Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet für Verteilte Systeme und Betriebssysteme

Bachelorarbeit

Parallelisierung von HRU-Safety-Analysen

Vorgelegt von:
Felix Neumann

Betreuer:	Prof. Dr.-Ing. habil. Winfried E. Kühnhauser Dipl-Inf. Anja Fischer
Studiengang:	Informatik 2006
Eingereicht:	Ilmenau, den 02. November 2009

Version zur Veröffentlichung

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Hinweise

Diagramme zur Software-Architektur folgen der UML 1.4-Notation. Namen von Teilen der Software, wie Klassen, Variablen, Methoden und ähnliches, sind in `Schreibmaschinenschrift` gesetzt.

KAPITEL 1

Einleitung

1.1 Themengebiet und Motivation

In der IT-Sicherheit spielen Zugriffsschutzsysteme eine entscheidende Rolle. Sie gehören zu den wichtigsten Mechanismen, wenn Vertraulichkeit und Integrität von Dokumenten gewahrt werden sollen, und gewährleisten dies selbst dann noch, wenn der Angreifer bereits Zugang zum IT-System hat.

Umso wichtiger ist es, den Zugriffsschutz eines gegebenen IT-Systems vollständig zu kontrollieren. Doch verwenden die heute weit verbreiteten Unix-artigen wie auch Windows-basierten Betriebssysteme eine diskrete Zugriffssteuerungspolitik, bei der jeder Nutzer die Möglichkeit hat, Rechte an den ihm zugeordneten Dokumenten zu verändern. Durch diese Dynamik können sich Zugriffsrechte an Stellen ausbreiten, an denen die zugrunde liegenden Sicherheitsanforderungen dies untersagen. Analysetechniken wie die *Safety-Analyse* im formalen und ausdrucksstarken HRU-Kalkül [HRU76] können solche Rechteausbreitungen aufspüren.

Bei der HRU-Safety-Analyse wird ein Zustand eines Zugriffssteuerungssystems betrachtet. Kann sich ausgehend von diesem Zustand ein bestimmtes Recht nicht ausbreiten, heißt er *HRU-Safe* bezüglich diesen Rechtes. Allerdings ist diese Eigenschaft im allgemeinen Fall nicht entscheidbar. Ein Weg, dem zu begegnen, besteht in der *Suche* nach solchen Rechteausbreitungen. Doch auch diese ist sehr rechenintensiv: So benötigten die Analysen von Verzeichnissen realer, privat genutzter Linux-Systeme mehrere Minuten [Joh09]; die dabei betrachteten Zugriffssteuerungsmatrizen umfassten weniger als eine halbe Millionen Zellen. Schon bei der Analyse mittlerer Serversysteme wird dieser Wert um Größenordnungen überschritten [AFK09].

Diese Bachelorarbeit wird die Implementierung eines Verfahrens zur Suche nach Rechteausbreitungen in einem gegebenen HRU-Modell betrachten und eine skalierbare Methode erarbeiten, um die Performanz der Analyse zu steigern. Sie wird sich dabei der Technik der Parallelisierung bedienen. Der zunehmende Trend, die Hardware handelsüblicher Computersysteme mit mehreren Prozessorkernen auszustatten, unterstützt dies.

1.2 Zielstellung

Um die Performanz der HRU-Safety-Analyse zu steigern, soll ihr *Durchsatz* erhöht werden. Bei der Suche nach Zuständen, welche eine Rechteausbreitung enthalten, wird ein Suchbaum erkundet – in Bezug auf diesen Baum bedeutet das Erhöhen des Durchsatzes, dass pro Zeiteinheit eine höhere Zahl an Knoten erkundet werden soll. Auf diese Weise kann ein Algorithmus (beispielsweise ein heuristisches Verfahren) in der selben Zeit einen breiteren oder tieferen Teil des Suchbaums analysieren. Dies ist Ziel der Bachelorarbeit.

Wie oben beschrieben, soll die Performanzsteigerung dabei über die Möglichkeiten der Parallelisierung erzielt werden. Zur Aufwandsbeschränkung wird sich die Arbeit auf nicht-verteilte Systeme konzentrieren, welche Parallelität mittels mehrerer Prozessoren

oder Prozessorkerne erzielen. Die spätere Erweiterung auf verteilte Systeme muss dabei stets möglich bleiben.

Neben der Erarbeitung der Parallelisierungskonzepte soll auch eine Beispielimplementierung entstehen. Als Grundlage hierzu dient die Software *I-Graphoscope* [Amt08, Joh09, Rin09] – dies ist ein Werkzeug zur Analyse von HRU-Modellen –, sowie eine bereits realisiertes Analyseverfahren. Die Konzepte und Algorithmen der Arbeit sollen unabhängig von diesem Verfahren entstehen.

1.3 Gliederung

Die zum Verständnis der Arbeit nötigen Grundlagen legt das Kapitel 2. Hierzu beschreibt es zuerst die HRU-Safety-Analyse und führt den formalen HRU-Kalkül ein. Anschließend stellt es *I-Graphoscope* vor und erläutert für diese Arbeit wesentliche Teile der Softwarearchitektur. Ein drittes Unterkapitel erarbeitet Möglichkeiten der Parallelisierung und präsentiert ein formales Speichermodell.

An das Grundlagenkapitel schließt sich die Problemanalyse an (Kapitel 3). Sie leitet aus der obigen Zielstellung konkrete Anforderungen her und sucht dann nach Ansatzpunkten zur Parallelisierung.

Diese nutzt Kapitel 4 zum Entwurf der Software. Es stellt hierzu eingangs ein allgemeines Konzept vor, aus dem nachfolgend zentrale Algorithmen sowie die Softwarearchitektur abgeleitet werden.

Die Implementierung des Entwurfs beschreibt Kapitel 5, das die Architektur zu einem Feinentwurf konkretisiert und Detaillösungen beschreibt.

Um die Leistung der Arbeit zu bewerten, erarbeitet die Evaluierung (Kapitel 6) eingangs Metriken und Methoden für konkrete Messungen an der implementierten Software. Die Ergebnisse dieser Messungen werden präsentiert und diskutiert; ebenso setzt sich ein Teil des Kapitels kritisch mit den Methoden der Arbeit auseinander. Ein anschließender Ausblick empfiehlt Richtungen zur weiteren Forschung.

KAPITEL 2

Grundlagen

Zum Verständnis der im weiteren Verlauf der Arbeit besprochenen Themen liefert dieses Kapitel Grundlagenwissen.

Es geht dabei zuerst auf die theoretischen Hintergründe ein, indem es im Abschnitt 2.1 den HRU-Kalkül und die damit verbundene Safety-Eigenschaft beschreibt. In einem zweiten Schritt bringt Abschnitt 2.2 relevante Teile der Software *I-Graphoscope* näher, welche Gegenstand der vorliegenden Arbeit ist. Zuletzt erläutert Abschnitt 2.3 die Parallelisierungskonzepte in der hier eingesetzten Programmiersprache *Java*.

2.1 HRU-Safety-Analyse

Unterliegt ein IT-System bestimmten Sicherheitsanforderungen, werden die Entwickler des Systems Regeln entwerfen, durch deren Einhaltung das System zugleich auch die Sicherheitsanforderungen erfüllt. Die Gesamtheit dieser (informellen) Regeln ist die *Sicherheitspolitik*.

Um Analysen und Beweise anhand solcher Sicherheitspolitiken durchführen zu können, bedarf es eines formalen Modells. Im von [HRU76] beschriebenen HRU-Kalkül lassen sich eben solche formalen Modelle von Sicherheitspolitiken angeben; der Kalkül zeichnet sich dabei durch seine Ausdruckskraft aus.

Eine für diese Arbeit wesentliche Sicherheitseigenschaft von Modellen im HRU-Kalkül ist die *HRU-Safety*, welche im Allgemeinen jedoch nicht entscheidbar ist. Dieser Tatsache kann beispielsweise mit Heuristiken begegnet werden.

Der vorliegende Abschnitt stellt die formale Definition und Semantik der Modelle im HRU-Kalkül vor, führt nachfolgend die Safety-Eigenschaft für jene HRU-Modelle ein und betrachtet schließlich zu dieser Safety-Eigenschaft ein heuristisches Analyseverfahren.

2.1.1 HRU-Modell

Das HRU-Modell wird hier übereinstimmend mit der in Vorgängerarbeiten gebräuchlichen Notation nach [Kü09] definiert.

Definition 2.1 (HRU-Modell).
Sei R eine endliche Rechtemenge, A eine Menge anwendungsspezifischer Operationen, S eine unendliche Subjekt- und O eine unendliche Objektmenge. Ein HRU-Modell ist ein deterministischer Zustandsautomat (Q, Σ, δ, q_0) mit

- $Q = \mathcal{P}(S) \times \mathcal{P}(O) \times M$, die Menge aller Zustände, dabei ist $M = \{m \mid m : S \times O \to \mathcal{P}(R)\}$ die Menge aller Zugriffssteuerungsmatrizen;
- $\Sigma = A \times (S \cup O)^k$, die Eingabemenge,
- $\delta : Q \times \Sigma \to Q$ als Zustandsübergangsfunktion, und
- $q_0 \in Q$ als initialer Zustand.

Jeder Zustand eines HRU-Modells räumt also einer Menge von Subjekten Rechte auf eine Menge von Objekten ein. Über die anwendungsspezifischen Operationen, welche zusammen mit einer Parameterliste als Eingabe für die Zustandsübergangsfunktion dienen, lässt sich das Modell in einen neuen Zustand überführen. Der Aufbau der Übergangsfunktion ist wie folgt definiert:

Definition 2.2 (Zustandsübergangsfunktion).
Gegeben sei ein HRU-Modell (Q, Σ, δ, q_0) zu den Mengen R, A, S und O wie in Definition 2.1. Die Beschreibung der Zustandsübergangsfunktion δ erfolgt durch partielle Definition; hierzu wird für jedes $\alpha \in A$ notiert:

- ein Bedingungsteil der Form $r_1 \in m(x_{s_1}, x_{o_1}) \wedge r_2 \in m(x_{s_2}, x_{o_2}) \wedge \ldots \wedge r_l \in m(x_{s_l}, x_{o_l})$, sowie
- ein Anwendungsteil, bestehend aus einer Reihe von HRU-Primitiven $p_1, \ldots, p_n$. Die möglichen Primitive mit der jeweils offensichtlichen Semantik sind:
  - `enter` r `into` $m(x_s, x_o)$,
  - `delete` r `from` $m(x_s, x_o)$,
  - `create subject` x_s,
  - `create object` x_o,
  - `destroy subject` x_s sowie
  - `destroy object` x_o.

Ein Zustandsübergang, ausgeführt durch Aufruf einer anwendungsspezifischen Operation $\alpha(x_1, \ldots, x_k)$, überführt bei Zutreffen aller Bedingungen im Bedingungsteil den Zustandsautomaten gemäß der spezifizierten HRU-Primitive in den Folgezustand.

2.1.2 HRU-Safety

Da Modelle im HRU-Kalkül formal notiert sind, können sie mit mathematischen Mitteln analysiert werden. Hier interessiert zumeist die Erreichbarkeit bestimmter Zustände, insbesondere, ob ein fest gewähltes Recht in eine Matrix eingetragen werden kann. [HRU76] benennt folgende Sicherheitseigenschaft, die hier nach [Kü09] definiert ist:

Definition 2.3 (HRU-Safety eines Zustandes).
Ein Zustand q eines gegebenen HRU-Modells heißt HRU-Safe in Bezug auf ein Recht r genau dann, wenn beginnend mit q keine Folge von Zuständen existiert, die r in eine Matrixzelle einträgt, welche dieses r nicht schon in q enthält.

Zur Entscheidbarkeit lässt sich nach [HRU76] und [Joh09] zusammenfassen:

Satz 2.4 (Entscheidbarkeit der HRU-Unsafety).
Zu einem gegebenen Zustand eines HRU-Modells und einem gegebenen Recht ist die HRU-Unsafety *semi-entscheidbar*: zwar existiert kein Algorithmus zur Lösung des Safety-Problems für allgemeine HRU-Modelle, doch lässt sich die Menge der Zustände des Modells, die für das gegebene Recht unsafe sind, rekursiv aufzählen.

Daraus folgt, dass es möglich ist, nach unsicheren Zuständen zu suchen, um so beispielsweise die Safety eines Modells zu widerlegen. Derartige Analysen bilden den Hintergrund der restlichen Arbeit; mit ihnen beschäftigt sich auch der nachfolgende Teil.

2.1.3 Analysen zur HRU-Safety

Die Safety-Eigenschaft der HRU-Modelle ist im Allgemeinen nur semi-entscheidbar, zur Untersuchung solcher Modelle können daher nur unsichere Zustände aufgezählt werden. Ein Weg, dies zu bewerkstelligen, besteht in der *Suche* nach solchen Zuständen; er wird nachfolgend beschrieben. Auf diesem Konzept baut [Joh09] auf und legt einen Algorithmus zur heuristischen Suche dar, welcher ebenfalls nahegebracht wird.

