Universität Bremen

Fachbereich Mathematik / Informatik
Studiengang Informatik

Dynamische Protokollerkennung für Multithreading-Anwendungen in Java

Diplomarbeit

Bernhard Scholz

Matrikelnummer: 1870843

26.11.2009

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Dr. Berthold Hoffmann

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 26.11.2009

. . . . . .
(Bernhard Scholz)

Danksagung

Mein besonderer Dank gilt Prof. Dr. Rainer Koschke für die Betreuung dieser Diplomarbeit. Ebenfalls bedanken möchte ich mich bei Dr. Jochen Quante für die hilfreiche Einführung in die Details seiner Arbeiten, sowie bei Dr. Berthold Hoffmann, der sich freundlicherweise als zweiter Gutachter zur Verfügung gestellt hat.

Nicht zuletzt danke ich auch meinen Eltern, die mir dieses überaus interessante Studium ermöglicht haben.

Abstract

Ein *Protokoll* definiert die Sequenzeinschränkungen für die auf einer Software-Komponente ausgeführten atomaren Operationen. Ein im Bauhaus-Projekt entwickeltes Protokollerkennungsverfahren extrahiert solche Protokolle aus existierender Software mittels dynamischer Analyse. Java-Anwendungen mit mehreren Threads können jedoch nur eingeschränkt analysiert werden. In der vorliegenden Diplomarbeit werden daher Ansätze für die Erweiterung des existierenden Verfahrens zur Unterstützung von Multithreading entwickelt und evaluiert.

A *protocol* defines the sequencing constraints for the atomic operations that are applied on a software component. The Bauhaus project has introduced a protocol recovery technique which is able to extract such protocols from existing software by means of dynamic analysis. However, multi-threaded Java applications cannot be analysed adequately. In this thesis, approaches to an extension of the existing technique for the support of multi-threading are developed and evaluated.

INHALTSVERZEICHNIS

1 Einführung **1**

1.1 Hintergrund . . . 1

1.1.1 Software-Wartung . . . 1

1.1.2 Software-Analyse . . . 2

1.1.3 Das Bauhaus-Projekt . . . 2

1.1.4 Protokolle . . . 2

1.1.5 Protokollerkennung . . . 3

1.2 Aufgabenstellung . . . 4

1.3 Aufbau der Arbeit . . . 5

2 Grundlagen **7**

2.1 Notation . . . 7

2.2 Endliche Automaten . . . 7

2.2.1 Nichtdeterministische endliche Automaten . . . 8

2.2.2 Deterministische endliche Automaten . . . 8

2.2.3 Grafische Darstellung . . . 9

2.3 Resource Flow Graphs . . . 9

3 Protokollerkennung im Bauhaus-Projekt **11**

3.1 Objektprozessgraphen . . . 11

3.1.1 Definition . . . 12

3.1.2 Beispiel . . . 14

3.1.3 Dynamische Extraktion . . . 15

3.2 Protokollextraktion aus Objektprozessgraphen . . . 22

3.2.1 Rekursionsauflösung . . . 23

3.2.2 Transformation in Protokollautomaten . . . 24

3.3 Protokollerkennung bei mehreren Threads . . . 24

3.3.1 Entstehung unvollständiger Protokolle . . . 25

3.3.2 Problemanalyse . . . 25

3.4 Andere Protokollerkennungsverfahren . . . 27

3.5 Zusammenfassung . . . 27

4 Linearisierung nebenläufiger Prozesse **29**
4.1 Motivation . . . 29
4.2 Technische Umsetzung . . . 30
4.2.1 Prozesse und Threads . . . 30
4.2.2 Scheduling . . . 30
4.2.3 Multiprozessor-Systeme . . . 31
4.2.4 Nebenläufigkeit in Java . . . 31
4.3 Petri-Netze . . . 31
4.3.1 Definition . . . 32
4.3.2 Beispiel . . . 33
4.3.3 Entscheidbarkeit . . . 34
4.4 Sequentielle Darstellung . . . 34
4.5 Interleaving und Objektprozessgraphen . . . 35
4.6 Verwandte Forschung . . . 36

5 Erweiterung der Protokollerkennung **37**
5.1 ITS-Graphen . . . 37
5.1.1 Eigenschaften . . . 38
5.1.2 Repräsentation im RFG . . . 38
5.1.3 Konstruktionsmethode ITS-Online . . . 39
5.1.4 Konstruktionsmethode ITS-Offline . . . 42
5.1.5 Auswahl eines Konstruktionsverfahrens . . . 46
5.1.6 Beispiel . . . 47
5.2 ITS-Protokolle . . . 48
5.2.1 Protokollextraktion aus ITS-Pfaden . . . 48
5.2.2 Unzulässige Sequenzen . . . 49
5.3 Shuffle-Sprachen . . . 51
5.3.1 Definition . . . 51
5.3.2 Eigenschaften . . . 51
5.3.3 Konstruktion eines Shuffle-Automaten . . . 52
5.3.4 Schnitt-Protokoll . . . 53
5.4 Anwendung an einem Beispiel . . . 54
5.4.1 Code . . . 55
5.4.2 Protokollerkennung mit dem bisherigen Verfahren . . . 56
5.4.3 Protokollerkennung mit dem ITS-Shuffle-Schnitt-Verfahren . . . 56
5.5 Zusammenfassung . . . 60

6 Evaluation **63**

6.1 Fragestellung 63
6.2 Testkonfiguration 64
6.2.1 Voraussetzungen 64
6.2.2 Messverfahren 64
6.2.3 Systeme 65
6.2.4 Testfälle 65
6.3 Erzeugen der globalen Spur 65
6.3.1 Durchführung 65
6.3.2 Auswertung 66
6.4 Protokollextraktion 67
6.4.1 Durchführung 67
6.4.2 Auswertung 68
6.5 Ergebnis 69

7 Fazit **71**
7.1 Zusammenfassung 71
7.2 Reflexion 72
7.3 Ausblick 72

A Werkzeuge **73**
A.1 Neu implementierte Werkzeuge 73
A.2 Angepasste Werkzeuge 73

B Code-Varianten **77**

C Vollständige Listings **79**
C.1 Code zur Veranschaulichung der OPG-Definition 79
C.2 Beispiel für eine Multithreading-Anwendung 79
C.3 Erzeuger-Verbraucher-System 80

Abbildungsverzeichnis **84**

Listings **85**

Literaturverzeichnis **88**

KAPITEL 1

Einführung

Das Einführungskapitel soll dem Leser einen Eindruck verschaffen, in welchem Themenumfeld sich die vorliegende Diplomarbeit bewegt und worin die Motivation für dieses Forschungsgebiet besteht. Daraus entwickelt sich im Anschluss die Problemstellung, deren Lösung zur zentralen Aufgabe dieser Diplomarbeit geworden ist.

Inhalt

1.1 Hintergrund **1**
1.1.1 Software-Wartung 1
1.1.2 Software-Analyse 2
1.1.3 Das Bauhaus-Projekt 2
1.1.4 Protokolle 2
1.1.5 Protokollerkennung 3
1.2 Aufgabenstellung **4**
1.3 Aufbau der Arbeit **5**

1.1 Hintergrund

Im Folgenden wird erläutert, wie aus der Notwendigkeit der Software-Wartung heraus der Bedarf an automatisierter Software-Analyse erwächst. Das *Protokoll* einer Komponente ist dabei derjenige Aspekt der Software-Analyse, der für diese Diplomarbeit das zentrale Thema darstellt.

1.1.1 Software-Wartung

Im Lebenszyklus eines Software-Systems entstehen nur ca. 20 % der Gesamtkosten durch die Erstentwicklung. Die restlichen 80 % entfallen auf die *Wartung* des bereits ausgelieferten Systems. Unter dem Begriff Software-Wartung versteht man den Prozess, in dem die Software in der Zeit nach ihrer Inbetriebnahme an veränderte Anforderungen angepasst, verbessert oder von Fehlern befreit wird.

Um die für die Wartung notwendigen Änderungen am Programmcode durchführen zu können, benötigt ein Wartungsprogrammierer genaue Kenntnis über die inneren Zusammenhänge des Systems. Idealerweise kann er dazu auf eine Dokumentation zurückgreifen, die während der Entwicklung erstellt wurde. Die Realität sieht jedoch anders aus – selbst wenn eine Dokumentation existiert, ist diese oftmals veraltet. So wurde möglicherweise bei vorausgegangenen Wartungsaktivitäten die Software verändert, die Anpassung der Dokumentation jedoch vergessen oder vernachlässigt.

Ein detailliertes Verständnis für ein bestimmtes Software-System besitzen aus diesen Gründen oft nur einzelne Mitarbeiter. Verlassen diese das Projekt, so geht auch das angesammelte Wissen verloren. So erklärt es sich, dass Wartungsprogrammierer durchschnittlich die Hälfte der Zeit allein damit verbringen, die vorhandene Software zu analysieren und zu verstehen.

1.1.2 Software-Analyse

Um den Prozess der Software-Wartung effizienter zu gestalten, können Verfahren zur werkzeuggestützten Programmanalyse eingesetzt werden. Diese extrahieren automatisch oder semi-automatisch Informationen unterschiedlicher Art aus dem vorhandenen Programmcode eines Software-Produktes. Solche Informationen können für einen Wartungsprogrammierer beim Verstehen des Systems nützlich sein. Sie können auch Hinweise auf mögliche Fehler in der vorhandenen Software liefern.

1.1.3 Das Bauhaus-Projekt

Die Entwicklung und die Untersuchung von Methoden und Werkzeugen für die Programmanalyse sind das erklärte Ziel des *Bauhaus-Projektes*[1]. Das Forschungsprojekt wurde 1996 an der Universität Stuttgart ins Leben gerufen und später auch an der Universität Bremen sowie vom Spin-Off Axivion GmbH[2] fortgeführt. Im Laufe der Jahre ist eine Sammlung von Werkzeugen für ein breites Spektrum an Analysen entstanden [Kos08]. Diese Werkzeuge nutzen eine mächtige Infrastruktur, die auch bei der Implementierung neuer Analyseverfahren eingesetzt werden kann. So werden in dieser Diplomarbeit, basierend auf dem Bauhaus-Projekt und dessen Infrastruktur, sowohl einige vorhandene Werkzeuge erweitert als auch neue Werkzeuge hinzugefügt.

1.1.4 Protokolle

Beim Programmverstehen und bei der Software-Analyse ist unter anderem von Interesse, in welche Komponenten sich ein System zerlegen lässt und welche Schnittstellen zwischen diesen Komponenten bestehen. Eines der Merkmale der Schnittstelle einer Komponente ist ihr *Protokoll*. Darunter versteht man, formal gesprochen, die Menge der gültigen Sequenzen so genannter *atomarer Operationen*, die während der Lebensdauer der Komponente ausgeführt werden dürfen.

Als erstes, anschauliches Beispiel für eine Komponente mit einem bestimmten Protokoll wird eine Datei betrachtet, wie sie z.B. von einem Textverarbeitungsprogramm zum Speichern eines Dokumentes verwendet wird. Dieses Beispiel wurde gewählt, da das Protokoll dieser „Komponente“ vom Benutzer selbst unmittelbar wahrgenommen wird und intuitiv nachvollzogen werden kann.

Ein mögliches Protokoll für ein Textverarbeitungsdokument ist in Abbildung 1.1 dargestellt. Die erste Operation im Lebenszyklus eines Dokumentes besteht in seiner Erzeugung über den Befehl „Neu“, der genau einmal pro Dokument ausgeführt wird. Das Dokument kann anschließend in beliebig vielen, also auch null, Schritten bearbeitet werden. Nachdem die Datei geschlossen wurde, kann sie beliebig oft umbenannt werden oder aber zum erneuten Bearbeiten geöffnet werden. Die letzte Operation im Lebenszyklus des Dokumentes ist immer

[1] `http://www.bauhaus-stuttgart.com`
[2] `http://www.axivion.com`

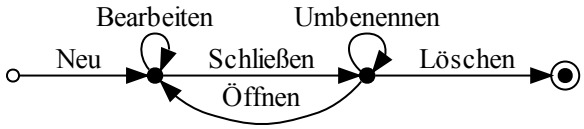


Abbildung 1.1: Beispiel für das Protokoll eines Dokumentes einer Textverarbeitung

„Löschen“. Dieser Befehl kann nur ausgeführt werden, wenn die Datei zuvor geschlossen wurde.

Die Besonderheit an diesem Beispiel ist, dass das Protokoll hier Teil der Benutzungsschnittstelle ist. Die Schnittstellen einzelner Software-Komponenten *untereinander* unterliegen jedoch oft ganz ähnlichen Protokollen. In objektorientierten Programmiersprachen wird eine Komponente in vielen Fällen z.B. durch eine Klasse mit ihren öffentlichen Methoden repräsentiert, wobei das Protokoll die Sequenzeinschränkungen für die Aufrufe dieser Methoden festlegt.

Das Protokoll trägt somit dazu bei, die *semantischen* Einschränkungen einer Schnittstelle zu definieren. In real existierenden Software-Systemen geschieht dies allerdings oftmals nur implizit. Die *syntaktischen* Vorschriften der Schnittstelle sind in der Regel formal definiert, da dies bereits von der verwendeten Programmiersprache verlangt wird. Das Protokoll ist hingegen häufig nicht explizit festgelegt, geschweige denn dokumentiert.

1.1.5 Protokollerkennung

Die implizite Definition eines Protokolls erlaubt es jedoch, daraus eine explizite Spezifikation des Protokolls herzuleiten, welche idealerweise mit der vom Programmierer ursprünglich entworfenen übereinstimmt.

Das Protokoll einer zentralen Komponente einer Anwendung kann so auch einen Beitrag zum Verständnis der gesamten Software liefern. Andererseits können implizit definierte Protokolle auch gegenüber einem vorhandenen Protokoll hinsichtlich ihrer Korrektheit überprüft werden. Auf diese Weise kann z.B. die fehlerhafte Verwendung einer Komponente im vorhandenen Code aufgedeckt werden.

Der Einsatz eines Werkzeugs, welches Protokolle automatisiert aus einem System extrahiert und sinnvoll repräsentiert, kann somit zur Effizienz der Software-Wartung beitragen.

Für die Analyse von Protokollen in existierender Software sind bereits unterschiedliche statische und dynamische Analyseverfahren erforscht worden. Von Whaley et al. [WML02] wird z.B. ein Ansatz vorgestellt, bei dem die Implementierung einer zu analysierenden Klasse dahingehend überprüft wird, auf welche Weise die korrekte Funktionsweise der einzelnen Methoden vom Zustand des jeweiligen Objektes abhängig ist und inwiefern die einzelnen Methoden den Objektzustand verändern.

Andere Verfahren ermitteln im Gegensatz dazu das Protokoll basierend auf den impliziten Annahmen, die von den anderen Programmteilen über die betrachtete Komponente getroffen werden. Die Voraussetzung dafür ist, dass Software-Systeme „überwiegend“ fehlerfrei sind. Unter diesen Umständen stimmt auch die tatsächliche Verwendung einer Komponente „über-

wiegend“ mit der beabsichtigten Verwendung überein. Diese Philosophie wird z.B. auch von Engler et al. [ECH+01] für deren Ansatz zum Auffinden von Protokollverletzungen vertreten.

Im Bauhaus-Projekt wurde ebenfalls ein Verfahren entwickelt, welches die tatsächliche Verwendung eines Objektes durch die anderen Programmteile betrachtet. Diese Information wird über einen so genannten *dynamischen Objektprozessgraphen* (DOPG) dargestellt. Ein Objektprozessgraph (OPG) ist eine Projektion des Kontrollflussgraphen der Software auf die für eine bestimmte Komponente (z.B. Instanz einer Klasse) relevanten Knoten und Kanten. Der DOPG wird durch ein dynamisches Analyseverfahren während der Ausführung der zu untersuchenden Software gewonnen.

Durch darauf folgende Transformationen kann anhand eines OPG schließlich das Protokoll der gewählten Komponente bestimmt werden. Dieses wird, wie in Abbildung 1.1, in Form eines *Zustandsautomaten* dargestellt. Basierend auf dieser Darstellung sind nun weiterführende Analysen möglich, insbesondere der Vergleich mit gegebenen Protokollen.

1.2 Aufgabenstellung

Die dynamische Protokollerkennung wurde im Bauhaus-Projekt bis jetzt für die Programmiersprachen C und Java implementiert. Bisher nicht explizit berücksichtigt wurde dabei allerdings das dynamische Verhalten von *Multithreading-Anwendungen*. Prinzipiell kann Multithreading zwar auch in C-Programmen auftreten, in der Programmiersprache Java ist die Verwendung jedoch sehr einfach und daher weit verbreitet[3]. Sobald eine Java-Anwendung z.B. über eine grafische Benutzungsschnittstelle (GUI) verfügt, ist ein Verzicht auf mehrere Threads nicht mehr möglich.

Solange auf eine zu analysierende Komponente nur von einem der Threads einer Anwendung zugegriffen wird, ergibt sich für die Protokollerkennung kein Unterschied im Vergleich zu einer Single-Thread-Anwendung. Sobald jedoch ein Objekt von mehr als einem Thread verwendet wird, scheitert die Analyse mangels einer passenden Darstellung. Im Allgemeinen ist das Resultat ein falsches Protokoll, in welchem nur die Aufrufe aus demjenigen Thread berücksichtigt werden, der das Objekt auch erzeugt hat.

Gerade die für das Programmverstehen besonders interessanten, zentralen Komponenten einer interaktiven Java-Anwendung mit einer GUI werden typischerweise von einer größeren Anzahl Threads verwendet. So gibt es häufig jeweils einen Thread zur Initialisierung des Systems, zur Verarbeitung der Benutzerinteraktion, zum Laden, Speichern und Drucken von Daten und für andere Aktivitäten. Alle diese Threads haben üblicherweise Zugriff auf die Datenstruktur, die z.B. das aktuell geöffnete Dokument repräsentiert.

Das Protokoll einer solchen Datenstruktur kann daher nur korrekt erkannt werden, wenn dabei auch alle beteiligten Threads berücksichtigt werden. Für viele andere, nicht OPG-basierte, dynamische Protokollerkennungsverfahren spielt es keine große Rolle, ob ein Objekt von einem oder mehreren Threads verwendet wird, da diese Verfahren direkt die Sequenz der auf das Objekt angewandten Operationen aufzeichnen und analysieren.

Gerade die besondere Eigenschaft des OPG-basierten Verfahrens, die syntaktische Struktur des Programms für die Protokollerkennung zu nutzen, erfordert im Gegensatz zu den anderen Verfahren eine spezielle Vorgehensweise zur Integration von zusätzlichen Informationen über das *nebenläufige Verhalten* des Programms.

An diesem Punkt setzt die Problemstellung dieser Diplomarbeit an. Die zentrale Aufgabe

[3] vgl. [Qua07] S. 6

besteht darin, ein Konzept zu entwickeln, durch das die OPG-basierte Protokollerkennung für die Programmiersprache Java um die Unterstützung von Multithreading erweitert werden kann.

Dabei soll die Verwendung eines Objektes einerseits aus der Sicht der einzelnen Threads betrachtet werden, andererseits aus der Perspektive des Objektes selbst. Letzteres bedeutet, das *Interleaving-Verhalten* der Threads zu analysieren, d.h. inwiefern die vorhandenen Threads tatsächlich nebenläufig ausgeführt werden und auf welche Art und Weise die einzelnen Threads miteinander „verzahnt“ sind. So ergibt sich aus der Sicht des Objektes wieder eine einfache Sequenz von Aufrufen, bei denen es für das Objekt selbst keine Rolle spielt, ob diese aus verschiedenen Threads stammen oder nicht. Das Protokoll des Objektes wird dann als eine Menge solcher Sequenzen betrachtet.

Das vorhandene, prototypische Protokollerkennungsverfahren für Java scheitert, abgesehen von der Multithreading-Problematik, auch an bestimmten Kontrollflüssen, wie sie zwar auch in C-Programmen, vor allem jedoch in Java-Anwendungen vorkommen. Dazu zählen dynamisches Binden virtueller Methoden sowie das Auftreten von Exceptions. Die Implementierung entsprechender Erweiterungen ist daher ebenfalls Teil der Aufgabenstellung.

1.3 Aufbau der Arbeit

Während in Kapitel 2 allgemeine Begriffe und Notationen definiert werden, erfolgt in Kapitel 3 eine Beschreibung der im Bauhaus-Projekt vorhandenen Konzepte und Werkzeuge zur Protokollerkennung. Insbesondere werden die Grenzen dieser Verfahren aufgezeigt, welche die Aufgabenstellung – die Repräsentation von Multithreading – begründen. In Kapitel 4 werden unabhängig vom Bauhaus-Projekt einige technische sowie theoretische Aspekte von Multithreading vorgestellt, um die selbst entwickelten Lösungsansätze einordnen zu können. Diese werden in Kapitel 5 ausführlich erörtert. Eine Untersuchung der Praxisrelevanz und -tauglichkeit des gewählten Lösungsansatzes wird in Kapitel 6 durchgeführt. Kapitel 7 enthält sowohl den Rückblick auf die Arbeit und ihre Ergebnisse als auch einen Ausblick auf weiterführende Fragestellungen.

Zusätzliche Informationen, Daten und Analyseergebnisse finden sich im Anhang sowie auf der beiliegenden DVD.

KAPITEL 2

Grundlagen

Dieses Kapitel fasst die verwendeten Grundbegriffe und Schreibweisen zusammen, die für das Verstehen der restlichen Kapitel notwendig sind. Dies umfasst Begriffe aus der Mathematik, der theoretischen Informatik und aus dem Bauhaus-Projekt. Sofern diese Grundlagen dem Leser bereits geläufig sind, kann das Kapitel auch übersprungen werden und lediglich als Referenz Verwendung finden.

Inhalt

2.1 Notation **7**
2.2 Endliche Automaten **7**
2.2.1 Nichtdeterministische endliche Automaten 8
2.2.2 Deterministische endliche Automaten 8
2.2.3 Grafische Darstellung 9
2.3 Resource Flow Graphs **9**

2.1 Notation

- ϵ bezeichnet das leere Wort
- Ein *Alphabet* ist eine endliche, nicht leere Menge.
- Für ein Alphabet Σ bezeichnet Σ^* die Menge aller Wörter über diesem Alphabet einschließlich ϵ.
- *String*, *Integer*, *Boolean* etc. bezeichnen die gleichnamigen Datentypen. Die Begriffe *Datentyp* und *Menge* werden hierbei synonym gebraucht. Es gilt somit z.B.

  $$\textit{Integer} \not\ni \texttt{abc} \in \textit{String}$$

  Auf eine formale Definition der Datentypen wird verzichtet, da sie für diese Arbeit nicht von Bedeutung ist.

2.2 Endliche Automaten

Betrachtet man die Menge der atomaren Operationen eines Objektes als ein Alphabet Σ, so ist das Protokoll des Objektes eine Sprache über diesem Alphabet. Aus verschiedenen Gründen ist es sinnvoll, sich bei der Protokollanalyse auf reguläre Sprachen zu beschränken. Jede reguläre Sprache lässt sich als ein endlicher Automat darstellen, der gerade die Wörter dieser Sprache erkennt, d.h. akzeptiert.

2.2.1 Nichtdeterministische endliche Automaten

Es wird eine Definition angegeben, wie sie auch von Hopcroft et al. verwendet wird[1].

Ein nichtdeterministischer endlicher Automat (NEA) ist ein System $A = (Q, \Sigma, \delta, q_0, F)$ mit

- der endlichen Menge der Zustände Q
- dem Eingabealphabet Σ
- der Zustandsüberführung $\delta \subseteq Q \times \Sigma \times Q$
- dem Startzustand $q_0 \in Q$
- und der Menge der Endzustände $F \subseteq Q$

Für einen Zustand $q \in Q$ und ein Eingabezeichen $x \in \Sigma$ bezeichnet $\delta(q, x)$ die Menge aller möglichen Folgezustände:

$$\delta(q, x) = \{q' \in Q \mid (q, x, q') \in \delta\}$$

Während die Zustandsüberführung δ die Übergänge für einzelne Zeichen definiert, gibt die fortgesetzte Zustandsüberführung $\delta^* : Q \times \Sigma^* \times Q$ die möglichen Zustandsübergänge für Wörter an. Sie ist für alle $q \in Q, x \in \Sigma, w \in \Sigma^*$ definiert durch

$$\begin{aligned} \delta^*(q, \epsilon) &= \{q\} \\ \delta^*(q, wx) &= \bigcup_{q' \in \delta^*(q,w)} \delta(q', x) \end{aligned}$$

Die von einem NEA A erkannte Sprache ist die Menge der Wörter, für die ein Zustandsübergang vom Startzustand zu mindestens einem der Endzustände existiert:

$$L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$$

2.2.2 Deterministische endliche Automaten

Ein deterministischer endlicher Automat (DEA) nach der in der Literatur zur theoretischen Informatik verbreiteten Definition unterscheidet sich vom NEA dadurch, dass die Zustandsüberführung eine Abbildung $\delta : Q \times \Sigma \to Q$ darstellt. Jeder Zustand hat also für jedes Eingabezeichen genau einen Folgezustand. Die fortgesetzte Zustandsüberführung und die erkannte Sprache sind entsprechend anders definiert. Gemäß Hopcroft et al. [HMU02] gelten für DEAs zwei wichtige Eigenschaften:

- Zu jedem NEA lässt sich durch Potenzmengenbildung ein äquivalenter DEA (der Potenzautomat) konstruieren, der dieselbe Sprache erkennt.
- Ferner existiert zu jeder regulären Sprache ein – bis auf die Benennung der Zustände – eindeutiger minimaler DEA, der durch einen Algorithmus bestimmt werden kann. Zwei reguläre Sprachen sind somit genau dann gleich, wenn ihre minimalen DEAs isomorph sind.

[1] vgl. [HMU02] S 66ff.

Wenn in dieser Diplomarbeit von einem deterministischen Automaten A gesprochen wird, ist – abweichend von der eben genannten Definition – ein spezieller NEA gemeint, dessen Zustandsüberführung δ die Eigenschaft einer *partiellen* Abbildung erfüllt, d.h. in der oben verwendeten Schreibweise für Relationen

$$\forall q \in Q, x \in \Sigma : |\delta(q, x)| \leq 1$$

Der Fall $\delta(q, x) = \emptyset$ ist insofern deterministisch, als das bis dahin gelesene Wort dann eindeutig nicht erkannt wurde. Durch Zulassen von $q \in Q, x \in \Sigma$ mit $\delta(q, x) = \emptyset$ können die endlichen Automaten wesentlich übersichtlicher grafisch dargestellt werden. Ansonsten müsste man einen weiteren Zustand $q' \notin Q$ mit der Eigenschaft einer „Sackgasse“ einführen, der nur den Zweck hat, das nicht erkannte Wort „zu Ende zu lesen“. Dazu würde man einen Automaten $A' = (Q \cup \{q'\}, \Sigma, \delta', q_0, F)$ konstruieren mit

$$\delta' = \{(q', x, q') \mid x \in \Sigma\} \cup \{(q, x, q') \mid q \in Q, x \in \Sigma, \delta(q, x) = \emptyset\}$$

Dieser Automat wäre äquivalent zum Automaten A, zugleich aber ein DEA im Sinne von Hopcroft et al..

Die abweichende Definition des deterministischen Automaten wird implizit in den Veröffentlichungen und in der vorhandenen Implementierung des Bauhaus-Projektes zur Protokollrepräsentation [QK07] verwendet, weshalb sie auch in dieser Diplomarbeit beibehalten werden soll.

2.2.3 Grafische Darstellung

Ein endlicher Automat $(Q, \Sigma, \delta, q_0, F)$ wird als Graph dargestellt, wobei jeder Zustand $q \in Q$ durch einen Knoten im Graph repräsentiert wird. Eine Kante vom Zustand q zu einem Zustand $q' \in Q$ mit der Beschriftung $x \in \Sigma$ existiert genau dann, wenn die Zustandsüberführung ein entsprechendes Tupel $(q, x, q') \in \delta$ enthält. Für die Knoten werden verschiedene Symbole verwendet, um Start- und Endzustände kenntlich zu machen. Diese sind in Tabelle 2.1 angegeben.

Symbol	Bedeutung
○	Startzustand ($q = q_0$)
◉	Endzustand ($q \neq q_0, q \in F$)
◎	Start- und zugleich Endzustand ($q = q_0 \in F$)
●	anderer Zustand ($q \neq q_0, q \notin F$)

Tabelle 2.1: Knotensymbole für Zustand $q \in Q$

2.3 Resource Flow Graphs

Die universelle Datenstruktur zur Darstellung von Graphen unterschiedlicher Art innerhalb der Bauhaus-Tools ist der so genannte *Resource Flow Graph* (RFG). Das RFG-Konzept wurde ursprünglich im Rahmen der Studienarbeit von Thomas Eisenbarth [Eis98] entwickelt. Für

diese Diplomarbeit ist allerdings nur die generelle Graph-Repräsentation relevant, nicht die in diesem Zusammenhang beschriebenen Analysen.

In diesem Abschnitt wird eine formale Definition angegeben, welche auf der vorhandenen Implementierung basiert. Auf diese Weise haben alle weiteren formalen Betrachtungen ihr unmittelbares Pendant in der im Rahmen dieser Arbeit entstandenen Implementierung. Im ursprünglichen Sinne war ein RFG ein Graph (N, E) mit $E \subseteq N \times N$. Tatsächlich repräsentiert die Implementierung einen RFG jedoch als Multigraph, so dass auch mehrere Kanten entlang desselben Knotenpaares verlaufen können. Von dieser Möglichkeit machen sowohl die vorhandenen als auch die während dieser Diplomarbeit entstandenen Protokollerkennungswerkzeuge Gebrauch.

Die hier angegebene Definition ist keine vollständige Spezifikation des RFG, sondern nur eine Betrachtungsweise, wie sie für die Bewältigung der in dieser Arbeit zu lösenden Probleme geeignet ist. Anders ausgedrückt, alle RFG-Instanzen, die in der zum Lösen der Probleme notwendigen Implementierung auftreten, lassen sich mit der folgenden Definition darstellen. Dagegen wird hier auf Konzepte wie z.B. das der *Sichten* (*Views*) gänzlich verzichtet.

Ein RFG ist ein gerichteter Graph mit Mehrfachkanten, Knoten- und Kantentypen sowie Knoten- und Kantenattributen. Formal lässt sich der RFG definieren als ein System

$$RFG = (N, E, \mathit{source}, \mathit{target}, \mathit{type})$$

Dabei gilt

- N ist die (endliche) Menge der Knoten.
- E ist die (endliche) Menge der Kanten mit $N \cap E = \emptyset$.
- Knoten und Kanten bilden zusammen die *Graphelemente* des RFG.
- Die Abbildungen $\mathit{source} : E \to N$ und $\mathit{target} : E \to N$ ordnen jeder Kante $e \in E$ einen Quellknoten $\mathit{source}(e)$ und einen Zielknoten $\mathit{target}(e)$ zu.
- Die Abbildung $\mathit{type} : N \cup E \to T_N \cup T_E$ definiert für jeden Knoten $n \in N$ einen Typ $\mathit{type}(n) \in T_N$ und für jede Kante $e \in E$ einen Typ $\mathit{type}(e) \in T_E$. Die endlichen Mengen T_N und T_E enthalten Typen für alle möglichen Darstellungen von Ergebnissen verschiedener Analysen und sind prinzipiell frei konfigurierbar. Die für diese Arbeit relevanten Typen werden jeweils an geeigneter Stelle vorgestellt. Ein RFG-Typ wird in serifenloser Schrift gedruckt, z.B. $\mathsf{Return} \in T_N$.
- In einem tatsächlichen RFG kann jedes Graphelement noch eine Attribut-Liste aus Schlüssel-Wert-Paaren besitzen, die zusätzliche Informationen beinhalten. Formal werden dadurch weitere Abbildungen definiert, auf die hier nicht näher eingegangen werden soll. Solche Abbildungen (z.B. $\mathit{id} : N \to \mathit{String}$ für die Knoten-ID) werden im Folgenden erst unmittelbar vor ihrem Einsatz definiert.

