



Bachelorarbeit

Eclipse-Plugin: Visualisierung von Threadeigenschaften

Le-Huan Stefan Tran

Aufgabensteller: Prof. Dr.-Ing. Gregor Snelting
Lehrstuhl für Programmierparadigmen
Institut für Programmstrukturen und Datenorganisation (IPD)

Betreuer: Dipl.-Inf. Univ. Jürgen Graf
Lehrstuhl für Programmierparadigmen
Institut für Programmstrukturen und Datenorganisation (IPD)

Abgabetermin: 31. Juli 2011

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, 31. Juli 2011

.....
Le-Huan Stefan Tran

Zusammenfassung

Die Beherrschung nebenläufiger Programme wird für den heutigen Software-Entwickler immer wichtiger. Die Verwendung moderner Programmiersprachen wie Java in Verbindung mit Threads führt jedoch zu Interferenzen zwischen nebenläufigen Programmabschnitten. Analysewerkzeuge helfen beim Verständnis und bei der Erkennung dieser Interferenzen; deren Ergebnisse werden bisher jedoch nur intern vorgehalten.

Wir präsentieren in dieser Arbeit ein intuitiv zu bedienendes Plugin für Eclipse, das auf den Analysen des ValSoft/Joana-Projekts des Instituts für Programmstrukturen und Datenorganisation (IPD) am Karlsruher Institut für Technologie (KIT) basiert und alle Threads sowie deren Interferenzen visualisiert. Das Plugin *ThreadViewer* extrahiert dazu die wichtigsten Ergebnisse aus den Analysen, erweitert diese um weiterführende Berechnungen und verwendet anschließend das Zest-Framework zur ansprechenden Visualisierung. Der Fokus wird dabei auf eine nahtlose Integration in Eclipse gelegt.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Aufgabenstellung	1
1.2. Fallbeispiel	1
1.3. Unser Beitrag	3
1.4. Aufbau der Arbeit	3
2. Grundlagen	5
2.1. Analyse sequenzieller Programme	5
2.1.1. Der Kontrollflussgraph	5
2.1.2. Der Systemabhängigkeitsgraph	7
2.2. Analyse paralleler Programme	8
2.2.1. Die Thread Invocation-Analyse	10
2.2.2. Der Kontrollflussgraph mit Threads	10
2.2.3. Die MHP-Analyse	11
2.2.4. Der Systemabhängigkeitsgraph für nebenläufige Programme	17
3. Implementierung	21
3.1. Das Plugin “ThreadViewer”	21
3.1.1. Verwendung von Informationen	21
3.1.2. Unterstützte Anzeigen	22
3.1.3. Mögliche Benutzerinteraktionen	22
3.2. Herausforderungen des Model-View-Controllers	23
3.2.1. Herausforderungen des Models	24
3.2.2. Herausforderungen der View	28
3.2.3. Herausforderungen des Controllers	32
3.3. Grenzen der Implementierung	34
3.3.1. Grenzen des Models	35
3.3.2. Grenzen der View	35
3.3.3. Grenzen des Controllers	36
3.4. Statistik	36
4. Verwendung von ThreadViewer	37
4.1. Bedienungsanleitung	37
4.2. Impressionen aus ThreadViewer	47
5. Evaluation	53
5.1. Testaufbau	53
5.1.1. Testszenarien	53
5.1.2. Testfälle	54
5.2. Laufzeiten	54

5.3. Visualisierte Thread Regionen	55
6. Fazit und Ausblick	57
6.1. Einsatzmöglichkeiten	57
6.1.1. Entwicklung nebenläufiger Anwendungen	57
6.1.2. Plausibilitätsprüfung für Slicing und Chopping	57
6.2. Verwandte Arbeiten	58
6.2.1. CodeSurfer	58
6.2.2. GraphViewer	58
6.3. Ausblick	59
A. CSDGs in ThreadViewer	61
Abbildungsverzeichnis	67
Listings	69
Algorithmenverzeichnis	71
Literaturverzeichnis	73

1. Einleitung

Um das Potenzial herkömmlicher Desktop-PCs optimal zu nutzen, sind heutige Software-Entwickler gefordert, nebenläufige Programme für Mehrkernprozessoren zu schreiben. Moderne Programmiersprachen wie Java[4] bieten dafür eine natürliche Unterstützung für sogenannte *Threads* an.

Diese potenziell parallel laufenden Ausführungsstränge verfügen über einen gemeinsamen Speicher, wodurch Wechselwirkungen von Programmabschnitten über Methoden- und Threadgrenzen hinweg möglich sind. Diese *Interferenzen* genannten Abhängigkeiten und deren Auswirkungen auf das Programm sind für den Entwickler allerdings ohne Weiteres nicht ersichtlich. Hier setzt die vorliegende Bachelorarbeit an. Mit einer Visualisierung wird dem Benutzer eine intuitive Möglichkeit gegeben, mögliche Interferenzen zwischen den Threads eines Java-Programms zu erkennen.

1.1. Aufgabenstellung

Die Analysen des *ValSoft/Joana-Projekts*[6] am Institut für Programmstrukturen und Datenorganisation (IPD) am Karlsruher Institut für Technologie (KIT) erkennen potenziell parallel laufende Threads sowie mögliche Interferenzen zwischen ihnen. Jedoch werden diese für den Entwickler wertvollen Informationen momentan nur in internen Datenstrukturen vorgehalten.

In dieser Bachelorarbeit wird ein Forschungsprototyp in Form eines *Eclipse-Plugins*[5] entwickelt, das die verschiedenen Threads eines Java-Programms und deren Interferenzen aus den Rohdaten extrahiert, aufbereitet und mithilfe geeigneter Visualisierungstechniken anzeigt. Eine enge Kopplung mit dem Eclipse-Editor ermöglicht dabei einen intuitiven Wechsel zwischen dem Visualisierungs-Plugin und dem Quellcode der betreffenden Programmabschnitte.

1.2. Fallbeispiel

Als durchgängiges Fallbeispiel dient ein leicht modifiziertes Java-Programm aus der Dissertation von Giffhorn[11], das anhand von zwei Threads die Nebenläufigkeit in Java demonstriert.

Das nachfolgend gelistete Programm `TimeTravel.java` besteht aus zwei Threads, die zum Teil parallel laufen. Der Hauptthread `main` instanziiert einen Thread der Klasse `Thread1`. Beide Threads greifen auf das Attribut `d` zu, wodurch Interferenzen entstehen.

```
1 package conc;
2
3 public class TimeTravel {
4     Data d = new Data(11);
5
6     public static void main(String [] args) {
7         new TimeTravel().m();
8     }
9
10    public void m() {
11        Thread1 t = new Thread1();
12        t.data = d;
13        t.start();
14
15        System.out.println(d.x);
16        d.x = 0;
17        System.out.println(d.x);
18    }
19
20    static class Thread1 extends Thread {
21        Data data;
22
23        public void run() {
24            int a = data.x + 4;
25            data.x = data.x * a;
26        }
27    }
28
29    static class Data {
30        int x;
31
32        public Data(int y) {
33            x = y;
34        }
35    }
36 }
```

Listing 1.1: Der Quellcode von `TimeTravel.java`

Der Quellcode in 1.1 veranschaulicht das von Krinke[18] als *time travel* bezeichnete Phänomen bei der Berechnung von Interferenzen.

Nach der Selbstinstanziierung in Zeile 7 gibt der Hauptthread den aktuellen Wert der Variable `x` in Zeile 15 aus. Nach dem Rücksetzen des Wertes von `x` wird in Zeile 17 wiederum der Wert von `x` ausgegeben. In der Zwischenzeit wird jedoch in den Zeilen 11–13 ein zweiter Thread gestartet, der mittels einer zugewiesenen Referenz auf dasselbe Objekt `data` zugreift wie der Hauptthread und dieses in Zeile 25 modifiziert. Damit sind die Ausgabe-Anweisungen in Zeile 15 und 17 abhängig vom Zeitpunkt der Berechnung in Zeile 25—die Anweisungen in Zeile 15 und 25 sowie in Zeile 17 und 25 *interferieren*. Durch die sequenzielle Reihenfolge der beiden Ausgabe-Anweisungen schließen sich beide Interferenzen jedoch gegenseitig aus. Wenn die Berechnung mit der zweiten Ausgabe interferiert, müsste der zweite Thread “in der Zeit zurückreisen”, um wiederum mit der ersten Ausgabe interferieren zu können. Analog,

wenn die Berechnung mit der ersten Ausgabe interferiert.

Für die Erkennung solcher *zeitsensitiven* Interferenzen haben sowohl Krinke[19] als auch Nanda und Ramesh[20] zeitsensitive Algorithmen entwickelt, auf die später in 6.1.2 als mögliches Einsatzgebiet des Visualisierungstools eingegangen wird. Die Unterscheidung in zeitsensitive und zeitinsensitive Interferenzen ist für die Bachelorarbeit nebensächlich. Für die Visualisierung sind vielmehr *alle potenziellen* Interferenzen zwischen Threads von Bedeutung.

1.3. Unser Beitrag

Selbst bei kleinen Java-Programmen sind die verschiedenen Threads und ihre Interferenzen ohne Weiteres nicht ersichtlich. Diese Interferenzen können ungewünschte Seiteneffekte und ein fehlerhaftes Programmverhalten verursachen. Die zu implementierende Visualisierung soll dem Benutzer zeigen, aus welchen Threads sein Programm besteht und welche Interferenzen zwischen welchen Threads bestehen. Er soll intuitiv zu den interferierenden Programmanteilen navigieren können. Das setzt den Benutzer in die Lage, sein Programm besser zu verstehen.

1.4. Aufbau der Arbeit

Das folgende Kapitel der vorliegenden Bachelorarbeit behandelt die zugrundeliegende Theorie der zu visualisierenden Interferenzen—insbesondere die Grundlagen zu Kontrollfluss- und Abhängigkeitsgraphen sowie Thread Regionen. Das dritte Kapitel stellt neben dem Plugin ausgewählte Besonderheiten und Herausforderungen bei der konkreten Implementierung dar. Anhand einer bebilderten Anleitung wird im vierten Kapitel die Verwendung des Plugins gezeigt.

In der obligatorischen Evaluation im fünften Kapitel wird die Funktions- und Leistungsfähigkeit des Plugins anhand verschiedener Testszenarien geprüft. Das sechste und letzte Kapitel verweist nach einem abschließenden Fazit auf thematisch verwandte Arbeiten sowie die Einsatzmöglichkeiten der finalen Visualisierung.

2. Grundlagen

Das Mittel der Wahl zur Modellierung komplexer Programme in der Informatik stellen Graphen dar. Dieses Kapitel gibt einen grundlegenden Überblick über in der Wissenschaft verbreitete Datenstrukturen zur Abbildung sequenzieller und paralleler Programme, die die Grundlage zur Berechnung von Threadeigenschaften darstellen.

Der erste Abschnitt präsentiert den Kontrollflussgraphen als Ausgangspunkt für die Analyse von Abhängigkeiten sequenzieller Programme. Im zweiten Abschnitt werden diese Konzepte auf die Evaluation nebenläufiger Programme erweitert. Insbesondere die Berechnung von Thread Regionen als potenziell parallel laufende Ausführungs-Teilstränge durch eine sogenannte *may happen in parallel*-Analyse (MHP) bildet die Basis für die Implementierung der Visualisierung.

2.1. Analyse sequenzieller Programme

Sequenzielle Programme bestehen aus einer Menge aufeinanderfolgender Methodenaufrufe, deren Anweisungen voneinander abhängen können. Dementsprechend intuitiv lassen sich Kontrollflüsse und Abhängigkeiten solcher Programme mit Graphen zum Zwecke weiterführender Analysen modellieren.

In 2.1.1 wird der *Kontrollflussgraph* eingeführt, um Kontrollflüsse sequenzieller Programme zu modellieren. Abhängigkeiten zwischen Anweisungen werden in 2.1.2 durch den Systemabhängigkeitsgraphen beschrieben.

2.1.1. Der Kontrollflussgraph

Die möglichen Ausführungsreihenfolgen der Anweisungen eines Programms werden meist durch einen *Kontrollflussgraphen* (control flow graph, CFG) repräsentiert[11]. Dabei bilden die Knoten dieses Graphen die Programmanweisungen ab, während eine Kante zwischen zwei Knoten die aufeinanderfolgende Ausführung dieser Knoten bedeutet. Eine Beschriftung der Kanten ist möglich, um beispielsweise Kontrollflusskanten für die Behandlung von Exceptions auszuzeichnen. Für diese Arbeit ist eine Beschriftung der Kanten jedoch nicht von Relevanz und wird daher nicht berücksichtigt.

Definition 2.1 (Kontrollflussgraph (CFG)). *Der Kontrollflussgraph $G = (N, E, s, e)$ eines methodenlosen Programms oder einer Methode m ist ein gerichteter Graph mit den folgenden Eigenschaften:*

- N ist eine Menge von Knoten als Repräsentation der Anweisungen von p .
- E ist eine Menge von Kanten als Repräsentation des Kontrollflusses zwischen den Knoten.
- s mit $s \in N$ ist der Startknoten, von dem aus alle Knoten in N erreichbar sind und der selbst keine eingehende Kante besitzt.

```

4 Data d = new Data(11);
5
6 public static void main(..) {
7     new TimeTravel().m();
8 }
10 public void m() {
11     ...
16     d.x = 0;
17     System.out.println(d.x);
18 }

```

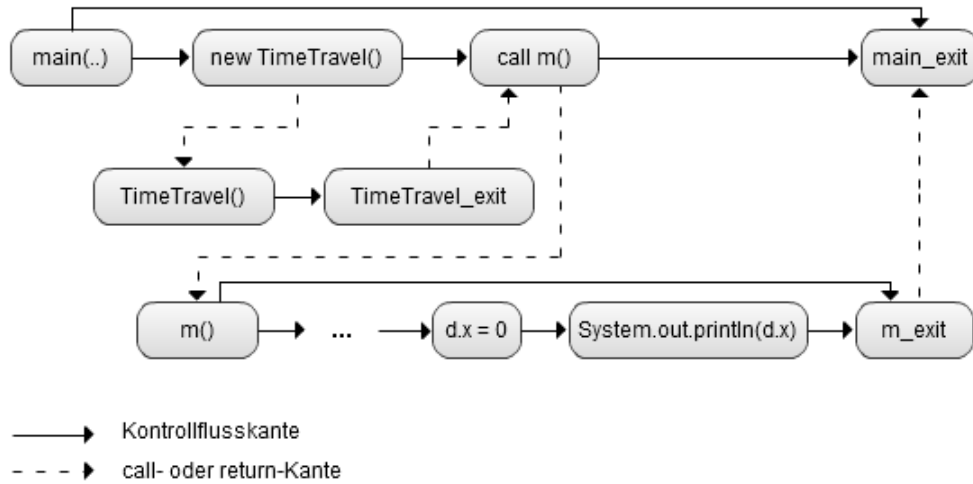


Abbildung 2.1.: Der ICFG eines Ausschnittes von TimeTravel.java

- e mit $e \in N$ ist der Zielknoten, der von allen Knoten in N erreichbar ist und der selbst keine ausgehende Kante besitzt.

Um den Umgang mit CFGs zu erleichtern, wird jedem CFG ein zusätzlicher Zielknoten hinzugefügt, der das Ende des CFG markiert und frei von Variablenzugriffen ist. Dieser Zielknoten ist über eine Kontrollflusskante direkt mit dem Startknoten verbunden. Darüber hinaus verbinden Kontrollflusskanten die call- mit den return-Knoten. Diese zusätzlichen Kanten vereinfachen die Berechnung des Kontrollflusses.

Um den Kontrollfluss eines Programms mit mehreren Methoden abzubilden, wird der *interprozedurale Kontrollflussgraph* (interprocedural control flow graph, ICFG) verwendet[11]. Die CFGs der einzelnen Methoden werden dabei mittels *call-* und *return-Kanten* zu einem Graphen verbunden. Eine call-Kante entspricht dabei einem Aufruf einer Methode; eine return-Kante entspricht der Rückkehr von derselben.

Definition 2.2 (Interprozeduraler Kontrollflussgraph (ICFG)). *Der interprozedurale Kontrollflussgraph $G = (CFG_q, main, Call)$ eines Programms q besteht aus einer Menge CFG_q von CFGs der Methoden von q , der Hauptmethode 'main' von q und einer Menge Call als Paare von call- und return-Kanten mit den folgenden Eigenschaften:*

- Die Knotenmengen N_{p_i}, N_{p_j} sowie die Kantenmengen E_{p_i}, E_{p_j} sind für alle Paare p_i, p_j von Methoden von q disjunkt.
- Sei c_p der Knoten der Methode p , der einen Aufruf der Methode p' repräsentiert. Dann existiert eine Aufrufstelle $(c_p \rightarrow_{call} s'_p, e'_p \rightarrow_{return} c'_p) \in Call$, für die gilt:

- $c_p \rightarrow_{\text{call}} s'_p$ ist die call-Kante von c_p zum Startknoten s'_p der Methode p' .
- $e'_p \rightarrow_{\text{return}} c'_p$ ist die return-Kante vom Zielknoten e'_p von p' zum direkten Folgeknoten c'_p von Methode p .

Wir bezeichnen c_p als call-Knoten und c'_p als return-Knoten.

- Jeder Knoten in G ist durch den Startknoten von 'main' erreichbar und erreicht selbst den Zielknoten von 'main'.

Abbildung 2.1 zeigt den ICFG exemplarisch an einem vereinfachten Ausschnitt des Fallbeispiels `TimeTravel.java` mit zwei Methoden. Jede Methode endet wie beschrieben mit einem expliziten Zielknoten, der im Kontrollfluss durch eine Kante direkt mit dem dazugehörigen Startknoten verbunden ist. Der Übersichtlichkeit wegen ist der Aufruf der Java-Methode `System.out.println(d.x)` in dieser und den nachfolgenden Abbildungen nicht enthalten.

2.1.2. Der Systemabhängigkeitsgraph

Aus dem Kontrollfluss eines Programms lässt sich berechnen, welche Anweisungen über die Ausführung anderer Anweisungen entscheiden. Wir unterteilen diese Abhängigkeiten in *Daten-* und *Kontrollabhängigkeiten*, die über die *dominieren*-Relation¹ berechnet werden. Kontrollabhängigkeiten bestehen beispielsweise zwischen einer `if`-Bedingung und den Anweisungen innerhalb der Rümpfe der Fallunterscheidung.

Definition 2.3 (Daten- und Kontrollabhängigkeit). *Seien m, n zwei Anweisungen eines Programms p .*

- Eine Datenabhängigkeit von m zu n liegt vor, wenn n Daten verwendet, die von den Daten von m abhängen.
- Eine Kontrollabhängigkeit von m zu n liegt vor, wenn die Ausführung von n abhängig ist von m .

Ferrante et al.[10] entwickelten als erste eine intraprozedurale Datenstruktur zur Modellierung dieser Abhängigkeiten—den *Programmabhängigkeitsgraphen* (program dependence graph, PDG). Der PDG eines Programms p wird aus dem CFG von p berechnet, indem jeder abhängige Knoten und für jede Abhängigkeit eine entsprechende Kante zum PDG hinzugefügt werden.

Definition 2.4 (Programmabhängigkeitsgraph (PDG)). *Der Programmabhängigkeitsgraph $G = (N, E)$ eines methodenlosen Programms oder einer Methode m ist ein gerichteter Graph mit den folgenden Eigenschaften:*

- N ist eine Menge von Knoten als Repräsentation der Anweisungen von m .
- E ist eine Menge von Kanten als Repräsentation der Daten- und Kontrollabhängigkeiten zwischen den Knoten.

¹Ein Knoten m *dominiert* einen Knoten n , wenn jeder mögliche Pfad vom Startknoten nach n durch m führt.

Als Erweiterung des PDG für Programme mit mehreren Methoden dient der *Systemabhängigkeitsgraph* (system dependence graph, SDG) von Horwitz, Reps und Binkley[15].

Definition 2.5 (Systemabhängigkeitsgraph (SDG)). *Der Systemabhängigkeitsgraph $G = (N, E)$ eines Programms p ist ein gerichteter Graph mit den folgenden Eigenschaften:*

- N ist eine Menge von Knoten als Repräsentation der Anweisungen von p .
- E ist eine Menge von Kanten als Repräsentation der Daten- und Kontrollabhängigkeiten zwischen den Knoten.
- G besteht aus den PDGs der einzelnen Methoden von p , die durch Aufrufstellen miteinander verbunden sind.
- Parameterkanten modellieren jede Übergabe einer Variable an die aufgerufene Methode.




Eine *Aufrufstelle* setzt sich aus dem call-Knoten c und dem Startknoten s der aufgerufenen Methode zusammen, die durch eine call-Kante $c \rightarrow_{call} s$ verbunden sind, sowie zusätzlich eingefügten Knoten und Kanten für Parameter. Damit werden Parameterübergaben und Methodenrückgaben modelliert.

Abbildung 2.2 zeigt einen vereinfachten Ausschnitt des Fallbeispiels `TimeTravel.java` mit mehreren Kontrollfluss- und Datenabhängigkeiten. Die Rechtecke repräsentieren dabei *Parameterknoten*, die durch *Parameterkanten* verbunden sind.

Berechnung des Systemabhängigkeitsgraphen

Hammers Dissertation[14] und Grafs Veröffentlichung[13] behandeln die effiziente Konstruktion des SDG von Programmen objektorientierter Sprachen wie Java. Die in diesen Arbeiten beschriebenen Algorithmen ermöglichen es, Java-Programme mit mehr als 40.000 Codezeilen vollständig zu analysieren. Für das Visualisierungs-Plugin ist der SDG eine Vorstufe bei der Analyse von Programmen.

