esecuzioni contenenti gli eventi relativi ad alcuni tra i più importanti costrutti di sincronizzazione e comunicazione presenti nel linguaggio Java.

6.1 Tracce di eventi Java

Come già detto più volte è stato sviluppato, in un precedente lavoro di tesi [31], un modulo di NetBeans in grado di instrumentare opportunamente un'applicazione Java, in modo da ottenere delle tracce di esecuzione contenenti gli eventi relativi ad alcuni tra i più importanti costrutti di sincronizzazione e comunicazione presenti in tale linguaggio. Per permettere al nostro strumento la visualizzazione delle tracce prodotte si è dovuto implementare una sottoclasse di *Session* chiamate *JavaSession* che operasse la traduzione necessaria. L'esperienza è significativa, sia per gli aspetti relativi all'applicazione del modello di computazione sviluppato, ad eventi "veri", sia per i problemi incontrati nella traduzione delle tracce.

Per motivi di eterogeneità degli eventi trattati, e di sovrapposizione di due lavori di tesi indipendenti, le tracce da tradurre costituiscono un buon esempio della varietà di fronte alla quale ci si può trovare.

Gli eventi Java di cui abbiamo curato la modellazione sono di due tipi:

- Eventi di creazione di thread.
- Eventi relativi ai costrutti di sincronizzazione su oggetti condivisi.

L'evento di creazione di un thread è stato rappresentato con un *SynEvent* appartenente al thread creante, avente un partner passivo. Tale partner è il primo evento del thread creato. Tra l'altro tale evento esiste sempre, perché all'inizio delle tracce originali relative a ciascun thread sono presenti eventi fittizi che rappresentano proprio la partenza del thread.

In Java ogni oggetto possiede una variabile di lock, invisibile al programmatore. Tale variabile è gestita tramite un meccanismo tipo monitor. Se in una classe si definisce un metodo con la parola chiave *synchronized*, tale metodo diventa una procedura del monitor associato a ciascuna istanza di quella classe. E' anche possibile definire un'intera classe *synchronized*, operazione equivalente a definire ogni metodo come tale. Infine è possibile inserire, in qualsiasi punto del codice un blocco *synchronized*, con il riferimento all'oggetto su cui si vuole ottenere il lock durante l'esecuzione di quel blocco.

All'interno di un metodo o un blocco *synchronized*, si possono inserire istruzioni di *wait* e di *notify* sulla variabile di lock associata all'oggetto a cui il blocco si riferisce. Per maggiori delucidazioni su quanto esposto finora si veda la documentazione necessaria indicata nella bibliografia [3].

Gli eventi di sincronizzazione sui lock tracciati dallo strumento in questione sono i seguenti:

- Ingresso in un blocco *synchronized*.
- Inizio di una *wait*.
- Fine di una *wait*.
- Inizio di una *notify*.

- Fine di una *notify*.
- Inizio di una *notifyAll*.
- Fine di una *notifyAll*.
- Uscita da un blocco *synchronized*.

La differenza tra una *notify* ed una *notifyAll*, consiste nel fatto di risvegliare uno o tutti i thread eventualmente bloccati sul lock dell'oggetto.

Le tracce da tradurre sono così strutturate:

- Esiste una traccia per ogni thread in cui sono presenti solo gli eventi di creazione e partenza dei thread stessi.
- Esiste una traccia per ogni oggetto condiviso, in cui sono presenti gli eventi ad esso relativi, con l'indicazione del tipo di evento e del thread che ha effettuato ciascuna operazione.
- Ciascuna traccia è su di un file separato.
- Tutti i file risiedono in una directory, il cui nome viene passato come argomento a *JavaSession*.
- I due tipi di tracce sono distinguibili in base alla struttura del nome del file in cui sono memorizzate.

Il lavoro di traduzione consiste quindi nel creare una traccia per ciascun oggetto, ed una per ciascun thread, aggiungendo nella traccia del thread (e nella posizione) opportuno, il partner sincrono di ciascun evento relativo ad ogni oggetto.