KAPITEL 3

Protokollerkennung im Bauhaus-Projekt

In diesem Kapitel wird erläutert, welches Verfahren zur Protokollerkennung im Bauhaus-Projekt entwickelt wurde. Kennzeichnend für das Verfahren ist dabei die Verwendung eines so genannten dynamischen Objektprozessgraphen. Es wird gezeigt, wie aus einer zu analysierenden Software ein dynamischer Objektprozessgraph gewonnen werden kann, der schließlich zu einem Protokollautomaten verarbeitet wird.

In Abschnitt 3.3 wird anhand eines Beispiels demonstriert, welche Probleme bei Anwendung des beschriebenen Verfahrens im Zusammenhang mit mehreren Threads auftreten.

Inhalt

3.1 Objektprozessgraphen 11

3.1.1 Definition 12

3.1.2 Beispiel 14

3.1.3 Dynamische Extraktion 15

3.2 Protokollextraktion aus Objektprozessgraphen 22

3.2.1 Rekursionsauflösung 23

3.2.2 Transformation in Protokollautomaten 24

3.3 Protokollerkennung bei mehreren Threads 24

3.3.1 Entstehung unvollständiger Protokolle 25

3.3.2 Problemanalyse 25

3.4 Andere Protokollerkennungsverfahren 27

3.5 Zusammenfassung 27

3.1 Objektprozessgraphen

Koschke und Quante [QK08] beschreiben detailliert das Konzept des dynamischen Objektprozessgraphen, dessen Erzeugung und verschiedene Anwendungsmöglichkeiten. Die Beispiele und Ansätze basieren im Wesentlichen auf der Annahme, dass die zu analysierende Software in der Programmiersprache C implementiert ist. In [Qua07] wird eine Erweiterung des Konzepts beschrieben, mit der auch das Auftreten und die Behandlung von Exceptions modelliert werden können. Im Folgenden sollen die vorhandenen Konzepte erläutert werden, allerdings speziell im Hinblick auf die Analyse von Java-Anwendungen. Die Beispiele und Abbildungen basieren auf den von Koschke und Quante vorgestellten.

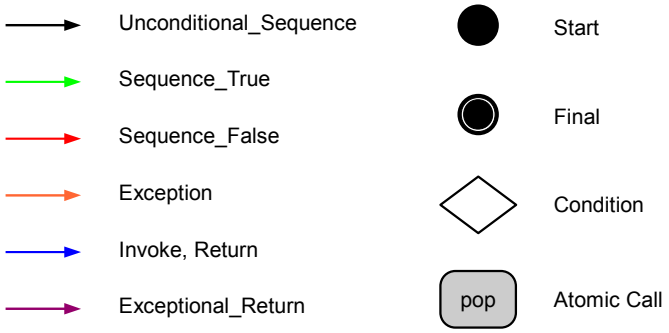


Abbildung 3.1: Legende für Objektprozessgraphen

3.1.1 Definition

Ein Objektprozessgraph (OPG) ist eine Sicht auf den Kontrollflussgraphen (CFG) einer Anwendung aus der Perspektive eines einzelnen Objekts. Er enthält genau diejenigen Knoten und Kanten des CFG, welche für das Objekt relevant im Sinne des Kontrollflusses sind. Ein OPG ist ein gerichteter, relationaler Graph:

$$(N, E) \text{ mit } E \subseteq N \times N$$

Ein OPG wird innerhalb der Implementierung als RFG abgebildet. Der relationale Graph (N, E) wird dabei als RFG $(N, E, \mathit{source}, \mathit{target}, \mathit{type})$ dargestellt durch Definition von $\mathit{source} : (s, t) \mapsto s$ und $\mathit{target} : (s, t) \mapsto t$.

Für die Typen eines Knotens $n \in N$ bzw. einer Kante $e \in E$ eines OPG gilt:

$$\mathit{type}(n) \in \{\mathsf{Start}, \mathsf{Create}, \mathsf{Read}, \mathsf{Write}, \mathsf{Condition}, \mathsf{Exception}, \mathsf{Call}, \mathsf{Entry}, \mathsf{Return}, \mathsf{Exceptional_Return}, \mathsf{Final}\}$$

$$\mathit{type}(e) \in \{\mathsf{Unconditional_Sequence}, \mathsf{Sequence_True}, \mathsf{Sequence_False}, \mathsf{Exception}, \mathsf{Invoke}, \mathsf{Return}, \mathsf{Exceptional_Return}\}$$

Der Typ einer Kante ist hierbei redundant, da er sich aus den Typen von Quell- und Zielknoten der Kante ergibt. Im Folgenden wird erläutert, welche Bedeutung die Knoten- und Kanten-Typen haben und welcher Zusammenhang zwischen den Typen von Knoten und denen der inzidenten Kanten besteht.

Knotentypen Die Knoten eines OPG repräsentieren Positionen im Quellcode, die Kanten die Kontrollfluss-Beziehungen zwischen den Quellcode-Positionen. Jeder OPG hat genau einen Knoten vom Typ Start, welcher den den Einstiegspunkt in den OPG repräsentiert, sowie genau einen Knoten vom Typ Create. Letzterer stellt die Position dar, an der das betrachtete Objekt erzeugt wird. Knoten der Typen Read und Write repräsentieren Lese- bzw. Schreibzugriffe auf die Attribute des Objektes. Abhängig von einer Bedingung (z.B. in einer `if`-Anweisung) teilt sich der Kontrollfluss an bestimmten Stellen in alternative Wege auf. Diese Stellen werden durch Condition-Knoten dargestellt. Exception-Knoten symbolisieren Exception-Handler, d.h. Stellen, an denen eine Exception gefangen wird (`catch`).

Ein Aufruf einer Methode wird durch einen Call-Knoten repräsentiert, das Betreten einer Methode durch einen Entry-Knoten. Ein Return-Knoten steht für die normale Rückkehr zum Aufrufer, während ein Exceptional_Return-Knoten das Verlassen der Methode aufgrund einer nicht gefangenen Exception darstellt.

Von allen drei Typen Entry, Return und Exceptional_Return existiert pro Methode jeweils maximal ein Knoten. Die vorliegende Implementierung weicht in diesem Punkt von der Beschreibung in der Literatur ab. Bei dem in [QK08] vorgestellten Konzept kann eine Methode nämlich auch mehrere Exceptional_Return-Knoten besitzen. In dieser Diplomarbeit wird jedoch nur das tatsächlich implementierte Konzept (d.h. mit maximal einem Exceptional_Return-Knoten) berücksichtigt.

Kantentypen Der Kontrollfluss wird durch die Kanten dargestellt, welche zwischen den Knoten verlaufen. Zunächst muss zwischen intraprozeduralem (innerhalb einer Methode) und interprozeduralem Kontrollfluss (über Methodengrenzen hinweg) unterschieden werden.

Der intraprozedurale Kontrollfluss kann unbedingt oder bedingt sein. Unbedingter Kontrollfluss wird durch Unconditional_Sequence-Kanten repräsentiert, bedingter Kontrollfluss wird durch Kanten der Typen Sequence_True und Sequence_False. Kanten des bedingten Kontrollflusses dürfen nur von einem Condition-Knoten ausgehen.

Zur Darstellung des interprozeduralen Kontrollflusses werden Kanten der Typen Invoke, Return und Exceptional_Return verwendet. Eine Invoke-Kante verläuft von einem Call-Knoten zum Entry-Knoten der aufgerufenen Methode. Eine Return- oder Exceptional_Return-Kante verbindet entsprechend den Return- bzw. Exceptional_Return-Knoten der Methode wieder mit dem Call-Knoten des Aufrufers.

Der Teilgraph eines OPG, der durch den Entry- und den Return und/oder Exceptional_Return-Knoten einer Methode begrenzt wird, heißt – analog zur Bezeichnung im Zusammenhang mit Quellcode – *Rumpf* dieser Methode.

Atomare Aufrufe Wird eine öffentliche Methode des zu analysierenden Objektes aufgerufen, so ist der Rumpf dieser Methode im OPG nicht sichtbar. Solche Aufrufe heißen *atomare Aufrufe* (engl. atomic calls), die Methoden entsprechend *atomare Methoden*. Die atomaren Aufrufe werden als Call-Knoten ohne Entry- und Return-Kanten repräsentiert. Sie sind für diese Diplomarbeit von besonderem Interesse, denn die in einem Programm möglichen Sequenzen von atomaren Methoden bilden das Protokoll des analysierten Objektes. So lässt sich die Menge der atomaren Methoden als das Alphabet betrachten, auf dem die gesuchte Sprache (d.h. das gesuchte Protokoll) basiert. Zum Alphabet gehören auch die Konstruktoren sowie das Create-Ereignis des Objektes.

Exceptions Einer besonderen Behandlung bedarf es beim Auftreten von Exceptions. Eine Exception kann entweder durch eine `throw`-Anweisung oder durch einen Methodenaufruf auftreten, wenn diese Methode über eine nicht gefangene Exception verlassen wird. Eine Exception-Kante verläuft von dem Knoten, an dem eine Exception auftritt, zu einem Exception-Knoten, falls die Exception innerhalb derselben Methode gefangen wird. Verursacht die Exception hingegen einen Sprung zu einem Exception-Handler außerhalb der Methode, so verläuft die Exception-Kante stattdessen – ggf. nach Passieren eines `finally`-Blocks – zu einem Exceptional_Return-Knoten. Von dort aus führt eine Exceptional_Return-Kante zurück zum Call-Knoten, der dadurch zum Quell-Knoten einer weiteren Exception-Kante wird.

```
1   void main() {
2       int i = 0;
3       Stack s1 = new Stack();
4       Stack s2 = readStack();
5       reverse(s2, s1);
6       do {
7           s1.pop();
8           i = i + 1;
9       } while (!s1.isEmpty());
10  }
11
12  void reverse(Stack from, Stack to) {
13      while (!from.isEmpty()) {
14          to.push(from.pop());
15      }
16  }
```

Listing 3.1: Code zur Veranschaulichung der OPG-Definition

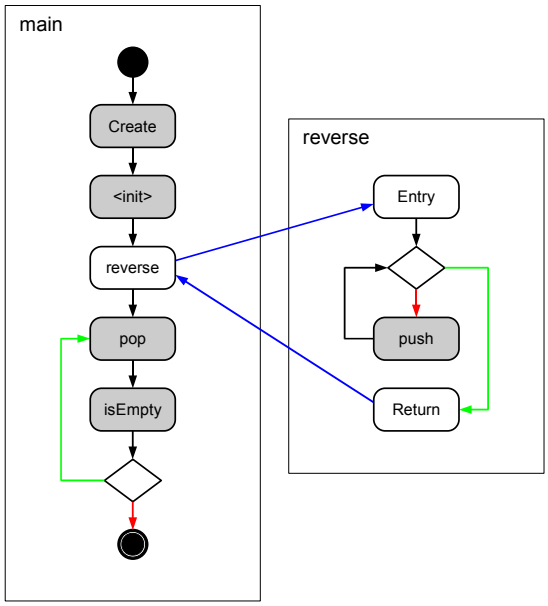


Abbildung 3.2: Beispiel-OPG für Objekt `s1` aus Listing 3.1

3.1.2 Beispiel

Abbildung 3.2 zeigt ein einfaches Beispiel für einen OPG. Dieser ergibt sich, wenn man in Listing 3.1 das Objekt `s1` betrachtet. Dabei handelt es sich um eine Instanz einer Klasse `Stack`, welche über die öffentlichen Methoden `isEmpty`, `push` und `pop` verfügt. Die Aufrufe dieser Methoden gelten im OPG als atomare Aufrufe, ebenso der Aufruf des Konstruktors `<init>` und der Create-Knoten. Die Methodenaufrufe für das Objekt `s2` werden hingegen in diesem OPG gar nicht repräsentiert, da sie für das betrachtete Objekt `s1` nicht relevant sind.

Zur Demonstration der Exception-Behandlung wird ein weiteres Beispiel angegeben, zu sehen in Abbildung 3.3. Eine Exception wird dort von der rechts abgebildeten Methode `h` am `throw`-Knoten geworfen. Die Exception wird erst vom Exception-Handler in der Methode `f` gefangen. Die Methoden `h` und `g` werden daher über ihre Exceptional_Return-Knoten verlassen.

Aus Gründen der Übersicht sind in den grafischen Darstellungen nur die einfachen Methodennamen zu sehen. Tatsächlich werden jedoch stets auch der Klassenname und die volle

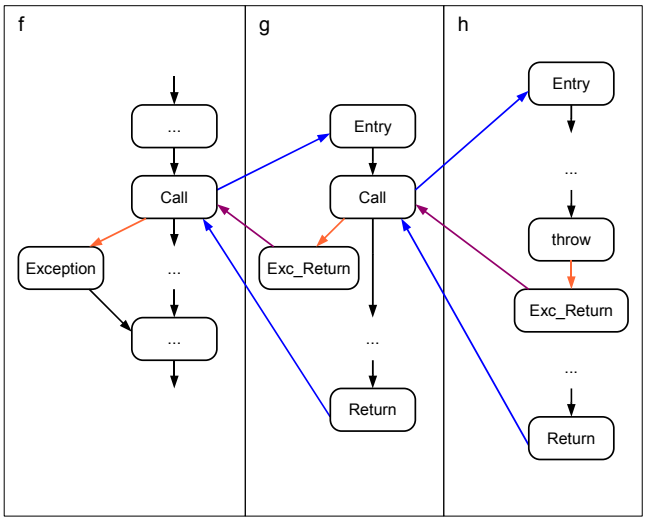


Abbildung 3.3: Werfen und Fangen einer Exception, dargestellt im OPG

Signatur einer Methode gespeichert. Ansonsten wäre die Eindeutigkeit nicht gewährleistet, da Methoden sowie Konstruktoren in Java überladen werden können.

3.1.3 Dynamische Extraktion

Ein OPG kann aus einem existierenden Programm durch eine dynamische oder statische Analyse erzeugt werden. Während die dynamische Analyse Informationen verwendet, die während der Ausführung des Programms gewonnen werden, basiert das statische Verfahren auf der Analyse des Programmcodes. Beide Ansätze haben ihre Vor- und Nachteile. So wird während der dynamischen Analyse in der Regel nur eine Teilmenge der möglichen Ausführungspfade tatsächlich durchlaufen. Die statische Analyse hingegen ist auf eine Datenflussanalyse angewiesen, um z.B. die Werte von Funktionszeigern vorherzusagen [EKV05]. Dabei müssen mitunter pessimistische Annahmen getroffen werden, wodurch auch Ausführungspfade in Betracht gezogen werden, die im tatsächlichen Kontrollfluss niemals auftreten. Die statische Analyse liefert somit eine Obermenge der möglichen Ausführungspfade.

Eine ausführliche Beschreibung statischer Objektprozessgraphen findet sich z.B. in [Vog06]. Diese Diplomarbeit beschränkt sich hingegen auf die Betrachtung der dynamischen Analyse. Ein OPG, der durch eine dynamische Analyse erzeugt wurde, heißt entsprechend *dynamischer Objektprozessgraph* (DOPG). Diese Bezeichnung wird im Folgenden aber nur verwendet, wenn die beschriebenen Zusammenhänge speziell für dynamisch extrahierte OPGs gelten.

Zur dynamischen Konstruktion eines Objektprozessgraphen werden eine oder mehrere so genannte *Spuren* (Traces) benötigt. Eine Spur ist eine Folge von Quellcode-Positionen, die bei einem bestimmten Lauf eines Programms in dieser Reihenfolge ausgeführt wurden. Eine *Objektspur* enthält schließlich nur diejenigen Quellcode-Positionen, die aus der Sicht des betrachteten Objekts relevant sind.

3.1.3.1 Instrumentierung

Zur Gewinnung von Spur-Informationen während der Programmausführung ist eine Instrumentierung des Programms notwendig. Dabei wird der Programmcode so modifiziert, dass

jede Ausführung einer potenziell relevanten Quellcode-Position in geeigneter Form protokolliert wird. Dazu wird jede dieser Quellcode-Positionen mit einer eindeutigen ID und einer Logging-Anweisung versehen, welche diese ID und weitere Informationen z.B. in eine Datei schreibt.

In [QK08] wird die Instrumentierung von C-Programmen dargestellt. Im Vergleich zur Instrumentierung von Java-Programmen bestehen einige Unterschiede, die im Folgenden erläutert werden.

Normalisierung Im Allgemeinen ist es erforderlich, dass die Kontrollstrukturen eines Programms vor der eigentlichen Instrumentierung *normalisiert* werden. Die Normalisierung ist eine semantikerhaltende Transformation, die sämtliche Schleifenkonstrukte und z.B. `switch`/`case`-Blöcke durch einfache Bedingungen (`if`), Sprunganweisungen (`goto`) und -marken (Labels) ersetzt. Auf diese Weise vermeidet man die Notwendigkeit einer Spezialbehandlung der einzelnen Schleifenarten. Die Information über die ursprüngliche Kontrollanweisung geht zwar verloren, sie hat aber für die Konstruktion des Objektprozessgraphen ohnehin keine Bedeutung.

Programmrepräsentation Diese Überlegung muss berücksichtigt werden, wenn man die Art der Programmrepräsentation festlegen möchte, auf deren Basis die Instrumentierung erfolgen soll. Prinzipiell kann ein Programm auf Ebene des Quellcodes, des Maschinencodes oder aber einer Zwischendarstellung instrumentiert werden. Jede Herangehensweise hat dabei gewisse Nachteile: Eine direkte Modifikation des Quellcodes ist ohne eine syntaktische und semantische Analyse schwierig. Die Transformation von Maschinencode unterliegt hingegen einer Abhängigkeit von der jeweiligen Zielplattform. Für C-Programme wird die Instrumentierung daher auf der Bauhaus-internen, sprachunabhängigen Zwischendarstellung *IML* (Intermediate Language) [KGW98] durchgeführt. Nachdem ein C-Programm in eine IML übersetzt worden ist, findet zuerst die Normalisierung und anschließend die Instrumentierung der IML statt, bevor diese schließlich wieder in C-Quellcode zurückübersetzt wird.

Bytecode-Instrumentierung Im Gegensatz dazu erfolgt die Instrumentierung einer Java-Anwendung durch Modifikation des Bytecodes. Von Vorteil ist hierbei, dass sämtliche Kontrollstrukturen im Bytecode bereits in „normalisierter“ Form vorliegen, d.h. in Form von Bedingungen und Sprunganweisungen. Andererseits besteht keine Abhängigkeit zu einer Prozessorarchitektur, d.h. die Implementierung zur Instrumentierung kann auf alle Java-Programme angewandt werden.

Die Listings 3.2 und 3.3 zeigen beispielhaft, wie der Java-Bytecode der Methode `reverse()` vor und nach der Instrumentierung aussieht. Vor der Instrumentierung erkennt man bereits, dass die `while`-Schleife aus dem Quellcode durch eine `goto`- (Z. 0) und eine `ifeq`-Anweisung (Z. 16) ersetzt, d.h. normalisiert wurde. Hinweis: Die Bytecode-Anweisung `pop` (Z. 11) bezieht sich auf den Stack in der virtuellen Maschine, nicht zu verwechseln mit der zu analysierenden Klasse `Stack`, deren Methoden in den Zeilen 5, 8 und 13 aufgerufen werden.

Nach der Instrumentierung enthält der Bytecode eine hohe Anzahl von Logging-Anweisungen, die an allen relevanten Stellen im Code eingefügt wurden. Relevante Stellen sind diejenigen, die eine Auswirkung auf den Kontrollfluss haben. Dies sind Sprunganweisungen (bedingt und unbedingt) und Methodenaufrufe, d.h. Eintritt in Methoden und Return-Anweisungen. Hinzu kommen die Stellen, an denen eine Exception ausgelöst werden kann. Die Instanziierung einer zu analysierenden Klasse ist ebenfalls eine für die Erzeugung des Objektprozessgraphen relevante Stelle.

Jede instrumentierte Code-Stelle erhält eine eindeutige ID (z.B. `1_42`), die als Parameter der Logging-Anweisung übergeben wird, ggf. zusammen mit Informationen über das aktuelle Objekt bzw. die aktuelle Methode.

Der Aufruf einer atomaren Methode der zu analysierenden Klasse wird hierbei nicht explizit gekennzeichnet. Vielmehr wird bei der Instrumentierung zusätzlich eine Liste der Methoden der entsprechenden Klasse erstellt, die später bei der Konstruktion des DOPG verwendet wird, um die Rümpfe der atomaren Methoden aus dem Graph zu entfernen.

Die folgenden Logging-Anweisungen können bei der Instrumentierung einer Java-Anwendung in den Bytecode eingefügt werden:

- `log_new` markiert die Erzeugung eines zu analysierenden Objekts durch das Schlüsselwort `new` oder durch Aufruf der Methode `Class.newInstance()` der Reflection-API im Package `java.lang.reflect`.

- `log_addr` wird in die Konstruktoren der zu analysierenden Klasse eingefügt, um die „Speicheradresse“ eines neu erzeugten Objektes zu registrieren. Da es in Java keine expliziten, für den Benutzer sichtbaren Speicheradressen gibt, wird stattdessen die Methode `System.identityHashCode()` verwendet, welche in der Praxis für jedes Objekt einen eindeutigen Wert liefert. Dieser wird im Folgenden synonym zum Begriff Speicheradresse gebraucht.

  Der separate Aufruf von `log_new` und `log_addr` ist notwendig, da zum Zeitpunkt der Erzeugung noch kein Zugriff auf den `this`-Zeiger des Objektes möglich ist, der zur Bestimmung der Adresse benötigt wird.

- `log_read` und `log_write` kennzeichnen Lese- und Schreibzugriffe auf Variablen. Für das hier beschriebene Protokollerkennungsverfahren sind jedoch vielmehr die Aufrufe der öffentlichen Methoden eines Objektes von Bedeutung (atomare Aufrufe). Daher werden nicht-statische Methoden so instrumentiert, dass zu Beginn eines jeden Methodenrumpfes eine `log_read`-Anweisung auf den `this`-Zeiger eingefügt wird. Auf diese Weise wird festgehalten, dass der Aufruf der Methode einen Zugriff auf ein Objekt darstellt und um welches Objekt es sich jeweils handelt. Dies geschieht unabhängig davon, ob die Methode tatsächlich auf die Attribute des Objektes zugreift oder nicht.

- `log_lock` und `log_unlock` markieren das Betreten und Verlassen des zum betrachteten Objekt gehörenden Monitors.

- `log_branch_true` und `log_branch_false` werden an bedingten Sprüngen (Verzweigungen) eingefügt, so dass am Beginn eines jeden Zweiges jeweils eine dieser Methoden aufgerufen wird. Dabei gilt die Konvention, dass der Aufruf `log_branch_true` immer dann ausgeführt wird, wenn der bedingte Sprung stattgefunden hat. Ist die Bedingung nicht erfüllt, so dass die Ausführung in der nächsten Zeile fortgesetzt wird, erfolgt der Aufruf `log_branch_false`.

- `log_exception` kennzeichnet Stellen, an denen Exceptions gefangen werden können (`catch`-Anweisungen).

- `log_call_before` und `log_call_after` werden vor und nach Methodenaufrufen eingefügt. Auf diese Weise kann für jeden Methodenaufruf die genaue Stelle im Code des Aufrufers ermittelt werden.

- `log_entry`, `log_return` und `log_exceptional_return` zeichnen das Betreten und Verlassen von Methoden auf.

```
void reverse(java.util.Stack, java.util.Stack);
  Code:
   0:   goto    12
   3:   aload_2
   4:   aload_1
   5:   invokevirtual   #42; //Method java/util/Stack.pop:()Ljava/lang/Object;
   8:   invokevirtual   #28; //Method java/util/Stack.push:(Ljava/lang/Object;)Ljava/lang/Object;
   11:  pop
   12:  aload_1
   13:  invokevirtual   #46; //Method java/util/Stack.isEmpty:()Z
   16:  ifeq    3
   19:  return

}
```

Listing 3.2: Java-Bytecode vor der Instrumentierung

Zur Feststellung einer nicht gefangenen Exception wird bei der Instrumentierung für jede Methode ein Exception-Handler in den Bytecode eingefügt, der für den gesamten Methodenrumpf sämtliche Exceptions fängt, die dort implizit oder explizit auftreten können, aber nicht gefangen werden. Dieser künstliche Exception-Handler registriert das Auftreten einer Exception durch Aufrufen von `log_exceptional_return`. Anschließend wirft er die gefangene Exception erneut, so dass die Methode tatsächlich verlassen wird. Dieser Mechanismus ist auch in Listing 3.3 zu erkennen (Z. 100–111).

- `log_label` wird überall dort eingefügt, wo der Kontrollfluss zusammenläuft. Auf diese Weise wird erkannt, wenn Code-Stellen mehrere Male besucht werden, so wie es in Schleifen der Fall ist.

Ein auf diese Weise instrumentiertes Programm verursacht somit während seiner Ausführung eine Folge von Aufrufen von Logging-Anweisungen. Diese Folge ergibt die Spur des jeweiligen Programmlaufs. Jeder Eintrag dieser Spur besteht aus einer Spur-ID und der Art des aufgetretenen Ereignisses.

Besteht ein Programm aus mehreren Threads, so wird für jeden Thread eine eigene Spur erzeugt.

3.1.3.2 Filterung der Spur

Die durch das instrumentierte Programm generierte Spur enthält Informationen über sämtliche Code-Stellen, da während der Instrumentierung noch nicht entschieden werden kann, welche Stellen für das betrachtete Objekt relevant sein werden. Dies würde eine Form der statischen Analyse bedeuten, die dem Ansatz der dynamischen Analyse widerspräche. Somit ist es notwendig, die erzeugte Spur durch ein Filterungsverfahren auf die Objektspur zu reduzieren, welche nur die tatsächlich relevanten Methodenaufrufe der Spur enthält. Ein Methodenaufruf gilt als relevant, wenn im Rumpf der Methode auf das Objekt zugegriffen wird oder wenn der Rumpf selbst einen Aufruf einer relevanten Methode enthält.

Diese Definition führt zu folgendem Algorithmus: Bei der Auswertung der Spur steht jeweils nach Verlassen eines Methodenrumpfes fest, ob darin ein Zugriff auf das Objekt erfolgt ist. Ist dies der Fall, so gilt der Methodenaufruf als relevant. Ebenfalls als relevant gelten sämtliche Aufrufer, die sich zu diesem Zeitpunkt auf dem Call-Stack befinden. Wird eine Methode verlassen, ohne dass dieser Aufruf der Methode als relevant markiert wurde, so wird der gesamte Aufruf aus der Spur entfernt.

Dasselbe gilt auch für die `run()`-Methoden von Threads. Findet in einem Thread kein Zugriff auf das betrachtete Objekt statt, so wird auch die Information über den gesamten Thread aus der Spur herausgefiltert.

```
void reverse(java.util.Stack, java.util.Stack);
  Code:
   0:   ldc     #183; //String bs/Main.reverse(Ljava/util/Stack;Ljava/util/Stack;)V
   2:   ldc     #185; //String 1_42
   4:   invokestatic    #17; //Method bauhaus/enhancer/Log2Graph.log_entry:(Ljava/lang/String;Ljava/lang/String;)V
   7:   ldc     #187; //String 1_43
   9:   invokestatic    #25; //Method bauhaus/enhancer/Log2Graph.log_label:(Ljava/lang/String;)V
   12:  goto    49
   15:  ldc     #189; //String 1_44
   17:  invokestatic    #25; //Method bauhaus/enhancer/Log2Graph.log_label:(Ljava/lang/String;)V
   20:  aload_2
   21:  aload_1
   22:  ldc     #191; //String 1_45
   24:  invokestatic    #60; //Method bauhaus/enhancer/Log2Graph.log_call_before:(Ljava/lang/String;)V
   27:  invokevirtual   #155; //Method java/util/Stack.pop:()Ljava/lang/Object;
   30:  ldc     #191; //String 1_45
   32:  invokestatic    #64; //Method bauhaus/enhancer/Log2Graph.log_call_after:(Ljava/lang/String;)V
   35:  ldc     #193; //String 1_46
   37:  invokestatic    #60; //Method bauhaus/enhancer/Log2Graph.log_call_before:(Ljava/lang/String;)V
   40:  invokevirtual   #109; //Method java/util/Stack.push:(Ljava/lang/Object;)Ljava/lang/Object;
   43:  ldc     #193; //String 1_46
   45:  invokestatic    #64; //Method bauhaus/enhancer/Log2Graph.log_call_after:(Ljava/lang/String;)V
   48:  pop
   49:  ldc     #195; //String 1_47
   51:  invokestatic    #25; //Method bauhaus/enhancer/Log2Graph.log_label:(Ljava/lang/String;)V
   54:  aload_1
   55:  ldc     #197; //String 1_48
   57:  invokestatic    #60; //Method bauhaus/enhancer/Log2Graph.log_call_before:(Ljava/lang/String;)V
   60:  invokevirtual   #161; //Method java/util/Stack.isEmpty:()Z
   63:  ldc     #197; //String 1_48
   65:  invokestatic    #64; //Method bauhaus/enhancer/Log2Graph.log_call_after:(Ljava/lang/String;)V
   68:  ifeq    79
   71:  ldc     #199; //String 1_49_F
   73:  invokestatic    #166; //Method bauhaus/enhancer/Log2Graph.log_branch_false:(Ljava/lang/String;)V
   76:  goto    87
   79:  ldc     #201; //String 1_49_T
   81:  invokestatic    #171; //Method bauhaus/enhancer/Log2Graph.log_branch_true:(Ljava/lang/String;)V
   84:  goto    15
   87:  ldc     #183; //String bs/Main.reverse(Ljava/util/Stack;Ljava/util/Stack;)V
   89:  ldc     #203; //String 1_50
   91:  invokestatic    #36; //Method bauhaus/enhancer/Log2Graph.log_return:(Ljava/lang/String;Ljava/lang/String;)V
   94:  return
   95:  nop
   96:  nop
   97:  nop
   98:  nop
   99:  athrow
   100: astore  4
   102: ldc     #183; //String bs/Main.reverse(Ljava/util/Stack;Ljava/util/Stack;)V
   104: ldc     #207; //String 1_52
   106: invokestatic    #43; //Method bauhaus/enhancer/Log2Graph.log_exceptional_return:(Ljava/lang/String;Ljava/
        lang/String;)V
   109: aload   4
   111: athrow
  Exception table:
   from   to  target type
     7   100   100   Class java/lang/RuntimeException


}
```

Listing 3.3: Instrumentierter Java-Bytecode

3.1.3.3 Roh-Graph-Konstruktion

Aus einer Objektspur kann zunächst ein so genannter Roh-Graph (N, E) konstruiert werden. Dieser zeichnet sich dadurch aus, dass er im Vergleich zum fertigen DOPG zahlreiche zusätzliche Knoten enthält. Diese sind notwendig, damit alle wiederkehrenden Code-Stellen, und somit das Vorhandensein von Schleifen, erkannt werden können. Außerdem gibt es nur einen allgemeinen Kantentyp. Erst durch weitere Graph-Transformationen geht aus dem Roh-Graphen schließlich ein DOPG hervor.