2.2. Analyse paralleler Programme

Moderne Programmiersprachen wie Java ermöglichen das  stellen von nebenläufigen Programmen mithilfe von Threads. ~~Die resultierenden Programme implizieren jedoch einen weiteren Typ von Abhängigkeiten: Interferenzen als Abhängigkeiten zwischen verschiedenen Threads.~~ Wie im Fallbeispiel in 1.2 beschrieben, können diese Interferenzen  Programmläufe beeinflussen, sind jedoch nicht ohne Weiteres ersichtlich. ~~Es ist  eine Erweiterung der Modelle der Kontrollfluss- und Systemabhängigkeitsgraphen erforderlich.~~

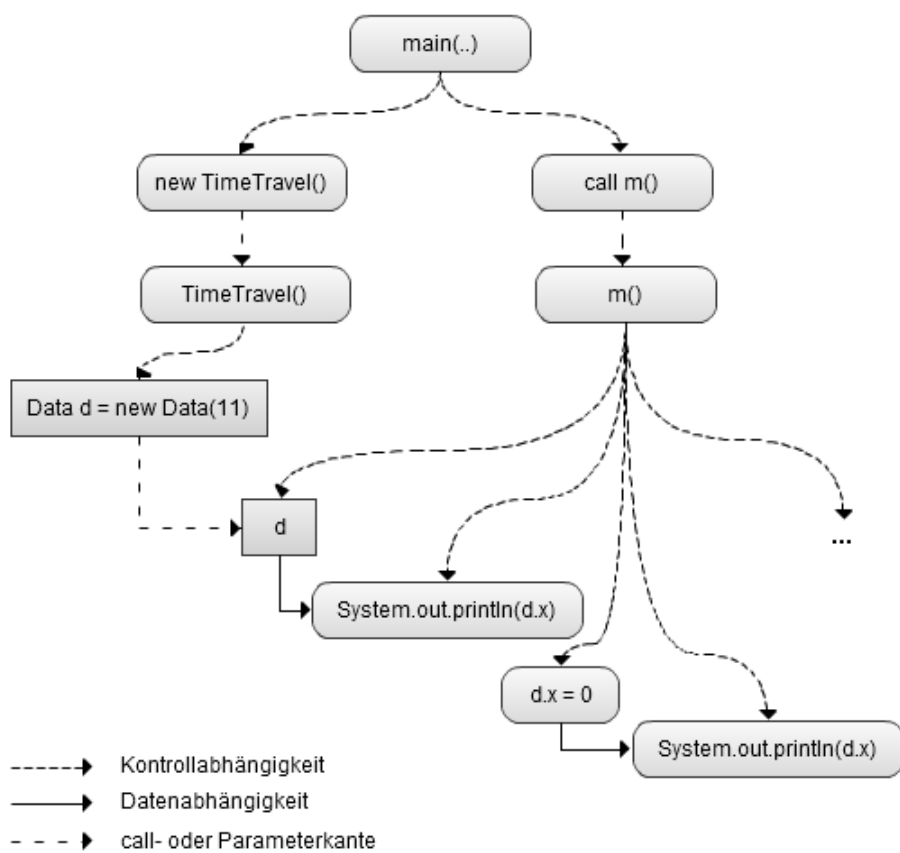
Nach einer Einführung in die Unterstützung von Threads durch Java wird in 2.2.1 die *Thread Invocation-Analyse* vorgestellt, um alle Threads eines Programms zu identifizieren. Die korrekte Abbildung von Threads erfolgt mittels des *Kontrollflussgraphen mit Threads* in 2.2.2. Darauf aufbauend berechnet die *may happen in parallel-Analyse* in 2.2.3 nebenläufige Programmabschnitte, um von einzelnen Anweisungen zu abstrahieren. Schließlich wird der Systemabhängigkeitsgraph in 2.2.4 um die Fähigkeit erweitert, Threads und deren Interferenzen zu handhaben.


```

3 public static void main(..) {
4     new TimeTravel().m();
5 }

25 public void m() {
26     ...
27
28     System.out.println(d.x);
29     d.x = 0;
30     System.out.println(d.x);
31 }

```

Abbildung 2.2.: Der SDG eines Ausschnittes von `TimeTravel.java`

Nebenläufigkeit in Java

Alle Threads in Java werden als Instanzen der `Thread`-Klasse realisiert. Das Verhalten eines Threads wird durch die vom Benutzer zu überschreibende `run()`-Methode beschrieben. Nach der Instanziierung und dem einmaligen Starten eines Threads mittels der `start()`-Methode läuft die `run()`-Methode fortan nebenläufig zum restlichen Programm. Der optionale Aufruf von `join()` beendet den Thread.

Der Java Language Specification[12] zufolge teilen sich alle Threads eines Programms den Heap und interagieren miteinander mittels Synchronisationsmechanismen und gemeinsam genutzter Variablen. Im Gegensatz zu einer Kommunikation via Nachrichtenaustausch können hierbei Abhängigkeiten zwischen Threads entstehen.

2.2.1. Die Thread Invocation-Analyse

In Java können Threads dynamisch zur Laufzeit auch innerhalb von rekursiven Aufrufen und Schleifen erstellt werden. Eine *Thread Invocation-Analyse* bestimmt und unterscheidet alle aktiven Threads einer Programmausführung. Dazu werden *Thread-Kontexte* nach Barik[8] eingesetzt.

Bei einer dynamischen Threaderstellung in einer Rekursion oder einer Schleife treffen wir die konservative Annahme, dass theoretisch unbeschränkt viele Threads der zugehörigen Thread-Klasse instanziiert werden. Diese werden als *Multi-Thread* durch einen einzigen Thread-Kontext repräsentiert.

2.2.2. Der Kontrollflussgraph mit Threads

Eine geeignete Modellierung des Kontrollflusses in nebenläufigen Programmen stellt der von Giffhorn[11] eingeführte *Kontrollflussgraph mit Threads* (threaded control flow graph, TCFG) dar. Ein TCFG besteht aus einer Menge von ICFGs der einzelnen Threads des Programms, die durch *fork-* und *join-Kanten* miteinander verbunden sind. Da die Anzahl dynamisch erstellter Threads theoretisch unbeschränkt ist, werden Threads durch ICFGs ihrer Thread-Klassen repräsentiert und damit aggregiert.

Definition 2.6 (Kontrollflussgraph mit Threads (TCFG)). *Der Kontrollflussgraph mit Threads $G = (ICFG_p, main, F, J)$ eines Programms p besteht aus einer Menge $ICFG_p$ von ICFGs der einzelnen Threads von p , einer speziellen ICFG 'main' und den beiden Mengen F und J von fork- und join-Kanten. Der TCFG hat folgende Eigenschaften:*

- *Von der Thread Invocation-Analyse unterschiedene Threads werden durch einen eigenen ICFG ihrer Thread-Klasse repräsentiert.*
- *Die Knotenmengen $N_T, N_{T'}$ und die Kantenmengen $E_T, E_{T'}$ sind für alle Paare (T, T') von ICFGs in $ICFG_p$ disjunkt.*
- *Die ICFGs sind durch fork- und join-Kanten miteinander verbunden. Es gilt:*
 - *Die Menge F enthält eine fork-Kante $(f_T, s_{T'})$ genau dann, wenn f_T ein fork-Knoten in ICFG T für den durch ICFG T' repräsentierten Thread ist und $s_{T'}$ den Startknoten von T' bildet.*

- Die Menge J enthält eine join-Kante $(e_T, j_{T'})$ genau dann, wenn $j_{T'}$ ein join-Knoten in ICFG T' für den durch ICFG T repräsentierten Thread ist und e_T den Zielknoten von T bildet.

- Jeder Knoten in G ist vom Startknoten des ICFG 'main' erreichbar.

Die Definition setzt ICFGs als disjunkt voraus. In der Realität können Threads jedoch gemeinsame Ressourcen nutzen— beispielsweise in Form von Bibliotheken. Um TCFGs in diesen Fällen dennoch korrekt zu berechnen, ist eine eindeutige Zuordnung von Knoten zu Threads zu gewährleisten. Für diese Bachelorarbeit werden die Ressourcen der Einfachheit halber dupliziert. Die Disjunktheit der ICFGs ist damit sichergestellt.

Abbildung 2.3 zeigt den TCFG am Fallbeispiel `TimeTravel.java` mit zwei Threads. Da Threads in Java nur optional beendet werden müssen, ist der Endzeitpunkt des zweiten Threads in diesem Fall unabhängig vom Hauptthread.

Mit dem TCFG werden nun alle durch die Thread Invocation-Analyse bestimmten Threads modelliert. Die Frage nach der Nebenläufigkeit von Teilen dieser Threads beantwortet der folgende Unterabschnitt.

2.2.3. Die MHP-Analyse

Eine *may happen in parallel*-Analyse (MHP) identifiziert alle potenziell parallel laufenden Programmabschnitte. Die einfachste MHP-Analyse trifft die streng konservative Annahme, dass alle Threads während der gesamten Programmausführung parallel laufen. Dies führt jedoch zu einer inakkuraten Berechnung von Interferenzen[11]. Im Folgenden werden daher die wichtigsten Optimierungen sowie darauf aufbauend ein effizienter Algorithmus zur MHP-Analyse erläutert.

Einführung von Thread Regionen

Nanda und Ramesh[20] segmentieren Java-Threads an fork- und join-Knoten in sogenannte *Thread Regionen*. Innerhalb einer Thread Region besitzen alle Knoten die gleiche MHP-Information. Statt auf der Ebene von Knoten ermöglicht das eine effiziente Speicherung der MHP-Information auf der Ebene von Thread Regionen.

Definition 2.7 (Thread Region). *Sei G ein TCFG. Ein Knoten n beginnt eine Thread Region R genau dann, wenn*

- n ein direkter Nachfolger eines fork-Knoten ist, oder
- n ein join-Knoten ist, oder
- n ein Knoten ist, an dem sich zwei verschiedene Thread Regionen treffen.

Ein Knoten m ist Teil von R , wenn n den Knoten m dominiert² und n der letzte Startknoten einer Thread Region auf jedem möglichen Pfad von n nach m ist.

Abbildung 2.4 visualisiert den Kontrollfluss des Fallbeispiels `TimeTravel.java` mit zwei Threads und vier Thread Regionen. Wie in 2.2 erwähnt, sind die Kontrollflusskanten von

```

3 public static void main(..) {
4     new TimeTravel().m();
5 }
6
7 public void m() {
8     Thread1 t = new Thread1();
9     t.data = d;
10    t.start();
11
12    ...
13 }
25 static class Thread1 ... {
26     Data data;
27
28     public void run() {
29         int a = data.x + 4;
30         data.x = data.x * a;
31     }
32 }

```

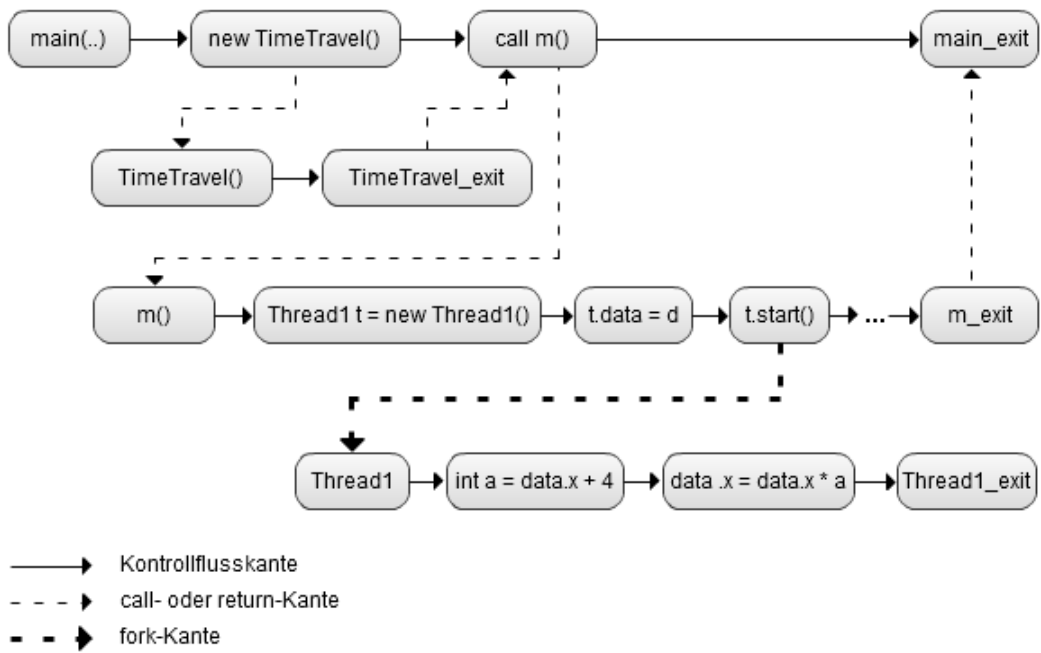
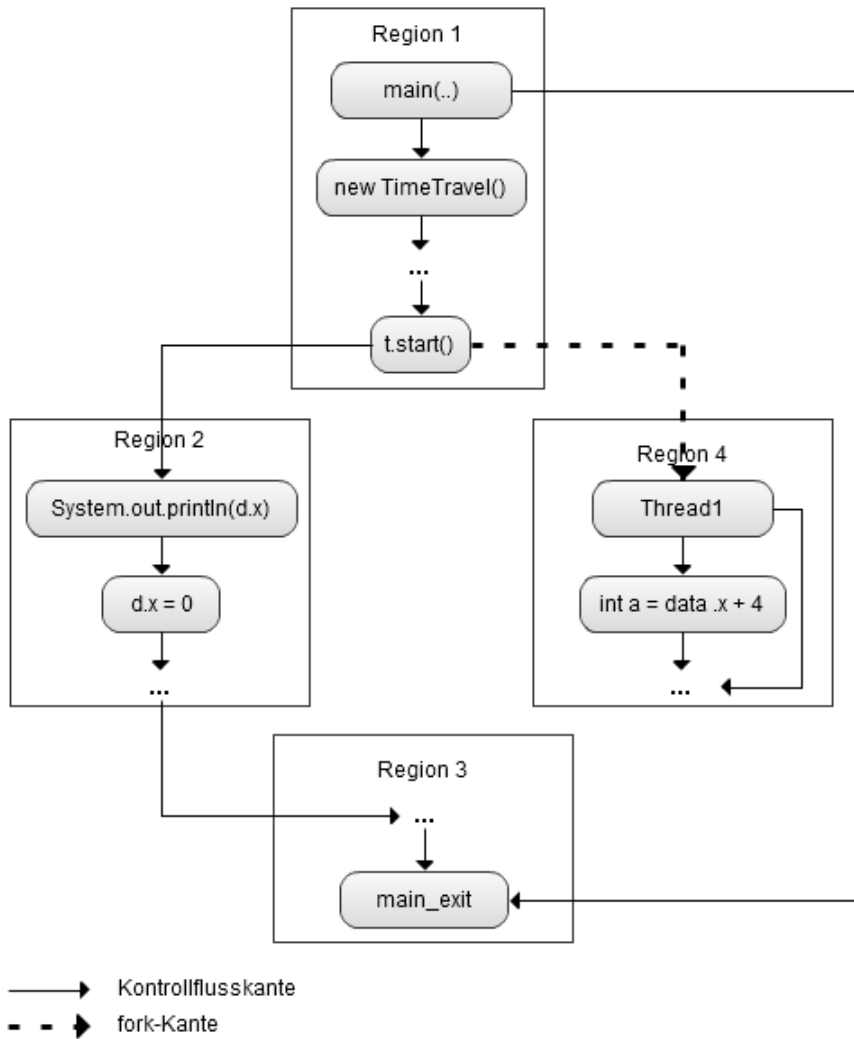


Abbildung 2.3.: Der TCFG von TimeTravel.java

Abbildung 2.4.: Die Thread Regionen von `TimeTravel.java`

den Startknoten der beiden Threads zu den zugehörigen Zielknoten hinzugefügt worden, um Berechnungen im Kontrollfluss zu vereinfachen.

Die MHP-Relation

Die potenzielle Parallelität von Programmabschnitten lässt sich nun auf der Ebene von Thread Regionen und Knoten sowie Thread-Kontexten und Threads formulieren.

Definition 2.8 (MHP-Relation \parallel_{region}). *Sei G ein TCFG. Für zwei Thread Regionen Q und R sind folgende Aussagen äquivalent:*

- a) Q und R laufen potenziell parallel ab, geschrieben $Q \parallel_{region} R$.
- b) Q und R gehören demselben Thread θ an und θ ist ein Multi-Thread.
- c) Q und R gehören verschiedenen Threads an und ein fork-Knoten f in G "verzweigt" nach Q und R , d.h. f erreicht den Startknoten der einen Region mit einer fork-Kante und den Startknoten der anderen Region mit einer anderen Kante.

Definition 2.9 (MHP-Relation \parallel). *Sei G ein TCFG.*

- Zwei Knoten m, n in G laufen potenziell parallel, geschrieben $m \parallel n$, wenn zwei Thread Regionen Q, R in G existieren mit $m \in Q, n \in R$ und $Q \parallel_{region} R$.
- Zwei Thread-Kontexte c_m, c_n in G laufen potenziell parallel, geschrieben $c_m \parallel c_n$, wenn $m \parallel n$ gilt.

Für zwei potenziell parallel laufende Threads θ_0 und θ_1 verwenden wir die Schreibweise $\theta_0 \parallel \theta_1$.

Die MHP-Analyse nach Giffhorn

Barik[8] stellte fest, dass Threads entweder vollständig parallel oder vollständig sequenziell laufen, wenn sie nicht (auch indirekt) mittels fork- oder join-Knoten miteinander verbunden sind. In diesem Fall genügt die Berechnung der MHP-Relation \parallel_{thread} auf der Ebene von Threads. In allen anderen Fällen erfolgt die MHP-Analyse auf der Ebene von Thread Regionen nach Nanda und Ramesh[20].

Die MHP-Analyse nach Giffhorn[11] erweitert nun beide Konzepte um die Technik der vorgestellten Thread Invocation-Analyse[8]. Durch die Repräsentation mittels Multi-Threads werden Programme mit dynamisch erzeugten Threads korrekt analysiert. Damit stellt Giffhorns Algorithmus die erste skalierbare MHP-Analyse mit vollständiger Unterstützung von Java zur Berechnung der MHP-Information dar.

Der nachfolgende Algorithmus 1 berechnet aus einem gegebenen TCFG in vier Schritten alle potenziell parallel laufenden Thread Regionen. Im ersten Schritt werden dazu alle Regionen innerhalb von Multi-Threads konservativ als potenziell parallel angenommen. Der zweite Schritt bestimmt all jene Threads, die vom selben Thread gestartet wurden. Deren Regionen erfüllen ebenfalls die MHP-Relation \parallel_{region} . Als drittes können alle nachfolgenden Regionen eines Threads zu allen vom selben Thread gestarteten Threads parallel laufen. Zuletzt werden alle bis dato gefundenen Thread Regionen nochmals verfeinert.

²Ein Knoten n dominiert einen Knoten m genau dann, wenn jeder mögliche Pfad vom Startknoten nach m durch n führt.

Algorithmus 1 Computation of MHP information**Require:** A TCFG G , the set T of thread contexts from the thread invocation analysis**Ensure:** A map $M : ThreadRegions \times ThreadRegions \mapsto \{true, false\}$ Compute the set R of thread regions.Initialize map M by setting all values to **false**.Let $regions(\theta)$ be the set of thread regions in thread θ .Let $\theta(c)$ be the thread of context c .Let $Desc(\theta)$ be the set of all threads of which θ is an ancestor.*/* 1) Handle multi-threads */***for all** multi-threads θ **do***// Collect all thread regions in θ and in descendant threads* $R' = \bigcup_{\theta' \in Desc(\theta) \cup \{\theta\}} regions(\theta')$ *// All these thread regions may happen in parallel to each other***for all** $(q, r) \in R' \times R'$ **do** $M(q, r) = true$ $M(r, q) = true$ */* 2) Determine the threads that are forked subsequently by the same ancestor */*Compute the set F of all fork site contexts (can be easily derived from set T).Let θ_f be the thread forked by fork site context f .**for all** $(f, f') \in F \times F : f \neq f' \wedge \theta(f) == \theta(f')$ **do***// All pairs of fork site contexts in the same thread***if** f reaches f' in the ICFG of $\theta(f)$ **then***// Use algorithm for reachability³**// The threads and their descendant threads may happen in parallel***for all** $(\theta, \theta') \in (Desc(\theta_f) \cup \{\theta_f\}) \times (Desc(\theta_{f'}) \cup \{\theta_{f'}\})$ **do***// Transfer the result to the thread regions***for all** $(q, r) \in regions(\theta) \times regions(\theta')$ **do** $M(q, r) = true$ $M(r, q) = true$ */* 3) All threads forked by a fork site f may happen in parallel to regions behind f */***for all** $f \in F$ **do**Compute the forward slice⁴ S for f of the ICFG of $\theta(f)$ **for all** $\theta \in (Desc(\theta_f) \cup \{\theta_f\})$ **do****for all** $q \in regions(\theta)$ **do****for all** thread regions r lying in S **do** $M(q, r) = true$ $M(r, q) = true$ */* continue on next page */*³Ein Algorithmus zur Berechnung der *Erreichbarkeit* zweier Kontexte in ICFGs findet sich bei Giffhorn[11].⁴Ein *forward slice* eines Knotens n enthält all jene Knoten m , die durch n beeinflusst werden—vergleiche auch 6.1.2.

/ continued from previous page */*

/ 4) Refine the result with must-join⁵information */*

Let J be the set of all join nodes in G that are identified as must-joins

for all $j \in J$ **do**

Let θ be the thread joined by j

for all $r \in R$: j dominates r 's start node **do**

for all thread regions q of θ **do**

$M(q, r) = false$

$M(r, q) = false$

return M

⁵Ein Join eines Threads θ ist ein *must-join* von θ , wenn an diesem Join einzig θ endet.

2.2.4. Der Systemabhängigkeitsgraph für nebenläufige Programme

Nach der Identifikation der im Programm enthaltenen Threads sowie der Nebenläufigkeit ihrer Thread Regionen werden als finaler Schritt, alle möglichen Interferenzen berechnet, und in den SDG integriert[11]. Der *Systemabhängigkeitsgraph für nebenläufige Programme* (concurrent system dependence graph, CSDG) bildet die Basis für das Visualisierungs-Plugin.

Definition 2.10 (Interferenz \rightarrow_{id}). *Ein Knoten m interferiert mit einem Knoten n genau dann, wenn n einen durch m berechneten Wert nutzt und $m \parallel n$ gilt. Wir schreiben dafür $m \rightarrow_{id} n$. Dabei ist m der interferierende Knoten und n der interferierte Knoten. Es lassen sich zwei Arten von Interferenzen unterscheiden:*

- *Eine Schreiben-Lese-Interferenz $m \rightarrow_{id_{read}} n$ liegt vor, wenn n einen von m berechneten Wert liest.*
- *Eine Schreiben-Schreiben-Interferenz $m \rightarrow_{id_{write}} n$ liegt vor, wenn n einen von m berechneten Wert überschreibt.*

In der Visualisierung der Interferenzen werden beide Arten von Interferenzen als gleichgestellt betrachtet und die Interferenzrichtung vernachlässigt— m und n *interferieren*. Wir schreiben dafür $m \leftrightarrow_{id} n$ oder gleichbedeutend $n \leftrightarrow_{id} m$.

Definition 2.11 (Interferenz \leftrightarrow_{id}). *Sei G ein TCFG.*

- *Zwei Thread Regionen Q, R in G interferieren, geschrieben $Q \leftrightarrow_{id} R$ oder $R \leftrightarrow_{id} Q$, wenn zwei Knoten m, n mit $m \in Q, n \in R$ existieren, die interferieren.*
- *Zwei Threads θ_0, θ_1 in G interferieren, geschrieben $\theta_0 \leftrightarrow_{id} \theta_1$ oder $\theta_1 \leftrightarrow_{id} \theta_0$, wenn zwei Knoten m, n mit $m \in \theta_0, n \in \theta_1$ existieren, die interferieren.*

Definition 2.12 (Systemabhängigkeitsgraph für nebenläufige Programme (CSDG)). *Der Systemabhängigkeitsgraph für nebenläufige Programme $G = (SDG_p, main, F, J, I)$ eines Programms p besteht aus einer Menge SDG_p von SDGs der einzelnen Threads von p , einem speziellen SDG 'main' und den drei Mengen F, J und I von fork-, join- und Interferenzkanten. Der CSDG hat folgende Eigenschaften:*

- *Von der Thread Invocation-Analyse unterscheidbare Threads werden durch einen eigenen SDG ihrer Thread-Klasse repräsentiert.*
- *Die Knotenmengen $N_T, N_{T'}$ und die Kantenmengen $E_T, E_{T'}$ sind für alle Paare (T, T') von SDGs in SDG_p disjunkt.*
- *Die Verbindung von SDGs erfolgt analog zu ICFGs durch fork- und join-Kanten. Es gilt:*
 - *Eine fork-Stelle repräsentiert den Aufruf eines Threads.*
 - *Eine join-Stelle repräsentiert das Beenden eines Threads.*
 - *Analog zu Parameterkanten modellieren fork- und join-Variablenkanten die Übergabe und Rückgabe gemeinsam genutzter Variablen eines aufgerufenen Threads.*
- *Jeder Knoten in G ist vom Startknoten des SDG 'main' erreichbar.*

Abbildung 2.5 zeigt den finalen CSDG des Fallbeispiels TimeTravel.java. Die beiden Threads main und Thread1 weisen insgesamt vier Interferenzen auf.