6.2 Esempio

In questo paragrafo descriveremo un esempio di visualizzazione di tracce di esecuzioni di applicazioni concorrenti che utilizzano i costrutti di sincronizzazione e comunicazione suddetti: inoltre utilizzando i meccanismi d'astrazione argomentati in questo lavoro faremo vedere come sia possibile ridurre notevolmente la complessità

della visualizzazione, e come queste tecniche permettano un'analisi ad un più alto livello.

Nello sviluppare tale discussione presenteremo immagini che raffigurano l'utilizzo dello strumento integrato nell'IDE.

6.2.1 AdvancedMailbox

L' applicazione concorrente che analizziamo si chiama *AdvancedMailbox*: è stata scelta questa perché, oltre ad essere un problema molto noto in letteratura, è critica dal punto di vista della densità degli eventi. Essa è costituita da 10 *Mailbox* a 5 posizioni, da 10 thread consumatori (*Consumer*) e da 10 thread Produttori (*Producer*) per ogni mailbox che effettuano ognuno rispettivamente 10 consumazioni e 10 produzioni a rotazione sulla mailbox assegnatagli.

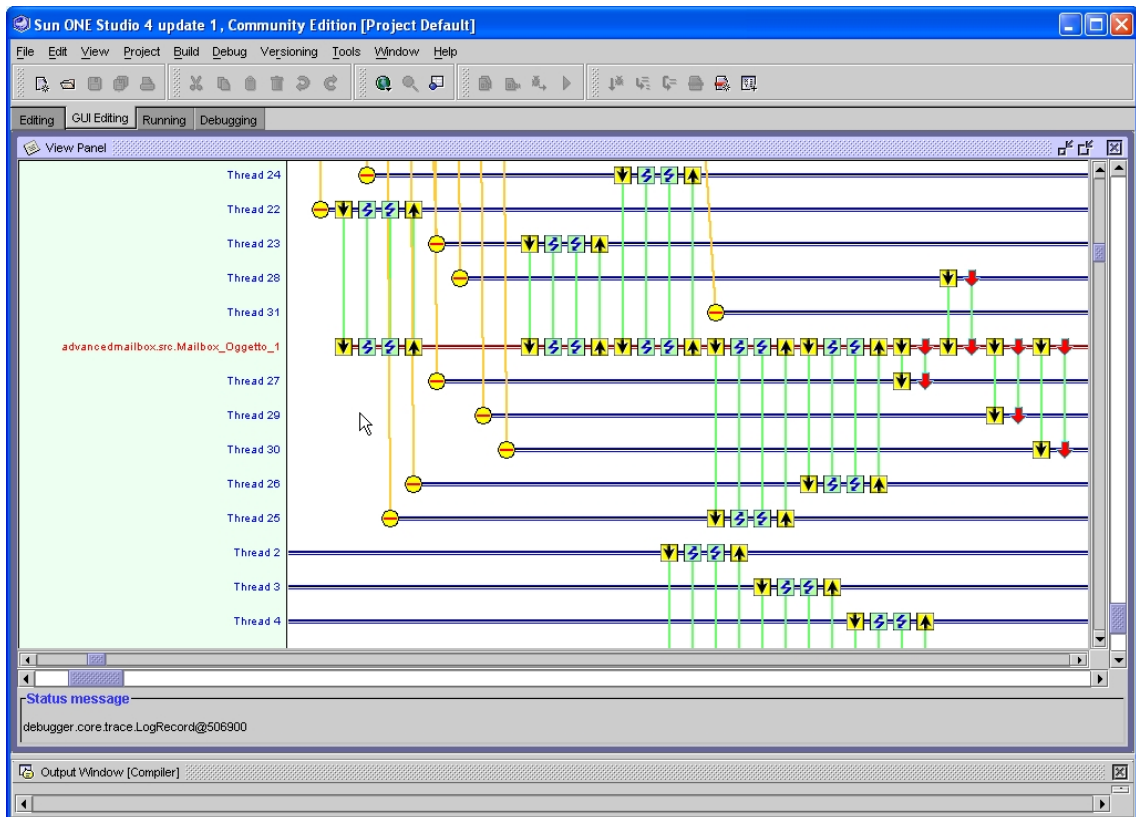
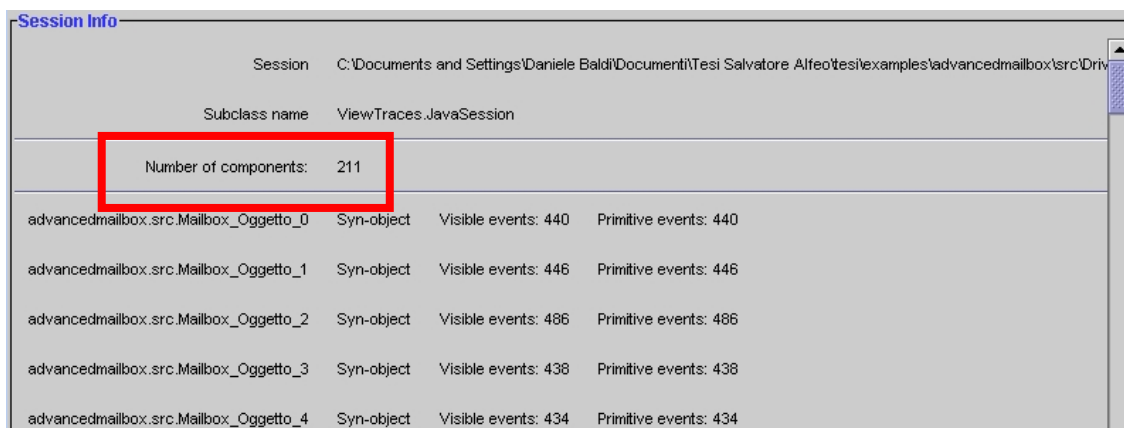