Algorithmus 3.1 skizziert das Konstruktionsverfahren im Pseudocode. Für jede erstmals auftretende Spur-ID erzeugt die Funktion *getOrCreateNode* einen neuen Knoten und versieht diesen mit der Spur-ID. Der Typ des Knotens ergibt sich aus der Art des aufgetretenen Ereignisses. Bei jedem weiteren Spur-Eintrag mit einer bereits bekannten ID wird kein neuer Knoten erzeugt, sondern der bereits vorhandene Knoten mit dieser ID zurückgeliefert.

Vor der eigentlichen Auswertung der Objektspur wird der spätere Start-Knoten des DOPG erzeugt und der ansonsten leeren Knotenmenge N hinzugefügt. Besteht das Programm aus mehreren Threads, so werden die Objektspuren der einzelnen Threads nacheinander eingelesen.

Durch die Funktionsweise von *getOrCreateNode* wird eine bijektive Abbildung zwischen Spur-IDs und Knoten definiert. Somit kann die Spur-ID auch als Knoten-ID betrachtet werden, welche zusätzlich die zugehörige Quellcode-Position eindeutig definiert. Die Begriffe Spur-ID und Knoten-ID werden daher in dieser Diplomarbeit synonym verwendet. In der Implementierung wird die Knoten-ID als ein entsprechendes RFG-Attribut des Knotens repräsentiert. Formal wird dadurch auch die umgekehrte Abbildung $id : N \rightarrow String$ bestimmt. Diese Information wird aber erst später benötigt und ist daher im Algorithmus nicht explizit dargestellt.

Algorithmus 3.1: Roh-Graph-Konstruktion

Input: object trace (filtered)
Output: raw graph (N, E)

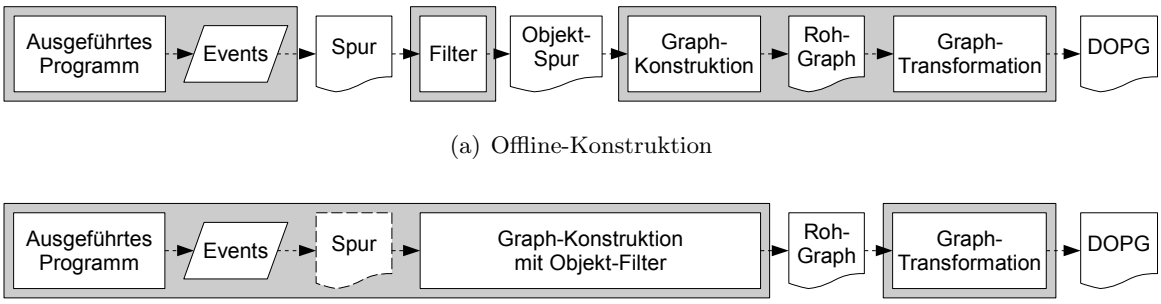
1 $N \leftarrow \{start\}$;
2 $E \leftarrow \emptyset$;
3 $lastNode \leftarrow start$;
4 **foreach** *event* of the object trace **do**
5 $\quad node \leftarrow getOrCreateNode(event)$;
6 $\quad N \leftarrow N \cup \{node\}$;
7 $\quad E \leftarrow E \cup \{(lastNode, node)\}$;
8 $\quad lastNode \leftarrow node$;

3.1.3.4 Offline- vs. Online-Konstruktion

Für die Konstruktion des Roh-Graphen existieren zwei Verfahren, deren Datenflüsse in Abbildung 3.4 gegenübergestellt werden. Die einzelnen Verarbeitungsschritte finden in unterschiedlichen Prozessen statt, welche durch graue Kästen symbolisiert werden.

Das Offline-Verfahren (a) zeichnet sich dadurch aus, dass sowohl die Filterung der Spur als auch die Roh-Graph-Konstruktion nach Beendigung des Programmlaufs erfolgen. Dazu ist es erforderlich, die gesamte Spur eines jeden Threads in einer Datei aufzuzeichnen. Diese Dateien können in Abhängigkeit von der Laufzeit des Programms beliebig groß werden.

Im Gegensatz dazu steht das Online-Verfahren (b), in welchem die Spur selbst nicht ge-



(a) Offline-Konstruktion

(b) Online-Konstruktion

Abbildung 3.4: Datenfluss bei der Konstruktion des DOPG [Qua07]

speichert wird. Vielmehr werden die auftretenden Events direkt zur Konstruktion des Roh-Graphen verwendet, die somit parallel zur Ausführung des Programms stattfindet. Die explizite Extraktion einer Objektspur ist hier nicht möglich, da schließlich erst nach Verlassen einer Methode feststeht, ob der seit Betreten der Methode generierte Teil der Spur zur Objektspur gehört oder nicht. Zu diesem Zeitpunkt ist die Graph-Konstruktion für den Methodenrumpf bereits erfolgt, und der entsprechende Teil der Spur nicht mehr verfügbar. Daher findet beim Online-Verfahren keine Filterung der Spur selbst statt. Vielmehr werden die entsprechenden Teilgraphen als relevant oder irrelevant markiert.

Die für die Graph-Konstruktion benötigten Datenstrukturen (zuletzt besuchter Knoten, Relevanz eines Methodenaufrufs etc.) werden für jeden Thread separat verwaltet. Laufen mehrere Threads parallel, so kann der Roh-Graph simultan an verschiedenen Stellen bearbeitet bzw. erweitert werden. Terminiert ein Thread, ohne auf das betrachtete Objekt zugegriffen zu haben, wird der gesamte Thread als irrelevant markiert. Treten mehrere Instanzen einer zu analysierenden Klasse auf, so wird für jede Instanz ein eigener Roh-Graph angelegt.

Auf diese Weise wird in Wirklichkeit der gesamte dynamisch erkannte Roh-Kontrollflussgraph erzeugt, von dem jedoch (je nach betrachtetem Objekt) nur eine Teilmenge der Knoten und Kanten als relevant markiert ist. Nach Beendigung des Programmlaufs wird dann für jedes Objekt der fertige Roh-Graph ausgegeben, der nur die als relevant markierten Graphelemente enthält. Im Gegensatz zu einer Spur, wie sie vom Offline-Verfahren erzeugt wird, ist die Größe des Roh-Graphen begrenzt, da der Roh-Graph maximal so groß werden kann wie der Roh-Kontrollflussgraph selbst.

Die Details des Online-Verfahrens werden in [Qua07] erläutert. Dort findet sich unter anderem eine Fallstudie, welche die Performanzunterschiede im Vergleich zum Offline-Verfahren untersucht.

3.1.3.5 Vereinfachende Transformationen

Nachdem der vollständige Roh-Graph konstruiert wurde, können einige Transformationen angewandt werden, die den Graph vereinfachen und von unnötigen Knoten befreien, wobei die Semantik stets erhalten bleibt.

- An Stellen, an denen sich der Kontrollfluss tatsächlich in einen `branch_true`- und einen `branch_false`-Knoten verzweigt, wird ein Condition-Knoten mit ausgehenden Kanten vom Typ Sequence_True und Sequence_False eingefügt. Die `branch`-Knoten werden durch einfache Label-Knoten ersetzt. Ebenfalls durch Labels ersetzt werden einzelne, d.h. nicht

paarweise auftretende `branch`-Knoten. Diese entstehen dadurch, dass die Bedingung für einen Sprung im analysierten Programmlauf immer denselben Wert annimmt, somit also einer der beiden möglichen Zweige nie ausgeführt wird.

- Besitzt eine Methode mehrere Return- oder Exceptional_Return-Knoten, so werden diese jeweils zu einem Return- bzw. Exceptional_Return-Knoten zusammengefasst.

- Unnötige Label-Knoten, irrelevante Kontrollstrukturen sowie Schlingen (Kanten mit identischem Quell- und Zielknoten) werden entfernt, bis keine weitere Vereinfachung mehr möglich ist.

3.1.3.6 Projektion

Für ein Objekt sind nur diejenigen Teile des Kontrollflussgraphen relevant, die während der Lebensdauer des Objektes, d.h. nach seiner Erzeugung, ausgeführt werden. Daher werden alle Knoten und Kanten entfernt, die vom Create-Knoten des Objektes nicht erreichbar sind. Dies betrifft insbesondere die Knoten und Kanten auf dem Pfad vom Start-Knoten des DOPG bis zum Create-Knoten, die nur einmal traversiert werden. Der Create-Knoten wird dadurch zum direkten, einzigen Nachfolger des Start-Knotens.

Das Ergebnis wird als *Projektion* des DOPG bezeichnet. Die Anwendung der Projektions-Transformation lässt sich – wie alle anderen Transformationsschritte auch – bei der Konstruktion des OPG deaktivieren. Dann entsteht ein nicht-projizierter DOPG, dem in den folgenden Kapiteln eine besondere Bedeutung zukommt.

3.1.3.7 Atomare Aufrufe

Im Roh-Graphen und auch nach Anwendung der beschriebenen Transformationen werden die Aufrufe atomarer Methoden im DOPG zunächst als gewöhnliche Methodenaufrufe dargestellt, d.h. der Rumpf einer atomaren Methode ist ebenfalls im DOPG sichtbar. Um die atomaren Aufrufe als solche zu kennzeichnen, werden alle Knoten entfernt, die zum Rumpf einer atomaren Methode gehören, einschließlich der Entry-, Return und/oder Exceptional_Return-Kanten, welche den Methodenrumpf mit den jeweiligen Call-Knoten verbinden. Auf diese Weise verbleiben von den atomaren Aufrufen nur die Call-Knoten. Zur Identifikation der zu entfernenden Methodenrümpfe wird die Liste der atomaren Methoden verwendet, die während der Instrumentierung erstellt wurde.

3.2 Protokollextraktion aus Objektprozessgraphen

Ein OPG enthält Informationen über die möglichen Sequenzen von atomaren Aufrufen und eignet sich daher zur Herleitung des Protokolls des betrachteten Objekts. Da das Protokoll in Form eines endlichen Automaten dargestellt werden soll, sind einige Transformationen notwendig, die aus einem OPG einen Protokollautomaten konstruieren. Diese sind u.a. in [QK07] beschrieben und sollen hier kurz erläutert werden.

Die gesamte Verarbeitungskette der beteiligten Werkzeuge ist in Abbildung 3.5 zu sehen.

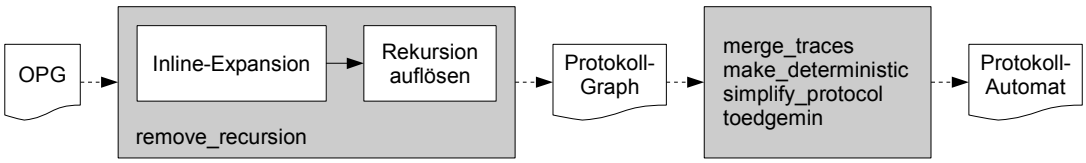


Abbildung 3.5: Transformation eines OPG zum Protokollautomaten

3.2.1 Rekursionsauflösung

Durch einen endlichen Automaten wird eine reguläre Sprache definiert. Ein OPG hingegen ist in der Lage, eine kontextfreie Sprache darzustellen. Dies wird durch die Modellierung rekursiver Methodenaufrufe möglich. Zur Transformation in einen endlichen Automaten ist daher zunächst eine Rekursionsauflösung notwendig. Dadurch ergibt sich eine Verallgemeinerung des durch den OPG definierten Protokolls zu einer regulären Sprache.

Die Motivation und das Verfahren zur Rekursionsauflösung wird in der Diplomarbeit von Haak [Haa04] ausführlich erläutert. Dort und in [QK07] wird die Funktionsweise des entsprechenden Werkzeugs `remove_recursion` im Zusammenhang vorgestellt. Tatsächlich besteht der darin implementierte Algorithmus aber aus zwei unabhängigen Schritten, was sich für diese Diplomarbeit noch als nützlich herausstellen wird.

1. Der erste Schritt der Rekursionsauflösung besteht in der *Inline-Expansion* von Methodenaufrufen. Für eine Methode, die von mehreren Stellen im Programm aufgerufen wird, entsteht bei der Konstruktion des OPG auch für jede dieser Stellen ein eigener Call-Knoten. Der Teilgraph für den Rumpf der Methode wird hingegen nur einmal angelegt. Der Entry-Knoten einer solchen Methode verfügt daher über eine entsprechende Anzahl eingehender Invoke-Kanten. Analoges gilt für Return- und Exceptional_Return-Knoten.

   Beim Inlining wird für jeden Aufruf einer Methode eine Kopie ihres Rumpfes erstellt, es sei denn es handelt sich um einen rekursiven Aufruf. Letzteres erkennt man durch Verwendung eines Call-Stacks beim Traversieren des OPG. Wird ein Call-Knoten besucht, der sich zu diesem Zeitpunkt bereits auf dem Call-Stack befindet, so liegt ein rekursiver Aufruf vor. In diesem Fall wird keine neue Kopie des Methodenrumpfes erzeugt, sondern stattdessen das auf dem Call-Stack vorhandene Exemplar verwendet.

   Das Resultat der Inline-Expansion wird in dieser Diplomarbeit als *expandierter Objektprozessgraph* bezeichnet.

2. Erst im zweiten Schritt findet ein Aufspalten von Call-Knoten in jeweils zwei einzelne Knoten statt. Ein neuer Call-Knoten verbindet dabei die eingehenden Kanten mit den ausgehenden Invoke-Kanten, ein anderer neuer Knoten die eingehenden Return- und Exceptional_Return-Kanten mit den ausgehenden Kanten des ursprünglichen Call-Knotens. Auf diese Weise eliminiert man die besondere Semantik der Call-Knoten bzw. des interprozeduralen Kontrollflusses.

   Nun können sämtliche Knoten aus dem Graph entfernt werden, die keine atomaren Aufrufe (oder ggf. Read-/Write-Knoten) darstellen, mit Ausnahme von Start- und Final-Knoten. Es werden somit alle Knoten der Typen Entry, Return, Exceptional_Return, Condition und Exception beseitigt.

   Der dadurch entstehende Graph heißt *Protokollgraph*. Er repräsentiert eine reguläre Sprache, die eine Obermenge der vom ursprünglichen OPG repräsentierten, kontextfreien Sprache ist.

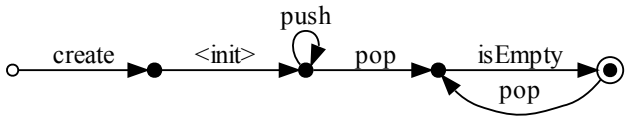


Abbildung 3.6: Protokollautomat, erzeugt aus dem Beispiel-OPG in Abbildung 3.2

3.2.2 Transformation in Protokollautomaten

Der aus der Rekursionsauflösung resultierende Protokollgraph kann in einen nichtdeterminischen endlichen Automaten (NEA, siehe Abschnitt 2.2.1) transformiert werden. Dazu erhält jede Kante die Beschriftung ihres Zielknotens, d.h. den Namen eines atomaren Aufrufs oder eines Lese-/Schreibzugriffs. Die Beschriftung der Knoten selbst wird entfernt. Auf diese Weise bilden die Knoten des Graphen die Menge der Zustände des endlichen Automaten. Die Kanten mit Quell-Knoten, Beschriftung und Ziel-Knoten definieren hingegen die Zustandsüberführung. Der Start-Knoten wird zum Startzustand des Automaten, während alle Knoten mit adjazentem Final-Knoten als Endzustände definiert werden.

Durch die Analyse von mehreren Instanzen einer Klasse ergeben sich mehrere solcher NEAs. Diese können durch Verschmelzen ihrer Startzustände zu einem Automaten vereinigt werden. Durch Transformation in einen Potenzautomaten und Anwendung von Algorithmen zur Minimierung [HMU02] sowie durch weitere, optionale Vereinfachungen [QK07] erhält man schließlich einen minimalen, deterministischen endlichen Automaten, der ein Protokoll in Form einer regulären Sprache repräsentiert. Ein solcher Automat wird daher auch Protokollautomat genannt und ist für das jeweilige Protokoll eindeutig bis auf die Bezeichnung der Zustände (vgl. Abschnitt 2.2.2).

Die Transformation in einen Protokollautomaten geschieht der Reihe nach durch Anwendung der Werkzeuge `merge_traces` (Verschmelzen der Automaten), `make_deterministic` (Potenzautomaten bilden), `simplify_protocol` (Vereinfachungen) und `toedgemin` (Knoten-Beschriftungen auf Kanten übertragen, Automaten minimieren). Aus technischen Gründen findet also die eigentliche Transformation zum NEA erst im vorletzten Schritt statt, während die anderen Transformationen noch auf dem Protokollgraphen – jeweils im Sinne des korrespondierenden NEA – durchgeführt werden.

Wendet man die beschriebenen Schritte auf den in Abbildung 3.2 (S. 14) gezeigten Beispiel-OPG an, so erhält man den Protokollautomaten in Abbildung 3.6.

3.3 Protokollerkennung bei mehreren Threads

In diesem Abschnitt werden die Probleme aufgezeigt, die bei der Anwendung des bisherigen Verfahrens zur Protokollerkennung im Zusammenhang mit Multithreading-Anwendungen auftreten. Das bisherige Verfahren liefert ein falsches Ergebnis, sobald ein Objekt der zu analysierenden Klasse von mehr als einem Thread verwendet wird. Dieser Sachverhalt soll zunächst anhand eines einfachen Beispiels erläutert werden. Dass dieses Problem auch eine praktische Bedeutung hat, wird später im Rahmen der Evaluation in Kapitel 6 demonstriert. Dazu werden einige reale Java-Anwendungen vorgestellt, in denen jeweils eines der zentralen

```
1  public class Main {
2      public static void main(String args[]) {
3          MySocket socket = new MySocket();
4          SocketConnectThread thread = new SocketConnectThread(socket);
5          thread.start();
6          ...
7          thread.join();
8          ...
9          int result;
10         do {
11             result = socket.read();
12         } while (result != -1);
13     }
14     ...
15 }
16
17 class SocketConnectThread extends Thread {
18     ...
19     public void run() {
20         socket.connect();
21     }
22 }
```

Listing 3.4: Beispiel für eine Multithreading-Anwendung

Objekte von mindestens zwei Threads verwendet wird.

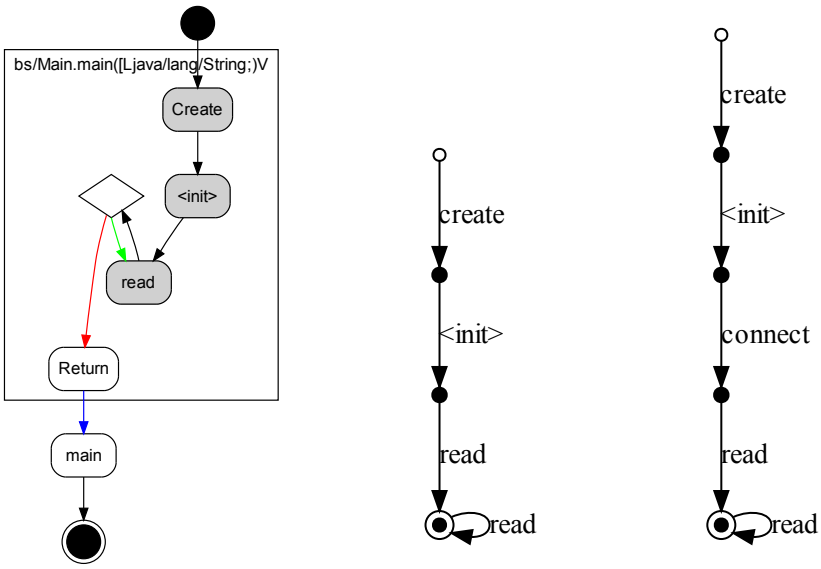
3.3.1 Entstehung unvollständiger Protokolle

Welche Probleme auftreten, wenn mithilfe der vorhandenen Werkzeuge eine Multithreading-Anwendung analysiert wird, zeigt sich bereits an dem einfachen Beispiel in Listing 3.4. Es könnte sich dabei um eine Anwendung mit einer Netzwerkanbindung und einer GUI handeln. Während die GUI aufgebaut wird, findet parallel dazu im Hintergrund eine Initialisierung der Netzwerkschnittstelle statt, hier durch die Klasse `MySocket` symbolisiert. Nach Abschluss des Initialisierungsvorgangs werden schließlich die vorhandenen Daten aus der Schnittstelle gelesen. Da die Initialisierung Zeit kostet, wird sie in einen Thread verlegt, so dass die GUI nicht blockiert wird. Ein solches Software-Design ist nicht unüblich, gerade bei interaktiven Anwendungen.

Das Protokoll der Klasse `MySocket` soll nun aus diesem Beispielprogramm extrahiert werden. Die bisherige Implementierung erzeugt daraus einen OPG und einen Protokollautomaten, zu sehen in Abbildung 3.7 (a) und (b). Wie man feststellen muss, wurden fälschlicherweise nur die Aufrufe der Methode `read()` berücksichtigt. Das vorgesehene Protokoll für die Klasse `MySocket` erfordert jedoch, dass vor dem ersten Aufruf von `read()` die Methode `connect()` aufgerufen wird, was von der Implementierung auch eingehalten wird. Es wäre somit wünschenswert, dass die Analyse ein Protokoll wie in Abbildung 3.7 (c) liefert.

3.3.2 Problemanalyse

Der OPG enthält offensichtlich nur diejenigen Knoten, die vom Create-Knoten aus innerhalb desselben Threads erreichbar sind. Es wird also nur derjenige Thread betrachtet, der das Objekt auch erzeugt hat. In vielen Anwendungen finden aber gerade Vorgänge wie Erzeugen, Initialisieren und Verwenden eines Objektes in unterschiedlichen Threads statt. Dabei kann es zu nebenläufigen Sequenzen von Atomic Calls kommen. Häufig besteht aber auch eine explizite Synchronisation. Dann treten die Atomic Calls in unterschiedlichen Threads durchaus in einer definierten Reihenfolge auf. Diese unterliegt wiederum einem bestimmten Protokoll,



(a) OPG (b) falsches Protokoll (c) erwartetes Protokoll

Abbildung 3.7: Falsches und erwartetes Protokoll für Listing 3.4

welches von der Schnittstelle des Objekts vorgegeben wird. Es wäre somit wünschenswert, auch derartige Protokolle aus einem Software-System extrahieren zu können.

Verantwortlich für das Entfernen der anderen Knoten ist die *Projektion* des OPG (siehe Abschnitt 3.1.3.6). Daher ist es naheliegend, für einen ersten Versuch einen OPG ohne Anwendung der Projektion zu erzeugen. Daraufhin erhält man das in Abbildung 3.8 (a) zu sehende Ergebnis.

Der direkte Nachfolger des Start-Knotens ist nun nicht mehr der Create-Knoten des Objektes, sondern der so genannte Initial-Entry-Knoten. Dort teilen sich die Pfade vom Start-Knoten in die einzelnen Threads auf. Für jeden Thread existiert ein Call-Knoten, der den Aufruf der `main()` bzw. `run()`-Methode symbolisiert. Jeder dieser Call-Knoten ist direkter Nachfolger des Initial-Entry-Knotens.

Die Existenz und der innere Kontrollfluss eines Threads werden also auf diese Weise im OPG dargestellt. Nicht enthalten ist jedoch eine Information darüber, welche Abhängigkeiten zwischen den Threads bestehen. Dementsprechend ist nachvollziehbar, dass die Projektions-Transformation den gesamten anderen Thread entfernt. Schließlich ist keiner der Knoten dieses Threads vom Create-Knoten aus erreichbar.

Versucht man, aus einem solchen nicht-projizierten OPG mit den vorhandenen Werkzeugen einen Protokollautomaten zu erzeugen, so erhält man ein Ergebnis wie in Abbildung 3.8 (b). Auf diese Weise tritt eine Änderung der Semantik ein, die so nicht beabsichtigt sein kann: Ein Initial-Entry-Knoten eines OPG mit mehreren direkten Nachfolgern repräsentiert *Nebenläufigkeit*. Ein Startzustand eines Protokollautomaten mit mehreren ausgehenden Zustandsübergängen symbolisiert hingegen *Alternativen*.

Um auch Nebenläufigkeit in Form eines Protokolls korrekt wiedergeben zu können, benötigt man also eine andere Art der Darstellung. Welche Möglichkeiten dafür in Frage kommen

und welche Informationen damit tatsächlich repräsentiert werden sollen, ist Gegenstand der folgenden Kapitel.

3.4 Andere Protokollerkennungsverfahren

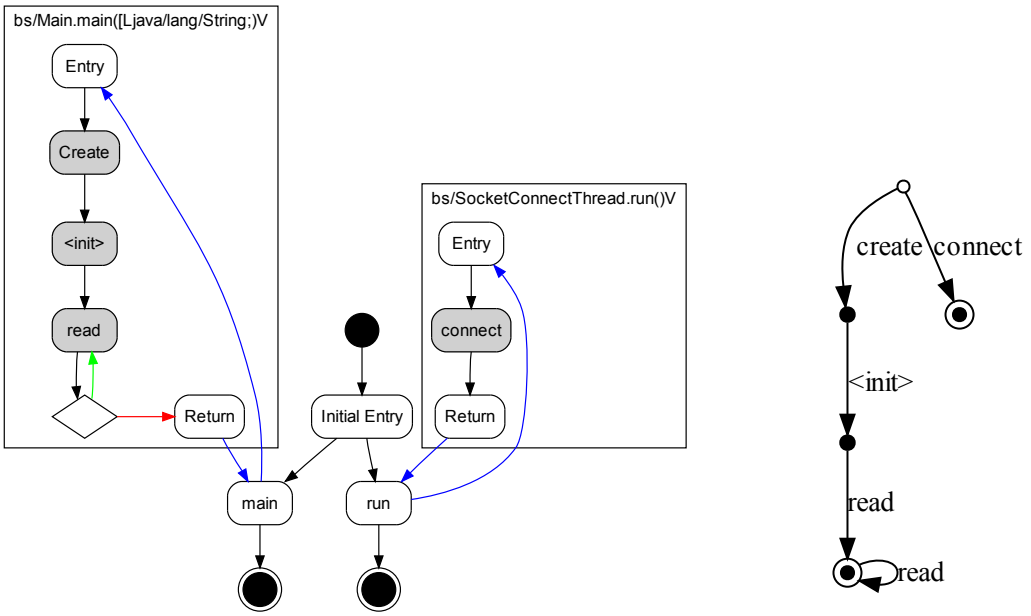
In [QK07] werden neben dem hier beschriebenen Verfahren noch andere Techniken zur Protokollerkennung vorgestellt und miteinander verglichen. Viele dieser Verfahren basieren auf der Konstruktion eines so genannten *prefix tree acceptor* (PTA), einem endlichen Automaten, der genau die (endliche) Menge der während der dynamischen Analyse beobachteten Sequenzen erkennt. Erst durch Verallgemeinerung des PTA nach heuristischen Kriterien ergibt sich dann ein Automat mit nicht endlicher Sprache.

Der entscheidende Unterschied zum DOPG-basierten Protokollerkennungsverfahren besteht darin, dass die PTA-basierten Techniken nur die Sequenz der atomaren Operationen verarbeiten, während das DOPG-basierte Verfahren den gesamten Kontrollflussgraph berücksichtigt. Dadurch erkennt das DOPG-basierte Verfahren Schleifen aufgrund wiederkehrender Quellcode-Positionen, während die PTA-basierten Techniken versuchen, Schleifen in der gegebenen Sequenz allein durch Anwendung der heuristischen Verfahren zu finden.

3.5 Zusammenfassung

Die im Bauhaus-Projekt entstandenen Werkzeuge erlauben es, durch eine dynamische Analyse das Protokoll einer Klasse in einem Java-Programm herzuleiten. Durch Instrumentieren und Ausführen des zu analysierenden Programms erhält man Spur-Informationen, die zu einem dynamischen Objektprozessgraphen (DOPG) verarbeitet werden können. Aus dem DOPG geht schließlich durch Graphtransformation ein Protokollautomat hervor.

Der Objektprozessgraph enthält jedoch keine Knoten außerhalb des Threads, der das betrachtete Objekt erzeugt hat. Wird ein Objekt von mehr als einem Thread verwendet, so ergibt sich aus diesem Grund im Allgemeinen ein falsches Protokoll. Der OPG bietet zwar bei Verzicht auf die Projektions-Transformation eine nebenläufige Darstellung der relevanten Threads, diese Information ist jedoch wiederum durch einen endlichen Automaten nicht direkt repräsentierbar.



(a) OPG ohne Projektion

(b) falsches Protokoll

Abbildung 3.8: Falsches Protokoll ohne OPG-Projektion für Listing 3.4

KAPITEL 4

Linearisierung nebenläufiger Prozesse

In diesem Kapitel werden verschiedene Konzepte zur Realisierung und Analyse von Nebenläufigkeit vorgestellt. Darauf basierend wird anschließend erörtert, wie die Linearisierung nebenläufiger Prozesse zur Protokollerkennung bei Multithreading-Anwendungen beitragen kann.

Inhalt

4.1 Motivation **29**
4.2 Technische Umsetzung **30**
4.2.1 Prozesse und Threads 30
4.2.2 Scheduling 30
4.2.3 Multiprozessor-Systeme 31
4.2.4 Nebenläufigkeit in Java 31
4.3 Petri-Netze **31**
4.3.1 Definition 32
4.3.2 Beispiel 33
4.3.3 Entscheidbarkeit 34
4.4 Sequentielle Darstellung **34**
4.5 Interleaving und Objektprozessgraphen **35**
4.6 Verwandte Forschung **36**

4.1 Motivation

In der bisherigen Implementierung werden die einzelnen Threads zwar im OPG dargestellt, jedoch ohne einen Zusammenhang zwischen den Threads herzustellen. Insbesondere fehlen Informationen darüber,

1. zu welchem Zeitpunkt ein bestimmter Thread gestartet und beendet wird
2. von welchem anderen Thread ein bestimmter Thread erzeugt wird
3. inwiefern die einzelnen Threads nebenläufig ausgeführt werden und welche Verzahnungsmöglichkeiten bestehen.

Für das Programmverstehen im Allgemeinen sind alle genannten Fragestellungen interessant. Für die Protokollerkennung hingegen besteht besonderes Interesse an Punkt 3. Schließlich ist aus der Sicht eines Objekts nur die Information relevant, die eine Aussage über die globale Sequenz der aufgetretenen Atomic Calls trifft. In welchem Thread ein Atomic Call jeweils

stattfindet, ist für das Protokoll nicht von Belang. Darüber hinaus könnte ein Refactoring unerwünschte Auswirkungen auf die Protokollerkennung haben. So kann der Fall eintreten, dass im Rahmen eines Refactorings ein bestimmter Atomic Call zwar in einen anderen Thread verschoben wird, die globale Reihenfolge der Atomic Calls jedoch erhalten bleibt. Dies würde keine Änderung am tatsächlichen Protokoll bedeuten. Die Analyse würde jedoch ein unterschiedliches Resultat ergeben. Das Ziel soll also sein, das Protokoll eines Objektes weiterhin als Sequenz atomarer Operationen zu betrachten.

4.2 Technische Umsetzung

Zunächst soll kurz auf die technischen Prinzipien eingegangen werden, welche die Implementierung nebenläufiger Anwendungen erst ermöglichen, sowohl im Allgemeinen als auch speziell im Hinblick auf die Programmiersprache Java.