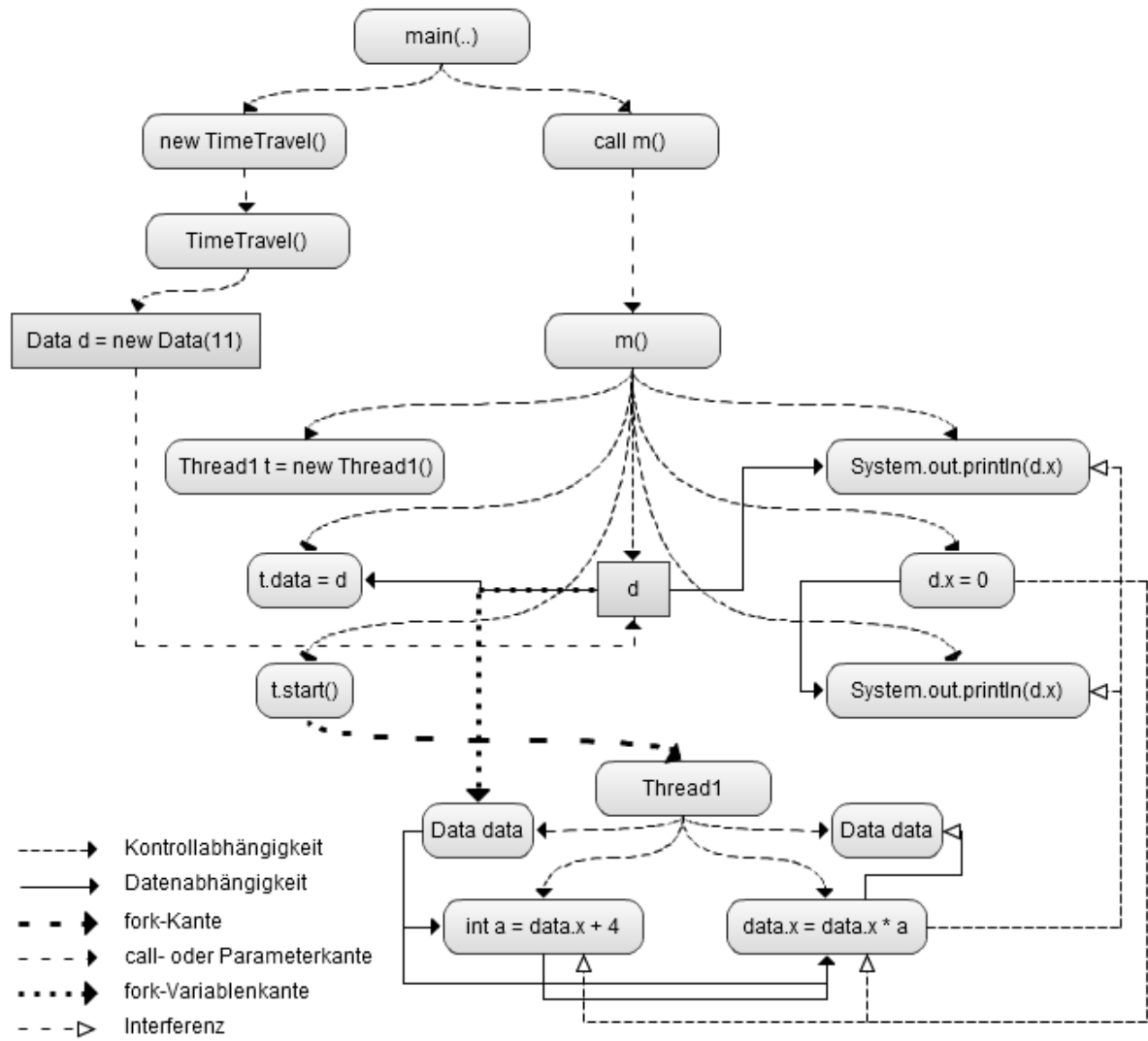


Abbildung 2.5.: Der finale CSDG von `TimeTravel.java`

Berechnung des Systemabhängigkeitsgraphen für nebenläufige Programme

Der CSDG ist das grundlegende Modell des Plugins zur Visualisierung von Threads und deren Interferenzen. Zur Berechnung des CSDG verbindet Giffhorn[11] die Ergebnisse der in diesem Kapitel beschriebenen Modelle und Algorithmen.

Ausgehend vom zu analysierenden Java-Programm wird für jede Thread-Klasse der SDG generiert. Dieser Schritt ist unabhängig von Abhängigkeiten zwischen Threads, da Kontroll- und Datenabhängigkeiten nur innerhalb eines Threads bestehen. Mithilfe der MHP-Analyse werden im zweiten Schritt alle Interferenzen zwischen den Threads berechnet. Diese Interferenzkanten verbinden schließlich die SDGs zum gewünschten CSDG. Damit ist die Analyse beendet und alle Informationen stehen bereit, um im nächsten Schritt visualisiert zu werden.

3. Implementierung

Mit dem Grundverständnis über die verwendeten Definitionen und Modelle zur Repräsentation von Abhängigkeiten zwischen Threads wird in diesem Kapitel der Kern der vorliegenden Bachelorarbeit vorgestellt—die Implementierung der Visualisierung. Der Fokus liegt dabei weniger in konkreten Implementierungsdetails als vielmehr in den Herausforderungen bei der Integration bestehender Visualisierungs- und Berechnungsklassen sowie der softwaretechnischen Vorgehensweise zu deren Lösung.

Die Kurzeinführung im ersten Abschnitt stellt das entwickelte Plugin vor und beschreibt prägnant seine Funktionalität aus Benutzersicht. Der zweite Abschnitt widmet sich eingehend ausgesuchter Implementierungskonzepte im Rahmen des Architekturmusters *Model-View-Controller* und deren Herausforderungen bei der realen Umsetzung. Die hierbei begrenzenden Faktoren der verwendeten Klassen zeigt das dritte Kapitel auf. Eine Statistik über die Implementierung rundet schließlich dieses Kapitel ab.

3.1. Das Plugin “ThreadViewer”

Wie in der Aufgabenstellung in 1.1 erwähnt, wird die Visualisierung als Forschungsprototyp in Form eines Eclipse-Plugins[5] entwickelt. Die verwendete Eclipse-Version ist 3.5.2. Als Programmiersprache dient Java[4] in der Version 1.6.0_23. Das Plugin selbst trägt den Namen *ThreadViewer*. ThreadViewer verwendet zur internen Berechnung des Graphenmodells das ValSoft/Joana-Framework[6] in der Version 0.0.2.

Der nachfolgende Unterabschnitt 3.1.1 listet auf, welche der Informationen aus dem CSDG von 2.2.4 und der MHP-Analyse von 2.2.3 verwendet werden. Unterabschnitt 3.1.2 nennt die vom ThreadViewer unterstützten Anzeigen, um Threads und Interferenzen zu visualisieren. Der letzte Unterabschnitt 3.1.3 führt mögliche Interaktionen des Benutzers mit dem Plugin auf.

3.1.1. Verwendung von Informationen

Für die Visualisierung der Threads und ihrer Interferenzen sind die folgenden Informationen aus dem CSDG und der MHP-Analyse eines Java-Programms von Bedeutung:

- Vorhandene Threads
- Thread Regionen und Informationen über ihre Nebenläufigkeit zueinander
- Knoten und die exakte Position der entsprechenden Anweisungen im Quellcode
- Kontrollflusskanten zwischen Knoten einschließlich der interprozeduralen Kanten
- Interferenzen zwischen Knoten

3. Implementierung

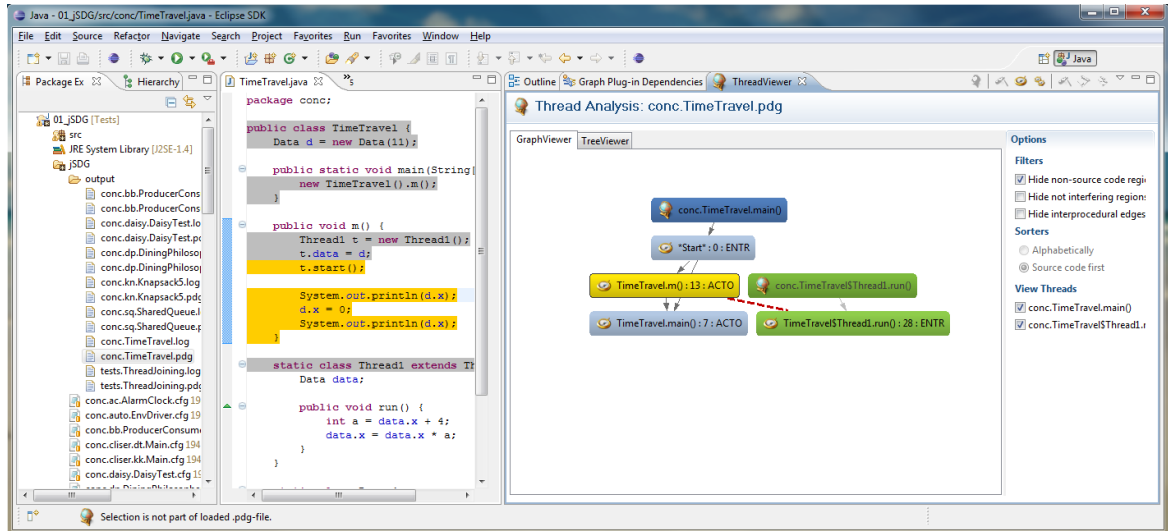


Abbildung 3.1.: Realisierung von *ThreadViewer* als Eclipse-Plugin

3.1.2. Unterstützte Anzeigen

Zur Visualisierung der Threads unterstützt das Plugin diverse graphische Anzeigen:

- Eine graphische Darstellung der Thread Regionen und der Interferenzen auf Ebene von Thread Regionen
- Eine Baumansicht der einzelnen Threads und der Interferenzen auf Ebene von Knoten
- Hervorhebungen des Quellcodes im Java-Texteditor

3.1.3. Mögliche Benutzerinteraktionen

ThreadViewer ermöglicht eine intuitive Bedienung über die folgenden Wege:

- Das Kontextmenü im Java-Texteditor
- Das Kontextmenü und die Doppelklick-Funktionalität in der Graphen- und Baumansicht
- Sortieren und Filtern im Optionen-Bereich des Plugins
- Die Werkzeugleiste des Plugins

Das *Folgen* von Objekten ist die zentrale Funktionalität des Plugins, um das gewünschte Objekt sowohl in der Graphen- und Baumansicht als auch im Java-Editor hervorzuheben. Abhängig vom selektierten Objekt sind in den Kontextmenüs und in der Werkzeugleiste die nachkommenden Interaktionen möglich:

- Einem Thread folgen
- Eine Thread Region zentrieren

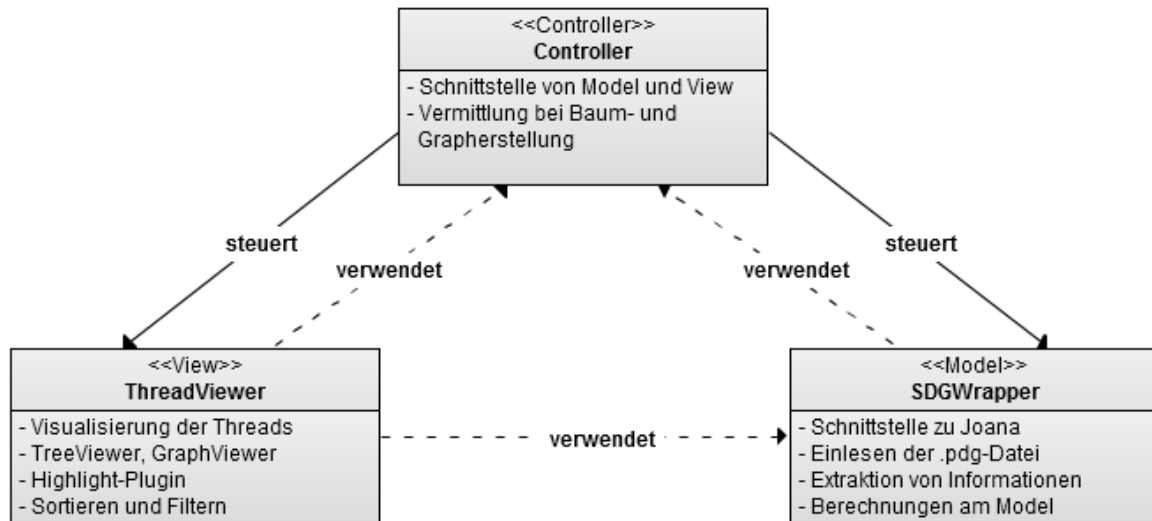


Abbildung 3.2.: Die MVC-Architektur von ThreadViewer

- Einer Thread Region folgen
- Parallelen Thread Regionen folgen
- Einen Knoten zentrieren
- Einem Knoten folgen
- Interferierenden Knoten folgen

Abbildung 3.1 zeigt das Plugin bei der Visualisierung des Fallbeispiels `TimeTravel.java`. Einzelheiten werden in den folgenden Abschnitten erläutert. Eine bebilderte Anleitung in Abschnitt 4.1 erklärt beispielhaft die Verwendung von ThreadViewer. Weitere Impressionen zum Plugin finden sich in 4.2.

3.2. Herausforderungen des Model-View-Controllers

Die Architektur von ThreadViewer orientiert sich am Muster *Model-View-Controller* (MVC). Durch die Kapselung der Einheiten Datenmodell, Präsentation und Programmlogik in die drei Komponenten *Model*, *View* und *Controller* wird eine Erhöhung der Flexibilität des Entwurfs bei gleichzeitiger Verminderung der Komplexität einer einzelnen Komponente erreicht. Anhand ausgewählter Aspekte werden nun die Herausforderungen bei der Implementierung der drei Komponenten des MVC sowie deren Lösung veranschaulicht.

Abbildung 3.2 veranschaulicht die MVC-Architektur von ThreadViewer. Jede Komponente ist dabei mit ihren Hauptaufgaben beschriftet. Die durchgezogenen Pfeile weisen auf die exponierte Stellung des Controllers als steuernde Schnittstelle der beiden anderen Komponenten. Die gestrichelten Pfeile repräsentieren die lose Kopplung der View und des Models zum Controller sowie den gelegentlichen Zugriff der View auf das Model.

3.2.1. Herausforderungen des Models

Der *SDGWrapper* als Model von ThreadViewer ist eine Wrapper-Klasse, die den Kontrollfluss in Form des CSDG und die Ergebnisse der MHP-Analyse kapselt. Als *Singleton* implementiert, bildet der SDGWrapper die Schnittstelle zum externen ValSoft/Joana-Framework.

Verarbeitung der Eingabe

Die Herausforderung beim Model liegt im Zusammenspiel der Klassen des ValSoft/Joana-Frameworks, um ein gegebenes Java-Programm in den gewünschten CSDG mit Informationen über Threads und Abhängigkeiten zu transformieren. Grundlage des SDGWrappers ist eine vom Framework erzeugte *.pdg-Datei*, die den CSDG des zu analysierenden Java-Programms enthält. Mit dem statischen Aufruf `SDG.readFrom(..)` liest das Model die *.pdg-Datei* ein. Eine Verarbeitung durch die Methode `ThreadRegions.createThreadRegions(..)` erzeugt die Thread Regionen des Programms. Der Aufruf `PreciseMHPAnalysis.analyze(..)` initiiert die in Algorithmus 1 vorgestellte MHP-Analyse. Sie liefert die entscheidende MHP-Information über nebenläufige Thread Regionen zurück. Abschließend bereinigt die Methode `cleanCSDG(..)` die MHP-Information um Interferenzen, die zwischen zwei nicht parallel laufenden Knoten bestehen. MHP-Information und CSDG bilden zusammen das Informationsmodell des Models mit allen Threads und deren Interferenzen.

Während der Verarbeitung der Eingabe wird zusätzlich eine Liste aller *.java-Dateien* erstellt, die im Projektverzeichnis der *.pdg-Datei* vorhanden sind. Die View nutzt später diese Liste, um Thread Regionen des zu analysierenden Programms von Thread Regionen der Java-Standardbibliotheken zu unterscheiden. Dabei wird die Annahme getroffen, dass eine Thread Region des Programms durch eine entsprechende *.java-Datei* im Projektverzeichnis repräsentiert wird.

Extraktion von Informationen

Mit dem CSDG verfügt der SDGWrapper über eine Datenstruktur, die umfassende Informationen zu den Abhängigkeiten und zum Kontrollfluss des Java-Programms enthält. So stehen neben Standardinformationen und -methoden wie eine eindeutige ID und Validierungsfunktionen auch erweiterte Methoden zur Abfrage bestimmter Thread Regionen bereit. Die Fülle an Informationen und die Verteilung derselben über mehrere Klassen erschweren jedoch eine direkte Extraktion gewünschter Daten und Abhängigkeiten. Daher erfolgt im Folgenden ein Auszug der nützlichsten Methoden des CSDG, gelistet nach den Klassen des Models.

Klasse	SDG
	Repräsentation des CSDG
<code>getEntry(..)</code>	Rückgabe des Entry-Knotens einer Region
<code>outgoingEdgesOf(..)</code>	Rückgabe der von einem Knoten ausgehenden Kanten
<code>getOutgoingEdgesOfKind(..)</code>	Rückgabe der von einem Knoten ausgehenden Kanten eines bestimmten Kantentyps
<code>incomingEdgesOf(..)</code>	Rückgabe der zu einem Knoten eingehenden Kanten
<code>getNodesOfProcedure(..)</code>	Rückgabe der Knoten einer Methode

Klasse	ThreadRegions Repräsentation der Thread Regionen
getThreadRegions()	Rückgabe aller Regionen
getThreadRegions(..)	Rückgabe der durch einen Knoten oder eine Thread-ID spezifizierten Regionen
Klasse	PreciseMHPAnalysis Repräsentation der MHP-Information
isParallel(..)	Test von zwei Knoten auf potenzielle Parallelität
Klasse	ThreadInformation Repräsentation der Threads
iterator()	Rückgabe eines Iterators für alle Threads
getThread(..)	Rückgabe des durch eine Region oder eine Thread-ID spezifizierten Threads

Eigene Berechnungen

Der Zugriff auf den CSDG und die MHP-Information ist zur Berechnung aller benötigten Abhängigkeiten alleine nicht ausreichend. Die Implementierung des Modells erfordert daher eigene Berechnungen von weitergehenden Informationen mithilfe der soeben vorgestellten Methoden. Diese Berechnungen werden vom Model durch die nachfolgenden Funktionen bereitgestellt.

Klasse	SDGWrapper::Threads Erweiterte Methoden für Threads
getThreads()	Rückgabe aller Threads Nutzt den Iterator der ThreadInformation-Klasse.

Klasse	SDGWrapper::Regionen Erweiterte Methoden für Thread Regionen
getRegions()	Rückgabe aller Regionen Nutzt den Iterator der ThreadInformation-Klasse.
getNextRegions(..)	Rückgabe der nachfolgenden Regionen einer Region Iteriert über alle ausgehenden Kontrollflusskanten.
getParallelRegions(..)	Rückgabe der parallel laufenden Regionen einer gegebenen Region Wendet die MHP-Information auf alle Regionen an.
getInterferedRegions(..)	Rückgabe der Regionen der Knoten, die von einem Knoten oder einer Region interferiert werden Iteriert über alle benachbarten Interferenzkanten.
Klasse	SDGWrapper::Knoten Erweiterte Methoden für Knoten
getNodes(..)	Rückgabe der Knoten eines Threads Iteriert über alle Regionen des Threads.
getNodesOfSameRegion(..)	Rückgabe der Knoten derselben Region eines gegebenen Knotens Iteriert über alle Regionen des gegebenen Knotens.
getInterferingNodes(..)	Rückgabe der interferierenden Knoten einer Region Iteriert über alle benachbarten Interferenzkanten.
getInterferedNodes(..)	Rückgabe der Knoten, die von einem Knoten oder einer Region interferiert werden Iteriert über alle benachbarten Interferenzkanten.
Klasse	SDGWrapper::Verschiedenes Allgemeine weitere Methoden
isInSourceCode(..)	Test mithilfe der Dateiliste, ob ein Knoten oder eine Region dem zu analysierenden Programm angehört Testet, ob das source-Attribut des Knotens mit einer Datei übereinstimmt. Bei der Region muss das für zumindest einen Knoten der Fall sein.
getCalledMethods(..)	Rückgabe der Methoden, die vom gegebenen Entry-Knoten aufgerufen werden Iteriert über alle ausgehenden call-Kanten.

Vorbereitung des Kontrollflusses

Der direkte Kontrollfluss im CSDG eines Programms ist durch intra- und interprozedurale Kontrollflusskanten vorgegeben. Für eine übersichtliche Darstellung in der Graphenansicht des Plugins wird der Kontrollfluss auf der Ebene von Thread Regionen aggregiert. Ein nicht unerheblicher Teil der Thread Regionen bildet jedoch lediglich die Funktionalität von Java-Standardbibliotheken ab oder besitzt keine Interferenzen. Um diese uninteressanten Regionen auszublenden, verwendet die View daher Filter.

Für jede Kombination der Filter ist eine Anpassung des Kontrollflusses auf die jeweils noch zu visualisierenden Thread Regionen erforderlich. Das Model berechnet diese Kontrollflüsse beim Laden der .pdg-Datei vor, um sie effizient der View zur Visualisierung bereitzustellen. Die folgenden Kontrollflüsse sind implementiert und in der View über den Optionen-Bereich durch den Einsatz der Filter realisierbar:

- Direkte Nachfolge-Regionen
- Nachfolge-Regionen ohne Standardbibliotheken
- Nachfolge-Regionen mit Interferenzen
- Nachfolge-Regionen mit Interferenzen ohne Standardbibliotheken

Die Berechnung der Relationen erfolgt jeweils mittels einer *Breitensuche* im CSDG. Gefundene Nachfolge-Regionen werden für einen effizienten Zugriff in *HashMaps* gespeichert. Die private Hilfsmethode `calculateSpecialControlFlows(..)` des Models im nachfolgenden Quellcode 3.1 implementiert beide Techniken.

```

1 // Input: ThreadRegion source
2
3 // Init
4 Collection<ThreadRegion> targets = new HashSet<ThreadRegion>();
5 Collection<ThreadRegion> visited = new HashSet<ThreadRegion>();
6 Queue<ThreadRegion> queue = new LinkedList<ThreadRegion>();
7 queue.add(source);
8 visited.add(source);
9
10 // Breadth First Search [BFS]
11 // Find all successor regions satisfying the given condition
12 while (!queue.isEmpty()) {
13     ThreadRegion current = queue.poll();
14
15     // Abort further search on region when region is in source code
16     if (current.equals(source) || !sdgWrapper.isInSourceCode(current)) {
17         for (ThreadRegion target : sdgWrapper.getNextRegions(current)) {
18             if (!visited.contains(target)) {
19                 queue.add(target);
20                 visited.add(target);
21             }
22         }
23     }
24 }

```

```
25 // Add found region when region is in source code
26 if (!current.equals(source) && sdgWrapper.isInSourceCode(current)) {
27     targets.add(current);
28 }
29 }
30
31 // Save successor relation in HashMap
32 memoryNextSourceCodeRegions.put(source, targets);
```

Listing 3.1: Die Breitensuche der Methode `calculateSpecialControlFlows(..)`

3.2.2. Herausforderungen der View

Die `ThreadViewer`-Klasse bildet die View des Plugins. Sie verantwortet damit die Interaktion mit dem Benutzer sowie die eigentliche Visualisierung anhand der Daten des Modells. Die Herausforderung der View liegt dabei insbesondere in der nahtlosen Integration der implementierten Benutzeroberfläche mit der bestehenden Eclipse-Plattform.