Figura 37: Visualizzazione tracce di AdvancedMailbox

La Figura 37 mostra una parte delle tracce di eventi dell'esecuzione di *AdvancedMailbox*; in tale visualizzazione si possono osservare le interazioni tra i vari componenti al più basso livello d'astrazione.

L'esecuzione di tale applicazione crea delle tracce contenenti 21013 eventi primitivi e 211 componenti (Figure 38, 39).



The screenshot shows a 'Session Info' window with the following data:

Session	Subclass name	Number of components	Visible events	Primitive events
C:\Documents and Settings\Daniele Baldi\Documents\Tesi Salvatore Alfeo\tes\examples\advancedmailbox\src\Driv	ViewTraces.JavaSession	211		
advancedmailbox.src.Mailbox_Oggetto_0	Syn-object		440	440
advancedmailbox.src.Mailbox_Oggetto_1	Syn-object		446	446
advancedmailbox.src.Mailbox_Oggetto_2	Syn-object		486	486
advancedmailbox.src.Mailbox_Oggetto_3	Syn-object		438	438
advancedmailbox.src.Mailbox_Oggetto_4	Syn-object		434	434

Figura 38: Numero di componenti



The screenshot shows a list of threads with the following data:

Thread	Sequential	Visible events	Primitive events
Thread 195	Sequential	43	43
Thread 196	Sequential	43	43
Thread 197	Sequential	63	63
Thread 198	Sequential	63	63
Thread 199	Sequential	63	63
Thread 200	Sequential	63	63
Thread 201	Sequential	63	63
Total number of visible events:		21013	
Total number of primitive events:		21013	

Figura 39: Numero totale di eventi primitivi

Data la struttura dell'applicazione descritta, il numero di thread effettivamente in esecuzione dovrebbe corrispondere al numero di mailbox moltiplicato per venti (dieci produttori e dieci consumatori). Quindi nella nostra analisi i componenti mostrati dallo strumento grafico dovrebbero essere 210 (200 threads e 10 risorse condivise).

In realtà i componenti sono 211; infatti è presente anche un thread *Driver*, contenente il *main* dell'applicazione, che si occupa della creazione dei threads consumatori e produttori (vedi Figura 40).