4.2.1 Prozesse und Threads

Moderne Betriebssysteme unterstützen die parallele (oder quasiparallele) Ausführung mehrerer Aufgaben. Diese sind in einzelne Prozesse aufgeteilt, welche wiederum aus einem oder mehreren Threads bestehen. Die Verwaltung eines solchen Systems betrifft viele Aspekte hinsichtlich der Zuteilung und Virtualisierung von Ressourcen für die einzelnen Prozesse und Threads, die von Stallings [Sta03] und Tanenbaum [TB09] ausführlich beschrieben werden. Die Virtualisierung führt dazu, dass konzeptionell jeder Prozess seinen eigenen (virtuellen) Arbeitsspeicher, seine eigenen Datei-Handles etc. besitzt. Diese Ressourcen werden wiederum von den einzelnen Threads eines Prozesses gemeinsam verwendet. Der Ausführungskontext wird dagegen bei jedem Thread separat verwaltet.

Diese Unterscheidung ist jedoch für eine abstrakte Betrachtung von Nebenläufigkeit nicht relevant. Es reicht daher aus dieser Sichtweise, einen Prozess als Zusammenfassung von Threads zu bezeichnen. Es ist mitunter sogar sinnvoll, die Begriffe Prozess und Thread synonym zu gebrauchen[1]. Daher wird im Folgenden auf eine entsprechende Unterscheidung verzichtet, sofern der Kontext eindeutig ist. Letztlich ist nämlich für die Analyse von Protokollen nur die Verwaltung der Ressource *CPU-Zeit* von Bedeutung.

4.2.2 Scheduling

Parallele Ausführung von Prozessen kann sowohl auf einem Multiprozessorsystem als auch auf einem Einprozessorsystem realisiert werden. Im letzteren Fall spricht man auch von *Quasi-Parallelität*[2]. Diese wird realisiert, indem die CPU-Zeit auf die einzelnen Prozesse eines Systems aufgeteilt wird. Die CPU führt dazu einen Prozess immer nur für eine kurze Zeitspanne aus, bevor sie diesen unterbricht und zum nächsten Prozess wechselt. Durch schnelles Hin- und Herwechseln zwischen Prozessen entsteht dabei der Eindruck der Parallelität. Die Unterbrechung eines Prozesses kann durch diesen selbst ausgelöst werden, wenn er einen blockierenden Systemaufruf ausführt (z.B. Ein-/Ausgabe) oder dadurch bedingt sein, dass die aktuelle *Zeitscheibe* abgelaufen ist, d.h. die maximale Zeitspanne für die dauerhafte Ausführung dieses Prozesses erreicht wurde.

Der *Scheduler* ist die Komponente eines Betriebssystems oder einer virtuellen Maschine,

[1] vgl. [Sta03] S. 236
[2] vgl. [TB09] S. 124

die für die Zuteilung der CPU-Zeit verantwortlich ist[3]. Die Arbeitsweise eines bestimmten Schedulers nennt man entsprechend *Scheduling-Strategie*. Jedes Mal, wenn die Ausführung eines Prozesses unterbrochen wurde, entscheidet der Scheduler nach bestimmten Kriterien, welcher Prozess als nächstes ausgeführt wird.

Diese Maßnahmen zur Zuteilung der CPU-Zeit sind auch auf Multiprozessorsystemen notwendig, da im Allgemeinen eine größere Anzahl Prozesse ausgeführt werden soll als CPUs vorhanden sind. Zu den Aufgaben des Schedulers kommt dort noch die Verteilung der einzelnen Prozesse auf die verschiedenen CPUs hinzu[4].

4.2.3 Multiprozessor-Systeme

Auf Rechnersystemen mit mehr als einem CPU-Kern können die Threads einer Anwendung nicht nur scheinbar parallel (durch Scheduling), sondern zeitgleich ausgeführt werden. Für das DOPG-basierte Protokollerkennungsverfahren erfordert dieser Unterschied jedoch keine besondere Beachtung. Alle für die DOPG-Konstruktion relevanten Ereignisse, die durch ein instrumentiertes Programm generiert und aufgezeichnet werden, laufen an einer zentralen Verarbeitungskomponente ein. Diese ist durch einen globalen Monitor geschützt, so dass unabhängig von der Rechnerarchitektur stets höchstens ein Thread zur Zeit ein aufgetretenes Ereignis speichern kann. Die globale Reihenfolge der Events ist daher nach jedem Programmlauf definiert, wenn auch – aus Sicht des Progammierers – dem Zufall überlassen.

4.2.4 Nebenläufigkeit in Java

Da der Bytecode von Java-Programmen in einer virtuellen Maschine (JVM) ausgeführt wird, liegt auch die Verwaltung der Nebenläufigkeit in der Verantwortung der JVM[5]. Dies betrifft sowohl die Realisierung der nebenläufigen Ausführung als auch die Synchronisationsmaßnahmen.

Wird die JVM auf einem Betriebssystem verwendet, das selbst das Konzept der Threads unterstützt, so kann jeder Thread eines Java-Programms in einem eigenen *nativen*, d.h. vom Betriebssystem verwalteten Thread verarbeitet werden. Andernfalls kann die JVM auch selbst mithilfe eines eigenen Schedulers eine Quasi-Parallelität herstellen. Aus Sicht des Betriebssystems werden dann alle Java-Threads in einem einzigen nativen Thread abwechselnd ausgeführt. Ein Nachteil dieser Lösung ist, dass auf diese Weise die Java-Threads, im Gegensatz zu nativen Threads, nicht auf mehrere Prozessoren verteilt werden können.

Die Spezifikation der virtuellen Maschine schreibt nicht vor, auf welche Art die Nebenläufigkeit realisiert werden muss. Für die Protokollerkennung ist auch diese Frage nicht von Bedeutung, denn für die unterschiedlichen Realisierungen gelten dieselben Überlegungen, wie sie bereits in Abschnitt 4.2.3 angestellt wurden.

4.3 Petri-Netze

Unabhängig von jeder technischen Umsetzung sind die theoretischen Konzepte zur Darstellung nebenläufiger Prozesse. Mit dem Petri-Netz wird hier eine Repräsentationsform vorgestellt, die sich in Theorie und Praxis als sehr nützlich erwiesen hat. Das Konzept der Petri-

[3] vgl. [Sta03] S. 458ff.
[4] vgl. [Sta03] S. 508
[5] vgl. [Ull06] S. 493f.

Netze und weitere Darstellungsformen werden von Priese und Wimmel [PW02] ausführlich erläutert und mathematisch formal definiert. Mit der Linearisierung wird dort ein wichtiger Aspekt aufgegriffen, der auch für diese Diplomarbeit von Bedeutung ist, da sie eine formale Grundlage für den Zusammenhang von nebenläufiger und sequentieller Darstellung bildet.

4.3.1 Definition

Ein Petri-Netz erinnert an einen endlichen Automaten und kann auch als dessen Verallgemeinerung bezeichnet werden. Ein Petri-Netz lässt sich ebenfalls als ein Graph darstellen, der allerdings über zwei unterschiedliche Arten von Knoten verfügt. Diese werden als *Stellen* und *Transitionen* bezeichnet. Die gerichteten Kanten des Petri-Netzes verlaufen niemals zwischen zwei Knoten derselben Art.

Während bei einem endlichen Automaten zu einem bestimmten Zeitpunkt genau einer seiner Zustände als aktuell gilt, verwendet ein Petri-Netz so genannte Marken, auch Token genannt. Jede Stelle enthält in einer bestimmten Konfiguration des Petri-Netzes eine Anzahl $n \geq 0$ von Token.

In der grafischen Darstellung eines Petri-Netzes werden die Transitionen als Rechtecke und die Stellen als Kreise gezeichnet. Innerhalb eines solchen Kreises kann sich sich eine Anzahl schwarzer Punkte befinden, welche die in der Stelle enthaltenen Token repräsentieren. Alternativ kann die Anzahl Token am selben Ort auch als Zahl angegeben werden.

Die Konfiguration (d.h. der Gesamtzustand) eines Petri-Netzes verändert sich, indem eine Transition t *feuert*. Es wird an allen Stellen, von denen eine Kante zu t verläuft (den Eingangsstellen), gleichzeitig je eine Marke pro Kante entfernt. Anschließend wird an allen Stellen, zu denen eine Kante von t hinführt (den Ausgangsstellen), je eine Marke pro Kante hinzugefügt. Es können also auch mehrere Kanten parallel verlaufen, so dass bei einem Feuervorgang an einer Stelle mehrere Marken entfernt bzw. hinzugefügt werden. Eine Transition kann immer dann feuern, wenn in allen Eingangsstellen die entsprechenden Marken vorhanden sind. Die Feuervorgänge der einzelnen Transitionen eines Petri-Netzes unterliegen keinem globalen Takt. Dadurch agieren sie weitgehend unabhängig und somit nebenläufig.

Betrachtet man jedoch das sequentielle Verhalten eines Petri-Netzes, so lässt sich auch eine so genannte *Petri-Netz-Sprache* finden. Das sequentielle Verhalten eines nebenläufigen Systems bezeichnet man auch als *Interleaving*-Verhalten.

Die für die Definition einer Petri-Netz-Sprache notwendigen Formalismen werden im Folgenden kurz erläutert.

Ein Petri-Netz ist ein System $N = (P, T, \mathbb{F}, \mathbb{B}, s_0)$ mit endlichen, angeordneten Mengen von Stellen $P = \{p_1, \ldots, p_{|P|}\}$ und Transitionen $T = \{t_1, \ldots, t_{|T|}\}$ mit $P \cap T = \emptyset$. $\mathbb{F}$ und $\mathbb{B}$ sind $|P| \times |T|$-Matrizen über $\mathbb{N}$, die für jedes Paar $(p_i, t_j) \in P \times T$ die Anzahl der inzidenten Kanten angeben. So verlaufen genau $\mathbb{F}_{i,j}$ Kanten von p_i nach t_j und $\mathbb{B}_{i,j}$ Kanten von t_j nach p_i.

Unter einem *Zustand* eines Petri-Netzes versteht man eine Abbildung $s : P \to \mathbb{N}$, welche jeder Stelle $p \in P$ eine Anzahl Token $s(p)$ zuordnet. Die Menge aller Zustände wird mit $\mathbb{N}^P$ bezeichnet. Der initiale Zustand oder *Startzustand* eines Petri-Netzes ist $s_0 \in \mathbb{N}^P$. Ein Zustand kann auch als Spaltenvektor mit $|P|$ Zeilen geschrieben werden.

Feuert eine Transition $t \in T$, so sind aus $\mathbb{F}$ und $\mathbb{B}$ jeweils die Spalten von Bedeutung, welche die mit t inzidenten Kanten angeben. Diese Spaltenvektoren bezeichnet man mit $\mathbb{F}(t)$ bzw. $\mathbb{B}(t)$. Befindet sich ein Petri-Netz in einem Zustand $s \in \mathbb{N}^P$, so geht es beim Feuern der Transition t in den Zustand $s' \in \mathbb{N}^P$ über, indem gemäß $\mathbb{F}$ und $\mathbb{B}$ die entsprechenden Marken

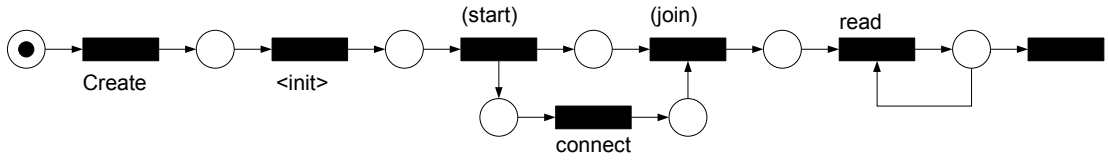


Abbildung 4.1: Petri-Netz für das Protokoll von `MySocket` in Listing 3.4

entfernt bzw. hinzugefügt werden. Die Voraussetzung für das Feuern ist, dass im Zustand s in den betroffenen Stellen genügend Marken vorhanden sind. Man schreibt „t feuert von s nach s'“ oder äquivalent $s[t{>}s'$ wenn gilt:

- $s \geq \mathbb{F}(t)$
- $s' = s - \mathbb{F}(t) + \mathbb{B}(t)$

Diese Definition lässt sich auf eine Sequenz von Transitionen $\sigma \in T^*$ erweitern. Für alle $t \in T, \sigma \in T^*$ wird definiert:

$$\begin{aligned} s[\epsilon{>}s' \quad &:\Longleftrightarrow \quad s = s' \\ s[\sigma t{>}s' \quad &:\Longleftrightarrow \quad \exists s'' \in \mathbb{N}^P : s[\sigma{>}s'' \wedge s''[t{>}s' \end{aligned}$$

Eine Sequenz $\sigma \in T^*$ mit $s[\sigma{>}s'$ heißt *Feuersequenz* von s nach s'.

Um mithilfe eines Petri-Netzes eines Sprache zu definieren, benötigt man nun noch, ähnlich wie bei einem endlichen Automaten, eine Beschriftung der Transitionen und eine Menge von terminalen Zuständen.

In einem *beschrifteten* Petri-Netz $N = (P, T, \mathbb{F}, \mathbb{B}, \Sigma, h_l)$ existiert zusätzlich ein Alphabet Σ, ein Zeichen $\tau \notin \Sigma$ und eine Beschriftungsfunktion $h_l : T \to \Sigma \cup \{\tau\}$. Die Beschriftungsfunktion lässt sich wie üblich auf Sequenzen von Transitionen bzw. Zeichen erweitern, wobei mit τ beschriftete Transitionen nicht in der Sequenz auftauchen.

In einem *terminalen* Petri-Netz $N = (P, T, \mathbb{F}, \mathbb{B}, \Sigma, h_l, S_f)$ ist darüber hinaus eine Menge von terminalen Zuständen $S_f \subseteq \mathbb{N}^P$ definiert. Dann ist die *(terminale) Sprache* des Petri-Netzes

$$L(N) = \{w \in \Sigma^* \mid \exists \sigma \in T^*, s_f \in S_f : s_0[\sigma{>}s_f \wedge w = h_l(\sigma)\}$$

Die Wörter der Sprache bestehen also aus den Beschriftungen von Transitionen einer Feuersequenz, durch die das Petri-Netz vom Startzustand in einen der terminalen Zustände übergeht.

Die von einem endlichen Automaten erkannten, regulären Sprachen sind Spezialfälle der Petri-Netz-Sprachen[6], d.h. jedes bisher als endlicher Automat dargestellte Protokoll lässt sich auch als Petri-Netz-Sprache formulieren.

4.3.2 Beispiel

In Abbildung 4.1 ist zum Beispiel das Protokoll der Klasse `MySocket` aus `lst:JavaSocketInit` in Form eines beschrifteten, terminalen Petri-Netzes dargestellt (der Endzustand ist hier erreicht, wenn keine Marken mehr vorhanden sind). Die Sprache des Petri-Netzes entspricht der des endlichen Automaten für das erwartete Protokoll in Abbildung 3.7 (c), sie ist also regulär.

[6] vgl. [PW02] S. 204ff.

Dennoch stellt das Petri-Netz im Gegensatz zum endlichen Automaten den Zusammenhang dar, dass das Beispielprogramm aus zwei Threads besteht. Feuert die mit (start) gekennzeichnete Transition, so entstehen zwei Marken, wodurch im Allgemeinen das Vorliegen von Nebenläufigkeit angezeigt wird. Es kommt aber zu keiner Parallelverarbeitung, denn die mit (join) gekennzeichnete Transition feuert erst, wenn in beiden Eingangsstellen je eine Marke vorhanden ist, d.h. wenn die Transition `connect` ausgeführt wurde. Somit werden die beiden Threads an dieser (join)-Transition wieder synchronisiert.

Die Bezeichnungen (start) und (join) in dieser Grafik dienen nur dem besseren Verständnis. Die so annotierten Transitionen haben in Wirklichkeit die Beschriftung τ, repräsentieren also keine Zeichen des Alphabets.

4.3.3 Entscheidbarkeit

Bezüglich der Entscheidbarkeit von Petri-Netz-Sprachen gelten zwei Eigenschaften, zu denen in der Literatur auch Beweise existieren:

- Das Wortproblem ist für Petri-Netz-Sprachen entscheidbar[7].
- Das Gleichheitsproblem ist jedoch für Petri-Netz-Sprachen unentscheidbar[8].

4.4 Sequentielle Darstellung

Wie in Abschnitt 4.3 gezeigt wurde, kann aus einem theoretischen Modell eines nebenläufigen Systems eine Linearisierung berechnet werden, d.h. die Menge aller möglichen Interleaving-Sequenzen. Bei der dynamischen Analyse eines realen Systems soll im Rahmen dieser Diplomarbeit eine Annäherung an die mögliche Linearisierung realisiert werden. Idealerweise liefert die Annäherung eine Teilmenge der tatsächlich möglichen Sequenzen, genauso wie ein dynamischer Objektprozessgraph eine Teilmenge der Spuren eines statischen Objektprozessgraphen repräsentiert. Durch Gewinnung von mehreren Interleaving-Sequenzen soll daher versucht werden, näherungsweise Informationen über das nebenläufige Verhalten des Systems zu erhalten.

Die Menge der gültigen Sequenzen soll dabei weiterhin als reguläre Sprache und nicht als Petri-Netz-Sprache dargestellt werden. Dafür sprechen unter anderem die folgenden Gründe:

Protokollvalidierung Die Möglichkeiten der Protokollvalidierung lassen sich nur bei Verwendung regulärer Sprachen voll ausschöpfen, da hier ein Gleichheitstest durch Konstruktion des minimalen DEA durchführbar ist. Für Petri-Netz-Sprachen ist zwar – ebenso wie für kontextfreie Sprachen – das Wortproblem entscheidbar, nicht jedoch die Sprachgleichheit. Ein Vergleich solcher Sprachen wäre daher z.B. durch ein heuristisches Verfahren möglich, wie es von Lo et al. [LK06] genutzt wird.

Zum Vergleich zweier Sprachen L_1, L_2 werden dort mithilfe der Repräsentation von L_1 zufällige Wörter generiert, für die getestet wird, ob sie auch in L_2 enthalten sind. Dieser Test ist möglich, wenn das Wortproblem für L_2 entscheidbar ist. Der Prozentsatz zufälliger Wörter aus L_1, die auch Element von L_2 sind, wird als *Präzision* von L_1 bezüglich L_2 bezeichnet. Vertauscht man die Rollen der beiden Sprachen, so erhält man den *Recall* von L_1 bezüglich

[7] vgl. [Wim08] S. 134
[8] vgl. [PW02] S. 192

L_2, d.h. den Anteil zufälliger Wörter aus L_2, die auch zu L_1 gehören. Beide Werte können gemeinsam als Maß für die Ähnlichkeit der Sprachen betrachtet werden.

Diese Art des Protokollvergleichs kann jedoch prinzipbedingt nur näherungsweise Informationen über die Ähnlichkeit bzw. Gleichheit der Protokolle liefern.

Kausalitäten Ein Petri-Netz beinhaltet Informationen über die Synchronisation der nebenläufigen Prozesse, welche die möglichen Interleaving-Sequenzen einschränken. Die tatsächlichen kausalen Abhängigkeiten zwischen Programmanweisungen in verschiedenen Threads sind jedoch bei der Protokollanalyse nicht ohne weiteres bekannt. Sie können schließlich sowohl durch explizite Synchronisationsmaßnahmen des Programmierers als auch durch bestimmte Abhängigkeiten z.B. bei der Ein- und Ausgabe unter Einbeziehung des Betriebssystems zustande kommen.

4.5 Interleaving und Objektprozessgraphen

Die besondere Herausforderung besteht nun darin, die in der dynamischen Analyse gewonnenen Interleaving-Sequenzen in das Konzept der OPG-basierten Protokollerkennung zu integrieren. Andere, PTA-basierte Protokollerkennungsverfahren (siehe Abschnitt 3.4, S. 27) nutzen nur Informationen über die Sequenz der Atomic Calls. Ob diese aus einer Single-Thread-Anwendung stammt oder durch Interleaving einer Multithreading-Anwendung entstanden ist, spielt für die weitere Analyse keine Rolle. Die Erkennung von Schleifen im Protokoll ist bei diesen Verfahren jedoch allein durch Generalisierung der vorhandenen, endlichen Sequenzen möglich.

Im Gegensatz dazu hat das OPG-basierte Verfahren den Vorteil, dass wiederkehrende Code-Stellen erkannt werden und so automatisch Schleifen im OPG entstehen, die sich auch nach den weiteren Verarbeitungsschritten im Protokoll wiederfinden. Bei Single-Thread-Anwendungen bedeutet dies, dass ein OPG den möglichen Kontrollfluss über die relevanten Teile der *syntaktischen* Struktur der Anwendung direkt repräsentiert.

Besonders wichtig sind hierbei die Informationen über Schleifen und Funktionsaufrufe. Für beide Konzepte gelten gewisse Einschränkungen bezüglich des Kontrollflusses. So hat die Sprunganweisung einer Schleife einen Knoten innerhalb derselben Funktion zum Ziel. Eine Funktion wiederum wird an genau einem Knoten betreten (Entry) und über maximal zwei Knoten (Return, Exceptional_Return) wieder verlassen. Nach dem Verlassen einer Funktion kehrt die Ausführung an den Knoten zurück, von dem aus die Funktion betreten wurde (Call).

Die dadurch möglichen Kontrollflüsse lassen sich dynamisch und statisch explizit an der Struktur des Sourcecodes ablesen bzw. vom Programmierer festlegen. Wird eine Funktion während der Laufzeit mehrfach aufgerufen, so wird der innerhalb der Funktion durchlaufene Pfad bei jedem Aufruf zu demselben Teilgraphen des OPG hinzugefügt. Ein solcher Funktions-Teilgraph (vom Entry-Knoten bis zum Return- oder Exceptional_Return-Knoten) repräsentiert somit im fertigen OPG alle dynamisch erkannten Pfade innerhalb der Funktion.

Das Ergebnis einer dynamischen Analyse von Interleaving-Sequenzen hängt dagegen wesentlich vom Verhalten des Schedulers während der Programmausführung ab. Genauer gesagt definiert der Scheduler die jeweilige Linearisierung des nebenläufigen Programms. Eine explizite Beeinflussung des Schedulings ist dabei nur durch Synchronisationsmaßnahmen möglich. Falls solche nicht erforderlich sind, wird ein Programmierer in der Regel auch darauf verzichten. So wird darüber hinaus sogar die automatisierte Beseitigung unnötiger Synchronisation angestrebt, da diese zu Lasten der Performanz geht. Ein entsprechendes Verfahren wird z.B.

von Bogda und Hölzle [BH99] vorgestellt. Außerdem spielt der tatsächliche Ablauf des Schedulings für das nach außen sichtbare Verhalten der Anwendung auch keine Rolle.

Gerade diese Informationen (d.h. die möglichen Interleaving-Sequenzen) sollen nun im Protokoll repräsentiert werden. Die Überlegungen zeigen aber, dass mit dem OPG und den Interleaving-Sequenzen zwei Konzepte aufeinander treffen, die sich nicht ohne weiteres miteinander vereinbaren lassen.

4.6 Verwandte Forschung

Die in dieser Diplomarbeit entwickelten Erweiterungen beschränken sich, ebenso wie das vorhandene Protokollerkennungsverfahren, auf die dynamische Analyse eines Programms durch Aufzeichnung einer Spur. Es wird also nicht gesondert berücksichtigt, wenn eine explizite Kommunikation bzw. Synchronisation zwischen Threads stattfindet, die das Zeitverhalten und damit das Interleaving-Verhalten beeinflussen kann. Als Beispiele für derartige Analyseverfahren werden die folgenden zwei Artikel vorgestellt.

Choi et al. [CLL+02] führen das Konzept des *Interthread Control Flow Graph* (ICFG) ein. Es wird versucht, unter anderem auf Basis der im Programm vorkommenden Anweisungen `start()`, `join()` und `synchronized` entsprechende kausale Abhängigkeiten zu finden mit dem Ziel, mögliche Race Conditions zu erkennen, d.h. Situationen, in denen ein bestimmtes Verarbeitungsergebnis (unbeabsichtigt) vom zeitlichen Verhalten nebenläufiger Prozesse abhängt. Die von Choi et al. gewählte Darstellung kommt der intuitiven Auffassung eines Programmierers sicherlich recht nahe, da sie das Konzept der Nebenläufigkeit möglichst beibehält. Die Transformation in einen endlichen Automaten würde aber auch hier eine Linearisierung bedeuten.

Von Boroday et al. [BPSH05] wird ein dynamisches Analyseverfahren vorgestellt, welches Protokollverletzungen erkennen soll, die auf Thread-Objekten selbst begangen werden. Auch dort wird explizit betrachtet, ob und in welcher Reihenfolge Methoden wie `start()`, `run()` oder `join()` betreten und verlassen werden.

KAPITEL 5

Erweiterung der Protokollerkennung

In diesem Kapitel werden die verschiedenen Ideen und Lösungsansätze, die im Rahmen der Diplomarbeit selbst entwickelt wurden, beschrieben und einer kritische Betrachtung unterzogen. Alle Überlegungen hatten zum Ziel, die Protokollerkennung so zu erweitern, dass auch die Analyse von Multithreading-Anwendungen ein nützliches Ergebnis liefert.

Inhalt

5.1 ITS-Graphen . . . 37
5.1.1 Eigenschaften . . . 38
5.1.2 Repräsentation im RFG . . . 38
5.1.3 Konstruktionsmethode ITS-Online . . . 39
5.1.4 Konstruktionsmethode ITS-Offline . . . 42
5.1.5 Auswahl eines Konstruktionsverfahrens . . . 46
5.1.6 Beispiel . . . 47
5.2 ITS-Protokolle . . . 48
5.2.1 Protokollextraktion aus ITS-Pfaden . . . 48
5.2.2 Unzulässige Sequenzen . . . 49
5.3 Shuffle-Sprachen . . . 51
5.3.1 Definition . . . 51
5.3.2 Eigenschaften . . . 51
5.3.3 Konstruktion eines Shuffle-Automaten . . . 52
5.3.4 Schnitt-Protokoll . . . 53
5.4 Anwendung an einem Beispiel . . . 54
5.4.1 Code . . . 55
5.4.2 Protokollerkennung mit dem bisherigen Verfahren . . . 56
5.4.3 Protokollerkennung mit dem ITS-Shuffle-Schnitt-Verfahren . . . 56
5.5 Zusammenfassung . . . 60

5.1 ITS-Graphen

Im Rahmen dieser Diplomarbeit sind mehrere Lösungsansätze zur sequentiellen Darstellung des nebenläufigen Verhaltens einer Anwendung enstanden. Allen Ansätzen gemein ist, dass sie auf der Konstruktion und Analyse eines so genannten *Inter-Thread-Sequence-Graphen* (ITS-Graph) basieren. In diesem Abschnitt wird zunächst eines allgemeine Beschreibung dieses Konzeptes vorgestellt. Anschließend wird für zwei verschiedene Ansätze beschrieben, wie ein ITS-Graph tatsächlich konstruiert werden kann.

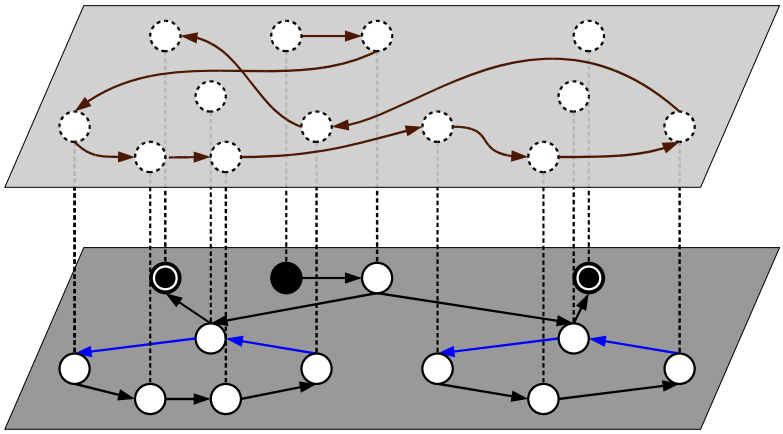


Abbildung 5.1: Schematische Darstellung eines OPG mit zwei Threads und Überlagerung durch einen ITS-Graphen mit $N_{ITS} \neq N$.

5.1.1 Eigenschaften

Unter einem ITS-Graphen kann man sich eine Art Überlagerung des OPG durch einen zweiten Graphen vorstellen, der dieselbe Knotenmenge besitzt. Es gelten die folgenden grundlegenden Eigenschaften:

- Die ITS-Kanten bilden denjenigen Pfad auf den Knoten des OPG, welcher die tatsächliche globale, zeitliche Ausführungsreihenfolge der Knoten über Threads hinweg darstellt.
- Die Menge der ITS-Kanten ist $E_{ITS} \subseteq N \times N$.
- Nicht alle tatsächlich besuchten Knoten liegen auf dem ITS-Pfad. Je nach Ansatz wird eine Teilmenge von Knoten $N_{ITS} \subseteq N$ definiert, so dass $E_{ITS} \subseteq N_{ITS} \times N_{ITS}$. Knoten aus $N - N_{ITS}$ werden beim Erzeugen der ITS-Kanten übersprungen.
- Das System (N_{ITS}, E_{ITS}) heißt *ITS-Graph* für den OPG (N, E).
- Die Semantik einer ITS-Kante ist wie folgt definiert: Existiert eine ITS-Kante $(s, t) \in E_{ITS}$, so war der Knoten t mindestens einmal im Programmlauf der direkte zeitliche Nachfolger von s, d.h. kein Knoten aus N_{ITS} wurde zwischendurch besucht.

Abbildung 5.1 zeigt schematisch eines OPG mit zwei Threads, der durch einen ITS-Graphen überlagert wird. Dabei gilt $N_{ITS} \neq N$, so dass einige Knoten übersprungen werden.

In diesem Kapitel werden mehrere Lösungsansätze vorgestellt, die alle auf der Konstruktion und Verwendung eines ITS-Graphen basieren. Die Ansätze unterscheiden sich jedoch zum einen in der Definition von N_{ITS}, zum anderen in der Wahl des Verarbeitungsschrittes der dynamischen Analyse, bei dem die Konstruktion von E_{ITS} erfolgt.

5.1.2 Repräsentation im RFG

Für die Implementierung muss das Konzept des ITS-Graphen innerhalb des RFG umgesetzt werden. Dazu wird ein neuer Kantentyp $\mathsf{Inter_Thread_Sequence} \in T_E$ eingeführt. Dadurch kann

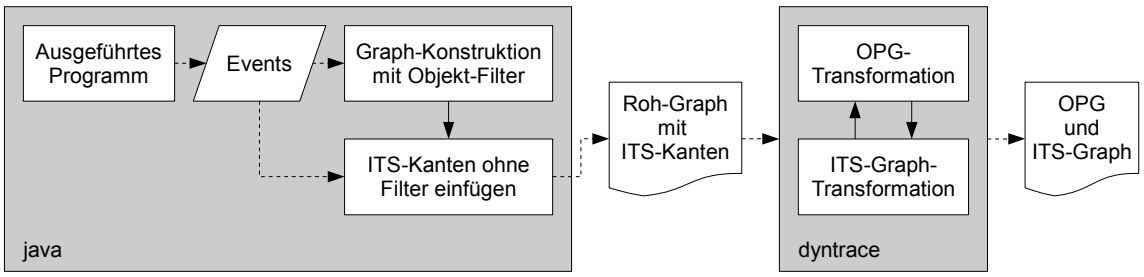


Abbildung 5.2: Datenfluss bei Verwendung des ITS-Online-Verfahrens

der ITS-Graph direkt als Überlagerung des OPG repräsentiert werden. Die ITS-Kanten verlaufen dabei zwischen den vorhandenen Knoten des OPG. Da der RFG ein Multigraph ist, also Mehrfachkanten erlaubt, kann eine ITS-Kante $e' \in E_{ITS}$ auch parallel zu einer OPG-Kante $e \in E$ verlaufen, wobei eine Unterscheidung der Kanten anhand ihrer Typen erfolgt.