Darstellung als Suchproblem. Um in einem gegebenen HRU-Modell (Q, Σ, δ, q_0) ausgehend von einem Zustand q nach einem Zustand zu suchen, der die HRU-Safety von q bezüglich r widerlegt, lässt sich das Modell als Zustandsgraph $G = (V, E)$ notieren, wobei die Knotenmenge V der Menge aller erreichbaren Zustände aus Q entspricht[1], und $(q_a, q_b) \in E \Leftrightarrow \exists(\alpha, x) \in \Sigma : \delta(q_a, (\alpha, x)) = q_b$. Darauf aufbauend ist das Suchproblem zur Safety-Analyse wie folgt beschrieben:

Definition 2.5 (Suchproblem zur HRU-Safety).
Gegeben ist zu einem HRU-Modell der Zustandsgraph G. Das Suchproblem zur HRU-Safety eines Zustandes q bezüglich eines Rechtes r besteht im Finden eines gerichteten Weges in G von q zu einem Zustand, welcher eine Ausbreitung des Rechtes r gegenüber q enthält.

Der Graph G entspricht dabei dem Suchgraphen. Zu beachten ist, dass es sich hier keineswegs um einen Such*baum* handeln muss, da (gerichtete wie auch ungerichtete) Kreise durchaus möglich sind.

[Joh09] untersucht zur Lösung des Problems mehrere Suchverfahren und betrachtet dabei zuerst Vertreter *klassischer Suchalgorithmen* wie die Tiefen- oder Breitensuche, welche ohne jegliche problemspezifische Informationen ablaufen. Die Arbeit stellt jedoch fest, dass solche Algorithmen komplexe Graphen nicht in annehmbarer Zeit durchsuchen können, und greift daher auf *heuristische Suchalgorithmen* zurück.

Heuristischer Ansatz. Heuristische Suchalgorithmen nutzen nach [Joh09] problemspezifische Informationen und sind so in der Lage, zielgerichtet zu suchen. Allen Heuristiken gemein ist dabei der Einsatz einer *heuristischen Funktion* $f : V \to \mathbb{N}$, welche – nach der Heuristik eigenen Maßstäben – einen Knoten bewerten kann; die Heuristik ist so in der Lage, den günstigsten, als nächstes zu untersuchenden Knoten zu bestimmen.

Die genannte Arbeit stellt den *Leak-Search*-Algorithmus vor, welcher das Suchproblem zur HRU-safety zu einem konkreten Zustand und Recht heuristisch bearbeitet. Der Algorithmus verfügt dabei über eine obere Laufzeitschranke und wurde unter der Anforderung der Speichereffizienz entworfen.

Der *Leak-Search*-Algorithmus ist für diese Arbeit relevant, da er ein funktionierendes und implementiertes Analyseverfahren zur HRU-Safety darstellt. Er wird später beispielhaft zur Untersuchung einiger Laufzeitmerkmale der *I-Graphoscope*-Simulationsumgebung eingesetzt werden und als Ausgangspunkt für die Implementierung der in der Arbeit entworfenen Konzepte dienen.

Arbeitsweise des Algorithmus. Gegeben sei hier erneut ein HRU-Modell (Q, Σ, δ, q_0) mit einer Menge A der anwendungsspezifischen Operationen und der Rechtemenge R. Als Eingabe für *Leak-Search* dient zusammen mit dem genannten Modell ein Recht $r \in R$.

[1] Da ein *Knoten* des Zustandsgraphs einem *Zustand* des HRU-Modells entspricht, werden beide Begriffe im Folgenden austauschbar verwendet.

Der Algorithmus sucht nun auf einem Suchbaum[2,3] $G = (V, E)$, wobei er in jedem Suchschritt einen Knoten des Baumes erkundet. Er führt eine Suchliste L, in der all jene Knoten abgelegt werden, die zwar bereits erkundet, jedoch noch nicht vollständig expandiert wurden[4]. Initial ist $L = \{q_0\}$.

In jedem Suchschritt wählt *Leak-Search* aus L einen Zustand q aus, bestimmt eine geeignete Eingabe für den Zustandsautomaten und berechnet den Zustandsübergang. Der so resultierende Zustand q' wird auf eine mögliche Rechteausbreitung überprüft; bei Erfolg terminiert der Algorithmus unter Ausgabe des gefundenen Zustandes, andernfalls wird q' zu L hinzugefügt.

Eine (in *I-Graphoscope* implementierte) Variante des Algorithmus terminiert erst bei Erreichen einer eingegebenen Höchstschrittzahl. Bis dahin vermerkt er alle gefundenen Zustände, die eine Rechteausbreitung enthalten.

2.2 Die Software *I-Graphoscope*

Die Arbeit baut auf eine bestehende Applikation auf: *I-Graphoscope* ist ein Werkzeug zur Visualisierung und Analyse der oben beschriebenen HRU-Modelle. Mit seinen Funktionen lassen sich einzelne Zustände wie auch die gesamte Dynamik solcher Modelle untersuchen. Die Software ist in Java geschrieben.

I-Graphoscope wurde von [Amt08] entwickelt, um damit aus gegebenen HRU-Modellen Informationsflussgraphen zu generieren und auf Verbandsklassen zu reduzieren. Damit auch die Dynamik solcher Modelle analysiert werden kann, erweiterte [Rin09] die Software nachfolgend um einen Simulator, mit dessen Hilfe die Zustandsübergänge eines gegebenen HRU-Modells berechnet werden können. Schließlich nutzte [Joh09] diesen Simulator, um eine erste Heuristik (siehe Abschnitt 2.1.3) zu implementieren.

Insbesondere die letzten beiden Komponenten, Simulator und Heuristik, werden eine zentrale Rolle in der vorliegenden Arbeit spielen.

Dieser Abschnitt gibt einen Überblick über die Architektur der Software, mit dem Ziel, einerseits die für diese Arbeit wesentlichen Komponenten zu erläutern, und andererseits deren Einbettung in die Systemumgebung nahezubringen.

2.2.1 Gesamtarchitektur

Die Architektur von *I-Graphoscope* gliedert sich in eine Reihe von Paketen, wie in Abbildung 2.1 dargestellt. Dies sind zunächst `gui`, welches Fenster und Steuerelemente implementiert, `core`, in dem Kernkomponenten und -algorithmen enthalten sind, sowie `model`, dessen Klassen HRU-Modelle, Informationsflussgraphen und andere fachliche Objekte modellieren.

Das `gui`-Paket. Klassen und Algorithmen zur grafischen Darstellung der Oberfläche werden in diesem Paket zusammengetragen, hier liegen also die Zuständigkeiten für das Erstellen der Steuerelemente, Aktualisieren der dargestellten Inhalte und ähnliches. Für alle Fenster und Dialoge der Anwendung enthält das Paket je eine Klasse.

[2] Mit *Suchbaum* ist der oben definierte Baum gemeint, und nicht etwa eine Datenstruktur, welche zur Laufzeit angelegt wird.

[3] Die *Baum*-Eigenschaft von G stellt hier eine Vereinfachung dar. Die Möglichkeit zweier verschiedener Knoten, welche identische Zustände repräsentieren, wird von [Joh09] in Kauf genommen.

[4] Ein Knoten heißt expandiert, wenn all seine Nachfolger erkundet wurden.

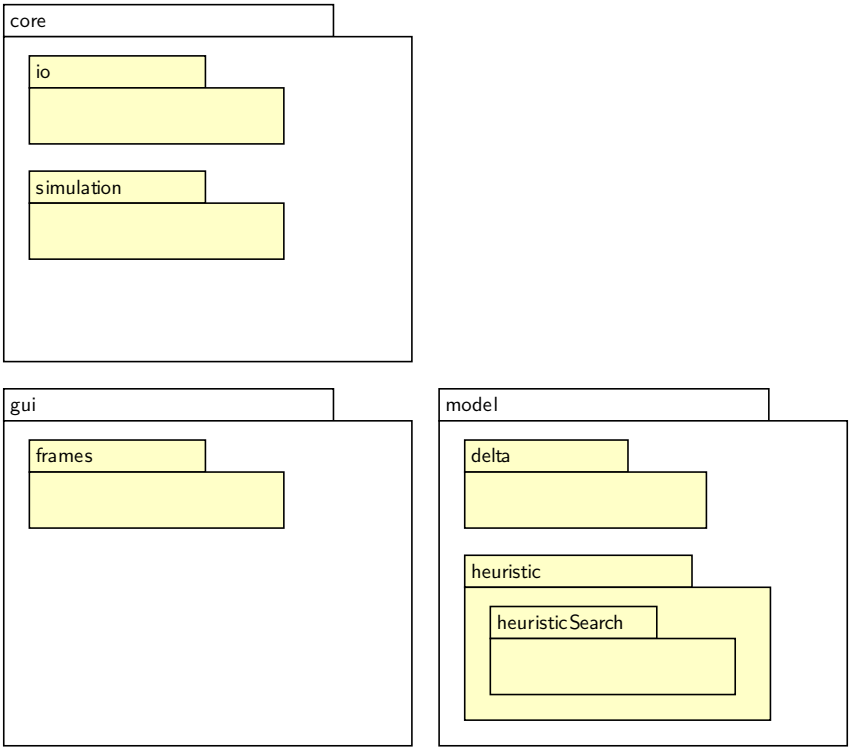


Abbildung 2.1: Die Pakete der Architektur.

Das `core`-Paket. Dieses Paket beinhaltet Klassen für verschiedenste Aufgaben, hierzu gehören zentrale Programmfunktionen, Im- und Exportfunktionen (Unterpaket `io`), sowie für diese Arbeit wesentliche Klassen zur Simulation der Dynamik von HRU-Modellen (Unterpaket `simulation`).

Besonders hervorzuheben sind hier zwei Klassen. Zum einen ist dies `IGraphToolMain`, welche wesentliche Programmabläufe steuert und im Sinne dieser Arbeit die Schnittstelle zwischen den Analyse-Komponenten und den restlichen Teilen von *I-Graphoscope* darstellt. [Amt08] spricht von `IGraphToolMain` „als koordinierende Instanz zwischen den GUI-Klassen auf der einen und den Klassen zur Modell- und Datenrepräsentation auf der anderen Seite."

Die andere wichtige Klasse in `core` ist die zentrale Schnittstelle zu den Simulationsfunktionalitäten. Die hierin ablaufenden Vorgänge werden in Abschnitt 2.2.2 näher erläutert.

Das `model`-Paket. Wurden fachliche Objekte modelliert, befinden sie sich in diesem Paket. Enthalten sind insbesondere Klassen für die Zugriffssteuerungsmatrix sowie die Heuristiken zur Safety-Analyse.

Letzteren steht das Paket `heuristic` zur Verfügung. Zusammen mit einer abstrakten Oberklasse für Heuristiken bestehen dort bereits zwei Implementierungen: `RandomHeuristic` [Rin09], welche Zustandsübergänge zufällig wählt, sowie `HeuristicSearch` [Joh09], die den *Leak-Search*-Algorithmus aus Abschnitt 2.1.3 realisiert. Genauer wird auf die Heuristiken ebenfalls im nachfolgenden Abschnitt eingegangen.

2.2.2 Komponenten zur Safety-Analyse

Die an der Safety-Analyse beteiligten Komponenten sind wesentlicher Gegenstand für diese Arbeit und werden daher im folgenden näher betrachtet. Dabei wird zunächst das Zusammenspiel der einzelnen Klassen vorgestellt, um anschließend auf die Details in Aufbau und Verhalten jener Klassen einzugehen.

Dynamik. Die Vorgänge, die während der Safety-Analyse durch eine Heuristik stattfinden, sind im Kollaborationsdiagramm in Abbildung 2.2 dargestellt.

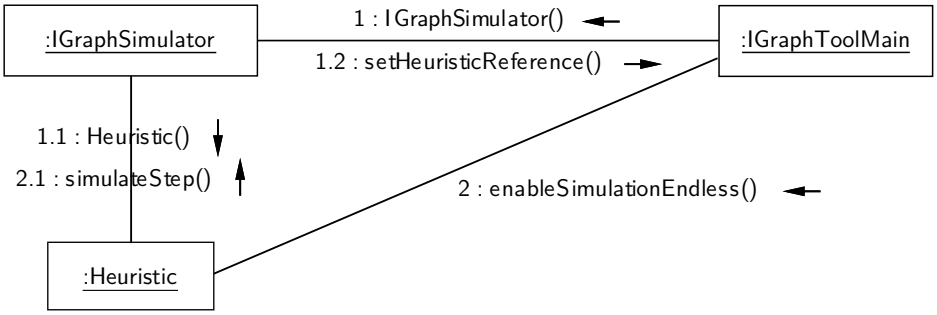


Abbildung 2.2: Die Safety-Analyse im UML-Kollaborationsdiagramm.