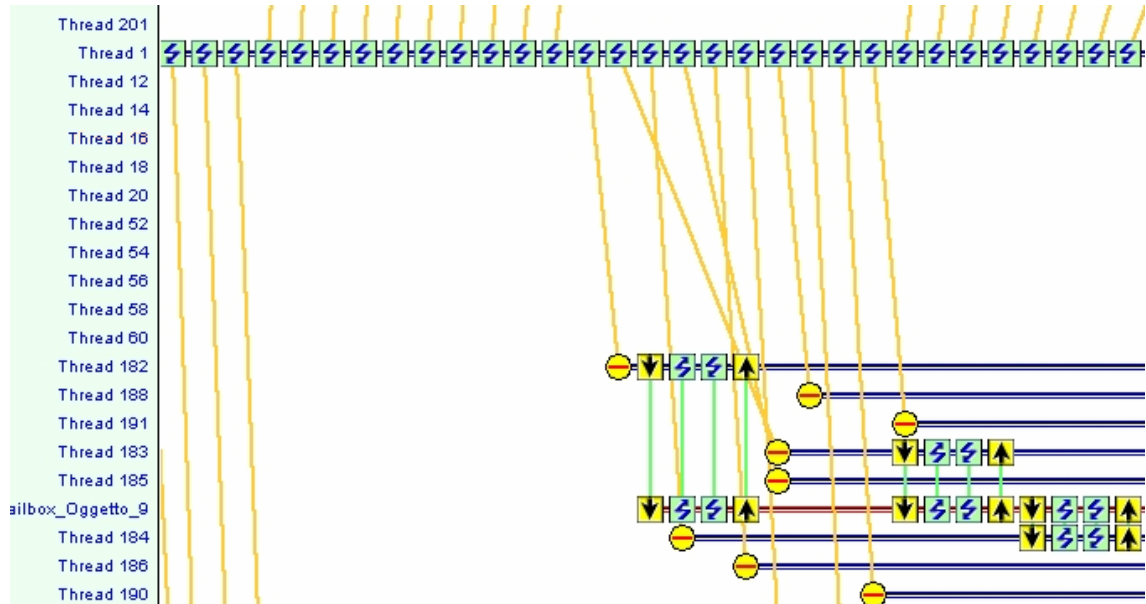


Figura 40: Creazione dei threads produttori e consumatori

Analisi di AdvancedMailbox ad un livello d'astrazione più alto

Come già spiegato nei precedenti capitoli può essere utile per l'utente avere una visualizzazione più astratta della computazione utilizzando le tecniche di astrazione e di ricerca di pattern.

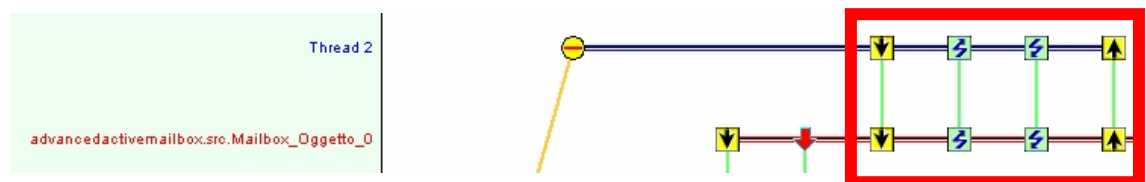


Figura 41: Insieme di eventi primitivi che intendiamo astrarre

L'insieme di eventi primitivi che intendiamo astrarre è mostrato in Figura 41.

Tale evento astratto corrisponde all'esecuzione da parte dei threads consumatori e produttori, dei metodi *synchronized public int get()* e *synchronized public void put(int*

data) della classe *Mailbox*, senza alcuna attesa dovuta al fatto che la casella di posta sia vuota o piena.

Si è scelto tale insieme di eventi perché si è notato, nelle figure sopra mostrate, che tale pattern si ripete molto spesso all'interno della computazione concorrente.