Abbildung 3.3 zeigt das von der konkreten Darstellung abstrahierte Layout des Plugins. Im Package Explorer (1) wählt der Benutzer die gewünschte zu visualisierende .pdg-Datei. In der Visualisierung (3) werden die Threads und alle Interferenzen als Baum oder als Graph angezeigt und können durch verschiedene Optionen (4) sortiert oder ausgeblendet werden. Gewünschte Knoten, Thread Regionen oder Threads werden im Java-Texteditor (2) entsprechend farblich hervorgehoben. Auf die Herausforderungen bei der Implementierung der wichtigsten View-Elemente wird im Folgenden weiter eingegangen.

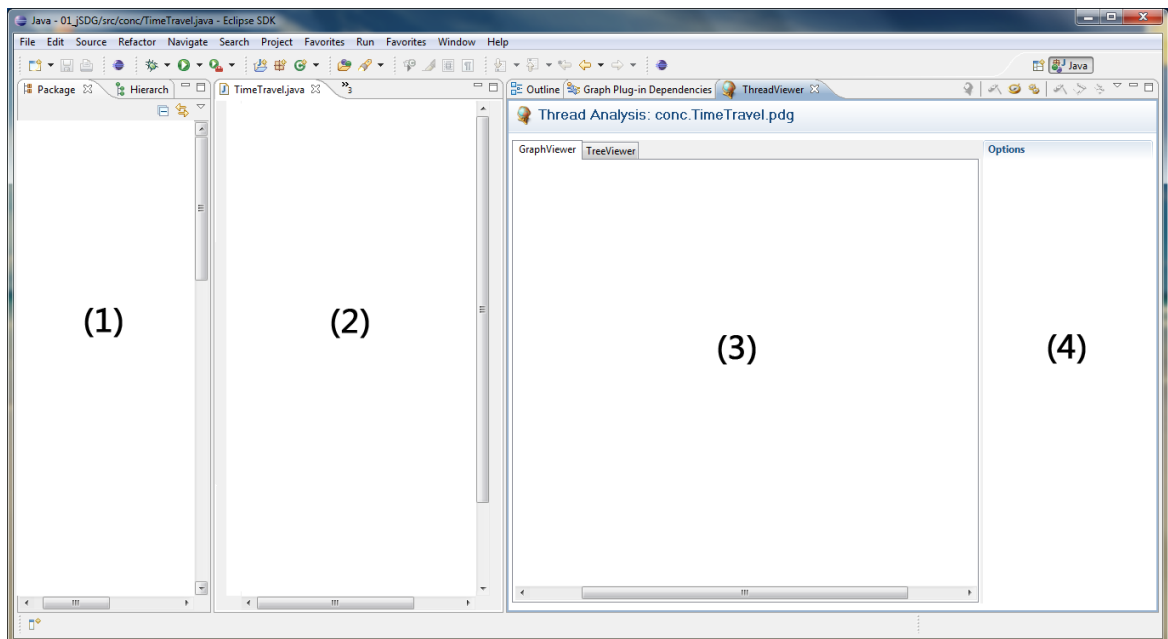


Abbildung 3.3.: Das Layout von ThreadViewer mit dem Package Explorer (1), dem Java-Texteditor (2), der Visualisierung (3) und den Optionen (4)

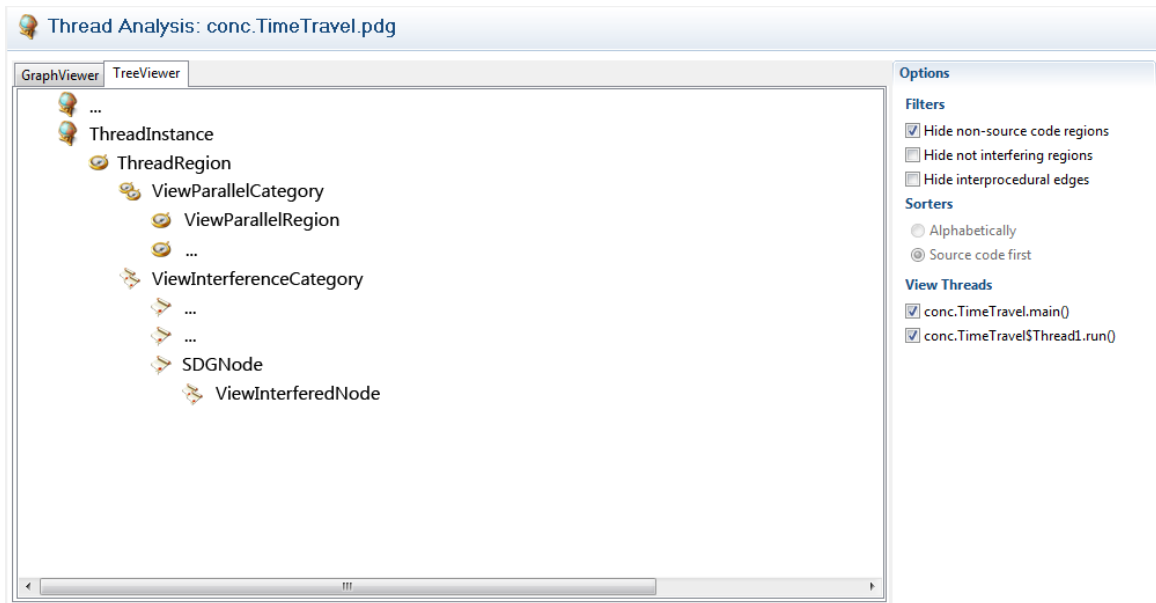


Abbildung 3.4.: Die verwendeten Klassen in der Baumansicht

TreeViewer

Die strukturierte Baumansicht der Threads in 3.4 erfolgt mithilfe der Klasse `TreeViewer` des Pakets `org.eclipse.jface.viewers.TreeViewer`. Der `TreeViewer` wird durch eine Instanz von `ITreeContentProvider` mit Elementen versorgt und durch eine Instanz von `LabelProvider` den Elementen entsprechend dekoriert. Angezeigt werden alle Entitäten des CSDG—von Threads über Thread Regionen zu interferierenden Knoten.

Dabei arbeitet der Baum direkt mit den Objekten des Models und unterscheidet die verschiedenen Baumebenen anhand der zugehörigen unterschiedlichen Klassen. Die Anzeige von `ThreadInstance` für eine Thread-Klasse, `ThreadRegion` für eine Thread Region und `SDGNode` für einen interferierenden Knoten ist damit direkt möglich.

Bei parallelen Thread Regionen und interferierten Knoten handelt es sich jedoch um Objekte der bereits verwendeten Klassen `ThreadRegion` bzw. `SDGNode`. Daher erfordert eine korrekte Implementierung die Kapselung dieser Objekte in zwei eigenen Java-Klassen, `ViewParallelRegion` und `ViewInterferedNode`. Auch für die Kategorie-Elemente werden separate Klassen als Unterklassen von der Klasse `ViewCategory` erstellt, um den Baum mit den entsprechenden Gegenstücken zu versorgen.

Abbildung 3.4 veranschaulicht alle im `TreeViewer` verwendeten Klassen.

GraphViewer

Die anschauliche Graphansicht von `ThreadViewer` beruht auf dem *Zest-Framework*[7] für Eclipse, das wiederum Teil des *Graphical Editing Framework* (GEF, [3]) ist. GEF nutzt SWT-Elemente und basiert auf dem leichtgewichtigen *Draw2d*-Toolkit. Damit werden das Layout sowie das Rendern von SWT-Elementen unterstützt.

Analog zum `TreeViewer` wird auch der `GraphViewer` von mehreren Klassen mit Elementen und Formatierungsvorgaben versorgt. Die zu zeichnenden Entitäten spezifiziert die Klasse

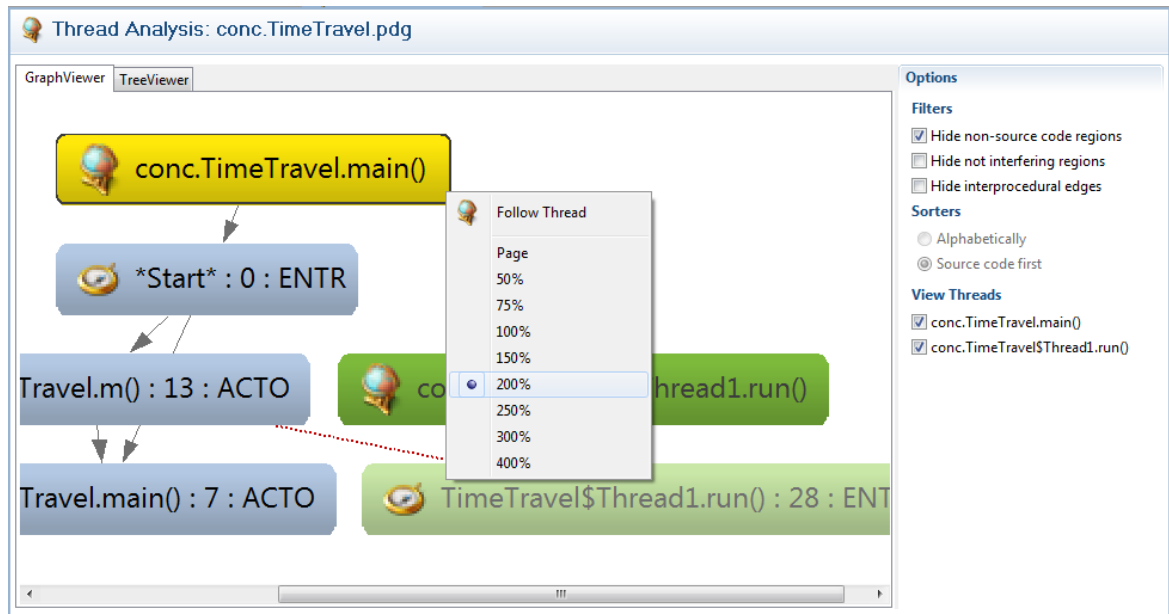


Abbildung 3.5.: Die Zoomfunktion des Zest-Framework in der Graphansicht im Einsatz

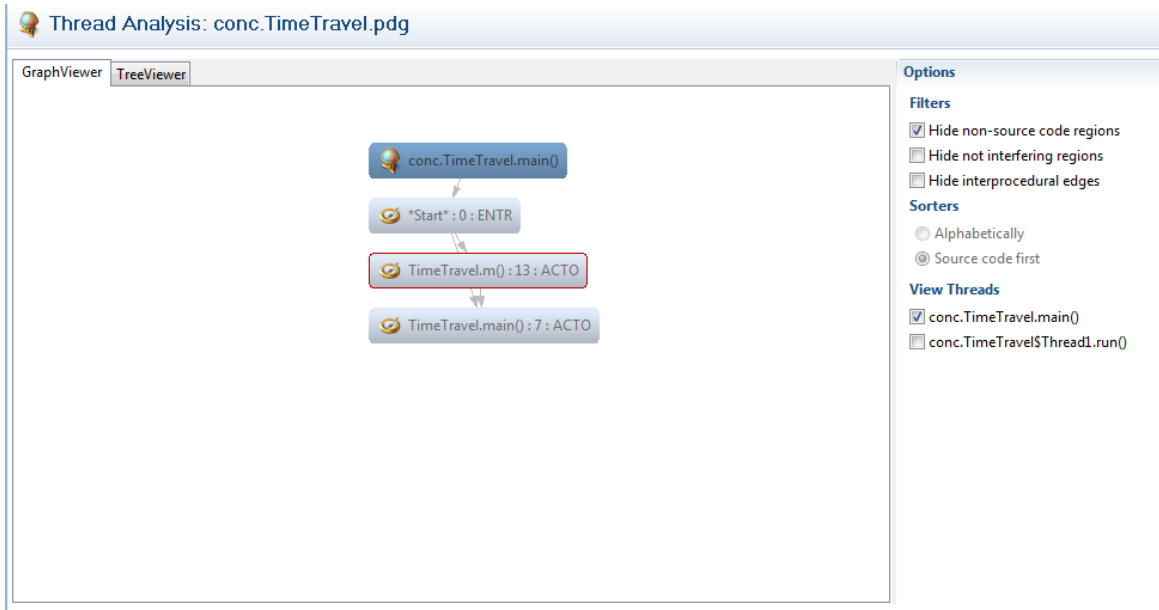
`IGraphEntityContentProvider`; die Beschriftung sowie das Kanten- und Entitätenformat geben die Klassen `ILabelProvider`, `IConnectionStyleProvider` bzw. `IEntityStyleProvider` vor. Die Anzeige beschränkt sich der Übersichtlichkeit halber auf Thread und Thread Regionen sowie Interferenzen auf Thread Region-Ebene.

Das Zest-Framework zeichnet sich durch zwei Aspekte aus, die einfach zu implementieren sind und eine große optische Wirkung erzielen. Zum einen bietet Zest eine eingebaute *Zoomfunktionalität* an. Nach der Instanziierung der Klasse `ZoomContributionViewItem` und der Implementierung des Interfaces `IZoomableWorkbenchPart` durch die View wird dem Kontextmenü noch ein Eintrag hinzugefügt. Dadurch steht dem Benutzer per Rechtsklick auf die Zeichenfläche direkt eine Palette möglicher Zoomfaktoren zur Verfügung, die die Ansicht des GraphViewers automatisch anpassen. Zum anderen unterstützt Zest verschiedene *Layout-Algorithmen*, um zu zeichnende Elemente automatisch optimal anzuordnen. ThreadViewer nutzt eine Komposition aus den Algorithmen `DirectedGraphLayoutAlgorithm` und `HorizontalShift`, um die Elemente mit gerichteten Kanten automatisch auf der Zeichenfläche anzuordnen und im Falle überlappender Elemente diese horizontal auseinander zu schieben. Das Ergebnis ist ein vom Zest-Framework optimal angeordneter Graph mit einer minimalen Anzahl von Überlappungen der Kanten. Bei einer Änderung des Graphen werden alle Elemente durch eine animierte Übergangssequenz automatisch repositioniert.

Abbildung 3.5 zeigt die Zoomfunktion des Zest-Frameworks im Einsatz.

Das HighlightPlugin

Der nahtlose Übergang vom Plugin zum Java-Editor bei der Benutzerinteraktion erfordert ein optisch stimmiges Design sowie eine einheitliche Benutzerführung. Zur farblichen Hervorhebung von Anweisungen im Java-Editor, die zu Knoten, Thread Regionen oder Threads gehören, wird daher eine Instanz des `HighlightPlugins` vom Paket `joana.sdg.textual.highlight`

Abbildung 3.6.: Der Graph von `TimeTravel.java` mit verdeckter Interferenz

in der Version 1.1.0 verwendet. Dazu bietet das `HighlightPlugin` die folgenden Methoden an:

Klasse	HighlightPlugin Hilfsklasse zur Hervorhebung von Code
<code>getDefault(..)</code>	Rückgabe der Instanz
<code>highlight(..)</code>	Hervorhebung des gegebenen Knotens mit der gegebenen Farbe im Quellcode
<code>highlightJC(..)</code>	Hervorhebung der gegebenen Knoten- und Farbpaare im Quellcode
<code>clearAll(..)</code>	Entfernung aller vorhandenen Hervorhebungen

Mit dieser Hilfsklasse können nun selektierte Thread Regionen und Threads nicht nur im `ThreadViewer`, sondern auch im Java-Editor optisch einheitlich hervorgehoben werden. Dies ermöglicht dem Benutzer eine intuitive Arbeit mit der klassischen Quellcode-Sicht und der abstrahierenden Graph- und Baumansicht.

Erweiterte Funktionalität

Zur flexiblen Anpassung der Graph- und Baumansicht bietet `ThreadViewer` im Optionen-Bereich diverse Funktionen zum Sortieren und Filtern der Thread Regionen an. Die Aktivierung der Klasse `AlphabeticalThreadRegionSorter` sortiert in der Baumansicht alle Thread Regionen alphabetisch aufsteigend, während die Klasse `SourcecodeThreadRegionSorter` sicherstellt, dass zunächst Regionen des zu analysierenden Programms zuerst gelistet werden. Letzteres stellt die Methode `isInSourceCode(..)` des Models sicher.

Das Filtern von Elementen ist in jeder Ansicht über den stets sichtbaren Optionen-Bereich möglich. `HideSourceCodeThreadRegionFilter` blendet alle Thread Regionen aus, die Java-Standardbibliotheken repräsentieren. Hingegen filtert `HideNotInterferingThreadRegionFilter` alle Regionen ohne Interferenzen. Dies führt bei größeren Programmen zu einer erhöhten Übersicht und eine Betonung der kritischen Interferenzen. Die abhängig von der Anzahl an Threads erscheinenden Checkboxes erlauben ein gezieltes Ausblenden einzelner Threads. Dies wird intern durch ein Array von Checkboxes realisiert, das bei der Verarbeitung der Eingabe einer .pdg-Datei eine entsprechende Anzahl von Checkboxes instanziiert. Interferiert nun eine Thread Region mit einer ausgeblendeten Region, visualisiert ein roter Rahmen um erstere die Existenz einer verdeckten Interferenz. Die Evaluation in 5.3 verdeutlicht später den enormen Zuwachs an Übersicht durch die Verwendung der Filter.

Abbildung 3.6 zeigt das Fallbeispiel mit einem über den Optionen-Bereich ausgeblendeten Thread. Die verbleibende interferierende Region ohne eingblendete interferierte Region wird rot umrandet.

Interaktion und Folgen

Die Interaktion mit dem Benutzer erfolgt über *Contributions* und *Actions*, die in Kontextmenüs und der Werkzeugleiste implementiert werden. Contributions werden direkt in der `plugin.xml`¹ notiert, um Eclipse selbst zu erweitern. So wird der Package Explorer um einen Kontextmenü-Eintrag zum Laden einer .pdg-Datei und der Java-Texteditor um ein Kontextmenü zum *Folgen* von Knoten, Region oder Thread im ThreadViewer erweitert. Mit dem Folgen wird das gewünschte Element in ThreadViewer selektiert und im Quellcode durch das HighlightPlugin farblich passend hervorgehoben.

Für die Kontextmenüs und die Werkzeugleiste in ThreadViewer werden die wesentlich flexibleren Actions verwendet. Dadurch ist eine Deaktivierung einzelner Schaltflächen in Abhängigkeit vom internen Plugin-Zustand möglich. Beispielsweise wird die Schaltfläche zum Folgen eines Threads aktiviert, wenn tatsächlich ein Thread markiert ist, während alle anderen Schaltflächen deaktiviert werden.

3.2.3. Herausforderungen des Controllers

Der *Controller* fungiert als Schnittstelle zwischen Model und View. Er veranlasst die tatsächliche Umsetzung von Benutzer-Interaktionen und versorgt die View mittels der Daten des Models mit den korrekten Threads und Interferenzen. Die wichtige Rolle des Controller spiegelt sich in der Größe der Klasse und der Anzahl der Methoden wieder. Hier bewähren sich klare Methodennamen und die Kapselung ähnlicher Programmabschnitte in private Hilfsmethoden.

Schema zum Folgen eines Objekts

Die wesentlichen Methoden des Controllers sind diejenigen zum Folgen von Knoten, Thread Regionen und Threads in den Ansichten des ThreadViewers und im Quelltext des Java-Editors.

Der nachfolgende Quelltext 3.2 zeigt beispielhaft das Folgen einer Thread Region, die der Benutzer im Java-Editor per Kontextmenü gewählt hat. Zunächst wird sichergestellt, dass

¹Die `plugin.xml` ist die Manifest-Datei eines Plugins zur individuellen Konfiguration von Eclipse.

ThreadViewer geladen und eine .pdg-Datei korrekt ausgelesen worden ist. Daraufhin werden alle durch die einzelnen Knoten der Thread Region spezifizierten Quellcode-Dateien geöffnet und durch das HighlightPlugin hervorgehoben. Zuletzt wird die Thread Region in der aktuellen Ansicht des ThreadViewers selektiert. Sollte eine der Vorbedingungen fehlschlagen, wird eine entsprechende Fehlermeldung in der Statusleiste von Eclipse eingeblendet.

```

1 public void runEditorFollowThreadRegion(IEditorPart targetPart) {
2     // Ensure ThreadViewer to be accessible
3     this.checkView();
4
5     // Ensure .pdg-file is correctly loaded and calculations are done
6     if (sdgWrapper.isCreated()) {
7         ThreadRegion region = this.getSelectedRegion(targetPart);
8
9         if (region != null) {
10            // Open corresponding file(s)
11            this.openFiles(region);
12            // Highlight region
13            this.followObject(region);
14            // Set selected region in ThreadViewer
15            ThreadViewer.getInstance().setSelectedObject(region);
16        } else {
17            ThreadViewer.getInstance().clearAllHighlights();
18            updateStatusBar("Selection is not part of loaded .pdg-file.");
19        }
20    } else {
21        updateStatusBar("Load .pdg-file into ThreadViewer first.");
22    }
23 }

```

Listing 3.2: Die Methode `runEditorFollowThreadRegion(..)` des Controllers

Das Sequenzdiagramm in Abbildung 3.7 verdeutlicht die zentrale Steuerfunktion des Controllers beim Folgen einer Thread Region. Ausgehend von einer Benutzer-Interaktion steuert der Controller nacheinander verschiedene Hilfsmethoden des Models und der View an. Der Übersichtlichkeit halber abstrahiert das Diagramm von möglichen Rückgabewerten und Verzweigungen und steigt nur bei der Methode `followObject(..)` detailliert weiter ab.