Grundlegend werden Safety-Analysen von der jeweiligen Heuristik aus gesteuert, doch erfolgt die Erzeugung der Datenstrukturen ausgehend von der GUI bzw. dem Simulator.

Richtet der Benutzer einen Simulationsvorgang über die grafische Oberfläche ein, veranlasst diese das Erstellen der Objekte. Hierzu wird in Schritt 1 (siehe Grafik) der Simulator (repräsentiert durch ein `IGraphSimulator`-Objekt) erzeugt, welcher dann das Laden und Instantiieren der entsprechenden `Heuristic`-Klasse veranlasst (Schritt 1.1). Wurde die Heuristik instantiiert, übergibt der Simulator der GUI eine Referenz darauf, wie in Schritt 1.2 dargestellt – anhand dieser Referenz wird die grafische Oberfläche dann direkt auf die Heuristik zugreifen können.

Die Oberfläche kann nun – wie in Schritt 2 – das Starten der Heuristik veranlassen. Die beiden vorhandenen Heuristikimplementierungen `RandomHeuristic` und `HeuristicSearch` leiten sich von der Klasse `Thread` ab, sodass der restliche Teil der Heuristik in einem eigenen Thread agiert. Die Heuristik fordert während der Analyse mehrmals das Simulator-Objekt dazu auf, einen Zustandsübergang zu berechnen (Schritt 2.1).

Simulator. Das Berechnen von Zustandsübergängen in HRU-Modellen obliegt einem Simulator-Objekt der Klasse `IGraphSimulator`, welche von [Rin09] beschrieben wurde. Der Simulator bietet darüber hinaus insbesondere Methoden, welche die Heuristik auf seine internen Daten zugreifen lassen, sowie Möglichkeiten, den Benutzer auf die Simulation Einfluss nehmen zu lassen.

Die Berechnung der Zustandsübergänge eines HRU-Modells (Q, Σ, δ, q_0) entspricht dabei, für einen Zustand $q \in Q$ und einer Eingabe $(\alpha, x) \in \Sigma$, der Ermittlung von $q' = \delta(q, (\alpha, x))$. Der Ausgangszustand q ist dabei Teil des internen Zustands des Simulators, die Eingabe (α, x) muss übergeben werden; der berechnete Zustand q' schließlich wird neuer Simulatorzustand.

Zu jedem errechneten Zustand erzeugt der jeweilige `IGraphSimulator` einen eindeutigen Bezeichner, mit dessen Hilfe zum einen Informationen über den jeweiligen Zustand

abgefragt werden können, zum anderen der interne Simulatorzustand auf einen beliebigen, bereits erkundeten Knoten umgesetzt werden kann.

Heuristik. Alle Heuristiken, die in *I-Graphoscope* implementiert werden, sollten von der abstrakten Klasse `Heuristic` [Rin09] abgeleitet werden, daher dient diese als Ausgangspunkt für die Überlegungen dieser Arbeit.

Heuristiken sind als eigener Thread implementiert und haben für ihre Berechnungen Zugriff auf den jeweiligen Modellzustand, die Zustandsüberführungsfunktion δ sowie das Simulator-Objekt.

2.3 Parallelisierung

Um Strategien zur Parallelisierung zu entwickeln ist es zwingend notwendig, ein Modell vom unterliegenden parallelen Computer zu haben, das Informationen liefert über Möglichkeiten der Parallelisierung, Synchronisation und Kommunikation, über bestimmte Garantien und Einschränkungen sowie über mögliche Eigenschaften (sowohl quantitativer als auch qualitativer Natur). Ist das Modell verbreitet, existiert möglicherweise schon Literatur hierzu, welche bei Spezifikation und Entwurf nützlich sein kann.

Wie in Abschnitt 2.2 erwähnt, wurde die Software, auf der diese Arbeit fußt, in Java entwickelt. Dies macht es sinnvoll, die Betrachtungen von vornherein auf das Maschinen- und Speichermodell von Java zu beschränken.

Referenz für die in diesem Abschnitt dargebotenen Informationen sind insbesondere [Lea99], erschienen im Rahmen einer offiziellen Reihe von Büchern zur Java-Plattform, sowie [MPA05], welches das Speichermodell formal definiert.

Als eine Einführung in die Modelle hinter Java betrachtet der erste, nachfolgende Abschnitt Javas Objektmodell im Kontext der Parallelität. Danach stellt Abschnitt 2.3.2 allgemeine Problemquellen bei der Entwicklung paralleler Programme in Java vor; Abschnitt 2.3.3 beschäftigt sich dann mit den bereitgestellten Mechanismen zur Synchronisation. Zuletzt wird in 2.3.4 ein formales Modell eingeführt, um die Semantik der Speicheroperationen in parallelen Java-Programmen zu beschreiben.

2.3.1 Objektmodell im Kontext der Parallelität

Java ist eine objektorientierte Sprache, sie arbeitet also auf Grundlage eines *Objektmodells*. Jedes Objekt verfügt dabei über bestimmte statische und dynamische Eigenschaften; insbesondere auch über Beziehungen zu anderen Objekten.

Es lassen sich *aktive und passive Objekte* unterscheiden. Während passive Objekte lediglich über einen Zustand sowie eine Reihe von Operationen verfügen, welche sich aufrufen lassen, treten aktive Objekte als *Aktoren* auf, die Operationen passiver Objekte *interpretieren* können. Solche aktiven Objekte sind *Threads*; die Operationen passiver Objekte sind *Methoden*.

Zentral in dieser Sichtweise ist, dass auf jedes passive Objekt von mehreren aktiven Objekten zugegriffen werden kann, was zu *Konkurrenzsituationen* führt, sobald mehrere Threads gleichzeitig auf ein Objekt zugreifen, mindestens einer davon schreibend.

Die Java-Sprachspezifikation bietet hier eine Reihe von Sprachkonstrukten, Semantiken und Garantien, die es ermöglichen, mit derartigen Situationen umzugehen. Diese werden im folgenden nahegebracht.

2.3.2 Problemquellen

Im Kontext der Konkurrenzsituationen gibt es eine Reihe üblicher Problemquellen, welche hier kurz umrissen werden.

Privater Arbeitsspeicher. Jeder Thread verfügt über einen privaten Arbeitsspeicher (eine Abstraktion für diverse Mechanismen realer Computer wie Caches und CPU-Register), in welchem er von ihm zugegriffene Werte zwischenspeichert. Der Inhalt dieser verschiedenen Thread-Arbeitsspeicher wird im Laufe der Programmausführung untereinander sowie vom Inhalt des Hauptspeichers abweichen; auf diese Weise entstehen inkonsistente Sichten der Threads auf die selben Objekte.

Abweichende Ausführungsreihenfolge. Operationen von Programmen in Java werden nicht in der Reihenfolge ausgeführt, in der sie der Entwickler festgelegt hat. Der Compiler, die virtuelle Maschine sowie die Computerhardware verfügen über Mechanismen, welche die Ausführungsreihenfolge der Operationen zu Optimierungszwecken abändern.

Zwar werden die Änderungen stets so getroffen, dass sie im Falle eines einzelnen, isoliert ablaufenden Threads t nicht durch t selbst beobachtet werden können; andere, parallel ablaufende Threads können sich jedoch nicht auf eine feste Reihenfolge der Befehlsabarbeitung in t verlassen.

Nicht-atomare Ausführung. In bestimmten Situationen ist es erforderlich sicherzustellen, dass eine Reihe von Zugriffen *atomar* ausgeführt wird. Insbesondere gilt dies, wenn auf ein oder mehrere Speicherfelder in mehreren Schritten lesend oder schreibend zugegriffen wird. Unterbrechen Operationen eines anderen Threads solche Zugriffe, kann es zu Inkonsistenzen kommen.

2.3.3 Synchronisationsmechanismen

Um verschiedene Threads zu synchronisieren, sei es bezüglich ihrer privaten Arbeitsspeicher oder der Ausführung von Operationen, bietet Java eine Zahl an Mechanismen, aus denen im folgenden eine Auswahl eingeführt wird.

Zugriffssemantik für Felder. Java garantiert atomare Lese- und Schreibzugriffe auf Felder beliebigen Typs außer `long` und `double`, jedoch einschließlich `volatile long` und `volatile double`. Darüber hinaus bietet die Bibliothek `java.util.concurrent.atomic` Objekte, auf denen Kombinationen von Lese- und Schreibzugriffen atomar durchgeführt werden können. Beispiele sind *Compare-And-Set-* oder Inkrement-Operationen.

Um zu verlangen, dass der Wert eines Feldes nicht im privaten Arbeitsspeicher der Threads abgelegt wird, kann dieses Feld als `volatile` deklariert werden. Ein Zugriff darauf erfolgt dann immer über den Hauptspeicher, sodass jeder Thread die selbe Sicht auf das Feld hat. Dies erhöht jedoch die Zugriffszeit und schränkt das ausführende System in seinen Möglichkeiten, die Ausführungsreihenfolge von Operationen zu optimieren, ein.

`synchronized`-Blöcke und -Methoden. Um mit Mitteln des wechselseitigen Ausschlusses eine Reihe von Operationen atomar auszuführen, bietet Java das `synchronized`-Konstrukt mit der in Listing 2.1 dargestellten Syntax.

`lockvar` ist hierbei eine Referenz auf ein beliebiges Objekt. Unter allen `synchronized`-Blöcken, welche `lockvar` verwenden, realisiert Java wechselseitigen Ausschluss. Das

```
synchronized(lockvar) {
    // Anweisungen
}
```

Listing 2.1: Das `synchronized`-Konstrukt.

Betreten des Blocks kommt dabei einem *Lock*, das Verlassen des Blocks einem *Unlock* auf das `lockvar`-Objekt gleich.

Gleichzeitig sorgt das Sprachkonstrukt für einen Abgleich des Hauptspeichers mit dem privaten Arbeitsspeicher des Threads. Bei Betreten eines `synchronized`-Blocks werden die Werte erreichbarer Felder aus dem Hauptspeicher gelesen; beim Verlassen werden Änderungen zurückgeschrieben.

Auf Terminierung warten. Ein `Thread`-Objekt bietet die Methode `join()` an. Ruft ein Thread t_1 die Methode t_2.join() für einen anderen Thread t_2 auf, wird t_1 so lange blockiert, bist t_2 terminiert.

Über `isAlive()` kann geprüft werden, ob ein Thread bereits terminiert ist.

Monitore. Jedes Java-Objekt verfügt über Monitor-Mechanismen, auf die über die Methoden `wait()`, `notify()` und `notifyAll()` zugegriffen werden kann. Ein Thread sollte diese Methoden nur dann aufrufen, wenn er (mittels `synchronized`) das Lock auf das Objekt hält.

Verkürzt dargestellt wird ein Thread, der `wait()` für ein Objekt aufruft, erstens blockiert; zweitens wird sein Lock auf das Objekt freigegeben. Ein Aufruf von `notify()` lässt einen auf das entsprechende Objekt per `wait()` wartenden Thread wieder anlaufen, sobald er das Lock zum Objekt wieder besetzen kann. `notifyAll()` funktioniert auf die selbe Art wie `notify()` für *alle* auf ein Objekt wartenden Threads.

Interrupts. Einem Thread kann mittels `Thread.interrupt()` ein Unterbrechungssignal gesendet werden. Das Vorliegen eines solchen Signals kann über `Thread.interrupted()` beziehungsweise `Thread.isInterrupted()` abgefragt und entsprechend behandelt werden. Ist ein Thread via `Object.wait()`, `Thread.sleep()` oder `Thread.join()` blockiert, veranlasst ein Interrupt auf diesen Thread seinen Wiederanlauf; die blockierende Methode wirft dann eine `InterruptedException`.

All diese Mechanismen können auf sehr komplexe Art und Weise zusammenwirken. Hierzu existiert in Java ein formales Modell, welches der anschließende Teil erläutert.

2.3.4 Formales Speichermodell

Für Java existiert ein Speichermodell, das definiert, wie und mit welchem Ergebnis Speicheroperationen in einem Programm ausgeführt werden. Es gibt Garantien über die Ausführungsreihenfolge von Operationen in Programmen, die mehrere Threads verwenden und bestimmten Regeln genügen. Im folgenden wird das Speichermodell nach [MPA05] formal definiert.

Das Speichermodell geht von einer kleinsten Einheit eines Programmes aus, der *Aktion*.

Definition 2.6 (Aktion).
Eine Aktion a ist ein Viertupel (t, k, v, u), bestehend aus dem die Aktion ausführenden Thread t, der Art der Aktion k, der involvierten Variable v sowie einem eindeutigen Bezeichner u.