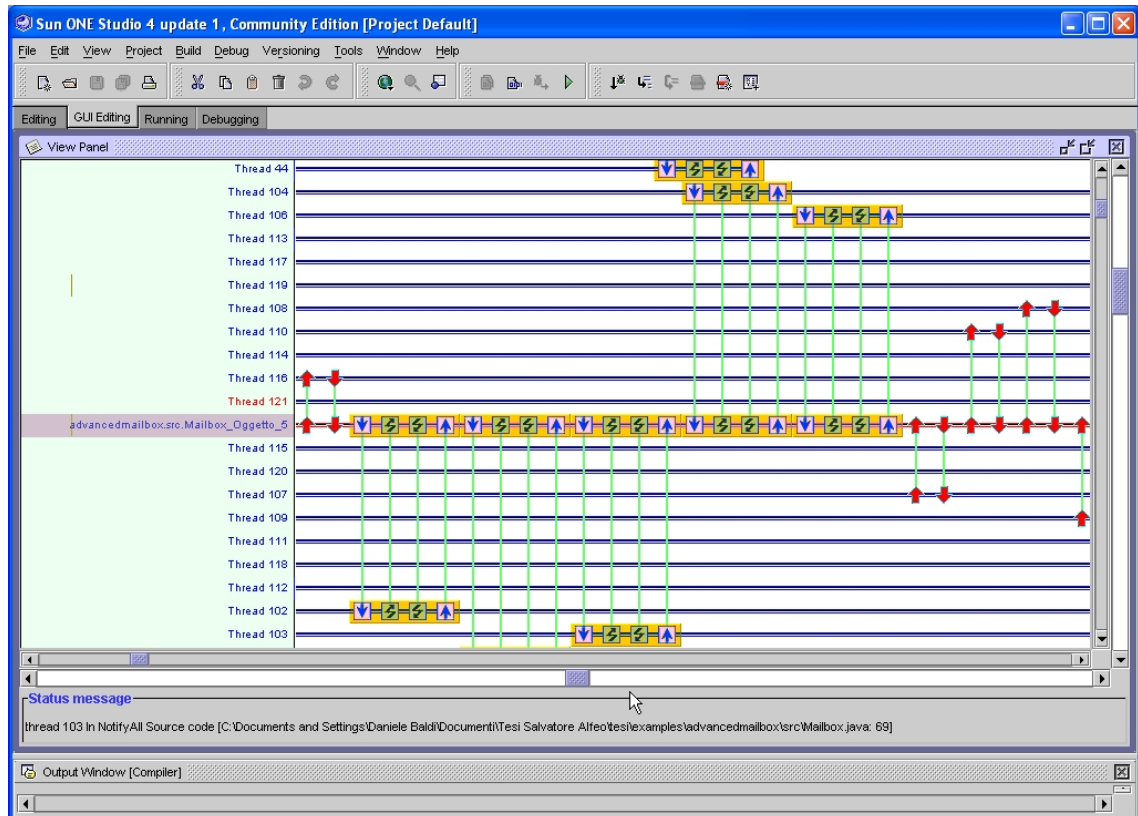


Figura 42: Pattern matched in AdvancedMailbox

In Figura 42 sono evidenziati tutti i patterns trovati all'interno delle tracce di esecuzione di *AdvancedMailbox* con l'ausilio dell'algoritmo di *Pattern Matching*. In totale sono ben 1803.

Tutti questi patterns sono stati astratti in modo tale che ognuno di essi formasse un unico evento; facendo ciò si è ridotta notevolmente la complessità dell'STD (Figura 43).