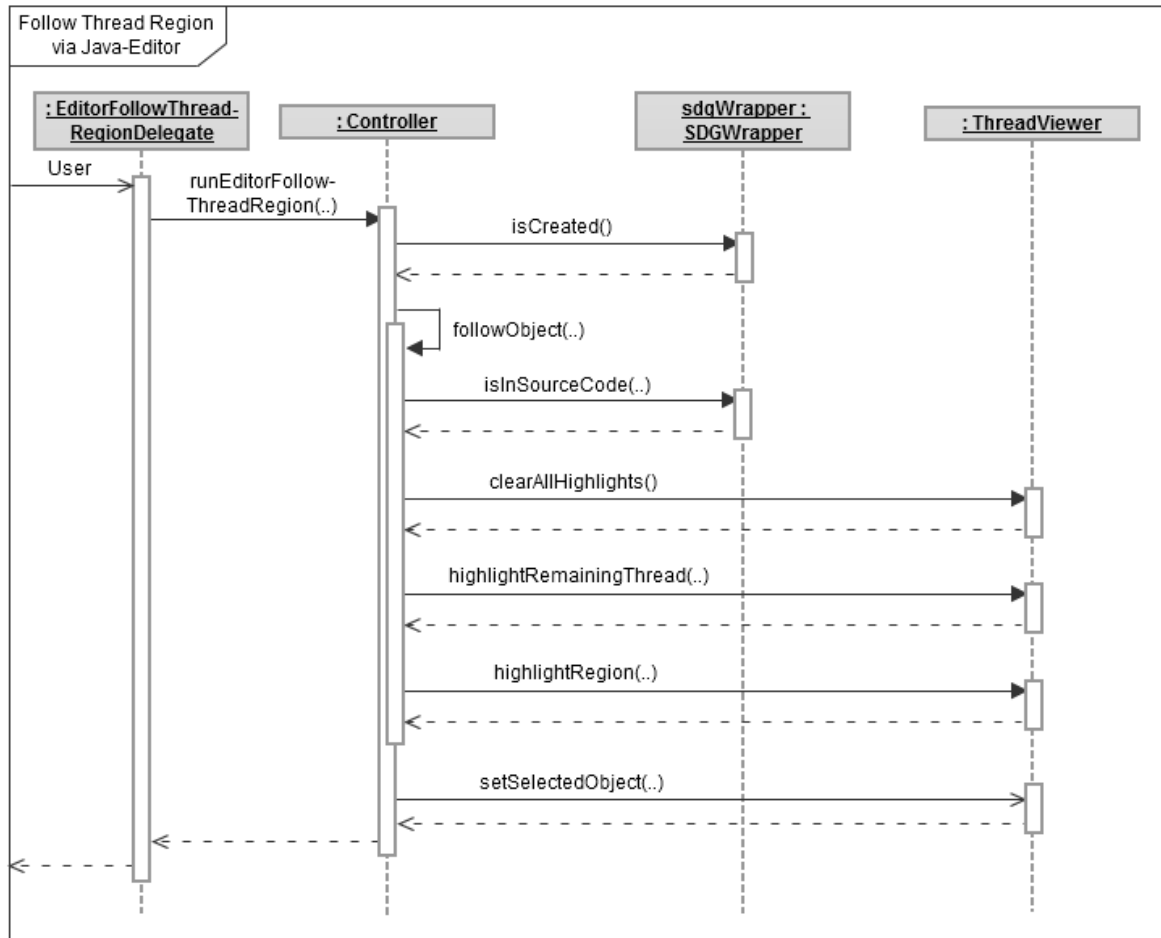


Abbildung 3.7.: Das vereinfachte Sequenzdiagramm von ThreadViewer beim Folgen einer Thread Region

Die Methoden-Schemata sowie die Sequenzdiagramme der weiteren von ThreadViewer unterstützten Methoden zur Benutzer-Interaktion sind analog zu den vorgestellten aufgebaut. Marginale Unterschiede ergeben sich beispielsweise beim Folgen von Objekten, wenn der Prozess vom ThreadViewer aus initiiert wird. In diesem Fall entfällt eine Überprüfung der View. Statt das selektierte Objekt über den Java-Editor zu ermitteln, wird direkt die View nach dem aktuellen Objekt erfragt.

3.3. Grenzen der Implementierung

Die vorgestellten Konzepte können nicht ohne Weiteres mit den Klassen des ValSoft/Joana-Framework implementiert werden. Die begrenzenden Faktoren und deren Umgang werden in diesem Abschnitt kurz dargelegt.

3.3.1. Grenzen des Models

Zur automatischen Anzeige der korrekten Quellcode-Dateien von Thread Regionen greift ThreadViewer auf das `source`-Attribut der einzelnen Knoten zurück. Eine Initialisierung dieses Attributs ist jedoch nicht für jeden Knoten gegeben. Daher iteriert ThreadViewer über alle Knoten der entsprechenden Thread Region und verwendet nur diejenigen `source`-Attribute, die initialisiert und nicht leer sind.

Eine weitere Schwäche der instanziierten Klassen des Models ist das Fehlen der wichtigen `getThread(int threadID)`-Methode der Klasse `ThreadsInformation`, um auf einen Thread über seine ID zuzugreifen. Wir erweitern die Klasse manuell um diese Methode.

Das Model trifft die Annahme, dass eine Thread Region des zu analysierenden Programms durch eine entsprechende `.java`-Datei im Projektverzeichnis repräsentiert wird. Wenn dies beispielsweise durch das manuelle Löschen von `.java`-Dateien nicht der Fall ist, können die Filter der View Thread Regionen des eigenen Programms fälschlicherweise der Java-Standardbibliothek zuordnen und damit ausblenden.

3.3.2. Grenzen der View

Die Implementierung der View als Plugin erfordert einige Einarbeitungszeit im Umgang mit der Manifest-Datei `plugin.xml` zur individuellen Konfiguration von Eclipse. Insbesondere die von einem Standardwerk zur Plugin-Entwicklung[9] empfohlenen *Contributions* erweisen sich als nicht flexibel genug, um den Anforderungen eines umfangreicheren Plugins zu genügen. Mit *Contributions* lassen sich die Kontextmenüs und Werkzeugleisten von Eclipse als auch von eigenen Plugins um eigene Einträge erweitern. Da diese in der `plugin.xml` zu definieren sind, lassen sich deren Abhängigkeiten vom aktuellen Pluginzustand nur unzureichend beschreiben. Es ist beispielsweise nicht möglich, bestimmte Schaltflächen zu deaktivieren, wenn in der Graphansicht eine einzelne Thread Region selektiert wurde. Zur Beseitigung dieses Mankos werden in der Oberfläche von ThreadViewer die üblichen *Actions* verwendet. Damit steht die volle Funktionalität von Java zur Verfügung.

Auch die Verwendung des *HighlightPlugin* stellt sich als begrenzender Faktor heraus. Zur farblichen Hervorhebung von Quellcode steht eine nur eingeschränkte Farbgebung in Form von Integer-Werten im Bereich von -3 bis 50 zur Verfügung. Dies limitiert das Design von ThreadViewer deutlich, da die optische Einheit von Plugin und Eclipse so nicht optimal gegeben ist. Die verwendeten Farben zeigt die folgende Tabelle:

Farbe	Integer-Wert	Konstante
Grau	-3	<code>HighlightPlugin.FIRST_LEVEL</code>
Blau	2	—
Grün	24	—
Dunkelgelb	40	—
Orange	50	<code>HighlightPlugin.LAST_LEVEL</code>

3.3.3. Grenzen des Controllers

In der Methode `getSelectedNode(...)` des Controller besteht eine Ungenauigkeit in der Ermittlung des im Java-Editor markierten Knotens. Da eine Quellcode-Zeile eines geladenen Java-Programms im CSDG theoretisch durch eine beliebige Anzahl von Knoten repräsentiert werden kann, ist unklar, welcher Knoten vom Benutzer gewünscht ist. ThreadViewer nimmt der Einfachheit halber den ersten gültigen Knoten.

Eine ähnliche Problematik ergibt sich bei der Bestimmung der markierten Thread Region in der Methode `getSelectedRegion(...)`. Ein ermittelter Knoten kann theoretisch Teil einer beliebigen Anzahl von Thread Regionen sein. Auch hier nimmt ThreadViewer als vereinfachende Annahme die erste gültige Thread Region.

3.4. Statistik

Als Abschluss des Implementierungskapitel folgt nun eine alphabetisch nach Java-Paketen geordnete Statistik von ThreadViewer. Die Anzahl der Klassen und der Codezeilen je Paket werden mithilfe des *Eclipse Metrics Plugin*[2] berechnet.

Paket	Anzahl Klassen	Anzahl Codezeilen	Anzahl Methoden
actions	25	460	58
controller	2	1040	46
model	1	401	38
view	1	91	4
view.filter	3	48	3
view.listener	2	23	2
view.provider	4	537	42
view.sorter	2	152	4
view.view	6	998	58
Σ	46	3750	255

Mit den in diesem Kapitel vorgestellten Konzepten und Herausforderungen ist die Implementierung des Plugins abgeschlossen. Die konkrete Verwendung von ThreadViewer wird im nächsten Kapitel gezeigt.

4. Verwendung von ThreadViewer

Bei der Entwicklung von ThreadViewer ist großer Wert auf eine intuitive Bedienbarkeit und eine nahtlose Integration des Plugins in Eclipse gelegt worden. Dennoch ist bei der erstmaligen Benutzung von ThreadViewer eine kurze Einführung erforderlich.

Im folgenden Abschnitt 4.1 findet sich daher eine bebilderte Bedienungsanleitung für die grundlegenden Funktionen des Plugins. Abschnitt 4.2 zeigt anschließend weitere Impressionen aus der Verwendung von ThreadViewer.

4.1. Bedienungsanleitung

In diesem Abschnitt wird beispielhaft ein vollständiger Prozess bei der Verwendung des ThreadViewers erklärt und mit Bildern veranschaulicht.

Laden einer .pdg-Datei

Die gewünschte *.pdg-Datei* des zu analysierenden Java-Programms wird über das Kontextmenü im Package-Explorer geladen. ThreadViewer wird daraufhin geladen, wenn dies noch nicht bereits erfolgt ist.

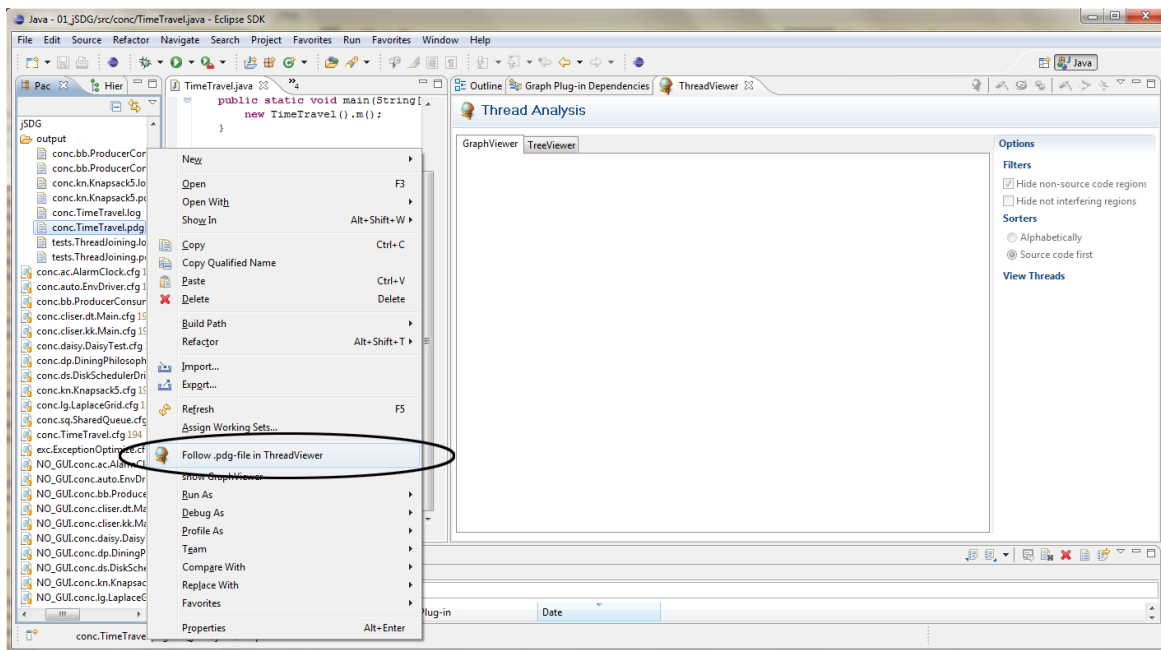


Abbildung 4.1.: Laden einer .pdg-Datei

Einblenden aller Thread Regionen

Alle *Thread Regionen*—einschließlich diejenigen von Java-Standardbibliotheken—werden über den Filter in den Optionen auf der rechten Seite eingblendet. Die Objekte werden automatisch optimal repositioniert.

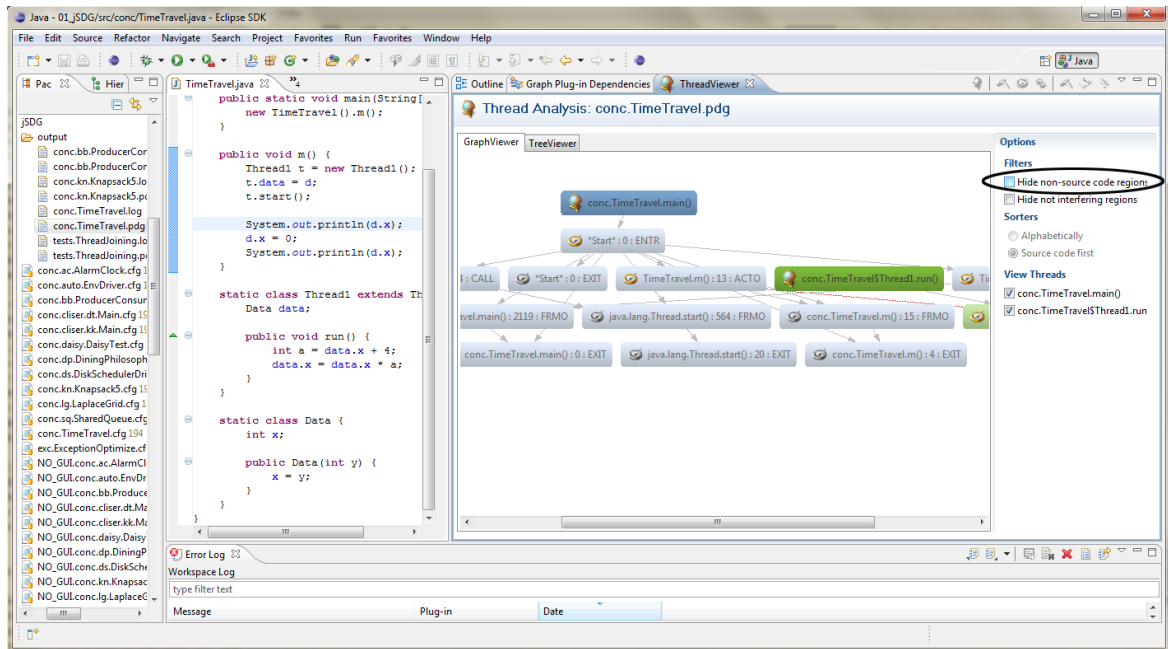


Abbildung 4.2.: Einblenden aller Thread Regionen

Doppelklick in der Graphenansicht

Im ThreadViewer bewirkt ein Doppelklick auf ein Objekt das *Folgen* derselben im Java-Editor.

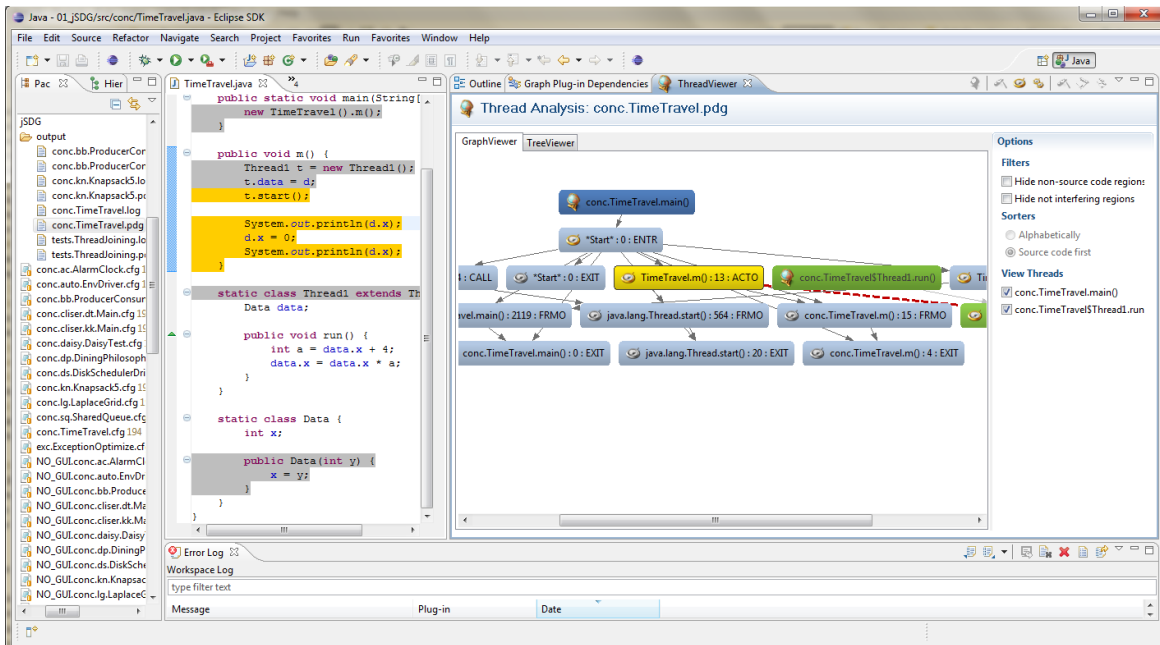


Abbildung 4.3.: Doppelklick in der Graphenansicht

Folgen paralleler Thread Regionen

Parallele Thread Regionen können alternativ über die Werkzeugleiste verfolgt werden. Die ursprüngliche Region wird im Quelltext grau markiert, während die parallele Region im Quelltext und im ThreadViewer gelb hervorgehoben wird.

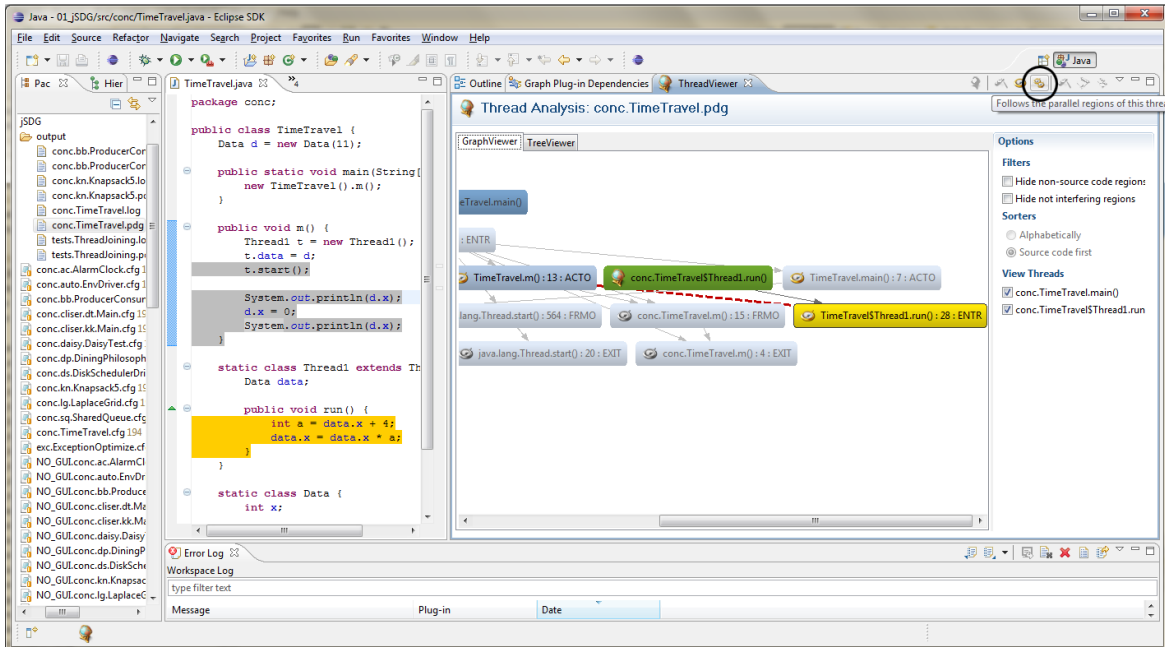


Abbildung 4.5.: Folgen paralleler Thread Regionen

Folgen in der Baumansicht

Die nachverfolgte parallele Thread Region wird ebenfalls automatisch in der *Baumansicht* markiert.

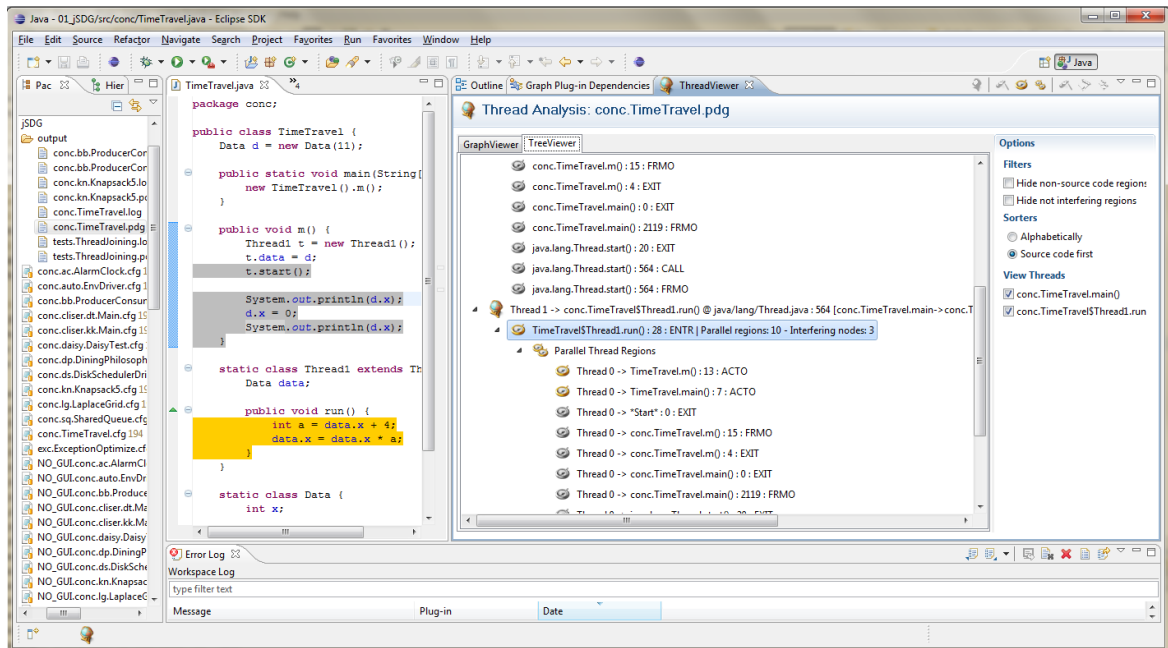


Abbildung 4.6.: Folgen in der Baumansicht

Folgen einer Interferenz

Eine *Interferenz* kann beispielsweise über das Kontextmenü der gewünschten Abhängigkeit verfolgt werden. Die interferierenden Thread Regionen werden optisch einheitlich im Java-Editor und im ThreadViewer hervorgehoben.

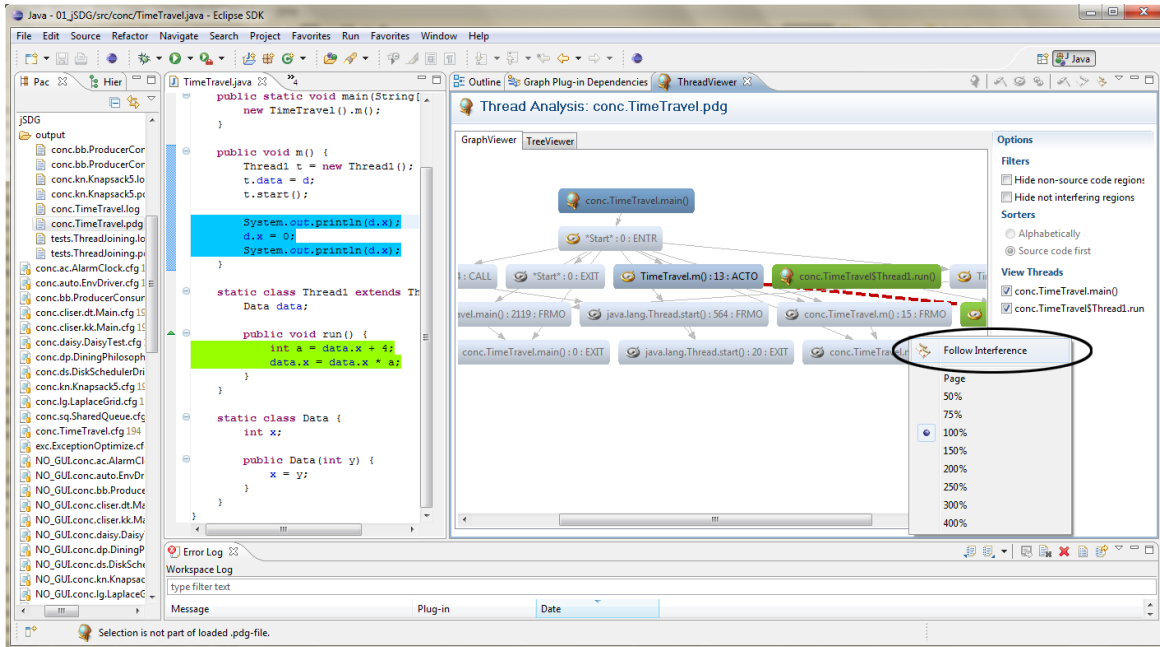


Abbildung 4.7.: Folgen einer Interferenz

Folgen eines interferierenden Knotens

Das Folgen eines *interferierenden Knotens* ist über das Setzen des Mauscurors an die entsprechende Quelltext-Zeile sowie einen Aufruf im Kontextmenü möglich. ThreadViewer wechselt daraufhin automatisch in die Baumansicht, um den Knoten zu selektieren.