Das Modell berücksichtigt folgende Arten von Aktionen:

- Lese- und Schreibzugriffe auf nicht-`volatile`-Felder
- Threaddivergenzaktionen (unendliche Schleifen)
- Ein-/Ausgabeoperationen
- *Synchronisationsaktionen*, nämlich
  - Lese- und Schreibzugriffe auf `volatile` Felder
  - Lock- und Unlock-Operationen (Betreten / Verlassen eines `synchronized`-Blocks)
  - Starten eines Threads
  - Erste / letzte Operation eines Threads
  - Entdecken der Terminierung eines Threads (beispielsweise via `join()`)

Von nun an wird eine *konkrete* Ausführung eines Programmes betrachtet. Die Aktionen in jedem individuellen Thread sind total geordnet, und zwar in der zeitlichen Reihenfolge ihrer Ausführung. Diese Ordnung heißt *Programmordnung* eines Threads.

Definition 2.7 (Programmordnung).
Für einen Thread t ist die Programmordnung eine totale Ordnung über alle Aktionen in t. Sie ist konsistent zur im Programmtext vorgegebenen Ordnung. Folgt laut Programmordnung einer Aktion a_1 aus t eine Aktion a_2 aus t nach, schreibe $a_1 \xrightarrow{po} a_2$.

Gleiches gilt für die Synchronisationsaktionen aller Threads: auch für sie gibt es eine feste, zeitliche Reihenfolge, die *Synchronisationsordnung*.

Definition 2.8 (Synchronisationsordnung).
Die Synchronisationsordnung ist eine totale Ordnung über alle Synchronisationsaktionen und ist konsistent zur Programmordnung. Folgt laut Synchronisationsordnung einer Aktion a_1 eine andere Aktion a_2 nach, schreibe $a_1 \xrightarrow{so} a_2$.

Es ist stets zu beachten, dass zu einem Programm viele verschiedene Programm- und Synchronisationsordnungen möglich sind.

Die Synchronisationsordnung bringt alle Synchronisationsaktionen in eine gemeinsame, zeitliche Reihenfolge. Um auch kausale Zusammenhänge zwischen den einzelnen Aktionen zu berücksichtigen, wird eine weitere, partielle Ordnung definiert.

Definition 2.9 (*Synchronizes-With*-Ordnung).
Die *Synchronizes-With*-Ordnung ist eine partielle Ordnung über Synchronsiationsaktionen. Folgt laut dieser Ordnung einer Aktion a_1 eine andere Aktion a_2 nach, schreibe $a_1 \xrightarrow{sw} a_2$.

Die Kanten der *Synchronizes-With*-Ordnung werden anhand einer gegebenen Synchronisationsordnung $\xrightarrow{so}$ nach festen Regeln gesetzt.

- Sei x eine Unlock-Aktion auf einem `synchronized`-Block mit Lock-Variable v. Für alle Lock-Aktionen y mit $x \xrightarrow{so} y$, die bezüglich derselben Variable v sperren, gilt: $x \xrightarrow{sw} y$.
- Sei x eine Schreiboperation auf einem `volatile`-Feld v. Für alle Lese-Operationen y auf demselben Feld mit $x \xrightarrow{so} y$ gilt: $x \xrightarrow{sw} y$.
- Es verläuft eine *Synchronizes-With*-Kante vom Start eines Threads zur ersten Aktion in diesem Thread.
- Sei x die Schreiboperation, die den Standardwert eines Feldes schreibt[5]. Eine *Synchronizes-With*-Kante verläuft von x zur ersten Aktion eines jeden Threads[6].
- Sei x die letzte Aktion eines Threads t und y eine Aktion eines anderen Threads, welche das Terminieren von t bemerkt. Es gilt $x \xrightarrow{sw} y$.
- Unterbricht ein Thread t_1 einen anderen Thread t_2 mit einer Aktion x, gilt $x \xrightarrow{sw} y$ für alle Aktionen y die bemerken, dass t_2 unterbrochen wurde.

Die so definierte *Synchronizes-With*-Ordnung induziert eine neue partielle Ordnung über *alle* Aktionen, die *Happens-Before*-Ordnung.

Definition 2.10 (*Happens-Before*-Ordnung).
Die *Happens-Before*-Ordnung ist der transitive Abschluss aus Programmordnung und *Synchronizes-With*-Ordnung. Folgt laut dieser Ordnung einer Aktion a_1 eine weitere Aktion a_2 nach, so schreibe $a_1 \curvearrowright a_2$.

Betont werden muss: die *Happens-Before*-Ordnung ist für jede Programmausführung individuell. Mithilfe der Ordnung lassen sich jedoch Aussagen darüber treffen, welche Ergebnisse ein konkretes Programm erzeugen *kann*, und welche nicht.

Die letzte Aussage des Speichermodells, die hier getroffen werden soll, stellt fest, welchen Wert eine Leseoperation lesen darf. Hierfür werden zwei *Konsistenzbedingungen* festgelegt.

Definition 2.11 (Konsistenzbedingungen).
Unter Happens-Before-Konsistenz *darf* eine Leseoperation r auf ein Feld v das Ergebnis einer Schreiboperation w auf v lesen, wenn:

- nicht gilt, dass $r \curvearrowright w$[7], und
- $\nexists w' : w \curvearrowright w' \curvearrowright r$, mit w' ist Schreiboperation auf v.

Unter Konsistenz der Synchronisationsordnung gilt erstens, dass die Synchronisationsordnung konsistent mit der Programmordnung ist, und zweitens, dass jede Leseoperation r auf eine `volatile`-Variable v die gemäß Synchronisationsordnung vorhergehende Schreiboperation w auf v sieht.

[5] In Java wird jedes Feld mit `0`, `false` bzw. `null` initialisiert.

[6] Die Vorstellung hinter dieser Regel ist, dass alle Objekte bereits zu Beginn der Programmausführung vorhanden sind.

Ist ein Java-Programm korrekt synchronisiert (gelten also die Konsistenzbedingungen), sind in jeder möglichen Ausführung alle Aktionen, die auf die selbe Variable zugreifen – mindestens einmal davon schreibend –, über die jeweilige *Happens-Before*-Ordnung in Reihenfolge gebracht.

Die Ergebnisse jeder Operation in jedem Thread sind somit über die obigen Definitionen genau bestimmbar. Dies wird im Implementierungs-Teil dieser Arbeit (Kapitel 5) hilfreich sein, wenn die zuvor entwickelten Algorithmen in Java implementiert werden sollen.

2.4 Zusammenfassung

Dieses Kapitel betrachtete die fachliche Landschaft, in der diese Arbeit angesiedelt ist. Es führte den HRU-Kalkül sowie die damit verbundene *Safety-Eigenschaft* ein, stellte jedoch fest, dass diese nicht grundsätzlich entscheidbar ist. Dennoch existieren Wege, in begrenztem Maße Aussagen über die Safety eines gegebenen HRU-Modells zu treffen; beispielsweise mittels heuristischer Suche. Hierzu skizzierte das Kapitel den Algorithmus *Leak-Search*, der über vier heuristische Funktionen zielgerichtet nach Rechteausbreitungen sucht.

Der zweite Teil des Kapitels beschrieb die Software *I-Graphoscope*; es handelt sich um ein Programm zur Visualisierung und Analyse von HRU-Modellen und dient auch als Plattform zur Implementierung der *Leak-Search*-Heuristik. Dieses Kapitel gab einen Überblick über die Gesamtarchitektur der Software und stellte Statik und Dynamik der an der Safety-Analyse beteiligten Komponenten detailliert vor.

Ausgehend vom Ziel dieser Arbeit, Laufzeiteigenschaften der Safety-Analysen mittels Parallelisierung zu verbessern, betrachtete der letzte Teil des vorliegenden Grundlagenkapitels die Methoden und Werkzeuge, welche die verwendete Programmiersprache, Java, zur Parallelisierung bietet. Er stellte dabei auch fest, dass für korrekt synchronisierte Programme genaue Garantien zur Ausführungsreihenfolge einzelner Operationen gegeben werden können, und formalisierte dies anhand einer *Happens-Before*-Ordnung.

[7] Happens-Before-Konsistenz fordert nicht, dass $w \curvearrowright r$.

KAPITEL 3

Problemanalyse

Der Softwareentwurf und die ihm folgende Implementierung benötigen als Ausgangspunkt Informationen rund um das zu lösende Problem. Diese soll das vorliegende Kapitel „Problemanalyse“ erarbeiten.

Es leitet dabei anfangs aus der Zielstellung konkrete Anforderungen her, welche in Abschnitt 3.1 dargelegt werden. Der darauf folgende Abschnitt 3.2 analysiert das Laufzeitverhalten der vorhandenen Implementierung, um so größeres Problembewusstsein zu schaffen und zu untersuchen, inwiefern Möglichkeiten bestehen, eine hohe Performanzsteigerung zu erzielen. Abschnitt 3.3 schließlich prüft, an welchen Stellen die HRU-Safety-Analyse in *I-Graphoscope* Nebenläufigkeiten aufweist. Diese dienen dann als Ausgangspunkt zur Arbeit des Entwurfskapitels.

3.1 Anforderungen

Gegeben sei das Suchproblem zur Safety eines gegebenen HRU-Modells wie in Abschnitt 2.1.3. Die Software *I-Graphoscope* bietet ein Framework für derartige Analysen; es wurde in 2.2 nahegebracht und dient im folgenden als Ausgangspunkt.

Die Zielstellung der Arbeit (siehe 1.2) besteht in der Steigerung der Zahl der erkundbaren Knoten pro Zeiteinheit mit Hilfe der Mechanismen der Parallelisierung. Aus dieser Richtungsvorgabe müssen nun konkrete Anforderungen entstehen, anhand derer später die Konzepte, Algorithmen und Architekturänderungen hergeleitet werden können.

Die Anforderungen teilen sich in zwei Arten ein: funktionale Anforderungen zum einen, welche zu realisierende Funktionalitäten in der Software beschreiben; zum anderen Anforderungen nichtfunktionaler Natur, die Aussagen über die Qualität der Implementierung treffen.

Die nachstehenden zwei Abschnitte leiten sowohl funktionale als auch nichtfunktionale Anforderungen her und erläutern diese im Kontext der Arbeit.

3.1.1 Funktionale Anforderungen

Gegenstand der Zielstellung ist ein nichtfunktionales Kriterium, nämlich der Durchsatz des Systems. Im Weg jedoch, der beschritten werden soll – dem Einsatz von Parallelisierungmechanismen – liegt die wesentliche funktionale Anforderung der Arbeit.

Anforderung 3.1 (Nutzung mehrerer Prozessoren).
Die Arbeit soll *I-Graphoscope* um die Funktion erweitern, Analysen zur HRU-Safety auf mehrere Prozessoren[1] aufzuteilen.

Alle weiteren Anforderungen und Einschränkungen sind nichtfunktionaler Natur.

[1] Im folgenden werden Multikernprozessoren vereinfacht als Mehrprozessorsystem behandelt.

3.1.2 Nichtfunktionale Anforderungen

Zu einem Programm, welches das funktionale Kriterium 3.1 erfüllt, also mehrere Prozessoren nutzt, stellt sich auch die Frage, wie viele Prozessoren das Programm nutzen kann.

Anforderung 3.2 (Skalierbarkeit bzgl. der Zahl der Prozessoren).
Um zukünftig auch komplexere Datenmengen bewältigen zu können, soll die Zahl der auslastbaren Prozessoren vor allem von der Komplexität der Eingabedaten abhängen. Insbesondere soll sie keiner konstanten Begrenzung unterliegen.

Aus dieser Anforderung folgt, dass die Bestrebungen zur Parallelisierung insbesondere darauf abzielen werden, auf verschiedenen Teilen der Eingabedaten die selben Algorithmen parallel abzuarbeiten (*Datenparallelität*), statt eine feste Zahl an Aufgaben parallel durchzuführen (*Taskparallelität*).

Ähnlich eng mit dem Laufzeitverhalten der Software verbunden wie die Skalierbarkeit bezüglich der Prozessorzahl ist die Performanz der zu entwickelnden Komponenten.

Anforderung 3.3 (Performanz).
Beim Entwurf der Algorithmen und Architektur soll die Arbeit auf Konzepte zurückgreifen, die auf ein günstiges Laufzeitverhalten abzielen.

Soweit vertretbar, soll dabei Speichereffizienz in den Hintergrund treten. Um einen Kompromiss zwischen Performanz und Wartbarkeit bzw. Übersichtlichkeit zu finden, sollten diese Eigenschaften nur *innerhalb* einer Komponente zugunsten der Performanz vernachlässigt werden.

Der Funktionsumfang von *I-Graphoscope* ist nicht starr; das Projekt wird vielmehr kontinuierlich im Rahmen vieler wissenschaftlicher Arbeiten weiterentwickelt. Änderungen in den hier angegebenen Anforderungen sind daher keinesfalls auszuschließen, und so muss für die Arbeit die Erweiterbarkeit eine Rolle spielen.