Der ITS-Graph (N_{ITS}, E_{ITS}) ist auf diese Weise also nur implizit repräsentiert. Alle weiteren grafischen Darstellungen in dieser Diplomarbeit zeigen ebenfalls immer die Überlagerung von OPG und ITS-Graph. Die ITS-Kanten werden dabei gestrichelt gezeichnet.

5.1.3 Konstruktionsmethode ITS-Online

Der ITS-Online-Ansatz stellt eine Erweiterung des Online-Ansatzes zur dynamischen OPG-Erzeugung dar. Während in der bisherigen Implementierung die einzelnen Threads im Roh-Graphen separat dargestellt werden, kommen nun – ebenfalls online – Informationen über die zeitliche Verzahnung der Threads in Form eines „ITS-Roh-Graphen“ hinzu (der Roh-Graph der bisherigen Implementierung wird in diesem Abschnitt zur besseren Unterscheidung als „OPG-Roh-Graph“ bezeichnet). Dieser ITS-Roh-Graph ist zunächst nicht als Überlagerung des OPG-Roh-Graphen zu sehen, da er auch Knoten enthält, die nicht Teil des OPG-Roh-Graphen sind. Erst der fertige ITS-Graph verwendet tatsächlich eine Teilmenge der Knoten des OPG.

Der für das ITS-Online-Verfahren modifizierte Datenfluss ist in Abbildung 5.2 dargestellt. Die neu hinzugekommenen Schritte werden im Folgenden erläutert.

5.1.3.1 Erzeugung des Roh-ITS-Graphen

Jedes Mal, wenn eine Kante im OPG-Roh-Graph erzeugt bzw. traversiert wird, wird auch im ITS-Roh-Graph eine Kante zum aktuellen Knoten eingefügt, falls dort noch keine mit dem entsprechenden Knotenpaar inzidente ITS-Kante vorhanden ist.

Globaler vorheriger Knoten Ein Unterschied besteht im Quellknoten der beiden Kanten. Während die OPG-Kante von dem Knoten ausgeht, der in dem jeweiligen Thread zuletzt ausgeführt wurde (vorheriger Knoten), gilt für die ITS-Kante ein globaler, Thread-übergreifender vorheriger Knoten als Quelle.

Irrelevante Knoten Ein weiterer Unterschied betrifft diejenigen Knoten, die für den OPG irrelevant sind. Während bei der Konstruktion des OPG-Roh-Graphen ein Filterungsverfahren dafür sorgt, dass nur relevante Methodenaufrufe im OPG sichtbar sind, werden für den

Roh-ITS-Graphen sämtliche Knoten verwendet. Der Grund dafür ist, dass das Filterungsverfahren ebenfalls „online“ arbeitet und jeweils direkt nach Verlassen einer Methode entscheidet, ob die innerhalb der Methode besuchten Knoten für den Roh-Graphen relevant sind oder nicht. Sind sie nicht relevant, werden sie gar nicht in den OPG-Roh-Graphen aufgenommen. Welche Knoten auch für den ITS-Graphen irrelevant sind, kann zu diesem Zeitpunkt noch nicht entschieden werden. Alle für den OPG-Roh-Graphen irrelevanten Knoten und die inzidenten ITS-Kanten existieren somit auch im fertigen ITS-Roh-Graphen. Erst durch die anschließende Transformation werden diese Knoten ebenfalls eliminiert.

5.1.3.2 Transformation

Da das Erzeugen des ITS-Roh-Graphen parallel zum OPG-Roh-Graphen erfolgt, muss in den anschließenden Transformationen zu einem OPG stets auch der ITS-Graph berücksichtigt werden. Insbesondere das Löschen von Knoten aus dem OPG-Roh-Graphen ist nicht ohne weiteres möglich, wenn diese Teil des ITS-Roh-Graphen sind. Das Entfernen irrelevanter bzw. im OPG nicht zulässiger Knoten ist jedoch das wesentliche Ziel der Graph-Transformationen. Daher werden die Transformationen entsprechend angepasst. Außerdem wird speziell für die Verarbeitung von ITS-Kanten und inzidenten Knoten ein neuer Verarbeitungsschritt eingeführt, die *ITS-Transformation*.

Im Gegensatz zur bisherigen Implementierung gelten folgende Änderungen:

1. Ein zu entfernender Knoten wird von den vorhandenen Transformationen nicht mehr gelöscht. Es werden nur die inzidenten OPG-Kanten (aus E) durchverbunden. ITS-Kanten werden nicht modifiziert.

2. Die neu hinzugekommene ITS-Transformation entfernt alle Knoten, die ausschließlich mit ITS-Kanten inzident sind. Diese Kanten werden anschließend entsprechend durchverbunden.

Die ITS-Transformation verbindet also Vorgänger- und Nachfolgerknoten von gelöschten Knoten mit entsprechenden neuen ITS-Kanten. Die Information über die vorhandenen Pfade im ITS-Graph bleibt dadurch erhalten.

Von der ITS-Transformation sind diejenigen Knoten betroffen, die von den existierenden Transformationen als irrelevant erkannt wurden, außerdem alle Knoten, die im bereits im OPG-Roh-Graphen aufgrund der Objekt-Filterung nicht enthalten waren.

Abbildung 5.3 zeigt beispielhaft, wie das Entfernen eines Knotens in zwei Schritten abläuft. In diesem Beispiel wird die *Label-Transformation* angewandt, die diejenigen Label-Knoten entfernt, welche genau einen Nachfolger besitzen. Diese Label-Knoten sind hier als kleine schwarze Punkte dargestellt. Wie in (b) zu sehen ist, werden die Knoten durch die angepasste Label-Transformation nicht entfernt. Vielmehr werden nur die Kanten aus E gelöscht und durch eine durchgehende ersetzt. Erst nach Anwendung der ITS-Transformation ergibt sich ein Graph wie in (c).

5.1.3.3 Interleaving von Methodenrümpfen

Da das Scheduling nicht explizit beeinflussbar ist, muss davon ausgegangen werden, dass die Thread-Wechsel an zufälligen Zeitpunkten stattfinden. Diese Zeitpunkte sind auch nicht von der Kontrollstruktur innerhalb eines Threads abhängig. Wird also eine Methode mehrere Male auf demselben Pfad durchlaufen, so können die Thread-Wechsel dennoch jedes Mal

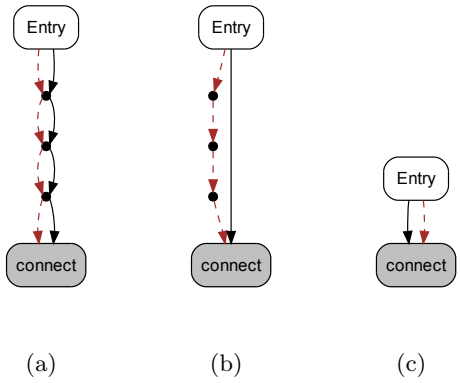


Abbildung 5.3: Ausschnitt aus einem Roh-Graphen (a), nach der Label-Transformation (b) und nach der ITS-Transformation (c).

zu anderen Zeitpunkten auftreten. Der daraus resultierende ITS-Graph enthält somit nicht nur sämtliche mögliche Pfade innerhalb der Funktion, sondern zusätzlich alle aufgetretenen Sprünge an Sourcecode-Positionen in anderen Threads. Diese können von einem bestimmten Systemzustand abhängen, der von der Methode selbst gar nicht beeinflussbar ist, sich jedoch von Aufruf zu Aufruf unterscheiden kann.

Nicht zuletzt kann sogar eine Methode nebenläufig zu sich selbst ausgeführt werden. Dann entstehen intraprozedurale ITS-Kanten, welche Sprünge zwischen Sourcecode-Position innerhalb der Funktion darstellen.

Versucht man, einen mithilfe des ITS-Online-Verfahrens konstruierten ITS-Graphen zu einem Protokollautomaten zu verarbeiten, so ergeben sich weitere Probleme. Ebenso wie bei einem OPG muss auch auf den ITS-Graphen zunächst die Rekursionsauflösung angewandt werden. Durch Kopieren von Methodenrümpfen im Rahmen der Inline-Expansion entstehen dabei weitere ITS-Kanten, die im tatsächlichen Programmlauf niemals traversiert wurden.

5.1.3.4 Pfade über irrelevante Knoten

Da die Konstruktion des ITS-Graphen prinzipbedingt ohne Anwendung des Objekt-Filters abläuft, enthält der Roh-ITS-Graph zunächst sämtliche Knoten des Kontrollflussgraphen. Diejenigen Knoten, die irrelevant und somit nicht Teil des OPG sind, sind auch ausschließlich mit ITS-Kanten inzident. Diese Knoten werden durch die ITS-Transformation entfernt. Dadurch befinden sich im fertigen ITS-Graphen auch keine irrelevanten Knoten.

Falls jedoch ein irrelevanter Knoten in einem ebenfalls irrelevanten Thread liegt, kann es zu unerwünschten Effekten kommen. Wenn nämlich der irrelevante Thread nebenläufig zu einem relevanten Thread ausgeführt wird, entstehen im ITS-Roh-Graphen Kanten zwischen Knoten des irrelevanten und des relevanten Threads. Daraus ergeben sich Pfade zwischen relevanten Knoten, die über die Knoten des irrelevanten Threads verlaufen. Werden die nun irrelevanten Knoten durch die ITS-Transformation entfernt, so bleiben die Pfade im Graph erhalten. Deren Herkunft ist dann nicht offensichtlich nachvollziehbar, da die Existenz des irrelevanten Threads im OPG gar nicht mehr repräsentiert ist.

Zur Umgehung dieser Schwierigkeiten wurde deshalb das ITS-Offline-Verfahren eingeführt, welches im folgenden Abschnitt vorgestellt wird.

5.1.4 Konstruktionsmethode ITS-Offline

Der Roh-Graph enthält während und nach seiner Erstellung eine große Anzahl irrelevanter Knoten. Welche Knoten tatsächlich relevant sind, ergibt sich erst durch die Transformationen, die aus einem Roh-Graphen einen OPG erzeugen. Da auch nur die relevanten Knoten im ITS-Graphen enthalten sein sollen, bietet es sich an, die Konstruktion des ITS-Graphen auf einen Zeitpunkt nach Fertigstellung des OPG zu verschieben. Auf diese Weise entstehen auch keine Pfade über irrelevante Knoten.

Außerdem bietet dieser Ansatz die Möglichkeit, das in Abschnitt 5.1.3.3 erläuterte Problem des Interleaving von Funktionsrümpfen zu umgehen.

5.1.4.1 Aufzeichnung einer globalen Spur

Durch das Konzept der Online-Konstruktion sind die für den Aufbau des ITS-Graphen notwendigen Informationen bereits mit Fertigstellung des Roh-Graphen nicht mehr vorhanden. Daher ist es erforderlich, die Information in Form einer *globalen Spur* der besuchten Knoten des Kontrollflussgraphen in einer temporären Datei aufzuzeichnen. Dadurch verzichtet man auf einen der Vorteile der Online-Konstruktion des OPG. Dementsprechend spricht man hier auch von der *Offline-Konstruktion* des ITS-Graphen.

Die globale Spur ist eine Folge von Einträgen der Form

$$(\mathit{threadID}, \mathit{nodeID}, \mathit{objAddr}) \in \mathbb{N} \times \mathit{String} \times \mathit{String}$$

Diese werden genau in der chronologischen Reihenfolge aufgezeichnet, in der sie während des Programmlaufs aufgetreten sind. Dabei verweist *nodeID* auf denjenigen Knoten im Roh-Graphen, der in diesem Moment vom Thread mit der ID *threadID* besucht worden ist.

Die Aufzeichnung der globalen Spur erfolgt – im Gegensatz zur Roh-Graph-Konstruktion – ohne Anwendung eines Objekt-Filters. Sobald in einem Programm mehrere Instanzen einer zu analysierenden Klasse auftreten, könnten die atomaren Aufrufe nicht mehr eindeutig einem Objekt zugeordnet werden. Damit diese Zuordnung beim Verarbeiten der globalen Spur noch nachvollzogen werden kann, erhalten Einträge für Create- und Atomic-Call-Knoten zusätzlich ein Attribut *objAddr*, welches die Adresse des von dem aktuell besuchten Knoten betroffenen Objektes enthält. Ist das Attribut *objAddr* nicht gegeben, so wird *objAddr* = **null** definiert.

Online- vs. Offline-OPG-Erzeugung Es stellt sich die Frage, weshalb der OPG überhaupt noch mithilfe des Online-Verfahrens konstruiert wird, wenn dessen Vorteile durch die Verwendung der globalen Spur wieder zunichte gemacht werden. Die globale Spur benötigt schließlich nur unwesentlich weniger Speicherplatz als eine Spur, wie sie für die Offline-Erzeugung eines OPG verwendet wird. Zumindest zwei Gründe sprechen für die Integration in das Online-Verfahren:

- Bereits aus dem fertigen Roh-Graphen lässt sich entnehmen, ob das zu analysierende Objekt überhaupt von mehr als einem Thread verwendet wurde. Ist dies nicht der Fall, so kann bezüglich der globalen Spur zumindest auf deren Auswertung verzichtet werden, da sie das gleiche Ergebnis liefern würde wie eine reine OPG-basierte Protokollanalyse.
- Bei einer Integration in das Offline-Verfahren müsste das Dateiformat der Offline-Spuren angepasst werden, was auch eine Anpassung des Objekt-Filters bedeuten würde. Die Filterung nach den relevanten Methodenaufrufen unter Beachtung des jeweiligen Threads benötigt einige Datenstrukturen, welche entsprechend verwaltet werden

müssen. Diese Art der Filterung ist im Online-Verfahren bereits implementiert. Somit wäre es ein wesentlich höherer Aufwand, den Objekt-Filter des Offline-Verfahrens anzupassen. Die Untersuchung, inwiefern das in dieser Diplomarbeit entwickelte Konzept nützlich ist, wird ebenso gut durch Verwendung des Online-Verfahrens ermöglicht.

5.1.4.2 Integration der globalen Spur

Die Ereignisse der globalen Spur sollen nun mit den vorhandenen Knoten im OPG abgeglichen werden, um einen ITS-Graph zu konstruieren. Anhand der globalen Spur soll also der Programmablauf im OPG nachvollzogen werden. Dabei ergibt sich durch Mitführen von Kontext-Informationen für jeden Thread stets ein eindeutiger Nachfolgerknoten über die Knoten-ID aus der globalen Spur. In welchem Verarbeitungsschritt des OPG die Integration der globalen Spur am sinnvollsten ist, beantwortet die folgende Überlegung.

Aufrufkontext einer Methode Ein OPG enthält zwar keine irrelevanten Knoten, dafür liegt aber der Rumpf einer jeden Methode im gesamten OPG nur einmal vor. Wird die Methode von verschiedenen Aufrufpositionen (Call-Knoten) aus aufgerufen, so verweisen die Invoke-Kanten dieser Call-Knoten alle auf denselben Entry-Knoten. Dadurch fehlt im Allgemeinen der jeweilige Aufrufkontext der Methode. Würde man die globale Spur also in den OPG integrieren, so würden sich bei mehrfachem Aufruf der Methode die einzelnen Sequenzen der globalen Spur im Rumpf der Methode überlagern, wodurch es zum Entstehen unzulässiger Pfade käme (siehe Abschnitt 5.1.3.4, S. 41).

Es wird also eine Darstellung benötigt, in der der Rumpf einer Methode stets im jeweiligen Kontext betrachtet werden kann. Dies erreicht man gerade durch die Inline-Expansion der Methodenaufrufe, welche im Zusammenhang mit der Rekursionsauflösung angewandt wird, genauer gesagt im ersten Schritt des bisherigen Tools `remove_recursion` (siehe Abschnitt 3.2.1).

Durch das Kopieren von Knoten bei der Inline-Expansion existieren anschließend im expandierten OPG auch unterschiedliche Knoten mit derselben Knoten-ID. Durch den Kontext eines Threads ist jedoch nach wie vor stets eindeutig, auf welchen Knoten der Eintrag aus der globalen Spur verweist. Außerdem lässt sich jede Kopie eines Methodenrumpfes (und somit jeder Knoten darin) eindeutig dem Thread zuordnen, der die Methode in einem bestimmten Kontext aufgerufen hat.

Deaktivierung der Projektion Da die Integration der globalen Spur erst anhand des expandierten OPG erfolgt, liegen zum Zeitpunkt der Transformationen des Roh-Graphen zum OPG noch keinerlei Informationen über das nebenläufige Verhalten der Anwendung vor. Die Projektions-Transformation entfernt daher die Teilgraphen für alle Threads bis auf denjenigen, in dem die Erzeugung des betrachteten Objekts stattgefunden hat (siehe S. 26). Somit muss für das ITS-Offline-Verfahren die Projektion bei der Konstruktion des OPG deaktiviert werden.

Anpassung der Werkzeuge Das existierende Tool `remove_recursion` erledigt neben der Expansion und der Rekursionsauflösung gleich einen weiteren Schritt, nämlich die Beseitigung aller Knoten, die keine atomaren Aufrufe darstellen. Das Ergebnis, der Protokollgraph, enthält für die Integration der globalen Spur allerdings wieder zu wenig Informationen, da dort die Eindeutigkeit einer Knoten-ID nicht mehr gewährleistet ist.

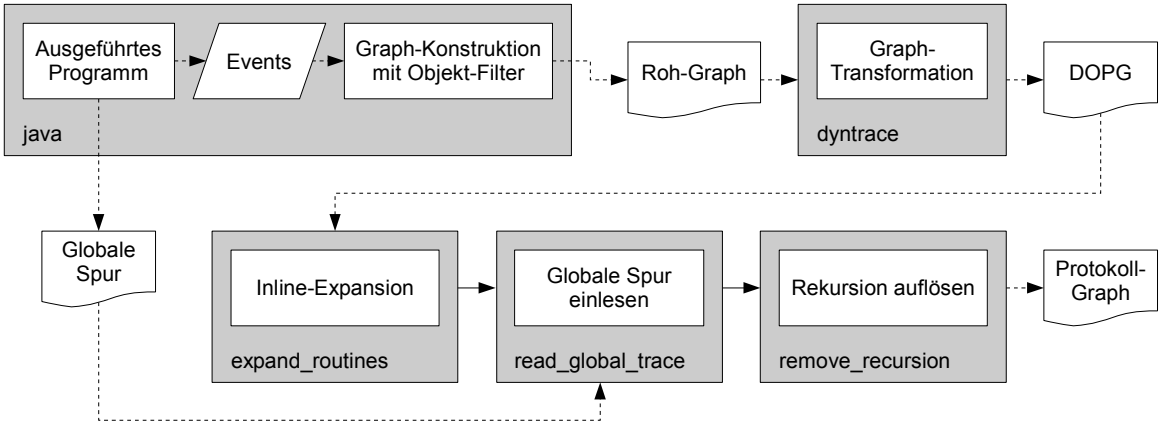


Abbildung 5.4: Datenfluss bei Vewendung des ITS-Offline-Verfahrens

Somit ist es erforderlich, das Tool `remove_recursion` in zwei Teile zu zerlegen. Der erste Schritt, die Inline-Expansion, wird vom neu eingeführten Tool `expand_routines` ausgeführt. Das Aufspalten der Call-Knoten sowie die Beseitigung der Knoten, die keine atomaren Aufrufe darstellen, wird dagegen weiterhin vom Tool `remove_recursion` erledigt.

Dadurch ergibt sich die Möglichkeit, ein weiteres neues Tool `read_global_trace` in die Verarbeitungskette zwischen die eben genannten einzufügen. Dieses Tool erzeugt aus der globalen Spur und dem expandierten OPG einen ITS-Graphen. Ob der Protokollgraph schließlich aus dem expandierten OPG oder aus dem ITS-Graph erzeugt werden soll, lässt sich beim Aufruf des Tools `remove_recursion` einstellen. Je nach Auswahl werden aus dem RFG die nicht benötigen Kanten gelöscht, d.h. entweder sämtliche OPG-Kanten oder sämtliche ITS-Kanten.

Daraus ergibt sich ein Datenfluss, wie er in Abbildung 5.4 dargestellt wird.

Algorithmus 5.1: `read_global_trace`

Input: recursionless OPG (N, E) with address *theObj* of analysed object
Input: $N_{ITS} \subseteq N$; global trace beginning with the creation of the object
Output: ITS graph (N_{ITS}, E_{ITS})

1 **foreach** $threadID \in \mathbb{N}$ **do**
2 $\quad lastNode(threadID) \leftarrow$ the initial entry node of the OPG;
3 $lastTargetNode \leftarrow$ the initial entry node of the OPG;
4 $E_{ITS} \leftarrow \{(\text{the Start node of the OPG}, lastTargetNode)\}$;
5 **foreach** element $(threadID, nodeID, objAddr)$ of the global trace **do**
6 $\quad$ **if** $\exists\, node \in N : (lastNode(threadID), node) \in E \wedge id(node) = nodeID$ **then**
7 $\quad\quad$ **if** $node \in N_{ITS}$ **then**
8 $\quad\quad\quad$ **if** $objAddr = \mathbf{null} \vee objAddr = theObj$ **then**
9 $\quad\quad\quad\quad E_{ITS} \leftarrow E_{ITS} \cup \{(lastTargetNode, node)\}$;
10 $\quad\quad\quad\quad lastTargetNode \leftarrow node$;
11 $\quad\quad lastNode(threadID) \leftarrow node$;

Algorithmus Das Tool `read_global_trace` verwendet für die Offline-Konstruktion des ITS-Graphen den in Pseudocode beschriebenen Algorithmus 5.1. Für dessen Entwurf folgt

nun eine Erläuterung der entscheidenden Zusammenhänge.

Der Algorithmus verwaltet zwei Variablen zur Zwischenspeicherung.

- Die Abbildung $lastNode : \mathbb{N} \rightarrow N$ speichert für jeden Thread, welcher Knoten des OPG für diesen Thread zuletzt besucht wurde. Es muss sich dabei nicht um einen Knoten des ITS-Graphen handeln. Die Abbildung wird benötigt, um die globale Spur im OPG nachzuvollziehen. Zu Beginn wird der $lastNode$ eines jeden Threads mit dem Initial-Entry-Knoten des OPG initialisiert.
- Die Variable $lastTargetNode \in N_{ITS}$ hingegen enthält immer den Zielknoten der zuletzt eingefügten ITS-Kante. Dieser Knoten wird dadurch zum Quellknoten der nächsten ITS-Kante. Die Variable ist also für die Konstruktion des ITS-Graphen erforderlich.

Die folgenden vorbereitenden Maßnahmen sind vor Beginn des eigentlichen Algorithmus notwendig:

- Der Anfang der globalen Spur wird soweit verworfen, bis der Create-Knoten des zu analysierenden Objektes besucht wird. Die erste ITS-Kante verläuft somit direkt vom Initial-Entry-Knoten des OPG zum Create-Knoten. Diese Vorgehensweise ersetzt die Projektion des OPG, die für das ITS-Offline-Verfahren deaktiviert werden musste (siehe S. 43).
- Da die globale Spur keinen Objekt-Filter durchläuft, muss während ihrer Auswertung beachtet werden, ob ein atomarer Aufruf zu dem zu analysierenden Objekt gehört oder nicht. Dies ist notwendig, obwohl der OPG nur die atomaren Aufrufe für das zu analysierende Objekt enthält.

  Falls nämlich ein weiteres Objekt derselben Klasse existiert, enthält die globale Spur die atomaren Aufrufe für beide Objekte. Wenn zwei Aufrufe für verschiedene Objekte nun auch noch jeweils von derselben Stelle im Code aus erfolgt sind, besitzen die Call-Knoten dieselbe Knoten-ID. Dann kann nur anhand der Objekt-Adresse entschieden werden, um welches Objekt es sich jeweils handelt. Um diese Entscheidung zu ermöglichen, wird vorher die Speicheradresse des im gegebenen OPG dargestellten Objektes dem Create-Knoten des Objektes entnommen und in *theObj* hinterlegt.

Die Auswertung eines Eintrags (*threadID*, *nodeID*, *objAddr*) der globalen Spur erfolgt in drei Schritten. Der erste Schritt dient dazu, die globale Spur im OPG nachzuvollziehen. Die weiteren Schritte entscheiden, ob ein im ersten Schritt gefundener Knoten bei der Konstruktion des ITS-Graphen übersprungen werden soll oder nicht.

- Im ersten Schritt (Z. 6) wird festgestellt, ob der Knoten aus der globalen Spur überhaupt im OPG gefunden werden kann. Dazu wird zunächst bestimmt, welchen Knoten der entsprechende Thread *threadID* zuletzt besucht hat. Dann wird überprüft, ob dieser Knoten *lastNode*(*threadID*) einen Nachfolger *node* im OPG besitzt, dessen ID *id*(*node*) mit der Knoten-ID *nodeID* aus der globalen Spur übereinstimmt. Ist dies nicht der Fall, so handelt es sich bei dem Spur-Eintrag offenbar um die Aufzeichnung eines irrelevanten Knotens, der im OPG nicht mehr existiert. Der entsprechende Spur-Eintrag wird daraufhin ignoriert.

  Wurde hingegen ein Nachfolger *node* gefunden, so wurde für den gegebenen Thread der letzte Abschnitt der globalen Spur im OPG erfolgreich nachvollzogen. Der Knoten wird daraufhin in *lastNode*(*threadID*) hinterlegt (Z. 11) und somit fortan für die Bestimmung der nächsten potenziellen Nachfolger innerhalb des Threads verwendet.

- Im zweiten und dritten Schritt (Z. 7–8) wird überprüft, ob auch im ITS-Graphen eine Kante zu dem gefundenen Nachfolgerknoten *node* erzeugt werden soll. Zum einen muss der Knoten dazu Element von N_{ITS} sein, zum anderen darf die Objekt-Adresse *objAddr* des Spur-Eintrags (falls vorhanden) nicht von der des zu analysierenden Objektes abweichen. Sind diese Bedingungen erfüllt, wird eine ITS-Kante $(lastTargetNode, node)$ eingefügt und die Variable *lastTargetNode* entsprechend aktualisiert (Z. 9–10).

5.1.4.3 Definition der Knotenmenge

Es bleibt die Frage, welche Teilmenge $N_{ITS} \in N$ der Knoten sich am besten zur Konstruktion des ITS-Graphen eignet. So wurde mit verschiedenen Definitionen experimentiert. Folgende Überlegung beantwortet jedoch diese Frage. Es müssen vor allem drei Möglichkeiten für N_{ITS} in Betracht gezogen werden:

- $N_{ITS} = N$

  In diesem Fall wäre der ITS-Graph von maximaler Größe. Er würde aber auch redundante Informationen enthalten. So müssen Call-Knoten, die keine atomaren Aufrufe darstellen, nicht Teil des ITS-Graphen sein. Sie werden nach Erzeugen des ITS-Graphen ohnehin durch das Tool `remove_recursion` in zwei Knoten gespalten.

  Außerdem enthält eine ITS-Kante parallel zu einer Invoke-, Return- oder Exceptional_Return-Kante enthält keine zusätzliche Information. Der Call-Knoten wird nämlich bei der Erzeugung des Roh-Graphen nicht durch eine eigene Logging-Anweisung erstellt. Er wird vielmehr künstlich erzeugt, wenn ein Entry-, Return- oder Exceptional_Return-Ereignis auftritt. Das geschieht zum selben Zeitpunkt wie das Ereignis selbst, d.h. zwischen dem Call- und dem weiteren Knoten kann kein anderer Knoten besucht worden sein.

- $N_{ITS} = \{n \in N \mid type(n) = \mathsf{Call} \wedge n \text{ ist atomar}\}$

  Mit dieser Konfiguration enthält der ITS-Graph die minimale Knotenmenge, die zur Protokollerkennung notwendig ist. Das erkannte ITS-Protokoll ist allerdings zu speziell, denn der ITS-Graph enthält keine Information über die Stellen, an denen die Abbruchbedingung einer Schleife geprüft wird. Wird diese erstmalig vor Betreten der Schleife evaluiert (entsprechend einer `while`-Schleife), so wird die Schleife im OPG so dargestellt, dass die Abbruchbedingung auch bereits vor dem ersten Durchlauf erfüllt sein kann. Der ITS-Graph hingegen enthält eine direkte Kante zum ersten atomaren Aufruf innerhalb der Schleife, welche den Condition-Knoten mit der Abbruchbedingung überspringt. Dadurch enthält der ITS-Graph eine andere Semantik, welche besagt, dass die Schleife mindestens einmal betreten werden muss.

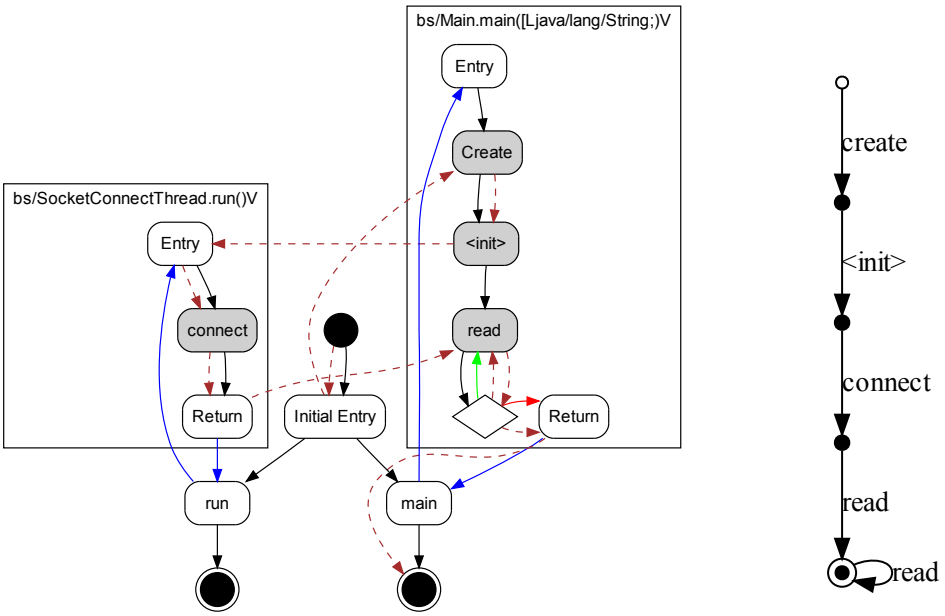
- $N_{ITS} = \{n \in N \mid type(n) \neq \mathsf{Call} \vee n \text{ ist atomar}\}$

  Diese Wahl von N_{ITS} hat sich als am sinnvollsten erwiesen. Sie vereint die Vorteile der eben beschriebenen Varianten. So enthält der ITS-Graph keine Redundanz, berücksichtigt aber alle möglichen bedingten Verzweigungen.

5.1.5 Auswahl eines Konstruktionsverfahrens

Nachdem die Konstruktionsmethoden ITS-Online und ITS-Offline vorgestellt wurden, sollen nun noch einmal die Vor- und Nachteile gegenübergestellt werden.

Da die Funktionsweise des ITS-Online-Verfahrens an die der DOPG-Online-Erzeugung angelehnt ist, zeichnet sie sich durch hohe Effizienz und einen geringen Speicherverbrauch aus. Auf



(a) ITS-Graph (b) ITS-Protokoll

Abbildung 5.5: Aus Listing 3.4 generierter ITS-Graph

diese Weise können die Vorteile der DOPG-Online-Erzeugung auch bei der Konstruktion des ITS-Graphen genutzt werden. Von Nachteil ist jedoch, dass sämtliche Tools und Dateiformate bis hin zum Protokollgraphen angepasst werden müssen. Dies betrifft das Roh-Graph-Format, die meisten OPG-Transformationen, das OPG-Format und die Rekursionsauflösung. Hinzu kommt die Problematik der Pfade über irrelevante Knoten.