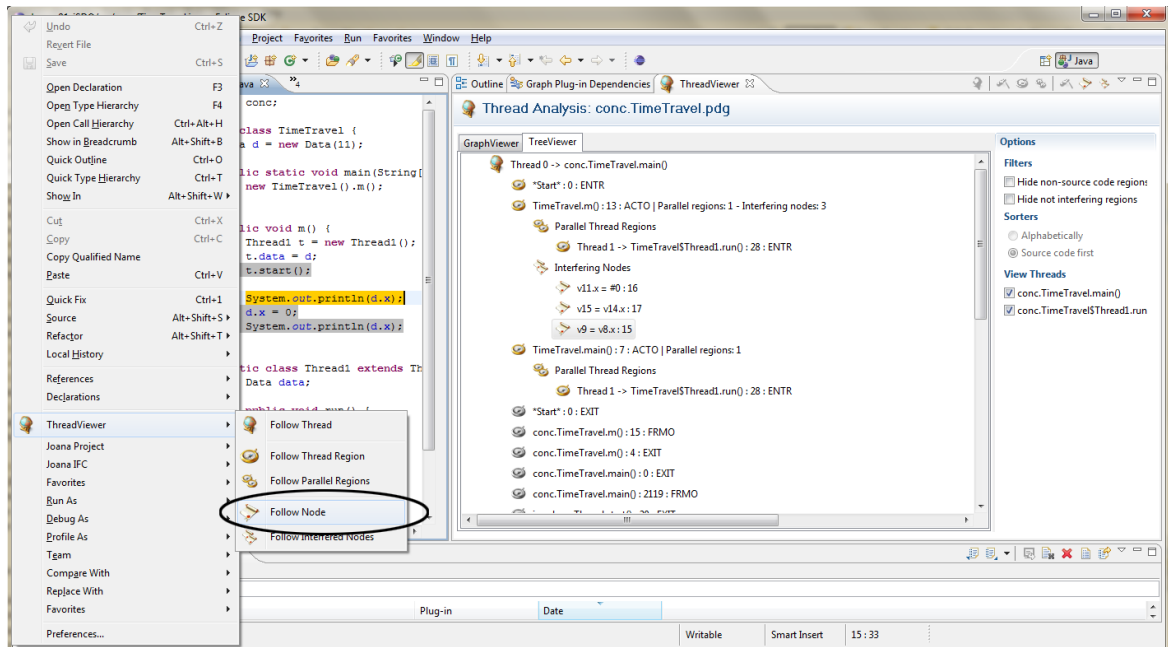


Abbildung 4.8.: Folgen eines interferierenden Knotens

Ausblenden von Regionen ohne Interferenz

Wenn im Graphen nur die Interferenzen von Bedeutung sind, können alle Thread Regionen *ohne Interferenz* ausgeblendet werden.

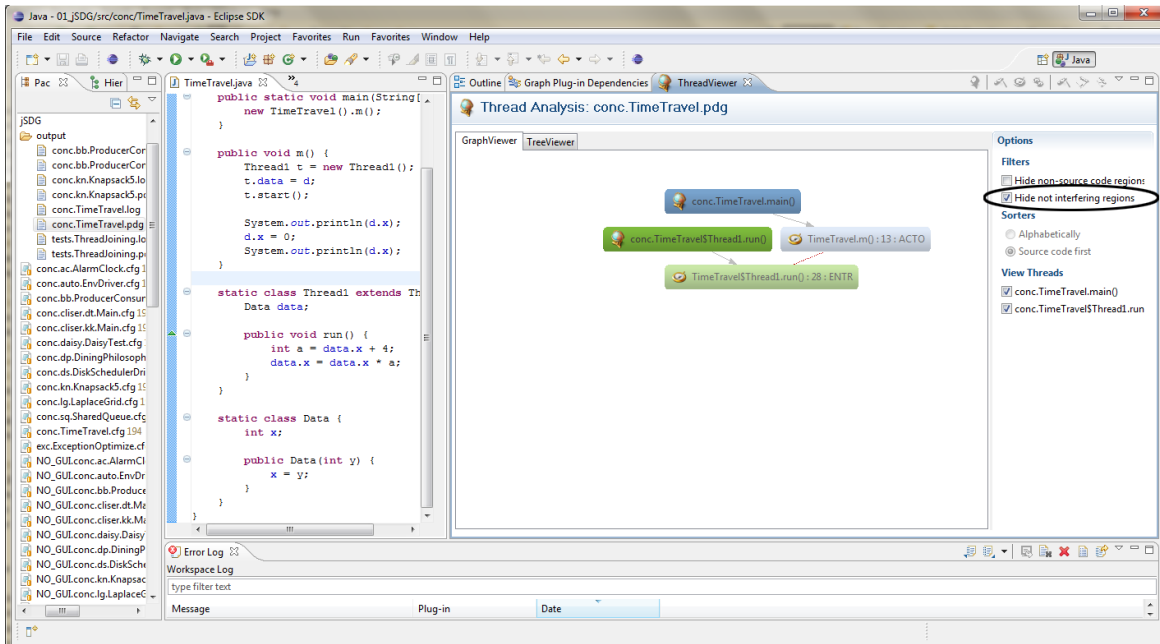


Abbildung 4.9.: Ausblenden von Regionen ohne Interferenz

Ausblenden einzelner Threads

Zur verbesserten Übersicht können einzelne Threads über den Optionen-Bereich ausgeblendet werden. Thread Regionen mit *versteckten Interferenzen* werden in diesem Modus mit einem roten Rand versehen.

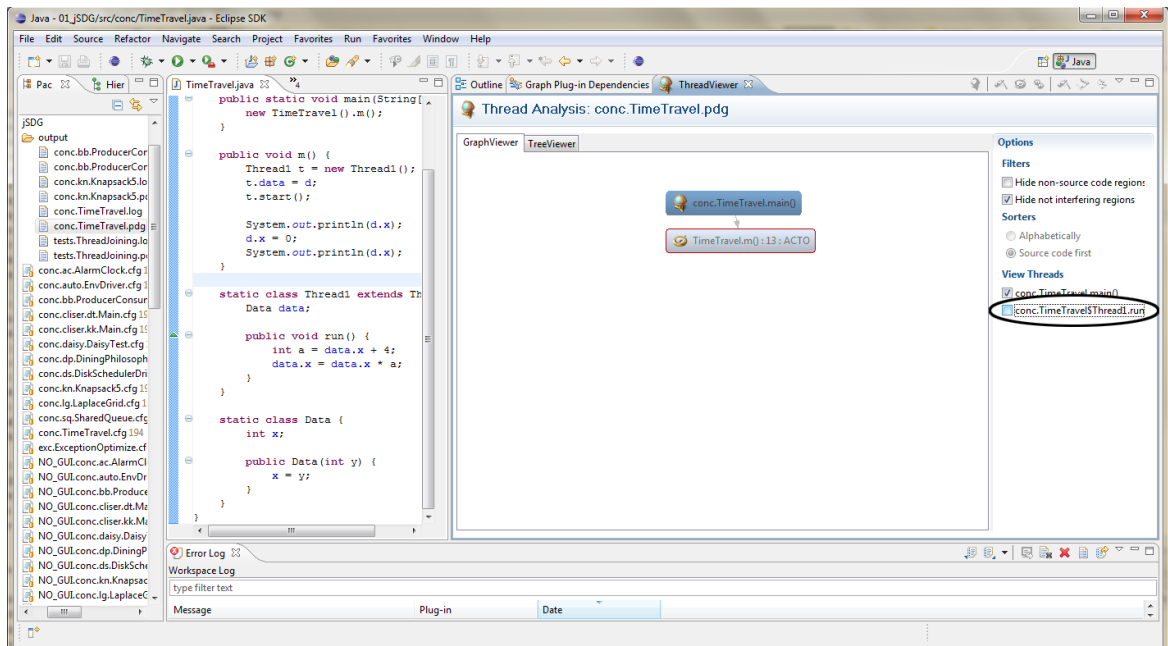


Abbildung 4.10.: Ausblenden einzelner Threads

4.2. Impressionen aus ThreadViewer

Im Folgenden ausgewählte Impressionen aus ThreadViewer.

Navigieren mit der Baumansicht

Der Baum lässt sich bis zu einzelnen interferierenden Knoten expandieren. Über das Kontextmenü können gewünschte Objekte jederzeit nachverfolgt werden.

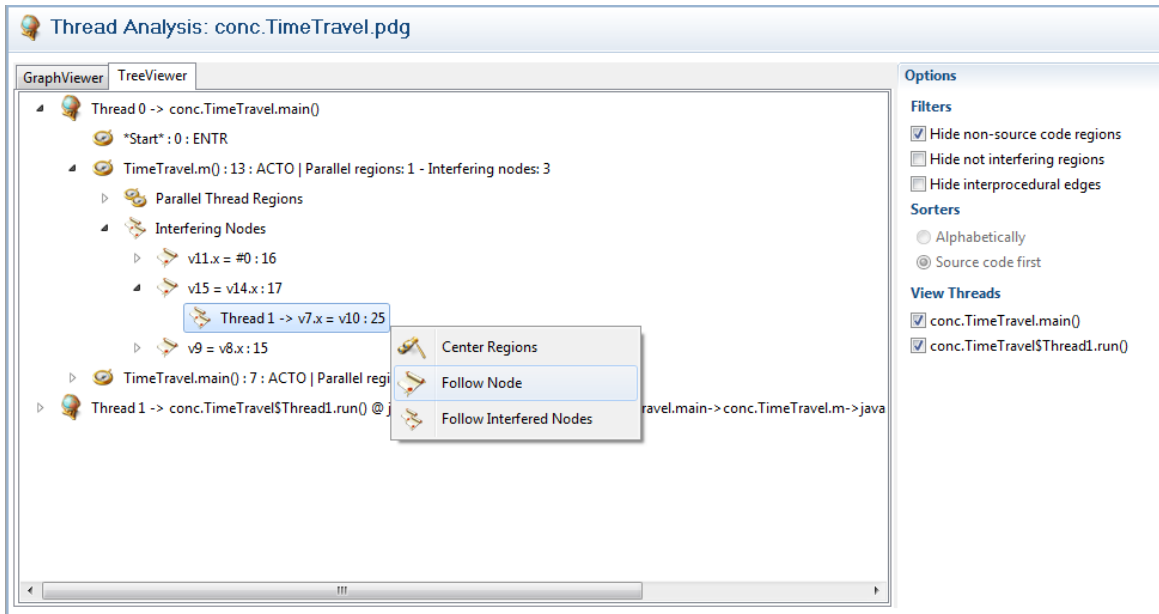


Abbildung 4.11.: Navigieren mit der Baumansicht

Aufbau des Graphen

Beim ersten Laden einer .pdg-Datei wird der Graph automatisch durch das *Zest*-Framework aufgebaut. Das weitere Filtern von Thread Regionen führt zu einer automatischen Repositionierung aller Elemente, wobei eine minimale Anzahl an Überlappungen von Kanten sichergestellt wird.

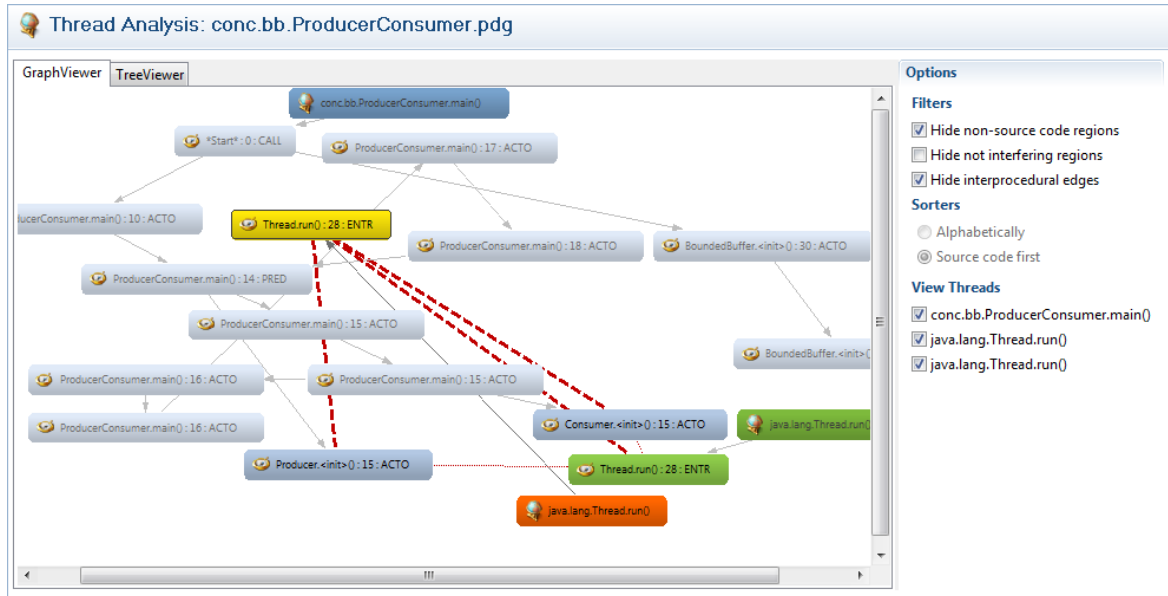


Abbildung 4.12.: Aufbau des Graphen

Folgen einer Interferenz

Die interferierenden Thread Regionen einer nachverfolgten Interferenz werden sowohl im ThreadViewer als auch im Java-Editor optisch einheitlich hervorgehoben.

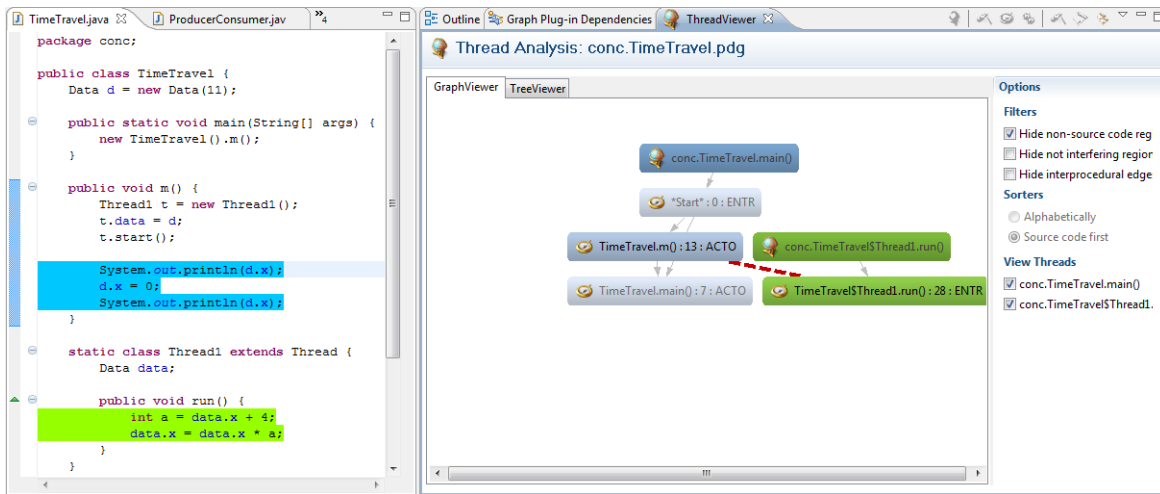


Abbildung 4.13.: Folgen einer Interferenz

Interferierende Regionen von DaisyTest

Die folgenden Impression zeigt alle interferierenden Thread Regionen von DaisyTest.java, einem weiteren Java-Programm.

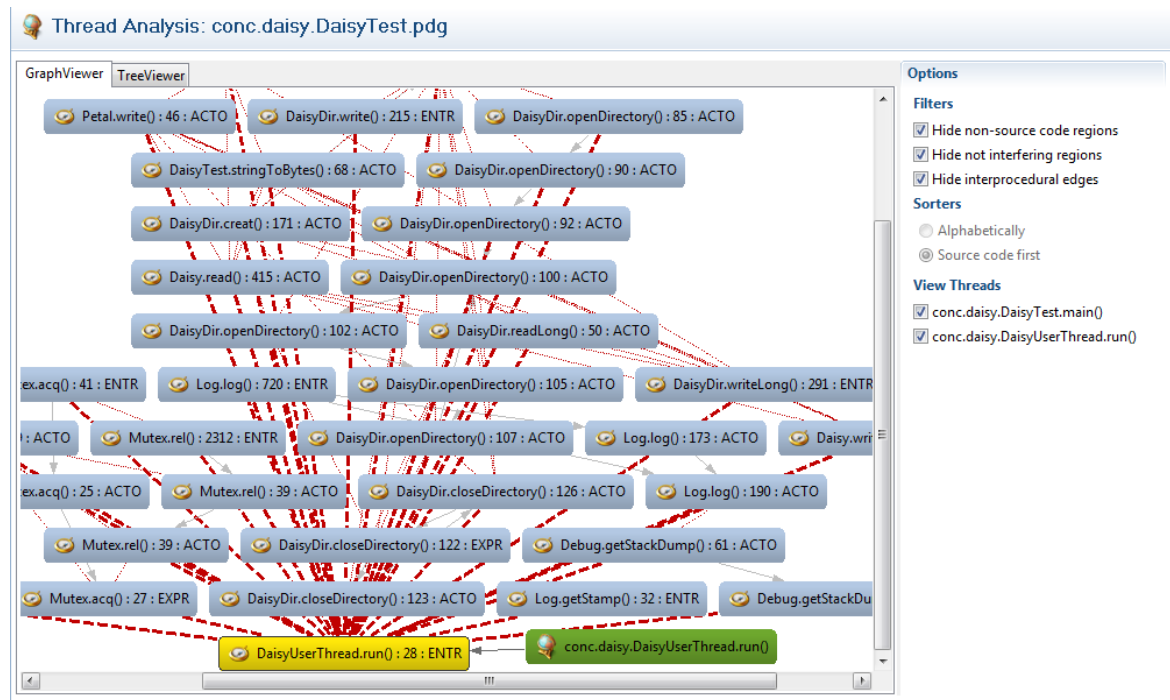


Abbildung 4.14.: Interferierende Regionen von DaisyTest.java

DaisyTest auf minimaler Zoomstufe

Abschließend alle Thread Regionen von `DaisyTest.java` auf minimaler Zoomstufe, sodass alle Regionen sichtbar sind.

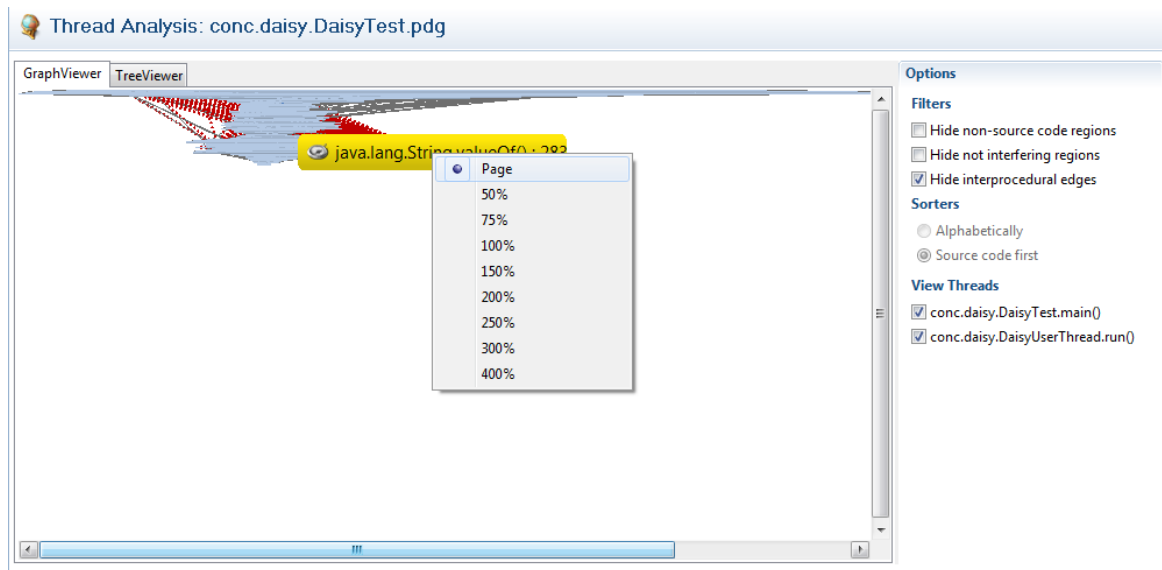


Abbildung 4.15.: aisyTest auf minimaler Zoomstufe

In diesem Kapitel sind die wichtigsten Funktionen von ThreadViewer veranschaulicht worden. Diese werden nun im nächsten Kapitel hinsichtlich ihrer Laufzeiten evaluiert.

5. Evaluation

Das vollständig implementierte Plugin wird in diesem Kapitel in Hinblick auf seine Funktions- und Leistungsfähigkeit evaluiert. Im Kern ist ThreadViewer ein Visualisierungstool, das vorhandene Daten über Graphen aggregiert und dem Benutzer optisch aufbereitet präsentiert. Während der Implementierung und nach Abschluss der Entwicklung ist der ThreadViewer durch das Laden und die Anzeige von verschiedenen Java-Programmen verifiziert worden. Insbesondere mit dem später in 6.2 vorgestellten *Graph Viewer* sind der Kontrollfluss und die visualisierten Interferenzen stichprobenartig auf Vollständigkeit und Korrektheit überprüft worden.

Im ersten Abschnitt wird der Testaufbau mit diversen Test-Programmen und einfachen Benutzerinteraktionen als Testfälle vorgestellt. Der zweite Abschnitt misst deren Laufzeiten. Der dritte Abschnitt evaluiert den Gewinn an Übersichtlichkeit durch das gezielte Filtern von uninteressanten Thread Regionen.

5.1. Testaufbau

Um die Leistungsfähigkeit des Plugins akkurat zu evaluieren, verwenden wir verschiedene Java-Programme als Testszenarien. In den Testszenarien werden jeweils die Laufzeiten einfacher Benutzerinteraktion als Testfälle gemessen. Die Laufzeit wird dabei mittels einfacher Aufrufe von `System.out.println(System.currentTimeMillis())` ermittelt und anschließend der Median von fünf Testläufen genommen, um unrealistische Laufzeiten auszuschließen.