Anforderung 3.4 (Erweiterbarkeit).
Damit zukünftige Entwicklungen und Erkenntnisse im Umfeld der Erweiterungen, welche diese Arbeit vornimmt, umgesetzt werden können, muss der Architekturentwurf auf Erweiterbarkeit Rücksicht nehmen.

Um möglichst gute Ansatzpunkte zur Entwicklung performanter Verfahren zu erhalten, schränkt die Arbeit die zu verwendenden Analyseverfahren ein. Die Art der Einschränkung wurde dabei so gewählt, dass die Klasse der *Best-First*-Algorithmen, einschließlich *Leak-Search* (siehe [Joh09]), enthalten ist.

Anforderung 3.5 (Art der Heuristik).
Das Framework zur Parallelisierung der Safety-Analyse wird für solche Analyseverfahren ausgelegt, die zu einem HRU-Modell auf einem Suchbaum wie in Abschnitt 2.1.3 nach Zuständen suchen, die eine Rechteausbreitung bezüglich des initialen Zustandes und des gesuchten Rechts enthalten. Die Verfahren sollen dabei die untersuchten Zustände mittels einer heuristischen Funktion bewerten.

Diese Arbeit soll an ihrem Ende die erzielten Ergebnisse auswerten. Um hier aussagekräftige Aussagen erzielen zu können, müssen während der Entwicklung Vorkehrungen getroffen werden, welche unter der folgenden Anforderung zusammengefasst werden.

Anforderung 3.6 (Evaluierbarkeit).
Die Software soll eine aussagekräftige Bewertung der Ergebnisse der Arbeit ermöglichen.

Selbstverständlich ist die Korrektheit der errechneten Ergebnisse entscheidend.

Anforderung 3.7 (Korrektheit).
Die Ergebnisse, welche die im Rahmen der Arbeit produzierte Software erzeugt, sollen stets korrekt bezüglich der Spezifikation sein.

Bei der Entwicklung von Software, welche parallele Berechnungen einsetzt, muss zudem stets auf die Lebendigkeit und Verhungerungsfreiheit geachtet werden.

Anforderung 3.8 (Verhungerungsfreiheit).
Kein Thread wird dauerhaft in seinen Berechnungen blockiert; jede Berechnung eines Teilproblems soll in endlicher Zeit terminieren.

Die Lebendigkeit folgt dann aus der Verhungerungsfreiheit.

3.2 Performanz

Um die durch die Zielstellung gegebene Aufgabe sinnvoll zu lösen, ist Wissen um die Laufzeitmerkmale der Implementierung erforderlich, insbesondere um deren *Flaschenhälse*. Die einzelnen Teilschritte der Safety-Analyse müssen also auf ihren Anteil an der Gesamtrechenzeit untersucht werden.

Ziel ist dabei erstens, eine Argumentationsgrundlage für Entwurfsentscheidungen zu erhalten, und zweitens, eventuell besonders geeignete Parallelisierungsmöglichkeiten aufzudecken.

Im Rahmen dieser Arbeit wurde eine solche Analyse vorgenommen. Nachstehend werden dazu die Herangehensweise, die erhaltenen Ergebnisse sowie deren Auswertung nahegebracht.

3.2.1 Herangehensweise

Im Rahmen der Herangehensweise wird zuerst die Art der vorgenommenen Untersuchung diskutiert, im Anschluss die Umgebung und Konfiguration zur Analyse erläutert.

Art der Untersuchung. Die Auswahl des anzuwendenden Verfahrens wurde durch folgende zwei Faktoren getrieben:

- Die tatsächlichen Laufzeiteigenschaften der Software lassen sich am Quelltext nur bedingt und unter hohem Aufwand erkennen. Dazu trägt neben der Komplexität des Quelltextes auch die Tatsache bei, dass die eingesetzte virtuelle Maschine, auf welcher das Java-Programm abläuft, einen starken Einfluss auf die Laufzeitmerkmale haben kann.
- Die Laufzeiteigenschaften der Software hängen stark vom analysierten Modell, und damit von der Eingabe ab.

Um hier also mit akzeptablem Aufwand möglichst realitätsnahe Ergebnisse zu erzielen, wurde die Software während der Analyse eines realitätsnahen HRU-Modells mit einem *Profiler* beobachtet. Mit diesem wurde zu jeder Methode der an der Safety-Analyse beteiligten Klassen deren Anteil an der Gesamtlaufzeit sowie die Zahl ihrer Aufrufe ermittelt. Mit den erhaltenen Daten kann die Parallelisierung des Analyseverfahrens zielgerichtet und problembewusst vorangetrieben werden.

Umgebung. Zur Messung kam folgendes System zum Einsatz:

- Der Prozessor war ein *Pentium M* 740 (1,73 GHz) mit einem 2 MiB Level-2-Cache sowie 1 GiB DDR2-Hauptspeicher (400 MHz).
- Als Betriebssystem kam Fedora Linux 11 zum Einsatz, es liefen keine weiteren Anwendungen.
- Die Virtuelle Maschine *IcedTea6* 1.5 ist Teil des *OpenJDK Runtime Environment*.
- Beim Profiler handelt es sich um den zur *NetBeans IDE* 6.5 ausgelieferten Profiler, Version 1.9.5.

Konfiguration. Als Analyseverfahren kam die in 2.1.3 beschriebene *Leak-Search*-Heuristik aus [Joh09] zum Einsatz, da sie die derzeit beste vorliegende Analysemethode darstellt. In der Auswertung jedoch sollte sie keine Rolle spielen, da gemäß Anforderung 3.5 von allgemeineren Verfahren ausgegangen wird.

Zur Evaluierung des *Leak-Search*-Algorithmus verwendete [Joh09] Modelle auf der Grundlage eines realen Linux-Systems, um so möglichst realitätsnahe Ergebnisse zu erzielen. Da dies zum einen den Anforderungen dieser Laufzeitanalyse entspricht, und zum anderen das Verhalten von *Leak-Search* auf den Modellen in jener Arbeit gut dokumentiert ist, kamen diese auch hier als Eingabe zum Einsatz.

Modelliert wurde die Unix-Zugriffssteuerung mit neun anwendungsspezifischen Operationen sowie fünf Rechten. Darüber hinaus wurden initiale Zugriffssteuerungsmatrizen verwendet, welche Daten basierend auf den Verzeichnissen „`/home/user1`“, „`/var`“ sowie „`/sys`“ eines realen Linux-Systems enthielten. Das System verzeichnete jeweils 29 Subjekte.

Aus der Menge der in [Joh09] beschriebenen Eingaben wurden aufgrund der dort erläuterten Ergebnisse drei Eingaben ausgewählt, welche drei verschiedene, charakteristische Verhaltensweisen der Heuristik widerspiegeln; sie werden im folgenden kurz erläutert.

- Analyse des „`/var`“-Verzeichnisses auf eine Ausbreitung des `read`-Rechtes in maximal 5000 Schritten. Die Analyse betrachtet eine Matrix mit 10244 Objekten und findet auch bei einer hohen Schrittzahl noch Rechteausbreitungen. Ältere Zustände werden selten erkundet.
- Analyse des „`/home/user1`“-Verzeichnisses auf eine Ausbreitung des `execute`-Rechtes in maximal 1000 Schritten. Die Matrix umfasst hier 19463 Objekte, die effektive Zahl an untersuchten Objekten ist jedoch circa 20 % kleiner als bei der vorhergehenden Matrix. Schon nach 100 Schritten werden kaum noch Rechteausbreitungen entdeckt. Nach etwa 300 Schritten werden auch ältere Zustände häufiger betrachtet.
- Analyse des „`/sys`“-Verzeichnisses auf eine Ausbreitung des `read`-Rechtes in maximal 1000 Schritten. Hier handelt es sich um die kleinste der drei Matrizen. Sie umfasst 8459 Objekte, ihre Rechteausbreitungen werden schon in den ersten 50 Schritten vollständig gefunden. Ältere Zustände werden sehr häufig betrachtet.

3.2.2 Ergebnisse

Während allen drei Analysen waren jeweils höchstens fünf Methoden für mehr als 95 % der Rechenzeit verantwortlich. Die konkreten Aufstellungen lassen sich Tabelle 3.1 entnehmen.

Methode	Zeit		Aufrufe
`IGraphSimulator.getMatrix`	58,2 %	9 146 545 ms	7 898
`CalculateValues.run`	23,6 %	3 711 348 ms	1 959
`Fstate.run`	12,6 %	1 973 220 ms	5 940
`Fentity.run`	4,9 %	763 818 ms	5 421

(a) Analyse des „`/var`“-Verzeichnisses auf eine `read`-Ausbreitung.

Methode	Zeit		Aufrufe
`IGraphSimulator.getMatrix`	62,4 %	1 365 313 ms	1 731
`CalculateValues.run`	25,7 %	562 365 ms	591
`Fstate.run`	5,8 %	126 208 ms	1 141
`Fentity.run`	2,3 %	50 839 ms	1 034
`Fselection.run`	1,5 %	32 642 ms	1
`IGraphSimulator.simulationFinished`	1,1 %	24 732 ms	1
`IGraphSimulator.simulateStep`	1,1 %	23 805 ms	731

(b) Analyse des „`/home/user1`“-Verzeichnisses auf eine `execute`-Ausbreitung.

Methode	Zeit		Aufrufe
`IGraphSimulator.getMatrix`	59,0 %	397 040 ms	1 552
`Fstate.run`	17,0 %	114 294 ms	1 120
`CalculateValues.run`	14,6 %	98 042 ms	433
`Fselection.run`	3,5 %	23 831 ms	1
`Fentity.run`	2,3 %	15 264 ms	1 050
`IGraphSimulator.simulationFinished`	2,1 %	13 950 ms	1
`IGraphSimulator.simulateStep`	1,3 %	8 855 ms	552

(c) Analyse des „`/sys`“-Verzeichnisses auf eine `read`-Ausbreitung.

Tabelle 3.1: Profiling-Ergebnisse für die Safety-Analyse dreier HRU-Modelle. Zu jeder Methode wird ihr prozentualer Anteil an der Gesamtlaufzeit der Analyse dargestellt, die Gesamtdauer ihrer Aktivität sowie die Anzahl ihrer Aufrufe. Methoden mit weniger als einem Prozent Anteil an der Gesamtlaufzeit werden nicht dargestellt, ebensowenig Methoden, die innerhalb der hier gezeigten Methoden aufgerufen werden.

Es zeigt sich, dass in allen drei Fällen die Methode `IGraphSimulator.getMatrix` mehr als die Hälfte der Gesamtlaufzeit über aktiv ist. Sie errechnet zu einem gegebenen Knoten die Zugriffssteuerungsmatrix. Diese wird dabei, ausgehend vom initialen Zustand, aus den bis dahin vorgenommenen Operationen *konstruiert* – ein Vorgehen, das die Laufzeit zugunsten der Speichereffizienz benachteiligt.

Einen ebenfalls hohen Anteil an der Laufzeit des Analyseverfahrens haben die Methoden der Klassen `Fstate` und `CalculateValues`. Es handelt sich hier um Algorithmen, welche vom *Leak-Search*-Algorithmus zur Zustandsbewertung eingesetzt werden, wie in 2.1.3 beschrieben.

Die verbleibenden Methoden sind nur noch für einen sehr geringen Anteil der Laufzeit verantwortlich.

3.2.3 Auswertung

Während der oben dargestellten Laufzeitbetrachtung der HRU-Safety-Analyse unter realistischen Bedingungen hatten sowohl Heuristik als auch Simulator einen hohen Anteil an der Rechenzeit.

Die besonders rechenaufwändigen Komponenten der Analyse, nämlich die Errechnung heuristischer Werte sowie die Konstruktion von Zugriffssteuerungsmatrizen, sind nicht nebenläufig zueinander, sodass hier keine Gewinne durch Parallelisierung erwartet werden können (siehe dazu auch Abschnitt 3.3).

Die restlichen Komponenten der Simulationsumgebung hatten einen sehr geringen Anteil an der Gesamtlaufzeit.

Die Untersuchung des Laufzeitverhaltens der HRU-Safety-Analyse zeigt daher keine wesentlichen Möglichkeiten zur Parallelisierung auf.

3.3 Datenabhängigkeiten

Notwendige Voraussetzung für das Parallelisieren zweier Berechnungen ist deren Nebenläufigkeit, sodass es Ziel diesen Abschnittes sein soll, solche Nebenläufigkeiten in der Safety-Analyse aufzuspüren.

Bezüglich der Parallelität zweier Berechnungen lässt sich zwischen Task- sowie Datenparallelität unterscheiden; entsprechend untersucht dieser Abschnitt zuerst die Tasks, die im Laufe einer Safety-Analyse durchzuführen sind; anschließend betrachtet er die Daten, auf denen solch eine Analyse abläuft.