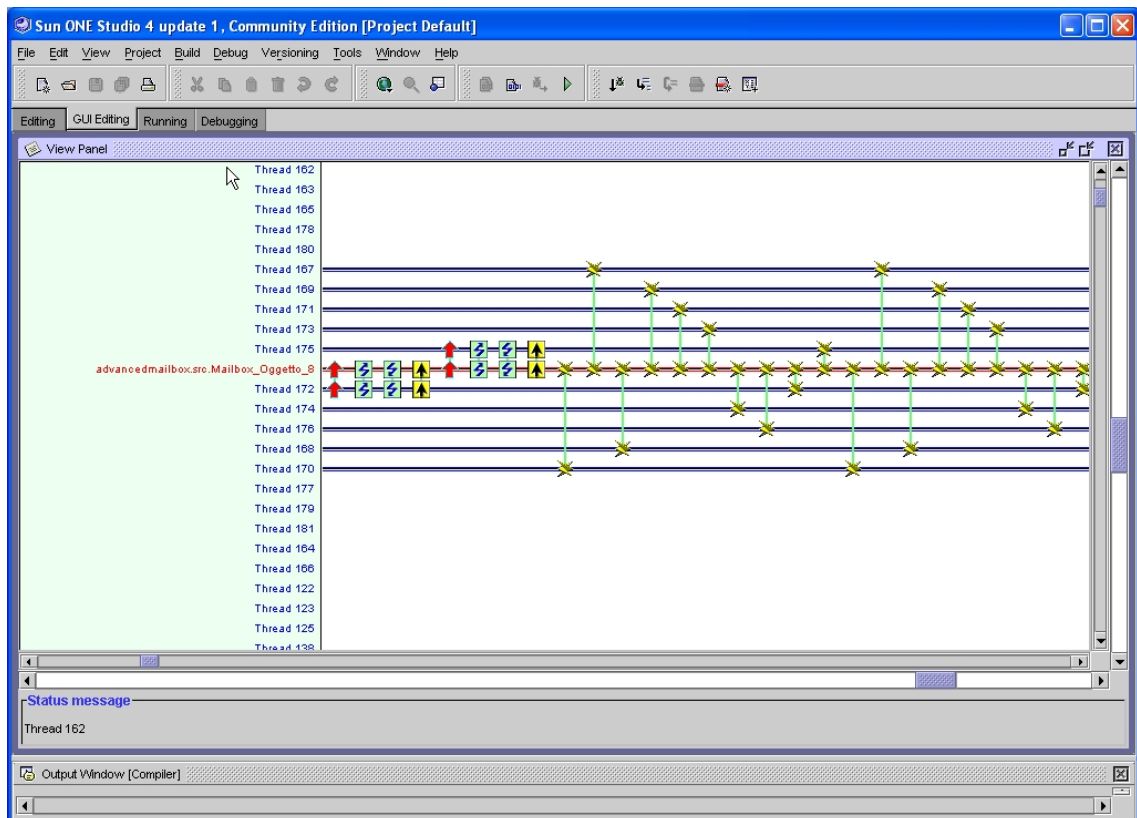


Figura 43: Visualizzazione delle tracce dopo astrazione patterns

Dopo tale operazione si è notato che il numero di eventi visualizzati è passato da 21013 a 10195 (Figura 44).

Thread 200	Sequential	Visible events: 36	Primitive events: 63
Thread 201	Sequential	Visible events: 36	Primitive events: 63
Total number of visible events:		10195	
Total number of primitive events:		21013	

Figura 44: Numero di eventi dopo astrazione patterns

6.3 Conclusioni

Nella prima parte di questo capitolo abbiamo descritto il tipo di tracce soffermandoci in maniera particolare sulle caratteristiche degli eventi Java registrati. Poi una parte importante del capitolo è stata dedicata alla descrizione dei vantaggi introdotti dalle tecniche d'astrazione, descritte nei precedenti capitoli, facendo uso di tracce prodotte dall'esecuzione di una particolare applicazione.

Conclusioni

Il presente lavoro di tesi ha come risultato finale l'implementazione di uno strumento di analisi di programmi concorrenti che permetta la visualizzazione di tracce di esecuzioni di una applicazione in cui siano presenti più processi/thread che collaborano/competono tra di loro. Lo strumento è stato specializzato per la visualizzazione di tracce che registrano informazioni su eventi di comunicazione e sincronizzazione Java, ma l'approccio seguito ne permette un riutilizzo completo per qualsiasi altro tipo di traccia. Avere a disposizione tracce di esecuzione da analizzare può essere utile a capire se e dove esistono delle interferenze che degradano le prestazioni. Data l'importanza di questi problemi, avere a disposizione uno strumento di analisi (per di più dotato di una interfaccia grafica) permette di aumentare la produttività e di ridurre i tempi di sviluppo del software.