Der ITS-Offline-Ansatz umgeht all diese Probleme. Es muss jedoch der hohe Speicherverbrauch durch die globale Spur in Kauf genommen werden. Daraufhin steht aber für jede Methode der Aufrufkontext zur Verfügung, es sind keine Änderungen an der OPG-Erzeugung notwendig und es steht vor Konstruktion des ITS-Graphen eindeutig fest, welche Knoten der globalen Spur relevant sind.

Aus diesen Gründen wird die Konstruktionsmethode ITS-Online nicht mehr weiter verfolgt. Die Methode ITS-Offline ist somit das Verfahren der Wahl zur Gewinnung eines ITS-Graphen.

5.1.6 Beispiel

Wendet man das ITS-Offline-Verfahren auf das Beispiel in Listing 3.4 an, so erhält man den ITS-Graphen in Abbildung 5.5 (a). Die Menge N_{ITS} enthält alle Knoten außer den nicht-atomaren Aufrufen. Somit werden die Call-Knoten für `main()` und `run()` übersprungen. Ebenfalls übersprungen wird der Pfad vom Initial-Entry-Knoten bis zum Create-Knoten des Objektes (siehe Abschnitt 5.1.4).

5.2 ITS-Protokolle

Aus einem ITS-Graph kann das Protokoll des analysierten Objektes gewonnen werden, indem auf den ITS-Graph dieselben Transformationen angewandt werden wie auf einen expandierten OPG in der bisherigen Implementierung (siehe Abschnitt 3.2.2). Dazu wird der ITS-Graph auf die Atomic-Call-Knoten reduziert und in weiteren Schritten in einen deterministischen endlichen Automaten transformiert. Abbildung 5.5 (b) zeigt den Prokollautomaten, der aus dem nebenstehenden OPG konstruiert wurde. Dieser Vorgang soll nun auch algebraisch auf Basis der formalen Sprachen beschrieben werden. Eine derartige Betrachtungsweise ist nämlich sehr hilfreich, um exakt formulieren zu können, welches Problem im Zusammenhang mit der ITS-Graph-basierten Protokollerkennung auftritt.

5.2.1 Protokollextraktion aus ITS-Pfaden

Ein ITS-Pfad $p \in N^*_{ITS}$ ist ein Pfad im ITS-Graphen (N_{ITS}, E_{ITS}), d.h. eine endliche Sequenz von Knoten $p = n_1 \cdots n_m$ mit den Eigenschaften

- $(n_i, n_{i+1}) \in E_{ITS}$ für $i = 1, \ldots, m-1$
- $\mathit{type}(n_1) = \mathsf{Start}$
- $\mathit{type}(n_m) = \mathsf{Final}$

Die Menge aller ITS-Pfade in einem ITS-Graphen ist $P_{ITS} \subseteq N^*_{ITS}$.

Durch Anwendung einer Projektion auf bestimmte Knotenmengen lassen sich aus ITS-Pfaden verschiedene Informationen gewinnen. Diese werden im Folgenden vorgestellt.

5.2.1.1 Projektion

Die *Projektion* löscht aus einem Wort alle Zeichen, die in einer bestimmten Menge nicht enthalten sind. Für Alphabete Σ und $\overline{\Sigma}$ sowie $w \in \Sigma^*, x \in \Sigma$ ist die Projektion definiert als $\pi_{\overline{\Sigma}} : \Sigma^* \to (\Sigma \cap \overline{\Sigma})^*$ mit

$$\begin{aligned} \pi_{\overline{\Sigma}}(\epsilon) &:= \epsilon \\ \pi_{\overline{\Sigma}}(wx) &:= \begin{cases} \pi_{\overline{\Sigma}}(w)x & \text{, falls } x \in \overline{\Sigma} \\ \pi_{\overline{\Sigma}}(w) & \text{, sonst} \end{cases} \end{aligned}$$

Die Bezeichnung Σ für das Alphabet dient hier nur der allgemeinen Definition. Bei dem Alphabet handelt es sich oft auch um eine Knotenmenge, die mit N etc. bezeichnet wird.

5.2.1.2 Atomic-Call-Sequenz

Die Menge der Atomic-Call-Knoten sei

$$N_\alpha := \{n \in N \mid \mathit{type}(n) = \mathsf{Call} \wedge n \text{ ist atomar}\}$$

Die *Atomic-Call-Sequenz* $\pi_{N_\alpha}(p)$ eines ITS-Pfades p erhält man durch Projektion des ITS-Pfades auf die atomaren Aufrufe:

$$\pi_{N_\alpha} : N^*_{ITS} \to N^*_\alpha$$

5.2.1.3 Intra-Thread-Sequenz

In einem expandierten OPG (N, E) ist jeder Knoten $n \in N$ mit Ausnahme des Start- und Initial-Entry-Knotens eindeutig einem Thread zugeordnet, da sämtliche Pfade vom Start-Knoten des OPG zum Knoten n über denselben Call-Knoten verlaufen, der den Aufruf eines bestimmten Threads repräsentiert. Für jeden Thread mit einer ID $t \in \mathbb{N}$ gibt es also eine Teilmenge $N_t \subset N$, welche gerade die Knoten dieses Threads enthält, außerdem den Start- und den Initial-Entry-Knoten des OPG. Für den ITS-Graphen (N_{ITS}, E_{ITS}) ist die Menge der Knoten des Threads entsprechend $N_t \cap N_{ITS}$.

Die *Intra-Thread-Sequenz* $\pi_{N_t}(p)$ des ITS-Pfades p bezüglich t ergibt sich durch die Projektion des ITS-Pfades auf N_t.

$$\pi_{N_t} : N_{ITS}^* \to N_t^*$$

$\pi_{N_t}(p)$ ist also ein Pfad, der nur die zum Thread mit der ID t gehörenden Knoten enthält und die anderen überspringt. Der Final-Knoten des ITS-Pfades muss dabei nicht unbedingt zur Intra-Thread-Sequenz gehören. Dies ist nur dann der Fall, wenn der ITS-Pfad auch auf dem Final-Knoten dieses Threads endet.

Projiziert man den ITS-Pfad auf die Schnittmenge $N_t \cap N_\alpha$, so erhält man die Atomic-Call-Sequenz $\pi_{N_t \cap N_\alpha}(p)$ für den Thread.

5.2.1.4 Protokoll

Jeder der Atomic-Call-Knoten eines ITS-Pfades steht für den Aufruf einer bestimmten Methode. Die Namen dieser Methoden bilden das Alphabet Σ für das zu extrahierende Protokoll $L \subseteq \Sigma^*$. Die entsprechenden RFG-Attribute der Atomic-Call-Knoten definieren eine Beschriftungsfunktion $l : N_\alpha \to \Sigma$, über die sich der Methodenname eines bestimmten Atomic-Call-Knotens bestimmen lässt. Eine entsprechende Definition $l : N_\alpha^* \to \Sigma^*$ gilt dabei für Sequenzen von Knoten bzw. Methodennamen.

Das mithilfe des ITS-Verfahrens dynamisch erkannte Protokoll $L \subseteq \Sigma^*$ ergibt sich somit aus der Menge aller ITS-Pfade P_{ITS} wie folgt:

$$L := \{l(\pi_{N_\alpha}(p)) \mid p \in P_{ITS}\}$$

5.2.2 Unzulässige Sequenzen

Die durch einen ITS-Graphen dargestellten Kausalitäten sind zwar thread-übergreifend, jedoch häufig nicht präzise genug. D.h. das aus einem ITS-Graphen extrahierte Protokoll enthält auch Sequenzen, die aufgrund der thread-internen Reihenfolge von Knoten gar nicht möglich sind. Anders gesagt ergibt die Projektion eines ITS-Pfades p auf einen einzelnen Thread t in bestimmten Fällen einen Pfad, der im Kontrollflussgraph dieses Threads gar nicht enthalten ist. Dementsprechend ist die beschriftete Atomic-Call-Sequenz dieses Pfades auch nicht Element des entsprechenden Intra-Thread-Protokolls L_t.

$$\exists t \in \mathbb{N}, p \in P_{ITS} : l(\pi_{N_t \cap N_\alpha}(p)) \notin L_t$$

5.2.2.1 Problem

Dass es sich dabei um ein prinzipielles Problem handelt, soll nun an einem kleinen Beispiel demonstriert werden. In Abbildung 5.6 ist ein OPG mit zwei Threads A und B dargestellt.

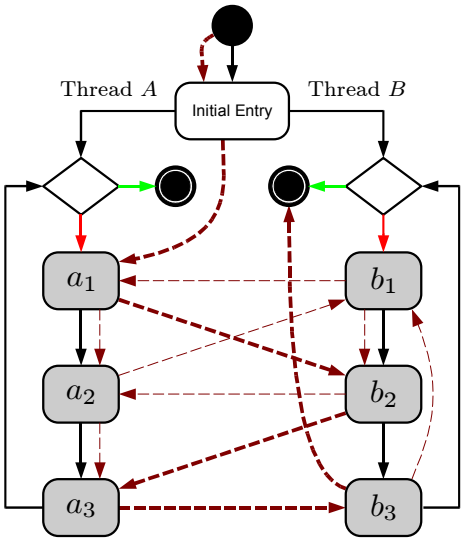


Abbildung 5.6: OPG (vereinfacht) mit ITS-Graph und unzulässigem ITS-Pfad p'

Dieser wurde der besseren Übersicht zuliebe etwas vereinfacht, was aber nichts an dem zu zeigenden Problem ändert. Die atomaren Aufrufe seien

$$N_\alpha = \{a_1, a_2, a_3, b_1, b_2, b_3\}$$

welche zur weiteren Vereinfachung mit der Identitätsfunktion beschriftet sind, d.h. $\Sigma = N_\alpha, l = \mathbf{id}$ ($\mathbf{id} : N_\alpha \to N_\alpha, n \mapsto n$). Des weiteren gibt es den Start-Knoten n_S, den Initial-Entry-Knoten n_E und die Final-Knoten a_F, b_F. Betrachtet man den Thread A separat, so erkennt man leicht das Intra-Thread-Protokoll $L_A \subseteq N_\alpha^*$ dieses Threads, hier mithilfe eines regulären Ausdrucks angegeben:

$$L_A = L((a_1 a_2 a_3)^*)$$

Es wird nun angenommen, das Programm sei gemäß der folgenden globalen Sequenz $p \in N_{ITS}^*$ abgelaufen (die Condition-Knoten seien nicht Element von N_{ITS}):

$$p = n_S n_E a_1 a_2 b_1 b_2 a_3 b_3 b_1 a_1 b_2 a_2 a_3 b_3 b_F$$

Bei genauer Betrachtung erkennt man, dass die Knoten innerhalb eines Threads jeweils die durch den OPG vorgegebene Reihenfolge einhalten ($a_1 \to a_2 \to a_3 \to \ldots$). Eine Projektion der globalen Sequenz auf die Knoten von Thread A ergibt somit z.B. einen zulässigen Pfad im OPG:

$$\pi_{N_A}(p) = n_S n_E a_1 a_2 a_3 a_1 a_2 a_3$$

Konstruiert man nun aus p einen ITS-Graphen, so erhält man die in Abbildung 5.6 gestrichelt gezeichneten ITS-Kanten, unter anderem $(n_E, a_1), (b_2, a_3), (a_3, b_3), (a_1, b_2), (b_3, b_F) \in E_{ITS}$. Dadurch existiert im fertigen ITS-Graph ein ITS-Pfad $p' \in P_{ITS}$ mit

$$p' = n_S n_E a_1 b_2 a_3 b_3 b_F$$

Die Kanten dieses Pfades sind in Abbildung 5.6 durch eine höhere Linienstärke gekennzeichnet. Projiziert man jetzt diesen ITS-Pfad auf die atomaren Aufrufe von Thread A, so ergibt sich:

$$\pi_{N_\alpha \cap N_A}(p') = a_1 a_3 \notin L_A$$

Offensichtlich ist die Atomic-Call-Sequenz $a_1 a_3$ nicht Element des Intra-Thread-Protokolls L_A, d.h. p' ist ein unzulässiger Pfad im ITS-Graph.

5.2.2.2 Lösungsidee

Daher soll nun verhindert werden, dass derartige Pfade im Protokoll auftauchen, indem zusätzlich zum ITS-Graphen die Intra-Thread-Protokolle $L_1, \ldots, L_m$ der einzelnen Threads berücksichtigt werden. Dazu könnte man die Menge der ITS-Pfade einschränken, so dass keines der Intra-Thread-Protokolle verletzt wird:

$$P'_{ITS} := \{p \in P_{ITS} \mid \forall t \in \{1, \ldots, m\} : l(\pi_{N_t \cap N_\alpha}(p)) \in L_t\}$$

Die tatsächliche Konstruktion der (nicht endlichen) Menge P'_{ITS} ist jedoch vermutlich nicht trivial. Eleganter ist es, die unzulässigen Sequenzen erst aus dem fertigen Inter-Thread-Protokoll herauszufiltern. Liegen die Protokolle nämlich in Form endlicher Automaten vor, so können die gängigen Sprachoperationen durch entsprechende Transformation der Automaten angewandt werden. Wie das Inter-Thread-Protokoll auf die zulässigen Sequenzen reduziert werden kann, wird im nächsten Abschnitt beschrieben.

5.3 Shuffle-Sprachen

Aus der Menge der Intra-Thread-Protokolle lässt sich ebenfalls ein Inter-Thread-Protokoll erzeugen, welches alle thread-internen, aber keine thread-übergreifenden Sequenz-Einschränkungen darstellt. Dies leistet gerade die aus der Literatur bekannte *Shuffle*-Operation auf Sprachen. Der Shuffle zweier Sprachen ist diejenige Sprache, die sämtliche Interleaving-Sequenzen von Wörtern der beiden Sprachen enthält.

5.3.1 Definition

Zunächst wird eine formale Definition des Shuffle auf Wörtern angegeben, die sich anschließend auf Sprachen erweitern lässt. Sind $w_1, w_2 \in \Sigma^*$ zwei Wörter, so ist der Shuffle $w_1 \sqcup\!\sqcup\, w_2$ die Menge aller Wörter, die sich durch „Ineinanderstecken“ von w_1 und w_2 ergeben können, vergleichbar mit zwei Stapeln von Spielkarten beim Mischen, wobei die Stapel den Wörtern und die Spielkarten den Zeichen entsprechen[1]. Er wird induktiv wie folgt definiert ($a, b \in \Sigma$):

$$\begin{aligned} w_1 \sqcup\!\sqcup\, \epsilon &:= \{w_1\} \\ \epsilon \sqcup\!\sqcup\, w_2 &:= \{w_2\} \\ w_1 a \sqcup\!\sqcup\, w_2 b &:= (w_1 a \sqcup\!\sqcup\, w_2) b \cup (w_1 \sqcup\!\sqcup\, w_2 b) a \end{aligned}$$

Der Shuffle zweier Sprachen $L_1, L_2 \subseteq \Sigma^*$ wird entsprechend als Vereinigung aller Shuffles aller Wort-Paare definiert:

$$L_1 \sqcup\!\sqcup\, L_2 := \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \sqcup\!\sqcup\, w_2$$

5.3.2 Eigenschaften

Die Shuffle-Operation hat zwei wichtige Eigenschaften, die im Folgenden nur genannt werden. In der Literatur finden sich ausführlichere Erläuterungen und entsprechende Beweise[2].

[1] vgl. [PW02] S. 40
[2] vgl. [CSV02] S. 187f.

- Die regulären Sprachen sind gegen die Shuffle-Operation abgeschlossen. D.h. wenn L_1, L_2 reguläre Sprachen über einem Alphabet Σ sind, so ist auch $L_1 \sqcup\!\sqcup L_2$ eine reguläre Sprache.
- Die Shuffle-Operation ist assoziativ und kommutativ.

Diese Erkenntnisse erlauben es, die Shuffle-Operation für die Protokollerkennung einzusetzen, ohne auf die Vorteile der regulären Sprachen verzichten zu müssen. Außerdem können so auch mehr als zwei Threads durch die Shuffle-Operation kombiniert werden. Sind also mehrere Intra-Thread-Protokolle $L_1, \ldots, L_n \subseteq \Sigma^*$ gegeben, so ergibt sich daraus (mit beliebiger Klammerung) das eindeutige Shuffle-Protokoll

$$L_{\sqcup\!\sqcup} := L_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup L_n$$

5.3.3 Konstruktion eines Shuffle-Automaten

Sind zwei reguläre Sprachen in Form zweier nichtdeterministischer endlicher Automaten A_1, A_2 gegeben, so kann daraus ein endlicher *Shuffle-Automat* $A_{\sqcup\!\sqcup}$ konstruiert werden, der gerade die Shuffle-Sprache der gegebenen Sprachen erkennt, d.h. so dass gilt

$$L(A_{\sqcup\!\sqcup}) = L(A_1) \sqcup\!\sqcup L(A_2)$$

Die Konstruktion erfolgt ähnlich wie die eines Produktautomaten, der die Sprache $L(A_1) \cap L(A_2)$ erkennen würde. So ist die Zustandsmenge des Shuffle-Automaten ebenfalls das kartesische Produkt der gegebenen Zustandsmengen. Der Unterschied besteht darin, dass der Produktautomat eine parallele Ausführung der gegebenen Automaten A_1 und A_2 „simuliert“, der Shuffle-Automat hingegen die nebenläufige Ausführung. Während der Produktautomat für jedes Zeichen des Eingabewortes den zeitgleichen Zustandsübergang beider gegebener Automaten simuliert, wählt der Shuffle-Automat bei jedem gelesenen Eingabezeichen nichtdeterministisch einen der beiden Automaten aus und simuliert nur den Zustandsübergang dieses einen Automaten, während der andere Automat seinen Zustand beibehält.

Von Câmpeanu et al.[3] wird ein ähnliches Konstruktionsverfahren vorgestellt. Dieses erzeugt jedoch gleich einen deterministischen Automaten. Dessen Zustandsmenge kann exponenziell groß werden, weshalb dieser Algorithmus in bestimmten Fällen nicht praktikabel ist. Die folgende, abgewandelte Beschreibung der Konstruktion hingegen lässt auch einen nichtdeterministischen Automaten als Ergebnis zu. Der entsprechende deterministische Potenzautomat kann daraus dann in einem zweiten Schritt konstruiert werden, falls die verfügbare Rechen- und Speicherkapazität ausreicht, abhängig von der Größe der Zustandsmenge.

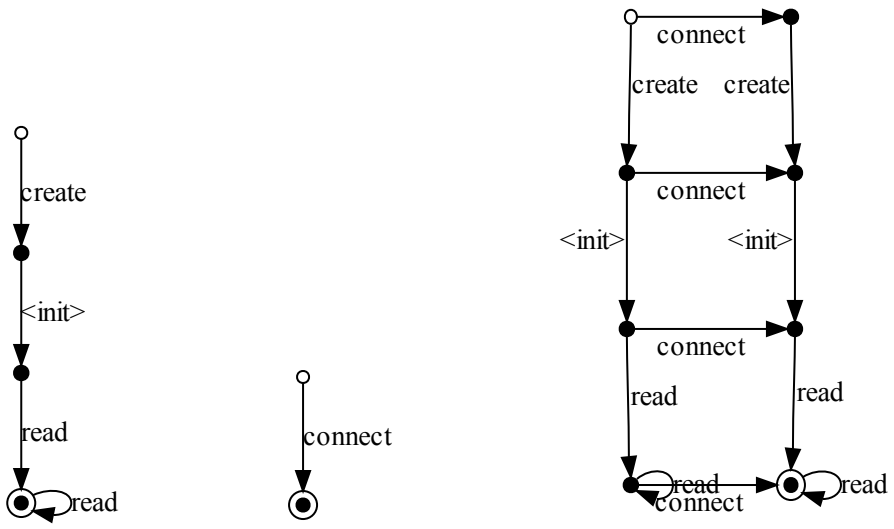
Seien $A_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, F_i)$ für $i = 1, 2$ nichtdeterministische endliche Automaten. Der Shuffle-Automat $A_{\sqcup\!\sqcup} = (Q_{\sqcup\!\sqcup}, \Sigma_{\sqcup\!\sqcup}, \delta_{\sqcup\!\sqcup}, q_{0\sqcup\!\sqcup}, F_{\sqcup\!\sqcup})$ ist ein nichtdeterministischer endlicher Automat und wie folgt definiert durch

- die Menge der Zustände $Q_{\sqcup\!\sqcup} = Q_1 \times Q_2$
- das Eingabealphabet $\Sigma_{\sqcup\!\sqcup} = \Sigma_1 \cup \Sigma_2$
- die Zustandsüberführung $\delta_{\sqcup\!\sqcup} \subseteq Q_{\sqcup\!\sqcup} \times \Sigma_{\sqcup\!\sqcup} \times Q_{\sqcup\!\sqcup}$ mit

  $$\delta_{\sqcup\!\sqcup}((q_1, q_2), x) = \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, x)\} \cup \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, x)\}$$

  Für ein Eingabezeichen $x \notin \Sigma_i$ sei $\delta_i(q, x) = \emptyset$ (für $i = 1, 2$, $q \in Q_i$).

[3] vgl. [CSV02] S. 195f.



(a) A_1 (b) A_2 (c) Shuffle-Automat $A_{\sqcup\!\sqcup}$

Abbildung 5.7: Automaten für Intra-Thread-Protokolle und Shuffle

- den Startzustand $q_{0_{\sqcup\!\sqcup}} = (q_{0_1}, q_{0_2})$
- die Endzustände $F_{\sqcup\!\sqcup} = F_1 \times F_2$

Der Shuffle-Automat unterscheidet sich vom Produktautomaten also nur im Eingabealphabet, welches eine Vereinigung statt einer Schnittmenge ist, sowie in der Zustandsübergangsfunktion, die eine nebenläufige statt einer parallelen Ausführung modelliert.

Ein Beispiel für die Konstruktion eines Shuffle-Automaten aus zwei gegebenen Automaten wird in Abbildung 5.7 dargestellt. Die Intra-Thread-Protokolle $L(A_1)$ und $L(A_2)$ stammen aus der Analyse von Listing 3.4 (siehe Abbildung 3.7 und 3.8).

Da der Shuffle-Automat aus den vorhandenen Intra-Thread-Protokollen konstruiert wurde, enthält er keine neue Information. Wie man beobachten kann, schränkt der Shuffle-Automat jeweils nur die Reihenfolge der Eingabezeichen innerhalb eines Threads ein. So darf in $L(A_{\sqcup\!\sqcup})$ ebenso wie in $L(A_1)$ kein `read` vor einem `<init>` erfolgen. Dass jedoch – gemäß dem erwarteten Protokoll – ein `connect` zwischen dem `<init>` und dem ersten `read` stehen muss, wird von dem Shuffle-Automaten nicht beachtet.

Der Shuffle-Automat ist somit nur eine andere Darstellung vorhandener Erkenntnisse. Seinen eigentlichen Nutzen entfaltet der Shuffle-Automat erst, wenn man ihn mit den Ergebnissen der ITS-Graph-basierten Analyse verknüpft.

5.3.4 Schnitt-Protokoll

Das ITS-Protokoll

⊕ kann Informationen darüber liefern, welche Interleaving-Sequenzen dadurch ausgeschlossen sind, dass zwischen bestimmten Operationen in verschiedenen Threads eine explizite

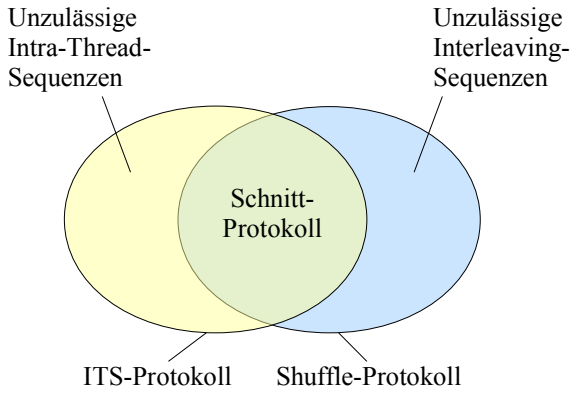


Abbildung 5.8: Venn-Diagramm zur Veranschaulichung von $L = L_{ITS} \cap L_{\sqcup\!\sqcup}$.

zeitliche Reihenfolge vorgegeben ist.

⊖ kann Sequenzen enthalten, die dem Intra-Thread-Protokoll eines Threads widersprechen.

Das Shuffle-Protokoll

⊕ stellt die Einhaltung der Intra-Thread-Protokolle sicher.

⊖ enthält keine Informationen über Einschränkungen bei den Interleaving-Sequenzen.

Beide Verfahren haben also das prinzipielle Problem, dass die extrahierten Protokolle auch unzulässige Sequenzen enthalten. Die zulässigen, durch die dynamische Analyse erkennbaren Sequenzen sind jedoch in den Ergebnissen beider Verfahren enthalten. Am nächsten kommt man dem tatsächlichen Protokoll also dadurch, dass man durch Konstruktion eines Produktautomaten die Schnittmenge aus beiden Ergebnissen bildet. Dieser Zusammenhang wird in Abbildung 5.8 grafisch dargestellt.

$$L = L_{ITS} \cap L_{\sqcup\!\sqcup}$$

Diese Schnittmenge heißt entsprechend *Schnitt-Protokoll* der Multithreading-Anwendung. Aufgrund der hier angestellten Überlegungen scheint das Schnitt-Protokoll eine ideale Darstellung des nebenläufigen Verhaltens einer Anwendung zu sein.

5.4 Anwendung an einem Beispiel

Die Arbeitsweise des zuvor beschriebenen Verfahrens zur Protokollerkennung soll nun noch einmal schrittweise anhand eines weiteren Beispiels nachvollzogen werden. Das bisher betrachtete, einfache Beispiel (Listing 3.4) zeichnet sich dadurch aus, dass dort zwar Nebenläufigkeit vorliegt, die zeitliche Reihenfolge durch den Code aber bereits eindeutig vorgegeben ist. Die Trennung in zwei Threads ist dabei eher technischer Natur. Durch Analyse eines komplexeren Beispiels lässt sich ein besserer Eindruck davon gewinnen, wie sich das ITS-Offline-Verfahren beim Auftreten „echter“ Nebenläufigkeit verhält und welchen Beitrag das Shuffle-Protokoll leistet, um das extrahierte ITS-Protokoll auf die tatsächlich gültigen Sequenzen zu reduzieren.

```
public class Main {
    ...
    public static void main(String args[]) {
        Queue queue = new Queue();
        queue.setMaxSize(100);
        Producer producer = new Producer(queue); producer.start();
        Consumer consumer = new Consumer(queue); consumer.start();
        ...
        producer.join();
        consumer.join();
        ...
    }
}

class Producer extends Thread {
    public Queue queue;
    public Producer(Queue queue) { this.queue = queue; }
    public void run() {
        int i = 1;
        while (i <= Main.ELEMENT_COUNT) {
            if (queue.size() < 2) {
                queue.push("A" + i);
                i++;
            }
        }
    }
}

class Consumer extends Thread {
    public Queue queue;
    public Consumer(Queue queue) { this.queue = queue; }
    public void run() {
        while (true) {
            if (!queue.isEmpty()) {
                String s = queue.pop();
                if (s.equals("A" + Main.ELEMENT_COUNT)) break;
            }
        }
    }
}
```

Listing 5.1: Erzeuger-Verbraucher-System

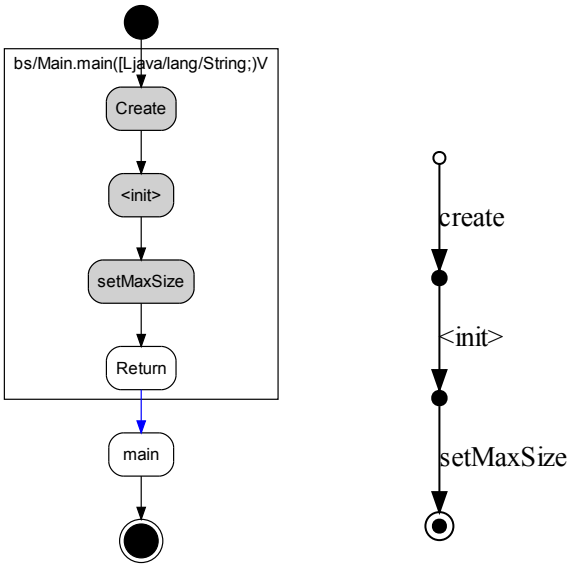
5.4.1 Code

Wie in vielen generellen Betrachtungen zur Nebenläufigkeit in der Literatur soll auch hier das *Erzeuger-Verbraucher-System* als Beispiel herangezogen werden. Der entsprechende Code ist in Listing 5.1 abgedruckt. Analysiert werden soll das Protokoll der *Queue*, der gemeinsam verwendeten Datenstruktur. Zu beachten ist dabei, dass es sich hier ausdrücklich *nicht* um eine „elegante“ Lösung der hier auftretenden Synchronisationsprobleme mit Semaphoren etc. handelt. Vielmehr wurde absichtlich ein „Polling“-Verhalten implementiert, um ein komplexeres Protokoll zu erhalten.

Das Protokoll der Queue sei so spezifiziert, dass vor jedem dem Einfügen mittels `push` der Füllungsgrad der Queue durch `size` überprüft werden muss. Ebenso sei festgelegt, dass jedem Aufruf von `pop` eine Überprüfung mittels `isEmpty` vorausgehen muss.

Im echten Beispiel-Code (siehe Anhang) befinden sich noch einige Sleep-Anweisungen mit zufälliger Länge, damit idealerweise alle Verzahnungsmöglichkeiten der Threads im Programmablauf vorkommen. So verringert sich in diesem Beispiel die Abhängigkeit des resultierenden ITS-Protokolls vom Verhalten des Schedulers.

Das Beispielprogramm besteht aus drei Threads, wobei ein Thread nur zum Erzeugen und



(a) Projizierter OPG (b) Protokoll

Abbildung 5.9: Erzeuger-Verbraucher-System: Unzureichende Protokollerkennung mit dem bisherigen Verfahren

Initialisieren des Objektes dient. Je ein weiterer Thread erfüllt die Rolle des Erzeugers bzw. des Verbrauchers. Die öffentlichen Methoden der Klasse `Queue` sind thread-sicher. Im hier abgebildeten Teil des Codes ist daher keine weitere Synchronisation notwendig.

5.4.2 Protokollerkennung mit dem bisherigen Verfahren

Zunächst soll das Erzeuger-Verbraucher-System mithilfe des bisherigen, in Kapitel 3 beschriebenen Protokollerkennungsverfahrens analysiert werden. Daran lässt sich erkennen, welcher Unterschied zum anschließend extrahierten ITS-, Shuffle- und Schnitt-Protokoll besteht.