3.3.1 Tasks

Finden sich bei der Safety-Analyse eines HRU-Modells nebenläufige Tasks, können diese parallel ausgeführt werden. Tasks sind dann nebenläufig, wenn zwischen ihnen kein kausaler Zusammenhang besteht, was insbesondere dann der Fall ist, wenn ein Task nicht von den Ergebnissen eines anderen Tasks abhängt.

Der Analysevorgang soll derart veranschaulicht werden, dass Datenabhängigkeiten leicht ersichtlich sind. Anforderung 3.5 beschriebt die Annahmen, die hier über die Heuristiken getroffen werden sollen. Abbildung 3.1 zeigt das entstandene Aktivitätsdiagramm, das den Ablauf der Safety-Analyse darstellt.

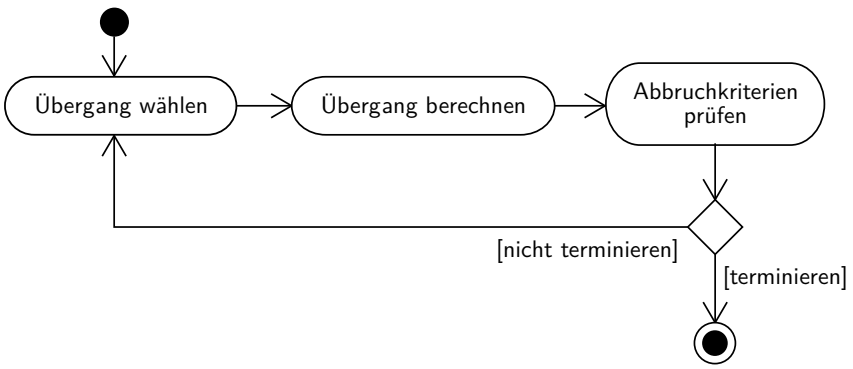


Abbildung 3.1: Die Safety-Analyse im UML-Aktivitätsdiagramm.

Die dort dargestellte Aktivität „Übergang wählen“ lässt sich kaum weiter verfeinern:

hier geht das jeweilige Analyseverfahren vollständig nach der ihm eigenen Strategie vor. Abhängig von dieser Strategie kann die Zustandswahl allerdings sehr zeitaufwändig sein; einerseits aufgrund der Berechnungen der Heuristik selbst, andererseits auch aufgrund von Methodenaufrufen auf andere Objekte. So wird eine Heuristik wahrscheinlich auf Informationen des Simulators zugreifen, um beispielsweise die Zugriffssteuerungsmatrix des ausgewählten Zustandes abzufragen. Die hohen Laufzeitkosten dieser Operation wurden bereits in Abschnitt 3.2 dargestellt.

Die zwei anderen Schritte der Safety-Analyse, das Berechnen des Übergangs sowie das Prüfen der Abbruchkriterien (zumeist wird hier ausgewertet, ob eine Rechteausbreitung vorliegt) lassen sich effizient implementieren und sind datenabhängig zur Zustandswahl, sodass sie hier nicht weiter betrachtet werden.

Weitere Teilschritte zeigt das betrachtete Modell der Safety-Analysen nicht auf; eine Verfeinerung des Modells auf eine Ebene, in der Nebenläufigkeiten erhalten werden hieße jedoch, die Heuristik beträchtlich einzuschränken. Taskparallelität lässt sich in diesem Fall also nicht sinnvoll einsetzen.

3.3.2 Daten

Möglicherweise sind Berechnungen auf verschiedenen Teilen der eingegebenen oder zwischenzeitlich produzierten Daten nebenläufig, sodass diese parallel durchgeführt werden können.

Im Rahmen der Safety-Analyse eines HRU-Modells gibt es hier zwei relevante Strukturen: dies ist zum einen der initiale Zustand, welcher als Eingabe für das Analyseverfahren dient, und zum anderen der Suchbaum, welcher während der Analyse abgearbeitet wird.

Initialer Zustand. Möglicherweise lässt sich die Zugriffssteuerungsmatrix des initialen Zustandes derart zerlegen, dass die so erhaltenen Teile sich von je einer Heuristik nebenläufig analysieren lassen, ohne die Güte der Analyse einzuschränken. Quantität und Qualität des Ergebnisses – und damit die Skalierbarkeit bezüglich der Zahl der auslastbaren Prozessoren – würden stark von der Beschaffenheit der Eingabematrix abhängen.

Die Erforschung solch einer Dekomposition geht über den Umfang dieser Arbeit hinaus, doch sollte die entsprechende Funktionalität leicht nachzurüsten sein (siehe dazu auch Anforderung 3.4, „Erweiterbarkeit“).

Suchbaum. Eine HRU-Safety-Analyse lässt sich, wie in Abschnitt 2.1.3 dargelegt, als Suchproblem darstellen. Das Analyseverfahren erkundet dabei einen Suchbaum, indem es in jedem Schritt einen Knoten auswählt und diesen auf eine Rechteausbreitung untersucht.

Wird nun die Nebenläufigkeit der Berechnungen betrachtet, sind wiederum die Datenabhängigkeiten entscheidend. Daher muss ermittelt werden, welche Informationen ein Suchschritt des Analysealgorithmus zu seiner Ausführung benötigt.

Auf diese Art und Weise lässt sich tatsächlich eine wesentliche Möglichkeit zur Parallelisierung finden, welche im folgenden als Satz festgehalten wird.

Satz 3.9 (Nebenläufigkeit der Analyse von Teilbäumen).
Gegeben sei die Safety-Analyse eines HRU-Modells mit Suchgraph G, initialem Zustand q und Recht r, wie in Definition 2.5. Seien (T, v) und (U, w) zwei disjunkte[2] Teilbäume in G mit Wurzeln v und w. Auf beiden Teilbäumen lassen sich korrekte Safety-Analysen so durchführen, dass diese nebenläufig zueinander sind.

Beweis. Die Suche nach einer Ausbreitung des Rechtes r auf dem Teilgraphen T ist ein Suchproblem zur HRU-Safety gemäß Definition 2.5, der initiale Zustand ist v. Zur Analyse des Teilbaumes genügen also die für G verwendeten Suchverfahren. Da T und U nicht ineinander enthalten sind, liegt U nicht im Suchgraphen des Suchproblems, somit ist die Safety-Analyse auf (T, v) nicht abhängig von den Daten des Teilbaumes (U, w).

Analog hierzu ist die Suche nach einer Ausbreitung des Rechtes r auf dem Teilgraphen U ein Suchproblem zur HRU-Safety mit initialem Zustand w und nicht abhängig von den Daten des Teilbaumes (T, v).

Beide Analysen sind somit nebenläufig. □

3.4 Zusammenfassung

Das vorliegende Kapitel „Problemanalyse“ verfolgte das Ziel, ein Fundament für Entwurf und Implementierung herzustellen, indem es aus der Zielstellung und dem vorhandenen Wissen neue, relevante Informationen herleitet.

Hierzu wurden zuerst einige funktionale und nichtfunktionale Anforderungen abgeleitet, nach welchen sich der Softwareentwurf richten wird. Demnach soll die durch diese Arbeit geschaffene Software insbesondere skalierbar (bezüglich der Prozessorzahl), performant und erweiterbar sein.

Da die Zielstellung der Arbeit auf Laufzeitkriterien abzielt, untersuchte ein zweiter Abschnitt das Laufzeitverhalten der bereits implementierten Komponenten zur HRU-Safety-Analyse, konnte jedoch keine konkreten Anhaltspunkte für Berechnungen liefern, deren Parallelisierung erhebliche Performanzsteigerungen liefern könnte.

Der dritte Teil des Kapitels betrachtete die Datenabhängigkeiten zwischen den einzelnen Aktivitäten, welche im Rahmen einer Safety-Analyse durchgeführt werden. Dabei konnten keine Ansatzpunkte für Taskparallelität entdeckt werden, doch ging aus der Untersuchung ein geeigneter, datenparalleler Ansatz hervor, welcher sich auch dadurch auszeichnet, dass er das Ziel der Skalierbarkeit bezüglich der Prozessorzahl unterstützt.

[2] Zwei Teilbäume sind disjunkt, wenn der Schnitt ihrer Knotenmengen leer ist.

KAPITEL 4

Entwurf

Nachdem Kapitel 3 problemspezifisches Wissen erarbeitete, wird das vorliegende Kapitel nun den Entwurf der herzustellenden Software leisten. Ziel ist es dabei, Struktur und Verhalten der zu erstellenden oder zu ändernden Komponenten zu beschreiben.

Hierfür stellt das Kapitel in Abschnitt 4.1 zuerst ein allgemeines Konzept vor, welches dann in den Abschnitten 4.2f. zu einem Algorithmus sowie einer Strategie und schließlich in Abschnitt 4.4 zu einer Architektur verfeinert wird.

4.1 Konzept

Abschnitt 3.3 stellte fest, dass die entscheidende Möglichkeit zur Parallelisierung darin liegt, die Analyse disjunkter Teilbäume des Suchbaumes zu parallelisieren. Der vorliegende Abschnitt soll hierzu ein konkretes Konzept vorstellen; einerseits bezüglich des *Ablaufes* einer HRU-Safety-Analyse, andererseits bezüglich der *Struktur* der beteiligten Komponenten.

Ablauf der Analyse. Eine Heuristik führt, wie in Abschnitt 2.1.3 erläutert und auch aus Grafik 3.1 ersichtlich, in jedem Suchschritt drei wesentliche Aktionen durch: sie wählt einen günstigen Folgezustand, berechnet den entsprechenden Zustandsübergang und prüft schließlich auf Eintreten der Abbruchkriterien, insbesondere also auf das Vorhandensein einer Rechteausbreitung.

Dieser grundlegende Ablauf muss nun so erweitert werden, dass die Heuristik an geeigneten Punkten die Analyse eines Teilbaumes in einem neuen Thread veranlassen kann (vgl. Satz 3.9). Dazu wird ein soeben berechneter Zustand an einer geeigneten Stelle als Ausgangspunkt für die Berechnungen eines neuen Threads genutzt, statt ihn der berechnenden Heuristik zur Weiterarbeit zu überlassen.

Mit dieser Änderung sind einige strategische Entscheidungen verknüpft:

- Wie viele Threads sollten zu einem Zeitpunkt maximal an der Analyse beteiligt sein? (*Höchstgrad der Parallelität*)
- Unter welchen Voraussetzungen sollte ein neuer Thread gestartet werden? (*Verzweigungskriterium*)
- Unter welchen Voraussetzungen sollte eine bereits laufende Analyse zugunsten einer anderen Analysemöglichkeit abgebrochen werden? (*Ersetzungskriterium*)

Die Beantwortung der Fragen ist strategieabhängig und nicht mit den allgemeinen Konzepten und Strukturen der Software verbunden. Sie werden daher getrennt in Abschnitt 4.3 betrachtet.

Struktur der Analysesoftware. Ein Teil der Software, welche im Rahmen dieser Arbeit erstellt wird, beschäftigt sich mit der Kontrolle der Parallelität. Ihm werden folgende Verantwortlichkeiten zufallen:

- Starten und Beenden von Threads,
- Koordination der ablaufenden Threads, dabei
  - Zuteilen der Aufgaben unter den ablaufenden Threads sowie
  - Austausch von Informationen zwischen den Threads.

Eine wichtige Entwurfsentscheidung besteht nun in der Frage, welche Komponente diese Verantwortlichkeiten tragen soll. Möglich wäre es einerseits, die entsprechenden Funktionen in eine vorhandene Komponente einzubauen, also in den Simulator oder die Heuristik, oder aber andererseits, hierfür eine getrennte Komponente zu entwickeln, welche dann nur mit der Kontrolle der Safety-Analyse beauftragt ist.

Hier spielen zwei Aspekte eine entscheidende Rolle. Einerseits muss die im Rahmen dieser Arbeit erstellte Software gemäß den Anforderungen 3.4 und 3.5 erweiterbar bleiben, und zwar bezüglich der eingesetzten Heuristik sowie der angewendeten Strategien. Um dies zu gewährleisten, sollten diese beiden Komponenten möglichst separat zu anderen Funktionalitäten angeordnet werden.

Andererseits stellt sich die Frage, ob die verschiedenen, an der Analyse beteiligten Threads zentral oder dezentral koordiniert werden sollen. Diesbezüglich ist festzustellen, dass das zugrundeliegende System lediglich parallel und nicht verteilt ist, sodass ein zentraler Koordinater vor allem aus Performanzgründen vorzuziehen ist.

Der folgende Abschnitt 4.2 beschreibt den resultierenden Algorithmus zur Synchronisation der Threads.