L'approccio teorico seguito ha messo in evidenza che il modo migliore per analizzare i dati raccolti all'interno delle tracce è mostrarli all'interno di diagrammi spazio tempo discreti (STD) in cui sono messe in luce le relazioni temporali tra gli eventi di sincronizzazione e comunicazione. Per migliorare la comprensione dell'applicazione che si sta analizzando sono state poi introdotte delle tecniche d'astrazione che permettono di ridurre il numero di eventi visualizzati.

L'applicazione espone i propri servizi sia in modalità *stand alone*, sia tramite una interfaccia grafica sviluppata come modulo plug-in per la piattaforma NetBeans su cui sono basati vari ambienti integrati di sviluppo, in primis SunOne Studio. In virtù di questa scelta, il prodotto risulta di immediato utilizzo pratico e disponibile per una

grande utenza: si cerca in questo modo di colmare la mancanza di uno strumento di analisi di programmi concorrenti senza introdurre un applicativo ad hoc che incontrerebbe maggiori difficoltà di inserimento nelle comunità di sviluppatori.

Lo strumento non è stato progettato prendendo particolari precauzioni per migliorare l'utilizzo della memoria, risorsa che, seppure abbondante negli odierni sistemi, rimane la più preziosa per problemi come quello affrontato.

La flessibilità dello strumento dovrà essere sfruttata per adattarlo alla soluzione di problemi specifici. Tale operazione ha ovviamente senso solo nell'ambito di un problema reale, che necessiti di uno strumento semanticamente simile all'applicazione per la quale viene utilizzato. Anche in questo senso, speriamo di aver indicato la strada da percorrere.

Bibliografia

- [1] J. Gosling, B. Joy, G. Steele, *“The Java language specification”*, Addison Wesley 1996
- [2] K. Arnold, J. Gosling, *“The Java programming language”*, Addison Wesley, 1996
- [3] B. Eckel, *“Thinking in Java”*, Prentice Hall PTR, 1998
- [4] S. Oaks, H. Wong,, *“Java threads”*, O Reilly 1997
- [5] D. Flanagan, *“Java in a nutshell”*, O Reilly 2002
- [6] J. Zukowski, *“Java AWT reference”*, O Reilly 1997
- [7] T. Boudreau et al., *“NetBeans, the definitive guide”*, O’Reilly, 2002
- [8] Martin Fowler, *“UML distilled”* Addison-Wesley, 1997
- [9] Martin Fowler, *“Applying UML and patterns”*, Prentice-Hall, 1997
- [10] P. Ancilotti e M. Boari, *“Principi e tecniche di programmazione concorrente”*, Utet Libreria, 1987

- [11] C. A. R. Hoare, “*Monitors: an operating-system structuring concept*”, Comm. ACM, Ott.1974
- [12] R. H. B. Netzer e B. P. Miller, “*What are race conditions? Some issues and formalizations*”, ACM Letters on Programming Language and Systems, vol.1,no. 1, March 92
- [13] L. Lamport, “*Time, clocks and the ordering of events in a distributed system*”, Comm. ACM, Vol. 21, no. 7, Jul. 1978
- [14] T. Kunz, J. P. Black, D. "J. Taylor, T. Basten, “*Poet: Target-System Independent Visualizations of Complex Distributed-Application Executions*”, The Computer Journal, Vol 40, no 8, 1997
- [15] Dean F. Jerding, John T. Stasko, Thomas Ball, “*Visualizing Interactions in Program Executions*”, 1997
- [16] D. Kranzlmuller, S. Grabner, J. Volkert, “*Event Graph Visualization for Debugging Large Applications*”, 1996
- [17] Sameer Shende, Allen D. Malony, “*Integration and application of Tau in parallel Java environments*”, 2003
- [18] J. Chassin de Kergommeaux, B. Stein, P. E. Bernard, “*Pajè, an interactive visualization tool for tuning multi-threaded parallel applications*”, 2000
- [19] Thomas Kunz, “*Process Clustering for Distributed Debugging*”
- [20] Michiel F.H. Seuren, “*Design and Implementation of an Automatic Event Abstraction Tool*”, Waterloo, Ontario, Canada, 1996