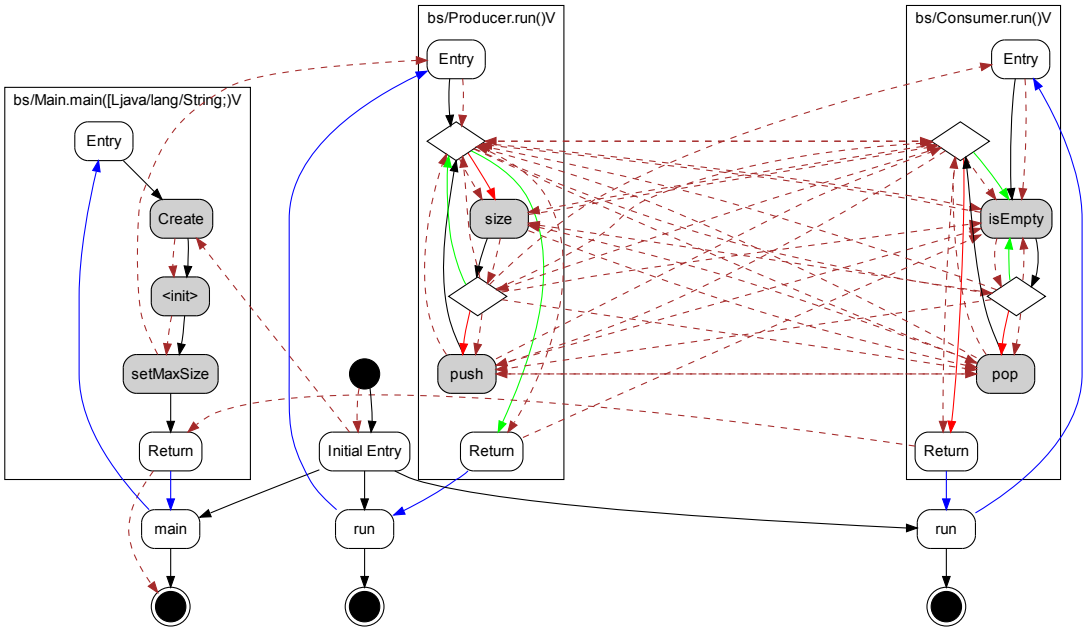
Der durch das bisherige Verfahren erzeugte OPG und das daraus resultierende Protokoll sind in Abbildung 5.9 zu sehen. Analog zum ersten Beispiel einer Multithreading-Anwendung (S. 26) sieht man auch hier, wie im projizierten OPG lediglich die Knoten enthalten sind, die vom Create-Knoten aus erreichbar sind, somit also zum selben Thread wie der Create-Knoten gehören. Im Protokoll ist dadurch nur die Initialisierung des Objektes durch den ersten Thread enthalten.

5.4.3 Protokollerkennung mit dem ITS-Shuffle-Schnitt-Verfahren

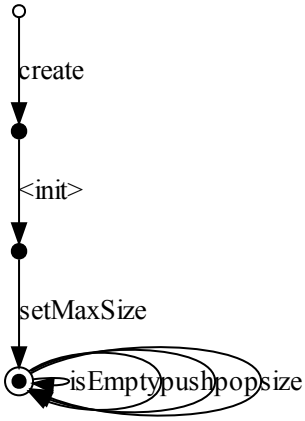
Es werden nun nacheinander das ITS-Protokoll, das Shuffle- und das Schnitt-Protokoll betrachtet. Dabei wird versucht, die erhaltenen Informationen sinnvoll zu interpretieren.

5.4.3.1 ITS-Protokoll

Zunächst sollen der ITS-Graph und das ITS-Protokoll des Beispielsystems betrachtet werden (Abbildung 5.10). Anhand des ITS-Graphen lassen sich eine Reihe von Erkenntnissen ablesen:



(a) OPG und ITS-Graph



(b) ITS-Protokoll

Abbildung 5.10: Erzeuger-Verbraucher-System: ITS-Protokoll

- Die ITS-Kante vom Aufruf der atomaren Methode `setMaxSize` zum Entry-Knoten der Thread-Methode `Producer.run()` zeigt an, dass der Aufruf von `setMaxSize` die letzte relevante Aktivität in der `main`-Methode war, bevor die Ausführung erstmals in einen anderen Thread wechselte.

- Die Ausführung von Thread `Producer` beginnt zu einem früheren Zeitpunkt als die von Thread `Consumer`.

- Die zahlreichen ITS-Kanten zwischen den Methodenrümpfen von `Producer.run()` und `Consumer.run()` lassen den Schluss zu, dass dort keine besondere Synchronisation stattfindet, sondern die Anweisungen beider Methodenrümpfe in beliebiger Weise verzahnt sein können.

- Die ausgehende ITS-Kante vom Return-Knoten der Methode `Producer.run()` führt zu einem Knoten innerhalb der Methode `Consumer.run()`. Zum Zeitpunkt der Termination des Threads `Producer` war der Thread `Consumer` offensichtlich noch aktiv.

- Die ausgehende ITS-Kante vom Return-Knoten der Methode `Consumer.run()` führt zum Return-Knoten der `main`-Methode. Dies ist die einzige ITS-Kante, deren Zielknoten innerhalb des Haupt-Threads, der Quellknoten jedoch in einem anderen Thread liegt. Seit dem Aufruf von `setMaxSize` gab es somit im Haupt-Thread keine weiteren Aktivitäten bis zur Termination der beiden anderen Threads. Dies lässt vermuten, dass in der `main`-Methode eine explizite Synchronisation stattfindet (`join`).

- Die letzte ITS-Kante verläuft vom Return-Knoten der `main`-Methode zum Final-Knoten des Haupt-Threads. Der Haupt-Thread war also derjenige Thread, der als letztes terminierte.

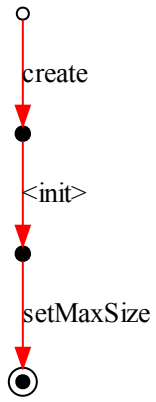
Durch den in Abschnitt 5.2.2 beschriebenen Effekt entstehen nun auch in diesem ITS-Graphen unzulässige Pfade, welche das Intra-Thread-Protokoll von einem oder mehreren Threads verletzen. So kann ein ITS-Pfad z.B. über den `push`-Knoten des `Producer`-Threads verlaufen, von dort aus zum `isEmpty`-Knoten, dann zum darauf folgenden Condition-Knoten und wieder zurück zum `push`-Knoten. Die Intra-Thread-Sequenz dieses ITS-Pfades bezüglich `Producer` enthält daher zwei aufeinander folgende Aufrufe von `push`, was dem spezifizierten Protokoll widerspricht.

Für die anderen atomaren Aufrufe in den Threads `Producer` und `Consumer` existieren ähnliche ITS-Pfade, so dass das resultierende Protokoll für die Phase nach der Initialisierung der Queue überhaupt keine Sequenzeinschränkungen mehr für die vier Methoden `size`, `isEmpty`, `push` und `pop` vorgibt. Dass die Sequenz `create`–`<init>`–`setMaxSize` vor der erstmaligen Verwendung der eben genannten Methoden ausgeführt wurde, wird hingegen durch das ITS-Protokoll korrekt wiedergegeben.

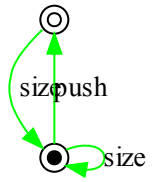
Durch die fehlenden Sequenzeinschränkungen für die Queue-Operationen stellt sich die Frage, inwiefern dieses ITS-Protokoll überhaupt von Nutzen ist. Andererseits lässt sich das ITS-Protokoll auch als (zu starke) Verallgemeinerung des spezifizierten Protokolls betrachten. Zumindest handelt es sich um eine Obermenge aller zulässigen Sequenzen.

5.4.3.2 Shuffle-Protokoll

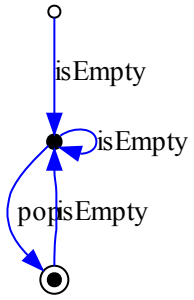
Nun soll die Konstruktion des Shuffle-Protokolls als „Gegenspieler“ zum ITS-Protokoll nachvollzogen werden. Dazu wird der nicht-projizierte OPG zunächst in jeweils einen OPG pro



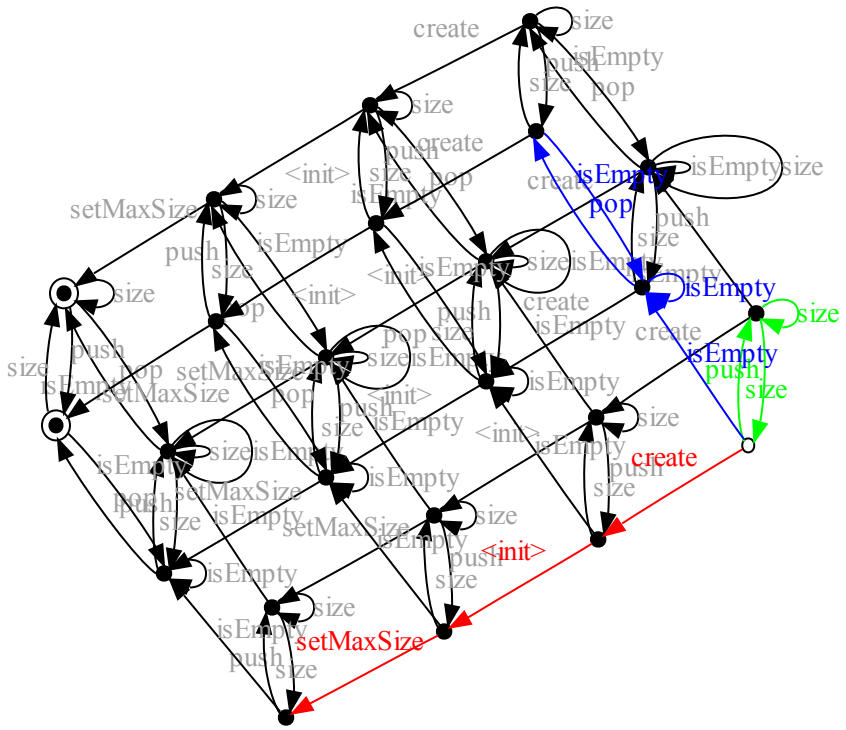
(a) Thread `Main.main()`



(b) Thread `Producer.run()`



(c) Thread `Consumer.run()`



(d) Shuffle-Protokoll aus den Threads (a) bis (c)

Abbildung 5.11: Erzeuger-Verbraucher-System: Shuffle-Protokoll

Thread zerlegt, und aus diesem mit der bisherigen Methode ein Intra-Thread-Protokoll generiert. Die Intra-Thread-Protokolle der drei Threads sind in Abbildung 5.11 (a) bis (c) dargestellt.

Durch Verknüpfung aller drei Intra-Thread-Protokolle mittels der Shuffle-Operation ergibt sich das Shuffle-Protokoll in Abbildung 5.11 (d). Offensichtlich eignet sich diese Art der Darstellung nicht dazu, nützliche Informationen über das analysierte System zu erfahren. Man erkennt aber die „dreidimensionale Struktur“ des Protokollautomaten. Für jeden Thread finden die Zustandsübergänge in einer eigenen „Dimension“ statt. Dies wird durch die farbliche Hervorhebung der Kanten entsprechend der Intra-Thread-Protokolle angedeutet.

Der Startzustand des abgebildeten Shuffle-Automaten befindet sich rechts unten. Würde jeweils nur einer der drei Threads ausgeführt, so würden auch nur die in der Farbe des entsprechenden Threads markierten Zustandsübergänge des Protokollautomaten genutzt.

Das Shuffle-Protokoll lässt hingegen jedes mögliche Interleaving-Verhalten der drei Intra-Thread-Protokolle zu. Dadurch erklärt sich auch die unübersichtlich große Knotenzahl des Shuffle-Automaten. Zu beachten ist ferner, dass die Endzustände des Shuffle-Automaten nur erreicht werden, wenn für alle drei Threads – egal ob nebenläufig verzahnt oder nacheinander – eine entsprechende Sequenz ausgeführt wurde, durch die der Automat des jeweiligen Intra-Thread-Protokolls in einen Endzustand übergegangen wäre.

5.4.3.3 Schnitt-Protokoll

Zur Bestimmung des Schnitt-Protokolls wird nun der Produktautomat aus den Automaten des ITS-Protokolls und des Shuffle-Protokolls konstruiert. Im Anschluss wird dieser noch in einen deterministischen, minimalen Automaten transformiert, der in Abbildung 5.12 zu sehen ist.

5.5 Zusammenfassung

Nachdem andere Konzepte zur Darstellung nebenläufigen Verhaltens betrachtet wurden, fiel die Wahl zugunsten des endlichen Automaten, mit welchem auch bereits die Single-Thread-Protokolle repräsentiert werden. Da ein endlicher Automat keine explizite Darstellungsmöglichkeit für Nebenläufigkeit bietet, kann ein Protokoll nur sequentiell in Form möglicher Linearisierungen des nebenläufigen Verhaltens repräsentiert werden. Zur Herleitung möglicher Linearisierungen wurde das Konzept des ITS-Graphen eingeführt und sowohl ein Online- als auch ein Offline-Verfahren zur Konstruktion vorgestellt.

Es wurde gezeigt, wie das dynamisch erkannte Interleaving-Verhalten eines Programms in Form eines ITS-Graphen repräsentiert werden kann und wie aus einem ITS-Graphen ein ITS-Protokoll gewonnen wird. Dabei wurde jedoch deutlich, auf welche Weise dieses Analyseverfahren auch unzulässige Pfade und somit unzulässige Intra-Thread-Sequenzen generieren kann.

Zur Einschränkung des erkannten Protokolls auf die zulässigen Sequenzen wurde die aus der Literatur bekannte Shuffle-Operation auf die Intra-Thread-Protokolle der einzelnen Threads angewandt, um ein Shuffle-Protokoll zu erhalten. Durch Schnittmengenbildung mit dem ITS-Protokoll ergibt sich das Schnitt-Protokoll, welches sowohl das tatsächliche Interleaving-Verhalten als auch die Sequenzeinschränkungen innerhalb der einzelnen Threads berücksichtigt.

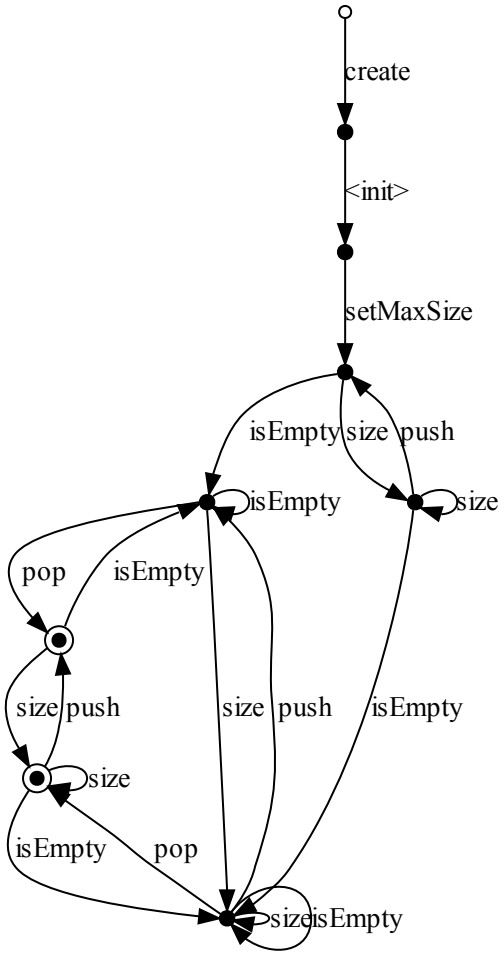


Abbildung 5.12: Erzeuger-Verbraucher-System: Schnitt-Protokoll

KAPITEL 6

Evaluation

In diesem Kapitel wird die in Kapitel 5 vorgestellte Erweiterung des Protokollerkennungsverfahrens hinsichtlich ihrer Praxistauglichkeit bei der Analyse realer Software-Produkte untersucht.

Inhalt

6.1 Fragestellung **63**
6.2 Testkonfiguration **64**
6.2.1 Voraussetzungen 64
6.2.2 Messverfahren 64
6.2.3 Systeme 65
6.2.4 Testfälle 65
6.3 Erzeugen der globalen Spur **65**
6.3.1 Durchführung 65
6.3.2 Auswertung 66
6.4 Protokollextraktion **67**
6.4.1 Durchführung 67
6.4.2 Auswertung 68
6.5 Ergebnis **69**

6.1 Fragestellung

Ein nützliches Protokollerkennungswerkzeug erfüllt in erster Linie zwei Eigenschaften: Es liefert ein korrektes bzw. brauchbares Ergebnis und tut dies innerhalb einer angemessenen Zeit. Um die Korrektheit eines extrahierten Protokolls festzustellen, benötigt man ein Vergleichskriterium, d.h. eine existierende Spezifikation des Protokolls, die für die analysierten Anwendungen in der Regel nicht verfügbar ist.

Die Nützlichkeit eines Protokollerkennungswerkzeuges im Allgemeinen wurde bereits durch Quante [QK07] begründet. Dass die in dieser Diplomarbeit entwickelten Erweiterungen des Protokollerkennungsverfahrens mehr Informationen berücksichtigen und dadurch ein besseres Ergebnis als das bisherige Verfahren liefern, konnte zumindest anhand der Beispiele nachvollzogen werden.

Daher soll sich die Fragestellung auf die Evaluation der Performanz beschränken. Durch die eingeführten Erweiterungen sind bei der Protokollerkennung einige Verarbeitungsschritte hinzugekommen. Diese funktionieren bei kleinen Beispielen wie gewünscht. Bisher unbekannt ist aber die Skalierbarkeit dieser Algorithmen, welche daher anhand echter Systeme untersucht

werden soll. Dies bezieht sich sowohl auf den Zeitaufwand der Berechnungen als auch auf den Speicherbedarf der Ergebnisse.

6.2 Testkonfiguration

Vor der Durchführung und Auswertung der Testprozedur wird zunächst festgehalten, unter welchen Bedingungen die Tests stattfinden.

6.2.1 Voraussetzungen

Für alle Performanzmessungen wurde der Rechner `sts6.informatik.uni-bremen.de` verwendet. Dieser hat die folgenden Eigenschaften:

Betriebssystem:	Ubuntu 8.10 (Linux)
Kernel-Version:	`2.6.27-14-generic`
Hardware-Architektur:	`x86_64` (64 Bit)
Anzahl CPUs:	8
CPU-Typ:	Intel® Quad-Core Xeon® X5365
CPU-Takt:	3 GHz
Arbeitsspeicher:	64 GB

6.2.2 Messverfahren

Für das in dieser Diplomarbeit vorgestellte Konzept musste die Implementierung an zwei Stellen geändert werden. Durch die Erstellung der globalen Spur ist zu erwarten, dass die Performanz der analysierten Anwendung im Vergleich zur bisherigen Online-DOPG-Erzeugung reduziert ist. Die Weiterverarbeitung des Roh-Graphen und der globalen Spur weist aufgrund der hinzugekommenen bzw. veränderten Daten voraussichtlich ebenfalls ein anderes Zeitverhalten auf.

Eine ähnliche Performanzmessung findet sich in [Qua07]. Dort wird der Zeitbedarf des vorgestellten Verfahrens zur Online-DOPG-Erzeugung im Vergleich zur Offline-Erzeugung bestimmt. Die Testfälle beinhalten jeweils eine interaktive Benutzung eines Programms, die nie exakt reproduzierbar ist. Deshalb wird für diese Messungen nicht die Realzeit gemessen, sondern die CPU-Zeit des jeweiligen Prozesses.

Für die im Folgenden durchzuführenden Messungen wird daher ebenfalls die CPU-Zeit gemessen. Da die Erzeugung der globalen Spur eine Ein-/Ausgabe-Operation ist, muss allerdings auch gemessen werden, inwiefern die Ein-/Ausgabe-Operation die Realzeit zusätzlich erhöht. Diese Ergebnisse hängen natürlich stark davon ab, in welchem Verhältnis die CPU-Leistung und die Datenübertragungsrate der Festplatte zueinander stehen, und wie stark insbesondere die Festplatte zum Zeitpunkt des Tests ausgelastet ist. Die vorhandenen CPUs des verwendeten Rechners sind in aller Regel nie ausgelastet, so dass dem Testlauf immer eine eigene CPU zu 100% zur Verfügung steht, die nicht mit anderen Prozessen geteilt werden muss.

Da sich gezeigt hat, dass die Verwendung eines Profilers (z.B. hprof) selbst einen großen Einfluss auf das Zeitverhalten einer Anwendung hat, wurde eine pragmatischere Lösung gewählt: Der Code der Anwendung wurde von Hand so modifiziert, dass vor und nach Ausführung einer internen „Haupt“-Methode die entsprechenden Zeiten registriert werden. Es ist davon auszugehen, dass diese Code-Änderung das Analyseergebnis nicht in relevanter Weise verfälscht,

da die eingefügten Anweisungen nur einmal im Programmlauf ausgeführt werden. Lediglich die Anzahl der bei der Instrumentierung eingefügten Logging-Anweisungen erhöht sich durch die zwei zusätzlichen Zeilen geringfügig.

Folgende Methoden der Java-API wurden zur Zeitmessung eingesetzt:

Realzeit: `System.currentTimeMillis()`

CPU-Zeit: `ManagementFactory.getThreadMXBean().getCurrentThreadCpuTime()`

Die beschriebenen Systeme, Testfälle und Ergebnisse befinden sich auf der beiliegenden DVD.

6.2.3 Systeme

ArgoUML 0.28.beta5[1] ist ein verbreitetes Open-Source-Werkzeug zur Erstellung von UML-Diagrammen. Untersucht wurde das Protokoll der Klasse `UMLClassDiagram`.

FindBugs 1.3.9[2] ist ein Werkzeug zum automatischen Auffinden von typischen Programmierfehlern in Java-Code. Es verfügt sowohl über eine GUI als auch über eine Kommandozeilenschnittstelle. Die Klasse `SortedBugCollection` wurde für die Protokollanalyse ausgewählt.

Beide Anwendungen lagen im Bytecode nach Java-1.5-Standard vor. Zur besseren Kompatibilität mit dem beim Instrumentieren eingesetzten ASM-Framework wurde der Quellcode jeweils neu kompiliert, um einen Bytecode nach Java-1.6-Standard zu erhalten.

6.2.4 Testfälle

Nach dem Starten der Anwendung *ArgoUML* wurde in dem vorhandenen, neuen Dokument ein kleines Klassendiagramm aus vier Klassen erstellt. Zwischen je zwei Klassen wurde eine „realize“-Beziehung erstellt, zwischen den beiden Basisklassen zusätzlich eine „dependency“-Kante eingefügt. Anschließend wurde das Dokument über „Save Project As...“ gespeichert und die Anwendung verlassen.

Die Anwendung *FindBugs* wurde mit der Analyse von kleineren Beispielen von Java-Code beauftragt. Für drei verschiedene Beispieldaten wurde jeweils vor dem Testlauf eine Projektdatei mit den zu analysierenden Verzeichnissen erstellt. Im eigentlichen Testlauf wurde diese Projektdatei der Anwendung übergeben, woraufhin die Analyse begann.

6.3 Erzeugen der globalen Spur

Die Ausgabe der globalen Spur während des Testlaufs eines zu analysierenden Programms verursacht eine zusätzliche Auslastung der Rechnerkapazitäten und einen höheren Zeitbedarf. Die folgende Messung soll einen Hinweis darauf geben, in welcher Größenordnung sich der erhöhte Zeitbedarf bewegt.

6.3.1 Durchführung

Zur Performanzbestimmung für das Erzeugen der globalen Spur soll neben der CPU-Zeit auch die Realzeit gemessen werden. Bei einer interaktiven Anwendung setzt sich die Realzeit nicht nur aus der CPU-Zeit und den Wartezeiten für Ein-/Ausgabe und Synchronisation

[1] `http://argouml.tigris.org`
[2] `http://findbugs.sourceforge.net`

zusammen. Sie wird vielmehr zusätzlich vom Verhalten des Benutzers beeinflusst. Da der genaue zeitliche Ablauf einer Benutzerinteraktion nicht reproduzierbar ist, eignet sich für diese Messung besser ein nicht-interaktives Programm.

Dabei muss es sich nicht einmal um eine Multithreading-Anwendung handeln, denn der zusätzliche Zeitbedarf durch das Erzeugen der globalen Spur ist unabhängig von deren Inhalt und somit unabhängig von der Anzahl der Threads. Gerade bei einer Single-Thread-Anwendung lässt sich das zeitliche Verhalten besonders gut reproduzieren und messen.

Die Anwendung *FindBugs* verfügt neben der GUI auch über eine Kommandozeilenschnittstelle. Verwendet man diese, so wird neben dem Haupt-Thread kein weiterer Thread gestartet. Diese Konfiguration eignet sich somit, eine möglichst genaue Zeitmessung vorzunehmen. So kann der zusätzliche Zeitbedarf für das Erzeugen der globalen Spur abgeschätzt werden. Der Testfall wurde wie beschrieben durchgeführt, lediglich das Speichern als XML-Datei entfiel bei der Kommandozeilenausführung.

Jeder Testfall wurde mit drei verschiedenen Varianten der Software durchgeführt:

1. Der ursprüngliche, nicht instrumentierte Code („Original-Code“).
2. Der für das bisherige Protokollerkennungsverfahren instrumentierte Code. Es wird also nur ein Online-DOPG konstruiert („DOPG (online)“).
3. Der für das ITS-Offline-Verfahren instrumentierte Code. Dieser erzeugt sowohl einen Online-DOPG als auch eine globale Spur („Globale Spur“).

Für jede der drei Varianten wurden die Testfälle drei Mal durchgeführt. Anschließend wurde aus den Messwerten das arithmetische Mittel gebildet. Gemessen wurde jeweils der gesamte Zeitbedarf einer bestimmten Konfiguration. Anschließend wurden die Differenzen zwischen den Gesamt-Zeiten berechnet, um die Dauer für die einzelnen Vorgänge (Original-Code, DOPG (online), Globale Spur) abschätzen zu können.

6.3.2 Auswertung

Die Ergebnisse der Zeitmessungen sind in Tabelle 6.1 zusammengefasst. Die Testfälle 1) – 3) beziehen sich auf die unterschiedlichen Beispieldaten, die durch *FindBugs* analysiert wurden. In der Zeile „Summe“ ist jeweils die Gesamtdauer für die Testkonfiguration mit Erzeugung der globalen Spur angegeben. In den Zeilen darüber steht der berechnete Zeitbedarf für die einzelnen Vorgänge.

Zunächst fällt auf, dass es sich um eine CPU-lastige Anwendung handelt, denn die Werte von CPU-Zeit und Real-Zeit unterscheiden sich jeweils nur geringfügig. Auffällig ist, wie unterschiedlich die Verhältnisse zwischen den Zeiten „DOPG (online)“ und „Original-Code“ ausgeprägt sind. So ist im Testfall 1) eine Erhöhung des Zeitbedarfs um den Faktor $13,36$ zu beobachten, während der Faktor im Testfall 3) bei $52,00$ liegt. Dies deckt sich aber grob mit den Beobachtungen aus [QK07].

Für die Untersuchung in dieser Diplomarbeit ist interessanter, inwiefern sich die Erzeugung der globalen Spur auf das Laufzeitverhalten auswirkt. Dazu wurde der „Zeitfaktor Glob./DOPG“ berechnet als

$$\frac{t_{\text{Summe}}}{t_{\text{Original-Code}} + t_{\text{DOPG (online)}}}$$

Es zeigte sich, dass bei Erzeugung der globalen Spur die verbrauchte Zeit (CPU-Zeit und Realzeit) im Mittel um den Faktor $\approx 1,67$ höher ist als bei der bisherigen DOPG-Online-

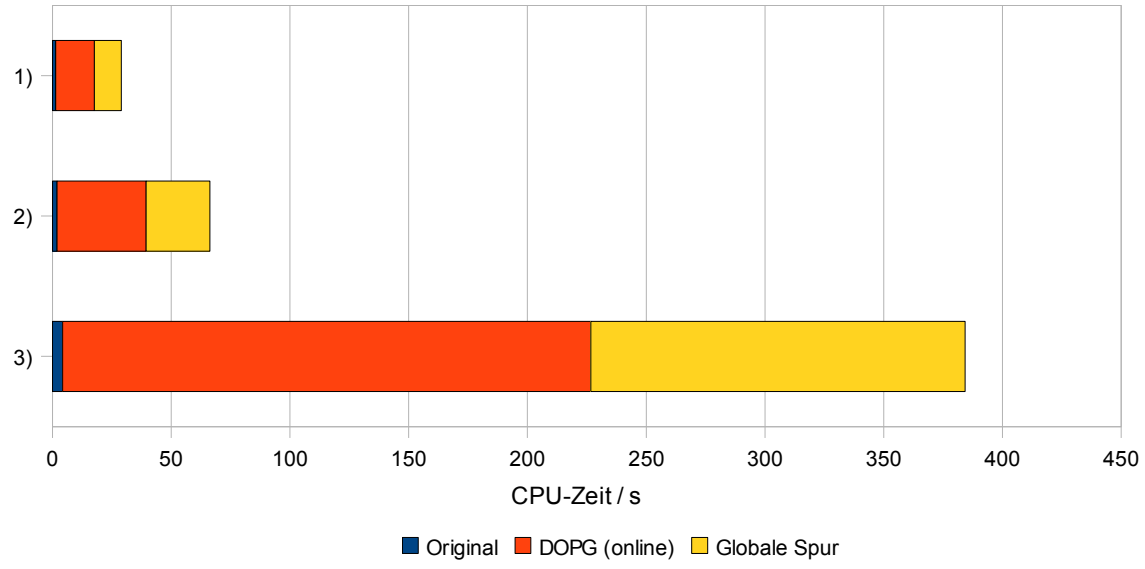


Abbildung 6.1: CPU-Zeiten und ihre Zusammensetzung laut Tabelle 6.1: Original-Code, DOPG-Erzeugung (online) und Erzeugung der globalen Spur

Erzeugung. Die einzelnen Abweichungen von diesem Faktor sind sehr gering. Die Ausgabeleistung für die globale Spur in Mio. Einträge/s nimmt ebenfalls recht einheitliche Werte an.

Die Zusammensetzung der Summe aus den einzelnen Zeiten ist in Abbildung 6.1 visualisiert. Auch dort ist erkennbar, dass der zusätzliche Zeitbedarf für das Erzeugen der globalen Spur in einem praktisch konstanten Verhältnis zum bisherigen Zeitbedarf steht.

Somit kann davon ausgegangen werden, dass für den Zeitverbrauch bei der Erzeugung der globalen Spur dieselben Überlegungen gelten wie bei der DOPG-Online-Erzeugung. D.h. bei interaktiven Anwendungen tritt eine kaum merkliche Verzögerung ein, während bei CPU-lastigen Programmen eine Erhöhung der Realzeit um bis zu Faktor $1,71 \cdot 50,43 = 86,24$ eintritt. In anderen Testkonfigurationen sind möglicherweise noch höhere Werte zu erwarten.

Problematischer hingegen ist der Speicherbedarf für die globale Spur. Bereits bei Testfall 3), der unter normalen Umständen innerhalb von $4,65s$ Realzeit beendet ist, wird eine globale Spur mit ca. 417,72 Mio. Einträgen und einem Speicherverbrauch von ca. 4,5 GB erzeugt.

6.4 Protokollextraktion

Im Folgenden wird gezeigt, wie bei Durchführung des Testfalls *ArgoUML* jeweils die Protokolle nach den vorgestellten Techniken extrahiert wurden. Von Interesse sind hierbei die CPU-Zeiten der einzelnen Verarbeitungsschritte und die Größen der verarbeiteten Graphen.

6.4.1 Durchführung

An der Erzeugung des OPG aus dem Roh-Graphen hat sich durch Einführung des ITS- und des Shuffle-Protokolls nichts geändert. Daher muss dieser Schritt auch nicht betrachtet werden. Für sämtliche Verarbeitungsschritte vom OPG zum ITS-, Shuffle- und Schnitt-Protokoll

Testfall	1)		2)		3)	
Globale Spur: Mio. Einträge	28,68		65,75		417,72	
Globale Spur: Megabytes	306,22		708,28		4.485,18	
Zeit in *s*	CPU	Real	CPU	Real	CPU	Real
Original-Code	1,32	1,38	1,90	1,98	4,36	4,65
DOPG (online)	16,31	17,18	37,56	39,22	222,38	229,81
Globale Spur	11,38	11,81	26,86	27,74	157,47	166,31
Summe	29,01	30,37	66,32	68,94	384,21	400,77
Zeitfaktor DOPG/Original	13,36	13,49	20,77	20,77	52,00	50,43
Zeitfaktor Glob./DOPG	1,65	1,64	1,68	1,67	1,69	1,71
Globale Spur: Mio. Einträge/s	2,52	2,43	2,45	2,37	2,65	2,51

Tabelle 6.1: Zeitmessungen (CPU-Zeit, Realzeit) für die Erzeugung der globalen Spur im Vergleich zur bisherigen Instrumentierung und zum Original-Code von *FindBugs 1.3.6*.

wurden jedoch Messungen der CPU-Zeit durchgeführt.