Die Messungen erfolgen an einem Notebook mit einem Intel® Core™ i7 720QM mit 4 Kernen à 1.60 GHZ, 3 GByte Hauptspeicher und Windows 7 Professional.

5.1.1. Testszenarien

Die zum Testen verwendeten Java-Programme sind aus der Dissertation von Giffhorn[11] entnommen und dienen in diesem Kapitel als Testszenarien:

- In *TimeTravel* greifen zwei Threads auf eine gemeinsame Variable zu.
- In *ProducerConsumer* teilen sich zwei Threads einen begrenzten Buffer.
- In *DiningPhilosophers* befindet sich eine beliebige Menge an Akteuren in einem Zustand gegenseitiger Abhängigkeit.
- In *SharedQueue* schreibt ein Thread Zeichenketten in eine Queue, die von einem anderen Thread ausgelesen wird.
- In *DaisyTest* werden Dateizugriffe simuliert.

Die große Bandbreite in der Anzahl an Thread Regionen, Knoten und Interferenzen der einzelnen Testszenarien veranschaulicht die folgende Tabelle:

Testszenario	Threads	Regionen	Knoten	Kanten	Interferenzen
TimeTravel	2	12	2717	41778	1
ProdCons	3	264	3561	46313	79
DiningPhils	2	318	5661	137934	77
SharedQueue	3	503	11889	110432	101
DaisyTest	2	2276	44770	511226	1323

5.1.2. Testfälle

Als Testfälle werden die folgenden einfachen Benutzerinteraktionen für die Testszenarien herangezogen und deren Laufzeit ermittelt:

- .pdg-Datei laden, MHP-Information vollständig berechnen und Graphen anzeigen.
- Einer Thread Region im Graphen folgen.
- Einer Thread Region im Java-Editor folgen.
- Alle Thread Regionen im Graphen einblenden.
- Alle Kanten im Graphen einblenden.
- Alle Thread Regionen in der Baumansicht einblenden.
- In der Baumansicht die Thread Regionen alphabetisch sortieren.

Die Testfälle decken die Hauptfunktionalitäten von ThreadViewer ab und sind somit für die Evaluierung der Leistungsfähigkeit repräsentativ.

5.2. Laufzeiten

Die Laufzeiten der Testfälle werden nun für jedes Testszenario am beschriebenen Testsystem jeweils fünf Mal ermittelt und der Median der fünf Werte bestimmt. Es ergeben sich die folgenden Werte:

Testszenario	Laden	Graph folgen	Editor folgen	Graph-Regionen	Graph-Kanten	Baum-Regionen	Sortieren
TimeTravel	0,671 s	0,094 s	0,078 s	0,296 s	0,078 s	0,296 s	0,063 s
ProdCons	1,575 s	0,125 s	0,125 s	1,139 s	0,156 s	1,076 s	0,187 s
DiningPhils	3,354 s	0,094 s	0,094 s	1,389 s	0,078 s	1,654 s	0,250 s
SharedQueue	6,131 s	0,124 s	0,125 s	2,589 s	0,468 s	2,512 s	0,484 s
DaisyTest	85,067 s	0,578 s	0,546 s	27,737 s	51,496 s	28,158 s	16,209 s

Die Testergebnisse zeigen eine steigende Laufzeit für einzelne Interaktionen bei zunehmender Komplexität der Testszenarien. Die Komplexität hängt dabei von der Anzahl an Thread Regionen und Knoten und dem Grad der Vernetzung durch die Anzahl an Kanten ab. So korreliert die Laufzeit für das Einblenden aller Thread Regionen in der Graphansicht natürlicherweise stark mit der Anzahl an Thread Regionen in 5.1.1.

Generell ist das Laden einer .pdg-Datei die mit Abstand teuerste Operation bei der Verwendung von ThreadViewer. Die zeitintensive Vorberechnung der Kontrollflüsse und die speicherintensive Bereitstellung in HashMaps führen jedoch zu guten Reaktionszeiten bei der Navigation und Bedienung des Plugins. Das mit Abstand komplexeste Testszenario Daisy-Test ist mit Laufzeiten im hohen zweistelligen Sekundenbereich nicht mehr bedienbar. Einzig einfache Folge-Operationen werden reibungslos ausgeführt.

Zusammenfassend ergibt die Evaluation der Testfälle, dass Java-Programme bis mittlerer Komplexität mit weniger als 1.000 Thread Regionen, ungefähr 10.000 Knoten und 100.000 Kanten durch ThreadViewer sehr gut visualisierbar sind. Insbesondere bei komplexeren Programmen liegt großes Potenzial in der Parallelisierung der Vorberechnungen.

5.3. Visualisierte Thread Regionen

Dieser Abschnitt zeigt, wie das gezielte Filtern von uninteressanten Thread Regionen die Übersichtlichkeit der Visualisierung von großen Programmen drastisch erhöht. Die nachfolgende Tabelle stellt anhand der Testszenarien die Anzahl zu visualisierender Thread Regionen nach dem Einsatz der Filter von 3.2.2 dar.

Testszenario	Thread Regionen	Regionen ohne Standardbibliotheken	Regionen mit Interferenzen	Regionen mit Interferenzen ohne Standardbibliotheken
TimeTravel	12	4	2	2
ProdCons	264	15	27	4
DiningPhils	318	11	36	3
SharedQueue	503	18	34	4
DaisyTest	2276	388	170	35

Alleine das Ausblenden von Thread Regionen von Standardbibliotheken reduziert die Anzahl der visualisierten Regionen um mindestens 75 %. Bei den ausgeblendeten Regionen handelt es sich zumeist um Programmabschnitte von Java-Bibliotheken oder statischen Initialisierungsmethoden. Durch die zusätzliche Aktivierung der Option, nur interferierende Regionen anzuzeigen, werden nochmals mindestens weitere 50 % der noch vorhandenen Regionen ausgeblendet. Beim Java-Programm *SharedQueue* verbleiben bei Aktivierung aller Filter von ursprünglich 503 nur noch 4 Thread Regionen, die auch tatsächlich für den Benutzer von Interesse sind. Dies sind weniger als 1 % der anfänglichen Anzahl an Thread Regionen. Die Übersichtlichkeit wird somit drastisch erhöht.

Nach der erfolgten Evaluierung von ThreadViewer folgt im nächsten Kapitel eine abschließende Zusammenfassung der Ergebnisse sowie ein Ausblick auf verwandte Arbeiten und mögliche Verbesserungen.

6. Fazit und Ausblick

Die vorliegende Bachelorarbeit stellt ein Eclipse-Plugin zur Visualisierung von Threads und ihren Interferenzen vor. Grundlage bilden die Analysen des ValSoft/Joana-Projektes, die um eigene Berechnungen erweitert wurden. Das Plugin kann kleine bis mittelgroße Java-Programme mithilfe einer Baum- und Graphenansicht anzeigen. Diverse Filter- und Sortier-Optionen ermöglichen eine individuell konfigurierbare Anzeige der Interferenzen zwischen Thread Regionen. Eine nahtlose Integration in Eclipse erlaubt den intuitiven Wechsel zwischen ThreadViewer und den entsprechenden Quellcode-Dateien im Java-Editor. Die einzelnen Funktionen wurden am Fallbeispiel `TimeTravel.java` veranschaulicht.

Als Fazit lässt sich festhalten, dass dem Software-Entwickler mit dem Plugin ein effektives Werkzeug zur Verfügung steht, um nebenläufige Programme mittlerer Komplexität auf kritische Abhängigkeiten zwischen einzelnen Programmabschnitten hin zu untersuchen. Damit ist er in der Lage, sein Programm besser zu verstehen und schließlich zu optimieren.

Dieses letzte Kapitel behandelt nun zunächst Anwendungsmöglichkeiten des Plugins, stellt danach thematisch verwandte Arbeiten vor und skizziert in einem Ausblick mögliche Erweiterungen von ThreadViewer.

6.1. Einsatzmöglichkeiten

Die strukturierte Übersicht des ThreadViewers über Thread Regionen und mögliche Interferenzen geben dem Benutzer die nachfolgenden Einsatzmöglichkeiten.

6.1.1. Entwicklung nebenläufiger Anwendungen

Bei der Entwicklung nebenläufiger Anwendungen kann der Software-Entwickler das vollständige Programm oder Zwischenstadien desselben durch ThreadViewer visualisieren lassen. Er erhält somit ein fundiertes Verständnis des Kontrollflusses und der Abhängigkeiten in seinem Code. Interferenzen erkennt er in der Graphenansicht von ThreadViewer direkt auf aggregierter Ebene der Thread Regionen und kann im Anschluss zu den interferierenden Knoten in der Baumansicht navigieren.

So ist vorstellbar, dass mehrere Entwickler an jeweils unterschiedlichen Thread-Klassen eines Software-Projekts arbeiten und dabei Referenzen von Objekten des jeweils anderen Threads modifizieren. ThreadViewer deckt in diesem Fall diese Interferenzen auf.

6.1.2. Plausibilitätsprüfung für Slicing und Chopping



Mit dem ThreadViewer können die Ergebnisse von Slicing- und Chopping-Verfahren ~~validiert~~ werden. Details zu beiden Verfahren erläutern die folgenden zwei Teilabschnitte.




Slicing

Beim *Program Slicing* werden alle Anweisungen eines Programms bestimmt, die eine gegebene Anweisung zu einem bestimmten Programmzeitpunkt beeinflussen oder aber von ihr beeinflusst werden. Im ersten Fall sprechen wir von einem *backward slice*; im zweiten Fall von einem *forward slice*.

Slicing ist eine universelle Technik, die beispielsweise beim Debuggen von fehlerhaftem Programmverhalten[16] oder zur Vermeidung von dupliziertem Code[17] verwendet wird. Mit einem zeitsensitiven Slicer[19, 20] können sich gegenseitig ausschließende Interferenzen wie im Fallbeispiel `TimeTravel.java` erkannt werden. Zur konkreten Berechnung von Slices in parallelen Programmen sei auf Giffhorns Dissertation[11] verwiesen. Nach erfolgtem Slicen können die resultierenden Slices anhand der tatsächlichen Interferenzen im ThreadViewer gegengeprüft werden.

Chopping

Bei dem eng mit dem Slicing verwandten *Chopping* von Programmen werden alle beeinflussten Anweisungen auf dem Pfad von einer gegebenen Anweisung s zu einer zweiten Anweisung t berechnet. Prinzipiell entsprechen diese Anweisungen dem Schnitt des forward slice von s mit dem backward slice von t . Hier sei wieder auf die Dissertation von Giffhorn[11] verwiesen.

Wie beim Slicing können auch nach erfolgtem Chopping die Ergebnisse durch die Visualisierung der Interferenzen im ThreadViewer ~~verifiziert werden.~~ 

6.2. Verwandte Arbeiten

Der vorgestellte ThreadViewer ist unseres Wissens nach erste Software-Anwendung zur Visualisierung von Thread, Thread Regionen und ihren Interferenzen. Jedoch sind bereits ähnliche Visualisierungswerkzeuge für andere Programmiersprachen oder mit einem anderen Fokus bei der Anzeige entwickelt worden. Es folgt daher ein kurzer Überblick über thematisch verwandte Entwicklungen.

6.2.1. CodeSurfer

Der *CodeSurfer*[1] von GrammaTech ist das bis dato einzig bekannte kommerzielle Framework zur Programmanalyse von C/C++-Programmen. Im Gegensatz zum ThreadViewer unterstützt der CodeSurfer kein Java. Sein Fokus liegt weniger in der einfachen Visualisierung von Interferenzen, als vielmehr in einer umfangreichen Analyse zum Slicen und Choppen von Programmen. Er ist mächtig genug, Programme von bis zu 300.000 Codezeilen zu analysieren.

6.2.2. GraphViewer

Der *GraphViewer*[6] des ValSoft/Joana-Frameworks visualisiert ebenfalls einen CSDG eines Java-Programms. Eine Sicht auf Ebene von Methoden und eine Sicht auf Ebene der Knoten einer Methode zeigen detailliert alle Kontrollflüsse und Abhängigkeiten des Programms. Jedoch mangelt es dem GraphViewer an einer aggregierten Sicht von Abhängigkeiten auf Methodenebene, um dem Benutzer einen Überblick über wesentliche Interferenzen zu ermöglichen. Durch die fehlende Klassifizierung von Knoten nach Threads und eine fehlende



~~Unterstützung des Springens in entsprechende Quellcode-Dateien~~ ist die Bedienung weniger intuitiv als beim ThreadViewer.

6.3. Ausblick

Nicht alle Ideen konnten innerhalb der vorgegebenen Bearbeitungszeit verwirklicht werden. Praktisch denkbar sind die folgenden Erweiterungen:

- ThreadViewer wird um eine *One Click*-Funktionalität erweitert: Ein einziger Klick des Benutzers wandelt das aktuell bearbeitete Java-Programm in eine .pdg-Datei um und visualisiert sie direkt in ThreadViewer.
- Die vielen Thread Regionen von Java-internen Methodenaufrufen des Hauptthreads werden zu einer repräsentativen Region zusammengefasst.
- Analog werden Thread Regionen bei Parameterübergaben einer Methode zu einer Thread Region zusammengefasst.
- Eine weitere Ansicht listet die wichtigsten Eigenschaften des visualisierten CSDG auf.
- Die Laufzeiten beim Laden eines Graphen und bei der Bedienung des ThreadViewers werden optimiert. Insbesondere die Vorberechnung der Nachfolge-Beziehungen kann parallelisiert werden. Der Grad der Interaktion des Plugins wird dadurch erhöht.

Diese möglichen Erweiterungen erhöhen den Komfort und die Übersichtlichkeit und würden hierdurch eine noch stimmigere Bedienung des ThreadViewers erlauben.

A. CSDGs in ThreadViewer

Dieser Anhang zeigt die Visualisierungen der Java-Programme aus den Testszenarien von 5.1.2. ThreadViewer stellt im Folgenden alle Thread Regionen mit vorhandenem Quellcode auf optimaler Zoomstufe dar, sodass der gesamte CSDG sichtbar ist.

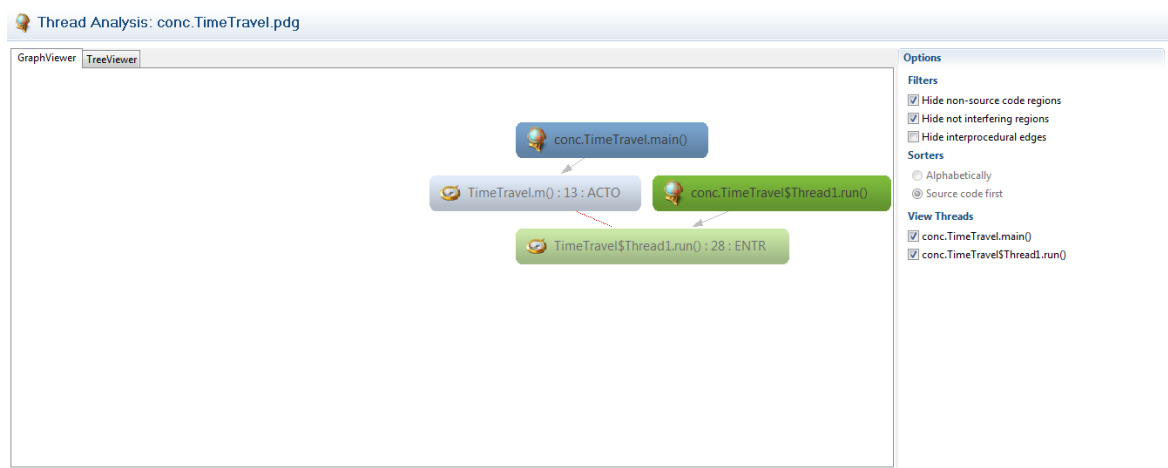
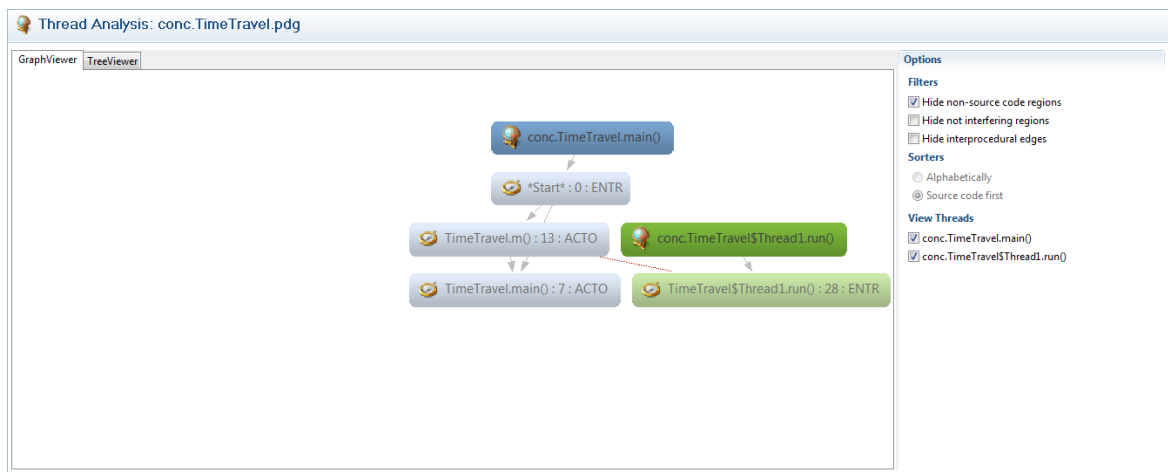
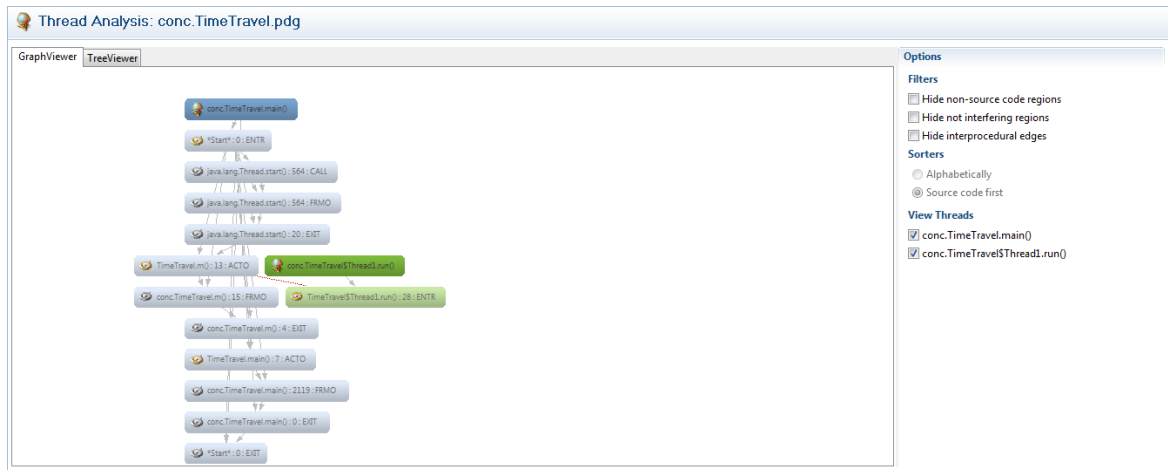


Abbildung A.1.: Visualisierungen des CSDG von TimeTravel.java bei der Aktivierung verschiedener Filter

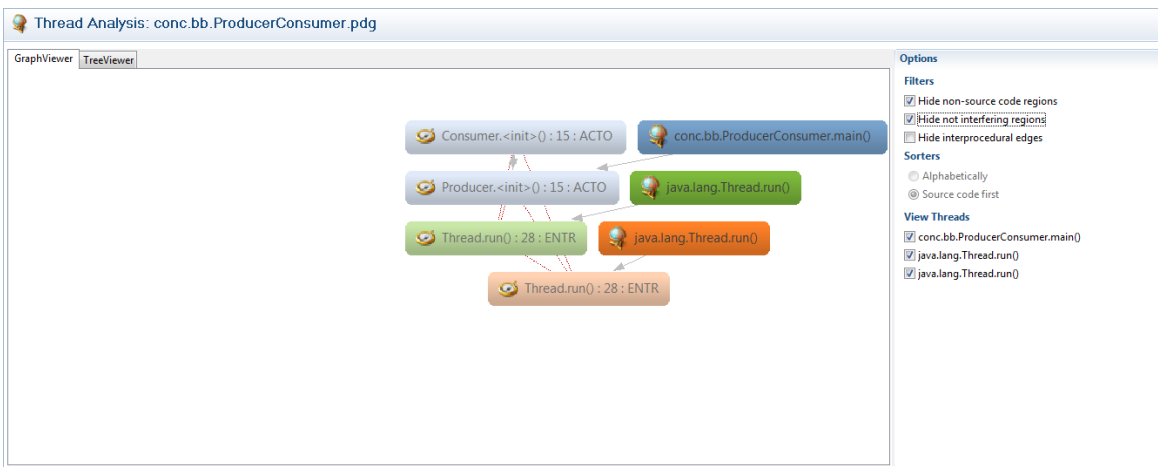
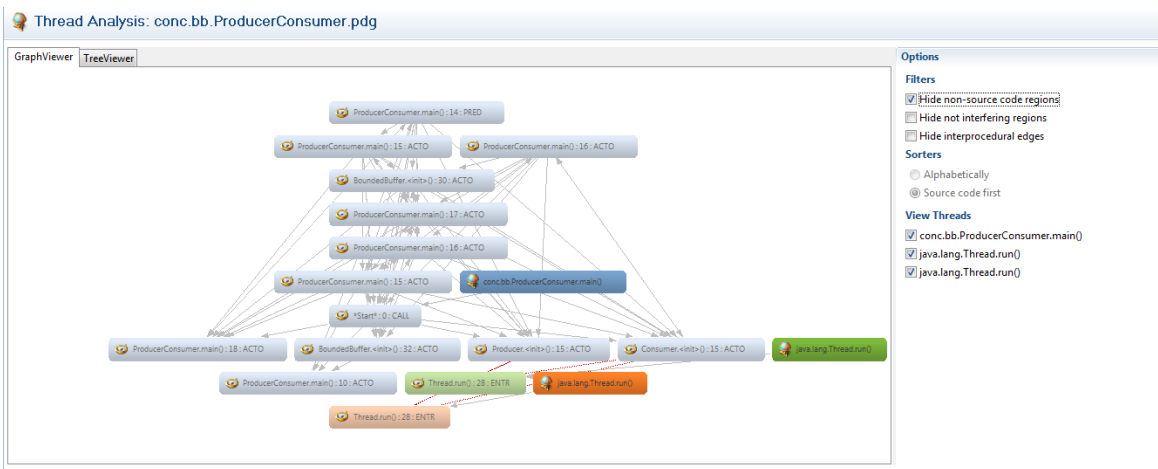
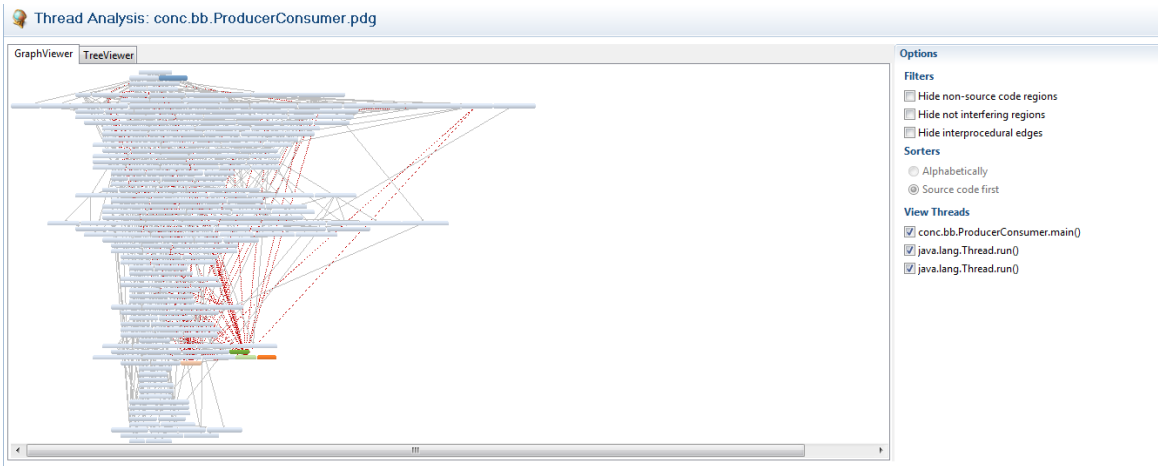