4.2 Der *Parallel-Leak*-Algorithmus

Dieser Abschnitt wird die in 4.1 erläuterten Konzepte zu einem Algorithmus verdichten. Der *Parallel-Leak*-Algorithmus leitet sich dabei vom prinzipiellen Vorgehen einer in Anforderung 3.5 beschriebenen Heuristik her und erweitert diese in einem ersten Schritt um Parallelität. In einem zweiten Schritt werden Kontrollmechanismen eingeführt, welche eine differenzierte Entscheidung darüber erlauben, an welchen Punkten Parallelität eingesetzt werden soll. Zuvor wird jedoch ein konkretes Modell der Heuristik entwickelt, auf dessen Annahmen der Algorithmus aufbauen kann.

4.2.1 Modell der Heuristik

Wie die Anforderungen 3.4 (Erweiterbarkeit) und 3.5 (Art der Heuristik) verlangen, geht diese Arbeit im Entwurf von einer verallgemeinerten Heuristik aus. Die folgenden Absätze erläutern die Anforderungen an solche Heuristiken.

Dieses Kapitel nimmt an, dass eine Heuristik ihren eigenen, inneren Zustand besitzt. Auf diesem Zustand wird eine Reihe von Operationen definiert, welche die Algorithmen der Heuristik kapseln.

Es wird weiterhin gefordert, dass die Suchgraphen, auf denen die Heuristiken arbeiten, Bäume sind (siehe Abschnitt 2.1.3), und dass die Berechnungen der Heuristik deterministisch ablaufen (siehe Anforderung 3.6, Evaluierbarkeit).

Operationen. Gegeben sei ein HRU-Modell $H = (Q, \Sigma, \delta, q_0)$ mit anwendungsspezifischen Operationen A und einer Rechtemenge R. Sei Z die Menge aller Zustände der Heuristik.

Definiert seien folgende Operationen mit $r \in R$; $z, z' \in Z$; $q \in Q$; $(\alpha, x) \in \Sigma$; $v \in \mathbb{N}$; $b \in \{\text{true}, \text{false}\}$:

- INITIALIZE : $H, r \mapsto z$ erzeugt den initialen Zustand z der Heuristik. Ihr gilt zu diesem Zeitpunkt noch kein Knoten als erkundet, insbesondere nicht q_0.
- DISCOVER_NODE : $z, q \mapsto z'$ markiert Knoten q im Zustand z' der Heuristik als „erkundet“ (es gilt dann: die Heuristik *sieht* den Knoten). Der erste Aufruf dieser Operation nach der Initialisierung bezeichnet den Wurzelknoten des Suchbaums. Weitere Aufrufe ereignen sich nach Zustandsübergängen.
- CHOOSE_TRANSITION : $z \mapsto (z', q, (\alpha, x))$ wählt einen Zustandsübergang im HRU-Modell so, dass dieser eine Ausbreitung von r gegenüber q_0 begünstigt. q ist dabei ein bereits erkundeter Knoten. Der Übergang $(q, (\alpha, x))$ ist noch nicht durch die Heuristik gewählt worden.
- HEURISTIC_VALUE : $z, q \mapsto v$ erlaubt eine Ausgabe des heuristischen Wertes v zu Zustand q. Konform zu [Joh09] bezeichnen hierbei höhere Werte eine bessere Bewertung.
- SEARCH_TREE_EXHAUSTED : $z \mapsto b$ prüft eine notwendige Bedingung für das Durchführen der Analyse: ist der Suchbaum, auf dem die Heuristik abläuft, endlich, und wurde darin jeder mögliche Zustandsübergang durchgeführt, gibt der Aufruf „true“, andernfalls „false“ zurück.

4.2.2 Parallelität

Aus dem allgemeinen Ablauf von Heuristiken zur Safety-Analyse entsteht im folgenden ein paralleler Algorithmus, welcher ein fork/join-Modell zur Parallelisierung einsetzt. Er verwendet mehrere, nebenläufige Tasks. Das „Fork“ entspricht dann der Erstellung eines neuen Tasks; das „Join“ dem bedingten oder unbedingten Beenden eines Tasks. Die hier eingesetzte „Join“-Semantik implementiert eine atomare Operation (siehe unten), eine darüber hinausgehende Synchronisation erfolgt nicht.

Der Algorithmus wird in Listing 4.1 dargestellt.

Merkmale des Algorithmus. Der Algorithmus nimmt ein HRU-Modell, ein Recht sowie eine Höchstschrittzahl als Eingabe entgegen und sucht dann heuristisch unter Einsatz mehrerer, paralleler Tasks nach Rechteausbreitungen, welche bei Fund ausgegeben werden. Er terminiert, wenn alle Zustände des Suchbaums erkundet wurden oder die Höchstschrittzahl erreicht ist. Der Algorithmus ist deterministisch, was Anforderung 3.6 (Evaluierbarkeit) zuträglich ist.

Die Zahl durchgeführter Suchschritte wird von allen ablaufenden Tasks gemeinsam genutzt (benennt also nicht etwa die Suchtiefe pro Task). Sie gestattet daher Aussagen über die Zahl der insgesamt erkundeten Knoten und ist damit ebenfalls zu Anforderung 3.6 konform.

Der Algorithmus verzichtet auf die explizite Darstellung von Synchronisation. Bezüglich der Schrittzahl wird atomarer Zugriff vorausgesetzt. Eine JOIN()-Operation in Zeile 9 ermöglicht das korrekte Durchsetzen der Höchstschrittzahl. Das JOIN(v, z) führt

```
1  Eingabe: HRU-Modell H = (Q, Σ, δ, q_0) mit anwendungsspez. Op. A und
       Rechtemenge R; Recht r ∈ R; Obergrenze für Schrittzahl ŝ ∈ ℕ
2  Ausgabe: Zustände, die eine Rechteausbreitung enthalten
3  Variablen: Zustände q, q' ∈ Q; Eingabe (α, x) ∈ Σ; Heuristikzustand z;
       Schrittzahl s ∈ ℕ (global genutzt)
4
5  z ← INITIALIZE(H, r)
6  z ← DISCOVER_NODE(z, q_0)
7  s ← 0
8  while ¬ SEARCH_TREE_EXHAUSTED(z) do
9      JOIN(s = ŝ, s ← s + 1)
10     (z, q, (α, x)) ← CHOOSE_TRANSITION(z)
11     q' ← δ(q, (α, x))
12     if UNSAFE(q_0, q', r) then
13         out q'
14     end if
15     case FORK() is
16         no_fork:
17             z ← DISCOVER_NODE(z, q')
18         child:
19             z ← INITIALIZE(H, r)
20             z ← DISCOVER_NODE(z, q')
21         parent:
22             (no operation)
23     end case
24 end while
25 JOIN()
```

Listing 4.1: Parallelisierte heuristische Safety-Analyse unter Einsatz von fork/join.

atomar zuerst den Vergleich v durch – wenn der Vergleich zutrifft, wird der betreffende Task beendet, andernfalls die Zuweisung z angewendet.

Der eingesetzte FORK()-Aufruf (Zeile 15) erzeugt einen Task, der zum aufrufenden Task bis auf den Rückgabewert des FORK()-Aufrufs identisch ist: es wird entsprechend *child* bzw. *parent* zurückgegeben. Wurde der Task nicht erzeugt, wird *no_fork* retourniert. Mit diesem speziellen Fork-Modell können Mechanismen, welche die Parallelität kontrollieren, verborgen werden.

Die eingesetzte Heuristik entspricht der Spezifikation aus Abschnitt 4.2.1.

Ablauf. In Zeile 5 beginnt die Initialisierung des Algorithmus, dies beinhaltet insbesondere die Initialisierung der Heuristik.

Die nachfolgende Schleife, welche den restlichen Teil des Algorithmus umschließt, sichert über deren Laufbedingung ¬ SEARCH_TREE_EXHAUSTED(z) in Zeile 8 das Vorhandensein von nicht genutzten Zustandsübergängen, und damit eine notwendige Bedingung zum Fortfahren, ab. Jeder Durchlauf durch die Schleife entspricht einem Schritt der Heuristik.

Das JOIN()-Kommando in Zeile 9 sichert eine weitere notwendige Bedingung zum Fortfahren ab: die Zahl der durchgeführten Suchschritte s darf deren obere Schranke $\hat{s}$ nicht überschreiten (siehe oben). Ist dem so, wird der Schrittzähler erhöht, andern-

falls der Task beendet. Die atomare Ausführung der Operationen wird benötigt, um die Schrittzählerkorrektheit zusichern zu können.

Anschließend führt die Heuristik einen Zustandsübergang durch. Dieser wird berechnet – hierbei wird Zustand q in Zustand q' überführt – und auf eine Rechteausbreitung geprüft (Zeilen 10–14). Bei Vorliegen einer Ausbreitung erfolgt deren Ausgabe über die „out“-Operation, welche keine Auswirkungen auf den weiteren Programmablauf hat.

Die Zeilen 15 ff. schließlich behandeln einerseits das Erstellen eines neuen Tasks über Fork(), andererseits auch das Vermerken des soeben erkundeten Knotens q' im Heuristikzustand z. Findet kein Fork statt (Zeile 17), wird der Zustand schlicht vermerkt; andernfalls erhält der erzeugte Task den Zustand q' als Wurzelknoten für den von ihm abzusuchenden Teilbaum, während der erzeugende Task den Zustand nicht weiter behandelt.

Ein Task, welcher in seinem Teil des Suchbaums alle Übergänge erschöpft hat, beendet sich über das abschließende, bedingungslose Join(). Erreicht die Schrittzahl ihre Obergrenze, beenden sich alle Tasks über das erwähnte Join() in Zeile 9.

4.2.3 Nachweis der Eigenschaften

Dieser Abschnitt weist zwei wesentliche Eigenschaften des Algorithmus nach. Dies ist einerseits gemäß Anforderung 3.7 dessen Korrektheit. Andererseits soll zur Unterstützung der Anforderung 3.6 (Evaluierbarkeit) gezeigt werden, dass die Ergebnisse einer parallelen Ausführung der Heuristik mit denen einer nicht-parallelen Ausführung vergleichbar sind.

Zuvor jedoch beweist ein Hilfssatz, dass sich jeder erkundete Knoten im Suchbaum genau einem Task des Algorithmus zuordnen lässt. Diese Eigenschaft wird später zum Beweis der Vergleichbarkeit benötigt.

Lemma 4.1 (Sichtbarkeit eines Knotens).
Es gilt für jeden erkundeten Knoten q des Suchbaums: q kann von der Heuristik genau eines Tasks gesehen werden.

Beweis. Die Behauptung des Lemmas wird induktiv über die Schritte gezeigt, die der Algorithmus durchführt.

Induktionsanfang (Initialisierung). Der betrachtete Task fügt den Knoten q_0 ein. Da zu diesem Zeitpunkt nur ein Task existiert, kann nur die Heuristik dieses Tasks den Knoten sehen. q_0 ist der zu diesem Zeitpunkt einzige erkundete Knoten, es folgt die Behauptung.

Induktionsschritt (Schleifendurchlauf). Es gelte die Behauptung bis zum Beginn des Schleifendurchlaufs. Der betrachtete Task führt nun genau eine der folgenden drei Aktionen durch.

Terminierung wegen Erreichen der Höchstschrittzahl (Zeile 9). Dann wird kein Knoten sichtbar, unsichtbar oder erkundet.

Erkunden eines Knoten q ohne anschließendes Fork. Dann wird q in Zeile 17 für genau diejenige Heuristik sichtbar, welche im betrachteten Task abläuft. Die Sichtbarkeit anderer Knoten wird nicht beeinflusst.

Erkunden eines Knoten q mit anschließendem Fork. Dann bleibt q für den erzeugenden Task unsichtbar, da kein Discover_Node() aufgerufen wird. Die Heuristik des erzeugten Tasks wird neu initialisiert und sieht daher genau einen Knoten, nämlich q (wird sichtbar in Zeile 20).

Es folgt die Behauptung. □

Zur besseren Darstellung der Sätze und Beweise wird im folgenden eine Funktion definiert, welche eine Eingabe des Algorithmus auf die zugehörige Ausgabe abbildet.

Definition 4.2 (Berechnete Funktion).
Zu einem HRU-Modell H mit anwendungsspezifischen Operationen A und Rechtemenge R sei die vom Algorithmus berechnete Funktion $f_H : R \times \mathbb{N} \to \mathcal{P}(Q)$ definiert als diejenige Funktion, welche für $f_H : r, \hat{s} \mapsto L$ die Eingabe $r, \hat{s}$ auf die Menge L der bei Ausführung des Algorithmus ausgegebenen Zustände abbildet[1].

Der folgende Satz schließt aus obigem Lemma, dass – unter bestimmten Voraussetzungen – die Ergebnisse einer linearen[2]Ausführung des Algorithmus eine Teilmenge der Ergebnisse einer parallelen Ausführung bilden. Dies hilft maßgeblich dabei, die Leistungen dieser Arbeit zu evaluieren.