Außerdem war es wichtig zu zeigen, dass in realen Software-Systemen tatsächlich diejenigen Konstellationen aus betrachtetem Objekt und mehreren Thread vorliegen, wie sie in der Problemstellung für diese Diplomarbeit beschrieben wurden. Auf diese Weise wird die Nützlichkeit der entwickelten Lösungsansätze begründet.

6.4.2 Auswertung

6.4.2.1 Relevante Threads

Während der Ausführung des Testfalls wurden innerhalb der Anwendung ArgoUML insgesamt 15 Threads gestartet. Vier dieser Threads enthielten relevante Methodenaufrufe für die untersuchte Klasse `UMLClassDiagram`, zu sehen in Tabelle 6.2. Neben dem Haupt-Thread gibt es dort, wie in einer GUI-Anwendung üblich, eine „EventQueue“ (Thread-ID 2) zur Verarbeitung der Benutzerinteraktion. Der Thread mit der ID 6 dient offenbar dazu, das vom Benutzer erstelle UML-Diagramm auf inhaltliche Probleme zu untersuchen. Diese Funktion wurde im Testfall nicht genutzt, daher hat der Thread offensichtlich immer nur eine einzige atomare Operation des Objektes ausgeführt. Durch das Speichern des UML-Diagramms wurde augenscheinlich der Thread mit der ID 15 gestartet.

6.4.2.2 Performanz

Tabelle 6.2 enthält alle gemessenen Zeiten bei der Transformation eines OPG zu einem ITS-Protokoll bzw. zu einem Shuffle-Protokoll. Die Berechnung des Schnitt-Protokolls wurde separat gemessen. Auffällig ist, dass in der Tat die neu hinzugekommenen Schritte, d.h. Einlesen der globalen Spur, Shuffle- und Schnitt-Protokoll im Vergleich zu den anderen Schritten bemerkenswert viel Zeit benötigen. Insbesondere die Shuffle- und die Schnitt-Operation scheinen die Eigenschaft zu haben, ab einer gewissen Eingabegröße nicht mehr praktikabel berechenbar zu sein.

Betrachtet man Tabelle 6.2, so erkennt man die Zusammenhänge zwischen den Automatengrößen und den Laufzeiten. Ein Shuffle-Automat hat als Knotenzahl immer das Produkt der Knotenzahlen der beiden ursprünglichen Automaten. Eine Minimierung oder Bestimmung des Potenzautomaten konnte bei Shuffle-Automaten dieser Größenordnung (7680 Knoten) auch

Globale Spur: Megabytes	95,72
Globale Spur: Mio. Einträge	8,44
Vorgang	Zeit / s
Inline-Expansion	0,15
Globale Spur einlesen	70,85
ITS-Protokollautomaten	0,70
Summe ITS-Protokoll	71,70
Inline-Expansion	0,15
Threads extrahieren	0,11
Intra-Thread-Protokollautomaten	0,64
Thread 15 ⊔ Thread 1	0,20
$(\cdots)$ ⊔ Thread 2	20,20
$((\cdots)$ ⊔ $\cdots)$ ⊔ Thread 6	287,12
Summe Shuffle-Operationen	307,52
Summe Shuffle-Protokoll	308,42
Schnitt-Protokoll (min. DEA)	31.329,30

Tabelle 6.2: CPU-Zeiten für die Analyse von ArgoUML

Intra-Thread-Protokolle			
ID	Name	Knoten	Kanten
1	`main`	32	96
2	`AWT-EventQueue-0`	40	125
6	`CritiquingThread`	1	1
15	`ArgoSaveProjectThread`	6	10
Inter-Thread-Protokolle			
Shuffle-Protokoll (NEA)		7.680	67.520
ITS-Protokoll		38	108
Schnitt-Protokoll		335	1.329

Tabelle 6.3: Automatengrößen für ArgoUML

auf der vorhandenen, modernen Hardware nicht mehr durchgeführt werden. Somit erhöht sich die Anzahl der Knoten im Shuffle-Automaten mit jedem zusätzlichen Thread um den Faktor, welcher der Knotenzahl des Intra-Thread-Protokolls des hinzukommenden Threads entspricht.

Derselbe Effekt stellt sich bei der Bestimmung des Schnitt-Protokolls ein, da dieselbe Anzahl Knotenpaare überprüft werden muss. Im Ergebnis ist das Schnitt-Protokoll jedoch wieder relativ klein, enthält jedoch immer noch mehr als das Zehnfache an Knoten im Vergleich zum ITS-Protokoll.

6.5 Ergebnis

Die Evaluation hat gezeigt, dass die Extraktion eines ITS-Protokolls aus einer realen Software zwar zusätzlich Zeit kostet, aber dennoch praktikabel ist. Die einzige Voraussetzung ist, dass auf dem entsprechenden Rechner genügend Festplattenkapazität zur Speicherung der globalen Spur zur Verfügung steht.

Bei der Bestimmung des Shuffle- und des Schnitt-Protokolls ist deutlich geworden, dass das Analyseverfahren ab einer gewissen Anzahl relevanter Threads schnell die praktikablen Zeit- und Speicherkapazitäten eines Rechners überschreitet. Dafür hat sich zumindest an einem praktischen Beispiel gezeigt, dass eine Erweiterung der Protokollerkennung zur Unterstützung von Multithreading, wie sie in dieser Arbeit entwickelt wurde, einen nützlichen Beitrag zur Software-Analyse leistet. Eine Einschränkung besteht lediglich darin, dass die Anzahl der relevanten Threads nicht zu groß werden darf.

KAPITEL 7

Fazit

Dieses Kapitel stellt den Inhalt und die Ergebnisse noch einmal in einer knappen Zusammenfassung dar. Nach einer persönlichen Reflexion werden im Ausblick die verbleibenden Probleme erwähnt, deren Lösungen zu einer weiteren Verbesserung des in dieser Diplomarbeit entwickelten Verfahrens beitragen können.

Inhalt

7.1 Zusammenfassung **71**

7.2 Reflexion **72**

7.3 Ausblick **72**

7.1 Zusammenfassung

In der vorliegenden Diplomarbeit wurden Ansätze entwickelt, mit denen das bisherige, DOPG-basierte Protokollerkennungsverfahren des Bauhaus-Projektes zur Unterstützung von Multithreading-Anwendungen erweitert werden kann.

Dazu wurde zunächst gezeigt, wie sich das bisherige Verfahren verhält, wenn das betrachtete Objekt von mehr als einem Thread benutzt wird. Es stellte sich heraus, dass die Protokollerkennung für das betrachtete Objekt nur diejenigen Operationen berücksichtigt, die auch in demselben Thread ausgeführt werden, durch den das Objekt erzeugt wurde. In einem OPG wird der Kontrollfluss innerhalb der einzelnen Threads zwar korrekt dargestellt, es sind jedoch keine Informationen über das Interleaving-Verhalten der Threads enthalten.

Daher wurde ein neues Verfahren eingeführt, welches die globale, thread-übergreifende Spur in einem ITS-Graphen repräsentiert. Daraus kann entsprechend ein ITS-Protokoll gewonnen werden. Bedingt durch das Konstruktionsverfahren kann ein ITS-Graph – und damit ein ITS-Protokoll – allerdings auch Sequenzen enthalten, die dem Kontrollfluss innerhalb einzelner Threads widersprechen. Diese Sequenzen lassen sich durch Schnittmengenbildung mit dem Shuffle-Protokoll beseitigen. Dazu muss das Shuffle-Protokoll zunächst aus den Intra-Thread-Protokollen der einzelnen Threads berechnet werden.

Zur Evaluation der neuen Verfahren wurde die Performanz der hinzugekommenen Verarbeitungsschritte bei der Analyse realer Software-Systeme gemessen. Dabei stellte sich heraus, dass insbesondere das Verfahren zur Bestimmung des ITS-Protokolls ein brauchbares Zeitverhalten aufweist. Die Komplexität der Algorithmen zur Konstruktion des Shuffle- und Schnitt-Protokolls lässt hingegen nur eine eingeschränkte Verwendung im praktischen Einsatz zu.

7.2 Reflexion

Mit dieser Diplomarbeit wurde meiner Meinung nach eine interessante Problemstellung präsentiert, für die auch eine elegante Lösung gefunden wurde.

Die Anfertigung einer Implementierung auf Basis des Bauhaus-Projektes erfordert jedoch stets eine längere Einarbeitungszeit – ein Programmverstehen im Sinne unseres eigenen Forschungsgebietes. Da die vorhandenen Werkzeuge zum Teil eher prototypischen Charakter haben, gelingt es selten ohne Hindernisse, anhand der entsprechenden wissenschaftlichen Publikationen die Zusammenhänge der vorhandenen Implementierung zu verstehen.

Somit wurde ein großer Teil der Zeit dadurch verbraucht, Eigenheiten des vorhandenen Codes zu verstehen, Fehler zu entdecken und zu beseitigen oder nützliche Erweiterungen zu integrieren. Die zahlreichen Bugfixes haben keinen wissenschaftlichen Charakter, weshalb sie im schriftlichen Teil dieser Diplomarbeit auch bisher nicht erwähnt wurden.

Daher ist die Fallstudie, die zur Evaluation der neu eingeführten Verfahren dienen sollte, recht knapp ausgefallen. Durch eine größere Anzahl Testkandidaten wäre hier ein verlässlicheres Ergebnis zustande gekommen.

7.3 Ausblick

Das DOPG-basierte Protokollerkennungsverfahren lässt sich noch in vielfältiger Weise erweitern. Speziell in dieser Diplomarbeit sind auch Lösungsansätze entstanden, die gewissen Schwächen unterliegen. So ist es nicht gelungen, das ITS-Online-Konstruktion zu einem nützlichen Verfahren zu entwickeln. Daher muss nun der Speicherbedarf einer globalen Spur in Kauf genommen werden. Daher besteht weiterhin Bedarf, auch ein brauchbares ITS-Online-Verfahren zu entwickeln.

Für die Berechnung des Shuffle-Protokolls wurde ein naiver Algorithmus eingesetzt. Möglicherweise lässt sich auch hier durch Einsatz eines effizienteren Verfahrens eine Optimierung erreichen, durch die ein praktischer Einsatz der Protokollerkennung durch Kombination von ITS-, Shuffle- und Schnitt-Protokollen realistischer wird.

ANHANG A

Werkzeuge

Abbildung A.1 zeigt den vollständigen Datenfluss und alle beteiligten Werkzeuge zur Bestimmung des ITS-Protokolls, der Intra-Thread-Protokolle, des Shuffle- und des Schnitt-Protokolls eines Objektes.

A.1 Neu implementierte Werkzeuge

- `expand_routines`

  Dieses Werkzeug wurde zwar als neues Projekt angelegt, enthält aber keine neue Implementierung. Aus dem vorhandenen Werkzeug `remove_recursion` wurde lediglich der erste Schritt in das neue Werkzeug `expand_routines` verschoben. Die Implementierung musste jedoch erweitert werden, da sie bisher keine Multi-Calls, d.h. Call-Knoten mit mehreren ausgehenden Invoke-Kanten und keine Exceptions unterstützt hatte.

- `read_global_trace`

  Dieses Werkzeug implementiert den Algorithmus 5.1.

- `separate_threads`

  Dieses Werkzeug sucht, ausgehend vom Initial-Entry-Knoten eines expandierten OPG, die Call-Knoten aller Thread-Einstiegspunkte. Für jeden Thread wird die Menge der vom Einstiegspunkt aus erreichbaren Knoten in einer eigenen Sicht (View) dargestellt. Die Zahl der zu bearbeitenden Threads ist dadurch auf 63 begrenzt, da ein RFG maximal 64 Views enhalten kann, wobei ein View den ursprünglichen expandierten OPG enthält.

- `Shuffle`

  Diese Java-Klasse implementiert den Algorithmus zur Konstruktion eines Shuffle-Automaten aus zwei gegebenen NEAs. Sie basiert auf der Implementierung der bisher existierenden Klasse `Intersect`.

A.2 Angepasste Werkzeuge

- `Enhancer`

  Die Java-Klassen zur Instrumentierung und Graph-Konstruktion wurden zur Ausgabe der globalen Spur angepasst. Dazu wurde insbesondere die Klasse `GlobalTraceLogger` angelegt.

  Da die Erzeugung eines Objektes und die Festlegung der Speicheradresse zeitversetzt erfolgen (siehe Seite 17), musste für die Ausgabe der globalen Spur ein Zwischenspeicher

eingerichtet werden. Darin wird der Teil der globalen Spur aufbewahrt, der zeitlich zwischen der Erzeugung und der Adress-Registrierung eines Objektes liegt. Erst wenn die Adresse bekannt ist, wird die globale Spur ab der Erzeugung ausgegeben. Nur auf diese Weise kann das Create-Ereignis eines Objektes mit der entsprechenden Objekt-Adresse versehen werden.

- `make_deterministic`, `simplify_protocol`, `toedgemin`

  Diese Werkzeuge wurden so angepasst, dass sie auch die Protokolle mehrerer Threads nacheinander bearbeiten können, die in dem gegebenen OPG als einzelne Views enthalten sind.

- `remove_recursion`

  Dieses Werkzeug musste wie `expand_routines` zur Unterstützung von Multi-Calls und Exceptions erweitert werden. Außerdem musste auch hier die Anpassung zur Unterstützung von mehreren Threads in einzelnen Views vorgenommen werden.

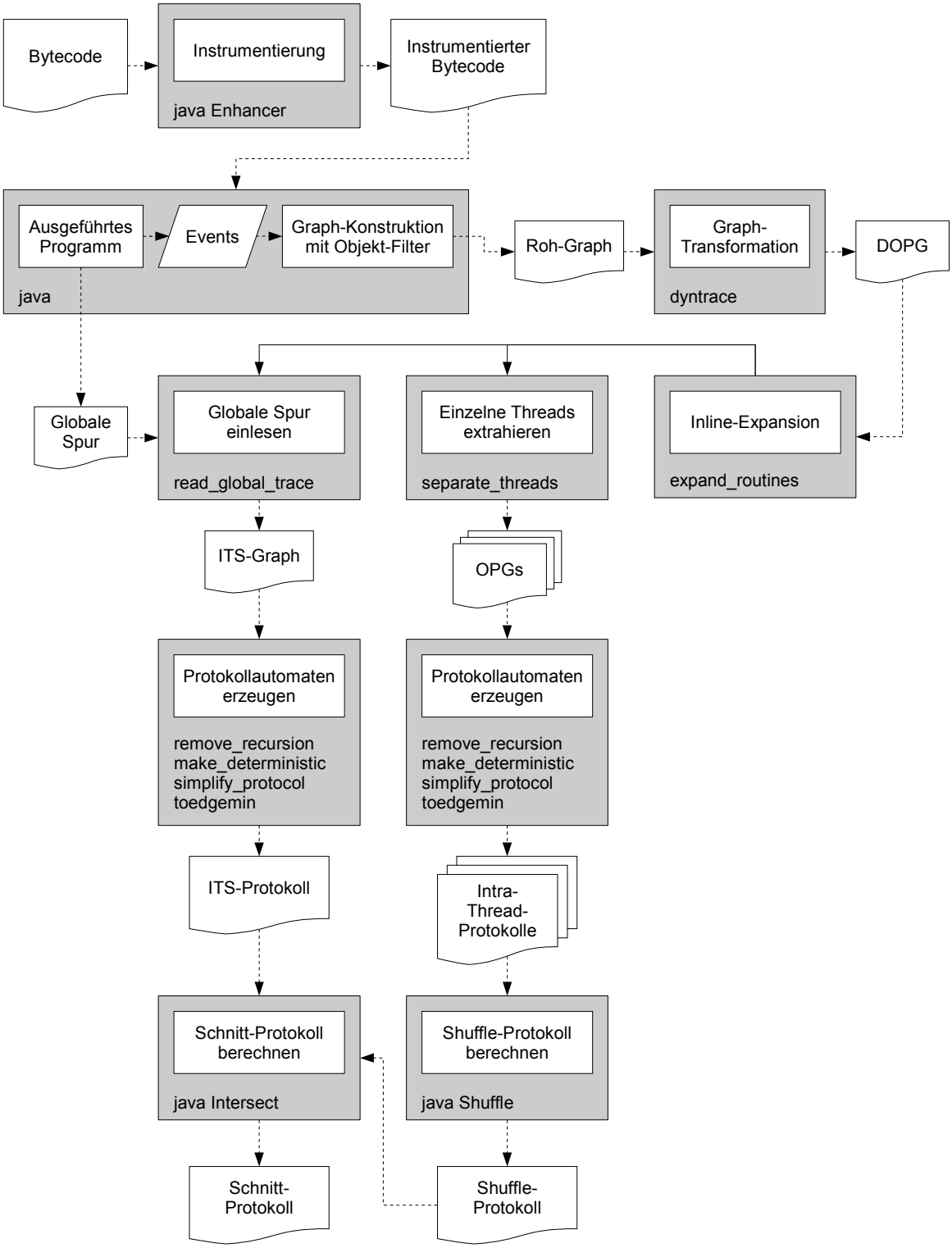


Abbildung A.1: Vollständiger Datenfluss zur Bestimmung des ITS-Protokolls, der Intra-Thread-Protokolle, des Shuffle- und des Schnitt-Protokolls eines Objektes

ANHANG B

Code-Varianten

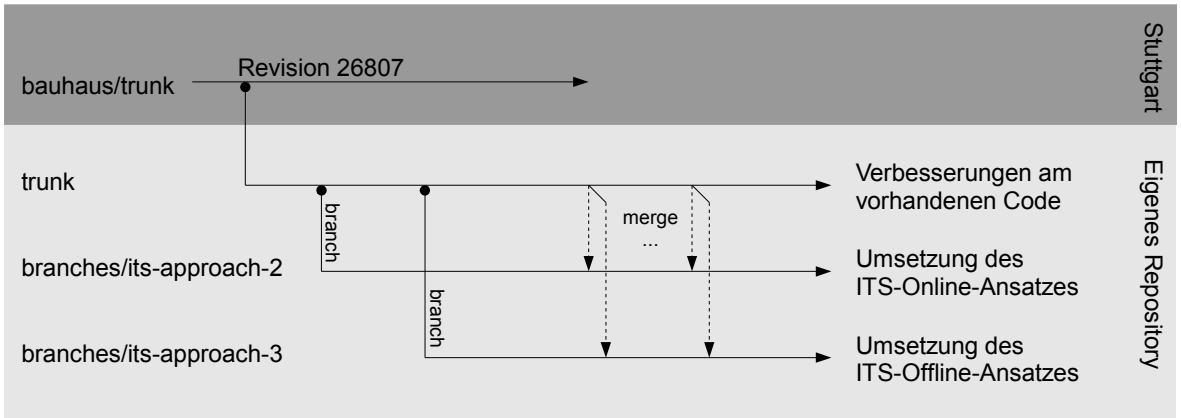


Abbildung B.1: Revisions-Graph des Subversion-Repositories

Alle Implementierungen in dieser Diplomarbeit basieren auf dem Bauhaus-Trunk der Revision 26807. Zum Kompilieren muss der Ada-Compiler `gnatpro-6.0.2` verwendet werden. Bedingt durch das ASM-Framework, welches für die Bytecode-Instrumentierung genutzt wird, muss das JDK 1.6 als Java-Umgebung eingerichtet sein.

Der Code zu dieser Diplomarbeit existiert in drei Varianten. Zu beachten ist, dass die eigentlich relevanten Implementierungen im Zweig `branches/its-approach-3` zu finden sind.

Inhalt der einzelnen Verzeichnisse:

- `trunk`

  enthält alle Änderungen, die aufgrund kleinerer Fehler in der vorhandenen Implementierung notwendig wurden und die nicht unmittelbar mit der Multithreading-Erweiterung zusammenhängen. Diese Änderungen wurden entsprechend regelmäßig in den Zweig `branches/its-approach-3` übertragen.

- `branches/its-approach-2`

  enthält die Modifikationen zur Umsetzung des ITS-Online-Ansatzes. Da sich dieser als ungeeignet herausgestellt hat, wurde der Code nicht weiter gepflegt und dient nur zur Anschauung.

- `branches/its-approach-3`

  enthält die Umsetzung des ITS-Offline-Ansatzes und aller weiteren in Anhang A beschriebenen Erweiterungen.

ANHANG C

Vollständige Listings

C.1 Code zur Veranschaulichung der OPG-Definition

Dieses Listing ist die vollständige Version von Listing 3.1.

```
package bs;

import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        new Main().main();
    }

    Stack readStack() {
        Stack result = new Stack();
        result.push("foo");
        result.push("bar");
        return result;
    }

    void main() {
        int i = 0;
        Stack s1 = new Stack();
        Stack s2 = readStack();
        reverse(s2, s1);
        do {
            s1.pop();
            i = i + 1;
        } while (!s1.isEmpty());
    }

    void reverse(Stack from, Stack to) {
        while (!from.isEmpty()) {
            to.push(from.pop());
        }
    }
}
```

Listing C.1: Code zur Veranschaulichung der OPG-Definition

C.2 Beispiel für eine Multithreading-Anwendung

Dieses Listing ist die vollständige Version von Listing 3.4.

```
package bs;

public class Main {
    public static void main(String args[]) {
        MySocket socket = new MySocket();

        // Initialisierung
```

```
        SocketConnectThread thread = new SocketConnectThread(socket);
        thread.start();

        initGUI();

        // Warten, bis Initialisierung abgeschlossen ist
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Lesevorgang starten
        int result;
        do {
            result = socket.read();
        } while (result != -1);
    }

    private static void initGUI() {
        // hier z.B. Initialisierung der GUI
    }
}

class SocketConnectThread extends Thread {
    private MySocket socket;
    SocketConnectThread (MySocket socket) {
        this.socket = socket;
    }
    public void run() {
        socket.connect();
    }
}

class MySocket {
    private int bytesAvailable = 0;
    public void connect() {
        bytesAvailable = 5;
    }

    public int read() {
        if (bytesAvailable > 0) {
            bytesAvailable--;
            return 1;
        } else {
            return -1;
        }
    }
}
```

Listing C.2: Beispiel für eine Multithreading-Anwendung

C.3 Erzeuger-Verbraucher-System

Dieses Listing ist die vollständige Version von Listing 5.1.

```
package bs;
import java.util.Random;
public class Main {
    public static final int ELEMENT_COUNT = 1000;
    private static Random random = new Random(12);
    public static void randomSleep(int ms) {
        try {
            Thread.sleep(random.nextInt(ms));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void main(String args[]) {
        Queue queue = new Queue();
        queue.setMaxSize(100);
        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);
        try {
            producer.start();
            Thread.sleep(10);
            consumer.start();
            producer.join();
            consumer.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Producer extends Thread {
    public Queue queue;
    public Producer(Queue queue) {
        this.queue = queue;
    }
    public void run() {
        int i = 1;
        while (i <= Main.ELEMENT_COUNT) {
            if (queue.size() < 2) {
                Main.randomSleep(5);
                queue.push("A" + i);
                i++;
            }
            Main.randomSleep(5);
        }
    }
}

class Consumer extends Thread {
    public Queue queue;
    public Consumer(Queue queue) {
        this.queue = queue;
    }
    public void run() {
        while (true) {
            if (!queue.isEmpty()) {
                Main.randomSleep(5);
                String s = queue.pop();
                if (s.equals("A" + Main.ELEMENT_COUNT)) {
                    break;
                } else {
                    System.out.println("pop():␣" + s);
                }
            }
            Main.randomSleep(5);
        }
    }
}

class Queue {
    java.util.LinkedList<String> list = new java.util.LinkedList<String>();
    public synchronized void setMaxSize(int maxSize) {
        // Dummy-Methode
    }
    public synchronized void push(String s) {
        list.offer(s);
    }
    public synchronized String pop() {
        return list.poll();
    }
    public synchronized boolean isEmpty() {
        return list.isEmpty();
    }
    public synchronized int size() {
```

```
        return list.size();
    }
}
```

Listing C.3: Erzeuger-Verbraucher-System

ABBILDUNGSVERZEICHNIS

1.1 Beispiel für das Protokoll eines Dokumentes einer Textverarbeitung 3

3.1 Legende für Objektprozessgraphen 12

3.2 Beispiel-OPG für Objekt `s1` aus Listing 3.1 14

3.3 Werfen und Fangen einer Exception, dargestellt im OPG 15

3.4 Datenfluss bei der Konstruktion des DOPG [Qua07] 21

3.5 Transformation eines OPG zum Protokollautomaten 23

3.6 Protokollautomat, erzeugt aus dem Beispiel-OPG in Abbildung 3.2 24

3.7 Falsches und erwartetes Protokoll für Listing 3.4 26

3.8 Falsches Protokoll ohne OPG-Projektion für Listing 3.4 28

4.1 Petri-Netz für das Protokoll von `MySocket` in Listing 3.4 33

5.1 Schematische Darstellung eines OPG mit zwei Threads und Überlagerung durch einen ITS-Graphen mit $N_{ITS} \neq N$. 38

5.2 Datenfluss bei Verwendung des ITS-Online-Verfahrens 39

5.3 Ausschnitt aus einem Roh-Graphen (a), nach der Label-Transformation (b) und nach der ITS-Transformation (c). 41

5.4 Datenfluss bei Vewendung des ITS-Offline-Verfahrens 44

5.5 Aus Listing 3.4 generierter ITS-Graph 47

5.6 OPG (vereinfacht) mit ITS-Graph und unzulässigem ITS-Pfad p' 50

5.7 Automaten für Intra-Thread-Protokolle und Shuffle 53

5.8 Venn-Diagramm zur Veranschaulichung von $L = L_{ITS} \cap L_{\sqcup\!\sqcup}$. 54

5.9 Erzeuger-Verbraucher-System: Unzureichende Protokollerkennung mit dem bisherigen Verfahren 56

5.10 Erzeuger-Verbraucher-System: ITS-Protokoll 57

5.11 Erzeuger-Verbraucher-System: Shuffle-Protokoll 59

5.12 Erzeuger-Verbraucher-System: Schnitt-Protokoll 61

6.1 CPU-Zeiten und ihre Zusammensetzung laut Tabelle 6.1: Original-Code, DOPG-Erzeugung (online) und Erzeugung der globalen Spur 67

A.1 Vollständiger Datenfluss zur Bestimmung des ITS-Protokolls, der Intra-Thread-Protokolle, des Shuffle- und des Schnitt-Protokolls eines Objektes 75

B.1 Revisions-Graph des Subversion-Repositories 77

LISTINGS

3.1 Code zur Veranschaulichung der OPG-Definition 14
3.2 Java-Bytecode vor der Instrumentierung 18
3.3 Instrumentierter Java-Bytecode 19
3.4 Beispiel für eine Multithreading-Anwendung 25
5.1 Erzeuger-Verbraucher-System 55
C.1 Code zur Veranschaulichung der OPG-Definition 79
C.2 Beispiel für eine Multithreading-Anwendung 79
C.3 Erzeuger-Verbraucher-System 80

LITERATURVERZEICHNIS

[BH99] Bogda, J. ; Hölzle, U.: Removing Unnecessary Synchronization in Java. In: *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*. Denver, CO, Nov. 1999

[BPSH05] Boroday, S. ; Petrenko, A. ; Singh, J. ; Hallal, H.: Dynamic analysis of java applications for multithreaded antipatterns. In: *ACM SIGSOFT Software Engineering Notes* 30 (2005), Nr. 4, S. 1–7

[CLL+02] Choi, J.D. ; Lee, K. ; Loginov, A. ; O'Callahan, R. ; Sarkar, V. ; Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: *ACM SIGPLAN Notices* 37 (2002), Nr. 5, S. 258–269

[CSV02] Câmpeanu, C. ; Salomaa, K. ; Vágvölgyi, S.: Shuffle Quotient and Decompositions. In: *Developments in Language Theory – 5th International Conference, DLT 2001 Wien, Austria, July 16-21, 2001 Revised Papers*. Berlin / Heidelberg : Springer, 2002

[ECH+01] Engler, D. ; Chen, D.Y. ; Hallem, S. ; Chou, A. ; Chelf, B.: Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In: *ACM SIGOPS Operating Systems Review* 35 (2001), Nr. 5, S. 57–72

[Eis98] Eisenbarth, T.: *GropiusSE, Eine Resource Flow Graph Bibliothek in Ada95 für das Speichern und Aufbereiten von Reengineeringinformationen. Studienarbeit Nr. 1663*. Universität Stuttgart, 1998

[EKV05] Eisenbarth, T. ; Koschke, R. ; Vogel, G.: Static object trace extraction for programs with pointers. In: *Journal of Systems and Software* 77 (2005), Nr. 3, S. 263–284

[Haa04] Haak, D.: *Werkzeuggestützte Herleitung von Protokollen*, Universtität Stuttgart, Diplomarbeit, 2004

[HMU02] Hopcroft, J. E. ; Motwani, R. ; Ullman, J. D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. 2., überarbeitete Auflage. München : Pearson Studium, 2002

[KGW98] Koschke, R. ; Girard, J.F. ; Würthner, M.: An intermediate representation for integrating reverse engineering analyses. In: *Fifth Working Conference on Reverse Engineering*, 1998, S. 241–250

[Kos08] Koschke, R.: *Zehn Jahre WSR – Zwölf Jahre Bauhaus*. Universität Bremen, 2008

[LK06] Lo, D. ; Khoo, S.-C.: QUARK: Empirical assessment of automaton-based specification miners. In: *Proceedings of 13th Working Conference on Reverse Engineering*, 2006, S. 21–60

[PW02] Priese, L. ; Wimmel, H.: *Theoretische Informatik: Petri-Netze.* Berlin, Heidelberg, New York : Springer-Verlag, 2002

[QK07] Quante, J. ; Koschke, R.: Dynamic protocol recovery. In: *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, S. 219–228

[QK08] Quante, J. ; Koschke, R.: Dynamic object process graphs. In: *The Journal of Systems & Software* 81 (2008), Nr. 4, S. 481–501

[Qua07] Quante, J.: Online construction of dynamic object process graphs. In: *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2007, S. 113–122

[Sta03] Stallings, W.: *Betriebssysteme.* 4., überarbeitete Auflage. München : Pearson Studium, 2003

[TB09] Tanenbaum, A.S. ; Baumgarten, U.: *Moderne Betriebssysteme.* 3., aktualisierte Auflage. München : Pearson Studium, 2009

[Ull06] Ullenboom, C.: *Java ist auch eine Insel.* 5., aktualisierte und erweiterte Auflage. Bonn : Galieo Press, 2006

[Vog06] Vogel, G.: Statische Extraktion von Protokollen. In: *Workshop on Software Reengineering*, Mai 2006

[Wim08] Wimmel, H.: *Entscheidbarkeit bei Petri-Netzen.* Berlin, Heidelberg : Springer-Verlag, 2008

[WML02] Whaley, J. ; Martin, M.C. ; Lam, M.S.: Automatic Extraction of Object-Oriented Component Interfaces. In: *ACM SIGSOFT Software Engineering Notes* 27 (2002), Nr. 4, S. 218–228