Abbildung A.2.: Visualisierungen des CSDG von `ProducerConsumer.java` bei der Aktivierung verschiedener Filter

A. CSDGs in ThreadViewer

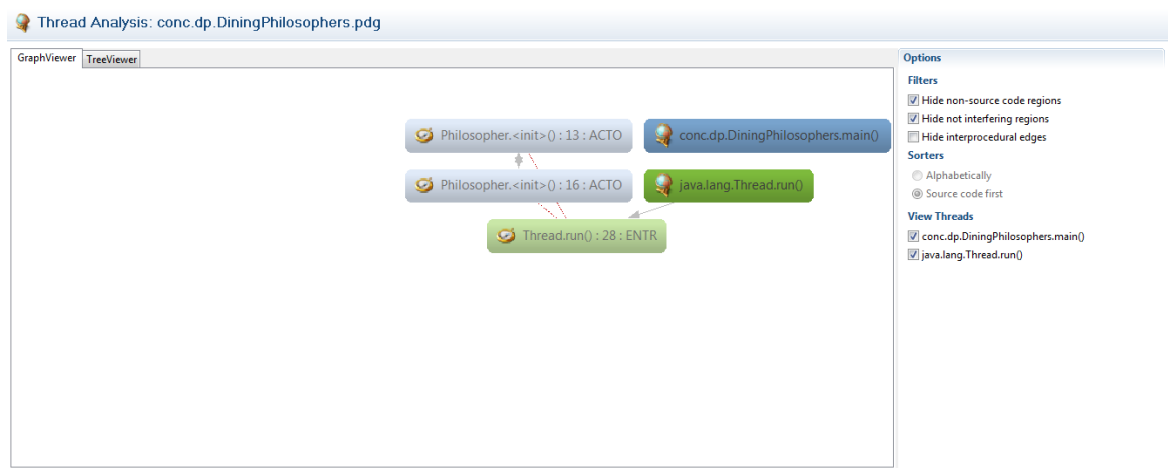
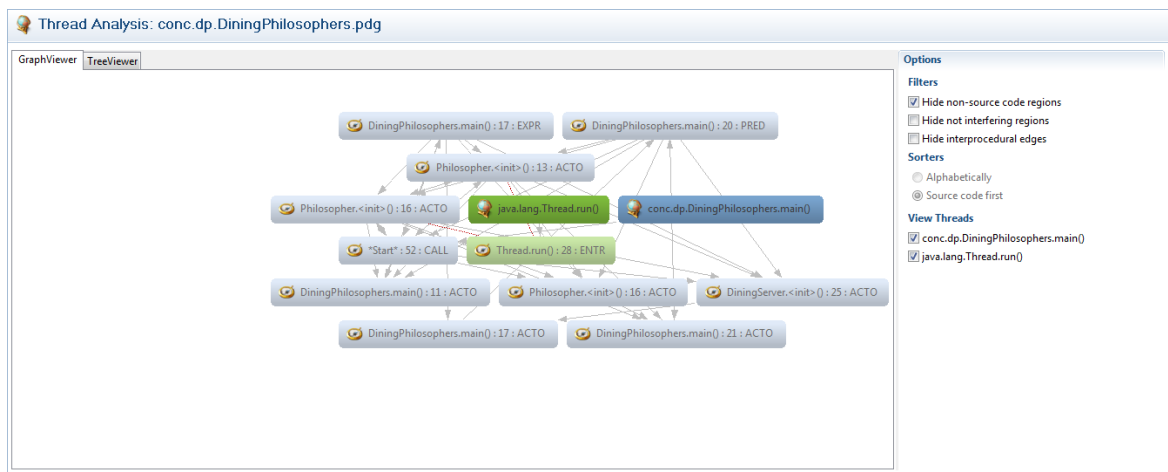
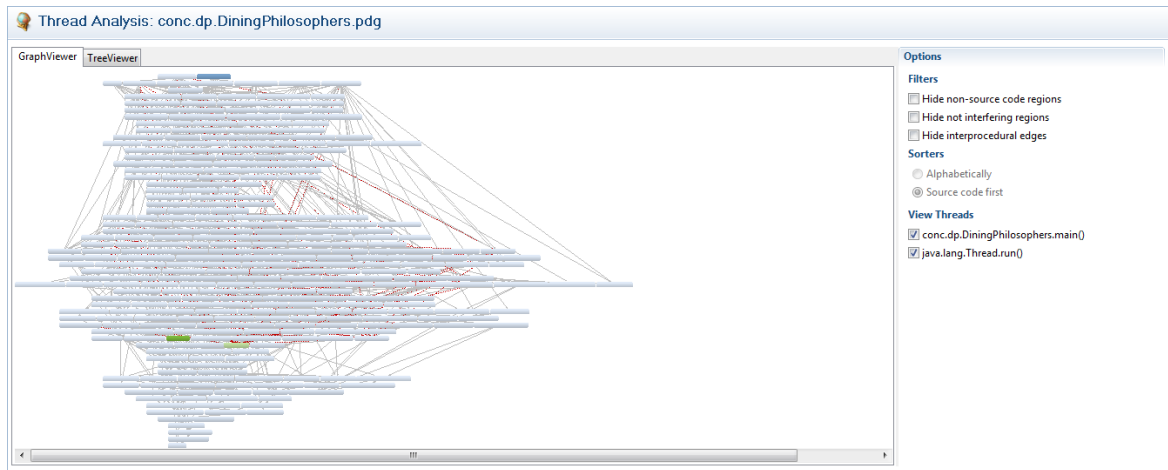


Abbildung A.3.: Visualisierungen des CSDG von `ProducerConsumer.java` bei der Aktivierung verschiedener Filter

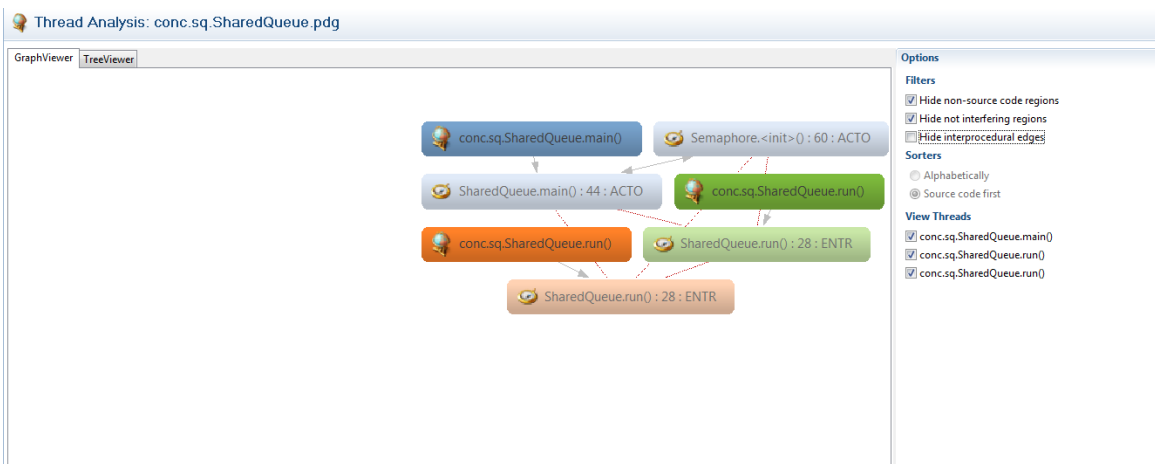
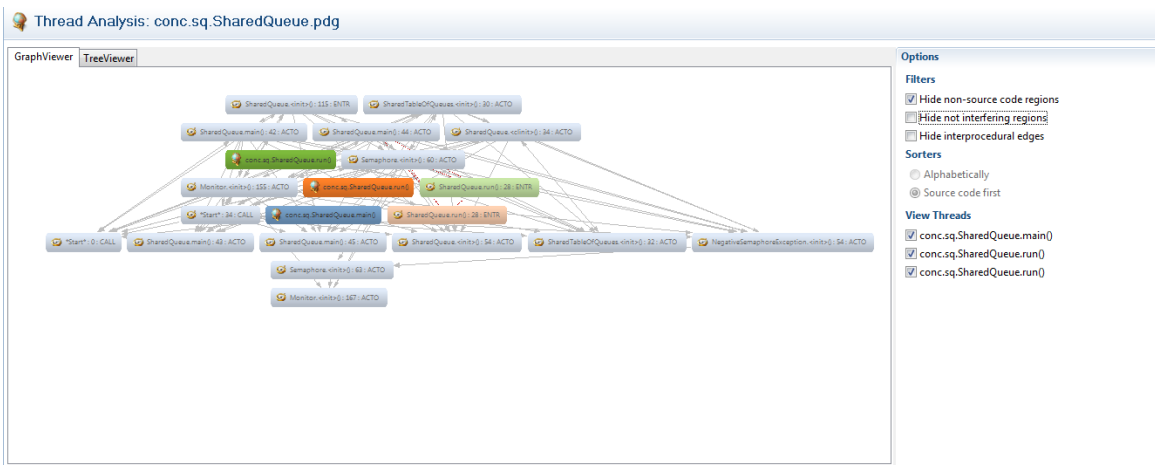
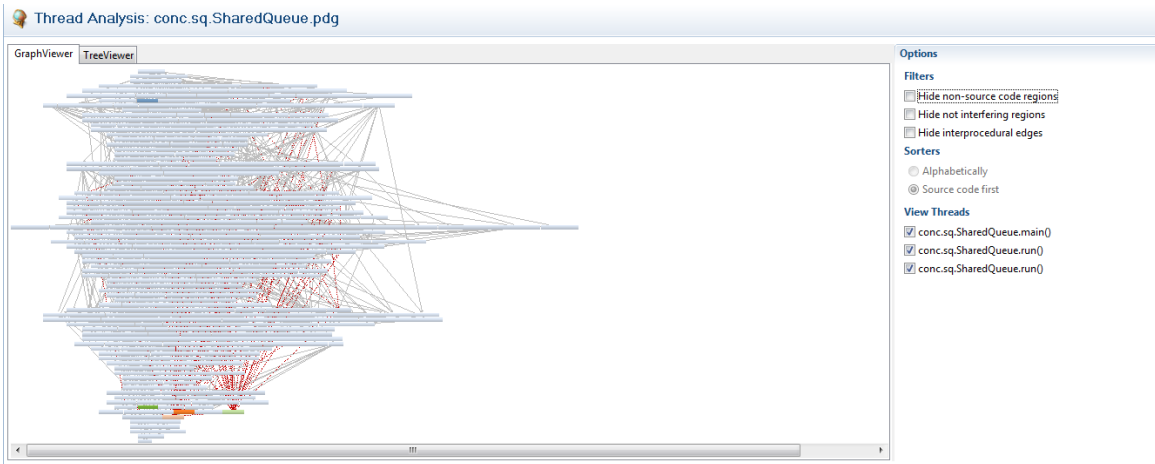


Abbildung A.4.: Visualisierungen des CSDG von SharedQueue.java bei der Aktivierung verschiedener Filter

A. CSDGs in ThreadViewer

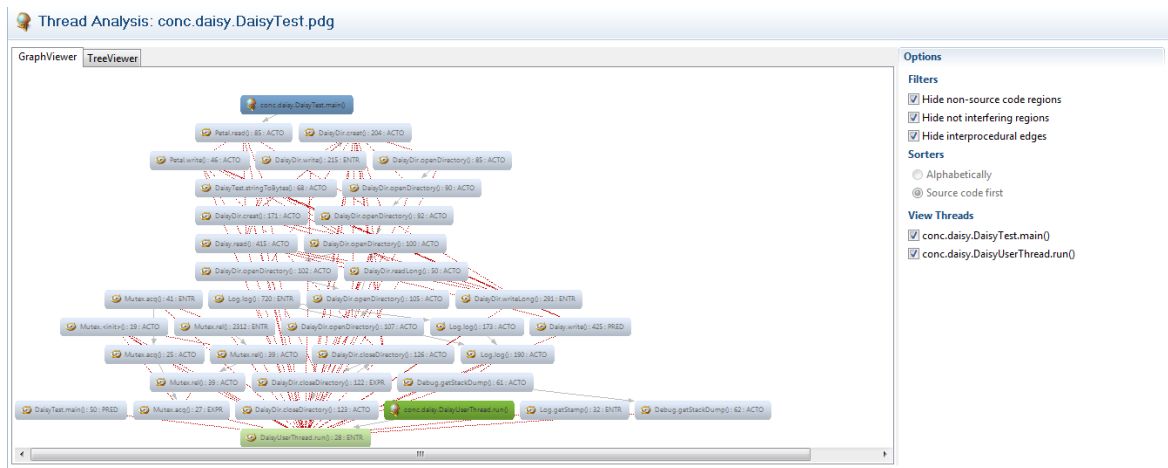
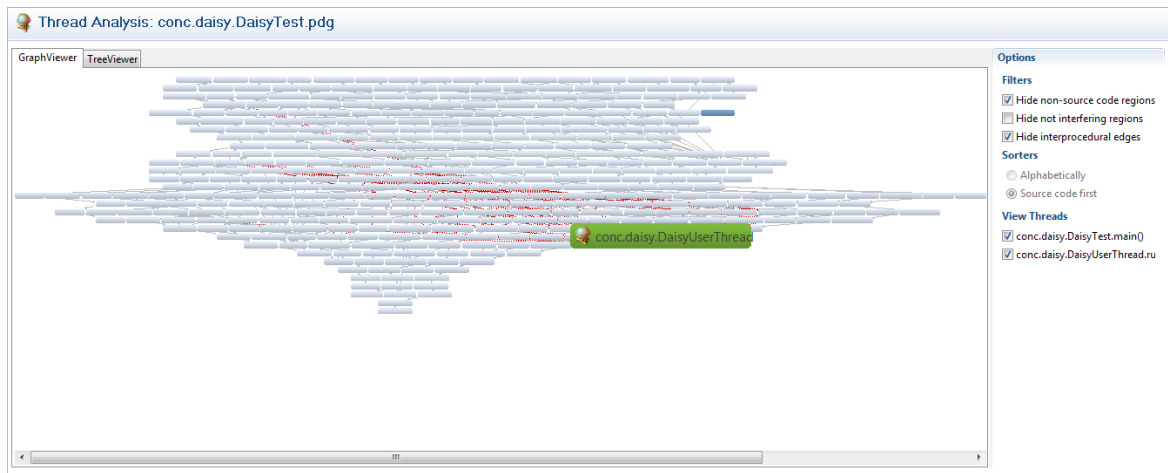
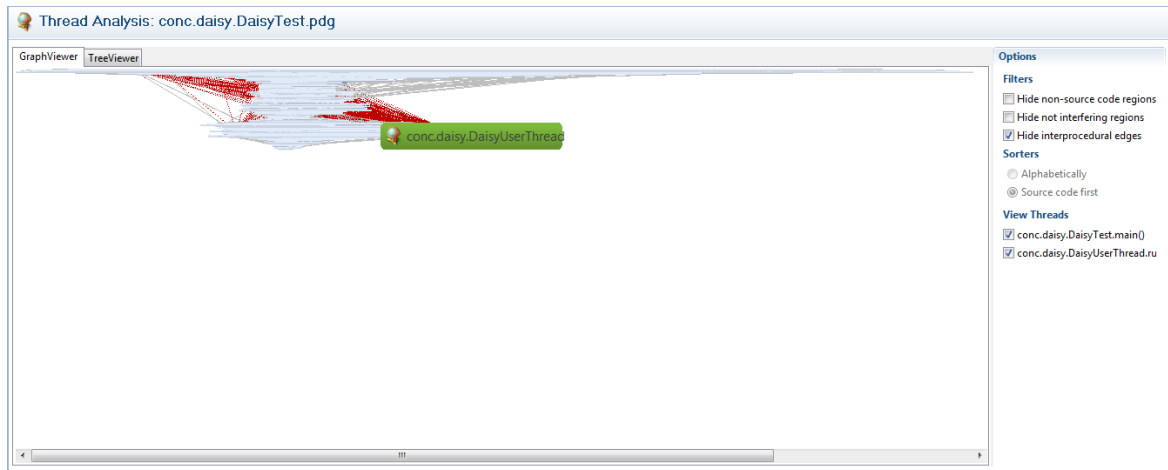


Abbildung A.5.: Visualisierungen des CSDG von DaisyTest.java bei der Aktivierung verschiedener Filter. Aus Performance-Gründen sind alle interproceduralen Kanten ausgeblendet.

Abbildungsverzeichnis

2.1.	Der ICFG eines Ausschnittes von <code>TimeTravel.java</code>	6
2.2.	Der SDG eines Ausschnittes von <code>TimeTravel.java</code>	9
2.3.	Der TCFG von <code>TimeTravel.java</code>	12
2.4.	Die Thread Regionen von <code>TimeTravel.java</code>	13
2.5.	Der finale CSDG von <code>TimeTravel.java</code>	18
3.1.	Realisierung von <i>ThreadViewer</i> als Eclipse-Plugin	22
3.2.	Die MVC-Architektur von <i>ThreadViewer</i>	23
3.3.	Das Layout von <i>ThreadViewer</i> mit dem Package Explorer (1), dem Java-Texteditor (2), der Visualisierung (3) und den Optionen (4)	28
3.4.	Die verwendeten Klassen in der Baumansicht	29
3.5.	Die Zoomfunktion des Zest-Framework in der Graphansicht im Einsatz	30
3.6.	Der Graph von <code>TimeTravel.java</code> mit verdeckter Interferenz	31
3.7.	Das vereinfachte Sequenzdiagramm von <i>ThreadViewer</i> beim Folgen einer Thread Region	34
4.1.	Laden einer <code>.pdg</code> -Datei	37
4.2.	Einblenden aller Thread Regionen	38
4.3.	Doppelklick in der Graphenansicht	39
4.4.	Folgen durch den Java-Editor	40
4.5.	Folgen paralleler Thread Regionen	41
4.6.	Folgen in der Baumansicht	42
4.7.	Folgen einer Interferenz	43
4.8.	Folgen eines interferierenden Knotens	44
4.9.	Ausblenden von Regionen ohne Interferenz	45
4.10.	Ausblenden einzelner Threads	46
4.11.	Navigieren mit der Baumansicht	47
4.12.	Aufbau des Graphen	48
4.13.	Folgen einer Interferenz	49
4.14.	Interferierende Regionen von <code>DaisyTest.java</code>	50
4.15.	<code>aisyTest</code> auf minimaler Zoomstufe	51
A.1.	Visualisierungen des CSDG von <code>TimeTravel.java</code> bei der Aktivierung verschiedener Filter	62
A.2.	Visualisierungen des CSDG von <code>ProducerConsumer.java</code> bei der Aktivierung verschiedener Filter	63
A.3.	Visualisierungen des CSDG von <code>ProducerConsumer.java</code> bei der Aktivierung verschiedener Filter	64
A.4.	Visualisierungen des CSDG von <code>SharedQueue.java</code> bei der Aktivierung verschiedener Filter	65

A.5. Visualisierungen des CSDG von <code>DaisyTest.java</code> bei der Aktivierung verschiedener Filter. Aus Performance-Gründen sind alle interprozeduralen Kanten ausgeblendet.	66
---	----

Listings

1.1. Der Quellcode von <code>TimeTravel.java</code>	2
3.1. Die Breitensuche der Methode <code>calculateSpecialControlFlows(..)</code>	27
3.2. Die Methode <code>runEditorFollowThreadRegion(..)</code> des Controllers	33

Algorithmenverzeichnis

1. Computation of MHP information	15
---	----

Literaturverzeichnis

- [1] <http://www.grammatech.com/>. The CodeSurfer Code Browser for C/C++.
- [2] *The Eclipse Metrics Plugin*. <http://sourceforge.net/projects/metrics/>.
- [3] *The Graphical Editing Framework*. <http://www.eclipse.org/gef/>.
- [4] *Java*. <http://www.java.com/de/>.
- [5] *The Plug-in Development Environment*. <http://www.eclipse.org/pde/>.
- [6] *VALSOFT/Joana*. <http://pp.info.uni-karlsruhe.de/project.php?id=30/>.
- [7] *Zest: The Eclipse Visualization Toolkit*. <http://www.eclipse.org/gef/zest/>.
- [8] BARIK, R.: *Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs*. In: AYGUADÉ, E., G. BAUMGARTNER, J. RAMANUJAM und P. SADAYAPPAN (Hrsg.): *Languages and Compilers for Parallel Computing*, Bd. 4339 d. Reihe *Lecture Notes in Computer Science*, S. 152–169. Springer Berlin / Heidelberg, 2006.
- [9] CLAYBERG, E. und D. RUBEL: *Eclipse Plug-ins, Third Edition*. AddisonWesley Prof., 2008.
- [10] FERRANTE, J., K. J. OTTENSTEIN und J. D. WARREN: *The program dependence graph and its use in optimization*. ACM Transactions on Programming Languages and Systems, 9:319–349, 1987.
- [11] GIFFHORN, D.: *Slicing of Concurrent Programs and its Application to Information Flow Control*. Doktorarbeit, Karlsruher Institut für Technologie, to be published in 2011.
- [12] GOSLING, J., B. JOY, G. STEELE und G. BRACHA: *The Java Language Specification..* AddisonWesley Prof., 3 Aufl., 2005.
- [13] GRAF, J.: *Speeding up context-, object- and field-sensitive SDG generation*. In: *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, September 2010. Accepted at SCAM 2010.
- [14] HAMMER, C.: *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. Doktorarbeit, Universität Karlsruhe (TH), July 2009. ISBN 978-3-86644-398-3.
- [15] HORWITZ, S., T. REPS und D. BINKLEY: *Interprocedural slicing using dependence graphs*. ACM Trans. Program. Lang. Syst., 12:26–60, January 1990.

- [16] KAMKAR, M., N. SHAHMEHRI und P. FRITZSON: *Bug Localization by Algorithmic Debugging and Program Slicing*. In: *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, S. 60–74, London, UK, 1990. Springer-Verlag.
- [17] KOMONDOOR, R. und S. HORWITZ: *Using slicing to identify duplication in source code*. In: *In Proceedings of the 8th International Symposium on Static Analysis*, S. 40–56. Springer-Verlag, 2001.
- [18] KRINKE, J.: *Static Slicing of Threaded Programs*. In: *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, S. 35–42, Montreal, Canada, June 1998.
- [19] KRINKE, J.: *Advanced Slicing of Sequential and Concurrent Programs*. Doktorarbeit, Universität Passau, April 2003.
- [20] NANDA, M. G. und S. RAMESH: *Interprocedural slicing of multithreaded programs with applications to Java*. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 28(6):1088–1144, 2006.