- [21] Thomas Kunz, “*Abstract behaviour of Distributed Executions with Applications to Visualizatin*”, 1994
- [22] Thomas Kunz, “*An Event Abstraction Tool: Theory, Design and Results*”, Gennaio 1994
- [23] Thomas Kunz, “*Visualizing Abstract Events*”, Toronto, Novembre 1994
- [24] Thomas Kunz, “*High-Level Views of Distributed Executions*”, St Malo, Francia, Maggio 1995
- [25] Dieter Kranzmuller, “*Event Graph Analysis for debugging massively Parallel Programs*”, Linz, Austria, Settembre 2000
- [26] Dieter Kranzmuller, “*Scalable Parallel Program Analysis*”, Linz, Austria, Ottobre 2002
- [27] H. Hallal, S. Boroday, A. Ulrich, A. Petrenko, “*An Automata-based Approach to Testing Properties in Event Traces*”, Sophia Antipolis, France, May 2003
- [28] G. H. Hwang, K. C. Tai, T. L. Huang, “*Reachability testing: an approach to testing concurrent software*”, International Journal of Software Engineering and Knowledge Engineering Vol. 5 No. 4, 1995
- [29] R. D. Yang e C. G. Chung, “*Path analysis testing of concurrent programs*”, Information and Software Technology, Vol. 34, no 1,Jan. 1992
- [30] R. H. Carver e K. C. Tai, “*Static analysis of concurrent software for deriving synchronizations constraints*”, Proc. of IEEE Int. Conf. On Distributed Systems, May1991

- [31] Tesi di Laurea di Salvatore Alfeo (studente della facoltà di ingegneria dell'università degli studi di Pisa), 2003
- [32] Tesi di Laurea di Luca Billi studente della facoltà di ingegneria dell'università degli studi di Pisa), 1999
- [33] K. C. Tai, R. H. Carver, E. E. Obaid, "*Debugging concurrent Ada programs by deterministic execution*", IEEE Trans. Software Engineering, Vol. 17, No. 1, Jan 1991, pp.45-63
- [34] H. Krawczyk, B. Wiszniewsky, "*Analysis and testing of distributed applications*", Research Study Press Ltd, 1998
- [35] L.J. White, "*General overview of software testing*", Software Testing: Methods & Techniques, Riva dei Tessali, vol.34, no. 1, Jan.92
- [36] R. N. Taylor, D. L. Levine, and C. D. Kelly, "*Structural testing of concurrent programs*", IEEE Transactions on Software Engineering, Vol.18,no 3, March 1992
- [37] R. Chow, T. Johnson, "*Distributed operating systems & algorithms*", Addison Wesley, 1997
- [38] K. C. Tai e R. C. Carver, "*Handbook of parallel and distributed computing*", Editor Albert Zomaya, McGraw Hill, 1995
- [39] M. S. Deutsch and R. R. Willis, "*Software quality engineering: a total technical and management approach*", Prentice Hall Series in Software Engineering, Englewood Cliffs, NJ, 1988
- [40] J. Gait, "*A probe effect in concurrent programs*", Software-Practice and Experience, Vol.16, No. 3, March 1986, pp. 225-233

- [41] Sun Microsystems, “*Java product versioning specification*”, White Paper, 1998
disponibile sul sito <http://www.java.sun.com>
- [42] J.Gosling, H.McGilton, “*The Java language enviroment*”, White Paper 1996
disponibile sul sito <http://www.java.sun.com>
- [43] D. Kramer, “*The Java platform*”, a White Paper by JavaSoft 1997 disponibile al
sito <http://www.java.sun.com>
- [44] “*The Java Tutorial*”, disponibile sito <http://www.java.sun.com>