Satz 4.3 (Vergleichbarkeit der Ergebnisse).
Gegeben sei ein HRU-Modell H mit anwendungsspezifischen Operationen A und Rechtemenge R. Seien f_H^l und f_H^p die von einer linearen bzw. parallelen Ausführung des Algorithmus berechneten Funktionen. Beide Funktionen setzen die selbe Heuristik ein, für die weiterhin gelte, dass CHOOSE_TRANSITION bei der Wahl des Übergangs $\delta(q, (\alpha, x))$ die Wahl der Eingabe (α, x) nicht davon abhängig macht, welche erkundeten Zustände sie sehen kann.

Sei nun $r \in R$ und $\hat{s}_l \in \mathbb{N}$. Es existiert ein $\hat{s}_p \in \mathbb{N}$, sodass gilt: $f_H^l(r, \hat{s}_l) \subseteq f_H^p(r, \hat{s}_p)$.

Beweis. Die Heuristik wählt in jedem Berechnungsschritt einen Übergang von einem erkundeten Zustand q zu einem nicht erkundeten Zustand q'. Die Heuristiken der parallelen und der linearen Ausführung verwenden identische heuristische Funktionen.

Da bei der parallelen Ausführung jeder erkundete Knoten des Suchbaums einem Task zugeordnet werden kann (Lemma 4.1), und da weiterhin die Auswahl des Übergangs nicht von den sichtbaren Zuständen abhängig ist, folgt, dass einige der parallel ausgeführten Heuristiken bei ausreichend hoher Schrittzahl auch all jene Zustände wählen, welche auch die Heuristik unter linearer Ausführung wählt.

Dies war zu zeigen. □

Zuletzt wird, analog zu [Joh09], die Korrektheit des Algorithmus über die Korrektheit der Ausgabe nachgewiesen.

Satz 4.4 (Korrektheit der Ausgabe).
Die Ausgabe des Algorithmus ist korrekt, das heißt, für alle gültigen Eingaben $H = (Q, \Sigma, \delta, q_0), r, \hat{s}$ gilt, dass für $f_H : r, \hat{s} \mapsto L$ gilt: alle $q \in L$ sind ausgehend von q_0 durch eine endliche Folge gültiger Zustandsübergänge zu erreichen und enthalten eine Ausbreitung des Rechtes r bezüglich q_0.

Beweis. Der Algorithmus führt durch die obere Schranke der auszuführenden Schritte endlich viele Schritte durch. Weiterhin wird ein Zustand (nicht jedoch q_0) genau dann erkundet, wenn er aus einem erkundeten Zustand berechnet wurde. Zu Beginn des Algorithmus ist nur q_0 erkundet.

Notwendige Bedingungen für die Ausgabe eines Zustandes sind erstens, dass er aus einem erkundeten Zustand berechnet wurde, und zweitens, dass er eine Ausbreitung des Rechtes r bezüglich q_0 enthält. Es folgt die Behauptung. □

[1] f_H ist eine Funktion, da sie rechtseindeutig (der Algorithmus ist deterministisch) und linksvollständig (zu jeder Eingabe liefert der Algorithmus eine Ausgabe) ist.

[2] Nicht-parallele bzw. lineare Ausführung des Algorithmus lässt sich erzielen, indem FORK() im Algorithmus grundsätzlich keinen neuen Task erzeugt, sondern immer „no_fork“ retourniert.

Satz 4.5 (Baum über Fork-Relation).
Sei $G_F = (N_F, E_F)$ ein Graph, dessen Knotenmenge aus allen Heuristik-Tasks besteht, die während der Abarbeitung des Algorithmus ausgeführt wurden, und dessen Kantenmenge definiert sei als $E_F := \{(t_x, t_y) \in N_F^2 \mid t_y$ wurde gestartet durch einen FORK()-Aufruf in $t_x\}$. Sei weiterhin $t_i \in N_F$ der zuerst angelaufene Heuristik-Task. Es gilt: G_F ist ein Baum mit Wurzelknoten t_i.

Beweis. Sei $\forall t \in N_F : \mathrm{In}(t) := |\{s \in N_F \mid (s,t) \in E_F\}|$ der Eingangsgrad des Knotens t. Zum Nachweis der Baumeigenschaft reicht es zu zeigen, dass $\forall t_x \in N_F \backslash t_i : |\mathrm{In}(t_x)| = 1$ und $|\mathrm{In}(t_i)| = 0$. Dies trifft zu, da t_i von keinem Knoten aus N_F erzeugt wurde, und jeder andere Task durch genau einen FORK()-Aufruf hervorgegangen ist. □

4.2.4 Kontrolle der Parallelität

Der in 4.2.2 gezeigte Algorithmus würde in jedem Schritt verzweigen, wenn der FORK()-Aufruf stets einen neuen Task erzeugt. Daher muss an dieser Stelle die Parallelität kontrolliert werden[3]. Dabei sollten zum einen harte Grenzen bezüglich des Grads der Parallelität anwendbar sein; zum anderen sollte eine Strategie eingesetzt werden, um Parallelität an möglichst günstigen Stellen einsetzen zu können.

Dieser Abschnitt wird einen parallelen Algorithmus, modelliert als verteilten, asynchronen Algorithmus, aufzeigen. Der Algorithmus wird an Stelle des FORK()-Aufrufes einen zentralen Koordinator ansprechen, welcher – ausgestattet mit dem Überblick über alle ablaufenden Tasks – die Strategieentscheidungen treffen kann. Um Fairness zu garantieren, wird diesem zentralen Koordinator eine Queue in Form eines weiteren Tasks vorgeschaltet.

Topologische Beschreibung. Die Topologie des Algorithmus wird über den Graphen $G_T = (N_T, E_T)$ beschrieben, wobei die Knotenmenge N_T die Menge der Tasks angibt und die Kantenmenge E_T die Kommunikationsbeziehungen zwischen den Tasks repräsentiert. Eine Menge N_0 beschreibt die Menge jener Tasks, die spontan zu Beginn des Algorithmus anlaufen.

Ein Task des Modells entspricht einem Thread der Implementierung. Es wird davon ausgegangen, dass insgesamt $n+2$ Tasks zum Einsatz kommen, wovon einer der zentrale Koordinator und einer die Queue ist.

$$
\begin{aligned}
N_T &= \{t_{\text{control}}, t_{\text{queue}}\} \cup \{t_i \mid \forall i \in \{1, \ldots, n\}\} \\
E_T &= \{(t_{\text{queue}}, t_{\text{control}}), (t_{\text{control}}, t_{\text{queue}})\} \cup \\
&\quad \{(t_i, t_{\text{queue}}), (t_{\text{control}}, t_i), (t_i, t_i) \mid \forall i \in \{1, \ldots, n\}\} \\
N_0 &= \{t_{\text{control}}\}
\end{aligned}
$$

Idee. Eine genaue Betrachtung des Algorithmus in Listing 4.1 ergibt, dass dieser am Anfang (JOIN()-Aufruf) sowie am Ende (FORK()-Aufruf) eines jeden Schleifendurchlaufs wechselseitigen Ausschluss unter allen beteiligten Heuristik-Tasks realisiert. Die Idee für den verteilten Algorithmus besteht nun darin, den Koordinator als Ausführungsort für beide Aufrufe zu nutzen.

[3] Mechanismen, die gemäß dem Ersetzungskriterium einen laufenden Heuristik-Task abbrechen, werden im hier vorgestellten Algorithmus zur besseren Übersichtlichkeit nicht dargestellt. Sie sind allerdings in einer Implementierung ebenfalls zu berücksichtigen.

Die Heuristiken werden dabei in den einzelnen t_i ausgeführt. Berechnete Zustände werden dann an den Koordinator gesendet, welcher diese nacheinander abarbeitet und so den wechselseitigen Ausschluss realisiert. Er sichert solche Zustände, die gesuchte Rechteausbreitungen enthalten, trifft die Entscheidung über das Fork und verwaltet den Schrittzähler.

Die Queue. Die Funktionsweise des Knotens t_{queue} wird hier nicht algorithmisch beschrieben, da er lediglich die Semantik der bekannten Warteschlangen realisiert.

Der Knoten empfängt zwei Botschaften, nämlich „enqueue“ sowie „dequeue“. Empfängt t_{queue} eine Botschaft „enqueue(x)“, fügt er x an das Ende der Warteschlange an. Empfängt er eine Botschaft „dequeue()“ von einem Knoten $t_j \in E_T$, wird er dasjenige Element y, das an vorderster Stelle der Warteschlange steht, entfernen und über eine „item(t_i, y)“-Botschaft an t_j senden. t_i bezeichnet dabei denjenigen Knoten, welcher das Element ursprünglich angefügt hat. Sollte die Warteschlange leer sein, wird die „item“-Botschaft erst dann verschickt, wenn ein Element angefügt wird.

Auf diese Art und Weise wird eine Serialisierungsreihenfolge für den Koordinator festgelegt und damit Unfairness vermieden, welche durch nichtdeterministische Botschaftenauswahl im Modell eines verteilten Algorithmus entstehen kann.

Die Heuristik-Tasks. Listing 4.2 zeigt den Algorithmus eines Knoten $t_i, i \in \{1, \ldots, n\}$. Dieser ist in der Lage, zwei Botschaften zu empfangen: einerseits die „start“-Botschaft, welche zur Initialisierung der Heuristik dient, und andererseits eine „do_step“-Botschaft, welche je einen Berechnungsschritt ausführt.

Die „start“-Botschaft in Zeile 8 ruft den INITIALIZE()-Befehl der Heuristik auf und startet – über eine an sich selbst gerichtete „do_step“-Botschaft (Zeile 12) – den ersten Berechnungsschritt.

Die „do_step“-Nachricht in Zeile 14 veranlasst die Auswahl und Berechnung eines Zustandsübergangs, wie bereits im Mittelteil des via Fork/Join modellierten Algorithmus gezeigt (Listing 4.1). Zu Beginn jedoch gestattet es die Verzweigung in Zeile 17, einen via Botschaftenparameter übergebenen Zustand q_{discover} als erkundet zu markieren. Diese Zeile realisiert sämtliche DISCOVER_NODE()-Aufrufe aus Algorithmus 4.1.

Die Laufbedingung $\neg$ SEARCH_TREE_EXHAUSTED() wurde hier als Verzweigung realisiert. Ist sie nicht erfüllt, hat dies zur Folge, dass der Knoten keine weitere Botschaft aussendet und damit seine Arbeit einstellt. Eine Benachrichtigung hierüber an den Koordinators wäre denkbar.

Zum Schrittzähler: dieser wird zentral durch den Koordinator verwaltet, welcher ihn bereits vor Beginn eines Berechnungsschritts erhöht. Dies ist aus Konsistenzgründen nötig, denn beim nächsten Durchlauf muss der Koordinator die Höchstschrittzahl wieder überprüfen können. Allerdings ist es möglich, dass Schritte nicht erfolgreich abgeschlossen werden können; dies ist beispielsweise der Fall, weil der Suchbaum eines Tasks erschöpft ist. Eine Korrektur ist dann gegebenenfalls nötig.

```
1  Async parallel-leak t_i
2
3  Variables:
4      z ∈ Z: Heuristikzustand
5      q, q' ∈ Q: Zustände im HRU-Modell (lokale Variable)
6      (α, x) ∈ Σ: Eingabe für das HRU-Modell (lokale Variable)
7
8  Input:
9      msg = start(H, r, q_start) from (t_control, t_i) ∈ In(t_i)
10 Action:
11     z ← INITIALIZE(H, r)
12     send do_step(q_start) to (t_i, t_i) ∈ Out(t_i)
13
14 Input:
15     msg = do_step(q_discover) from (t_j, t_i) ∈ In(t_i)
16 Action:
17     if q_discover ≠ nil then
18         z ← DISCOVER_NODE(z, q_discover)
19     end if
20     if ¬ SEARCH_TREE_EXHAUSTED(z) then
21         (z, q, (α, x)) ← CHOOSE_TRANSITION(z)
22         q' ← δ(q, (α, x))
23         send enqueue(node(q', HEURISTIC_VALUE(z, q'))) to
               (t_i, t_queue) ∈ Out(t_i)
24     end if
```

Listing 4.2: Ein Heuristik-Task als verteilter, asynchroner Algorithmus.

Der Koordinator. Der Koordinator-Task, dargestellt in Listing 4.3, ist verantwortlich für das Zusammenspiel der einzelnen Tasks t_i. Er hat folgende Verantwortlichkeiten:

- Initialisieren des Algorithmus,
- Speichern derjenigen Zustände, welche eine Rechteausbreitung enthalten,
- Durchsetzen der Höchstschrittzahl,
- Kontrolle der Verzweigung („Fork“) sowie
- Abbruch von Tasks mit geringem Erfolg (hier nicht dargestellt).

Abstrakte Aufrufe. Der Algorithmus nutzt neben UNSAFE() sowie den bekannten Heuristik-Operationen zwei weitere Aufrufe, welche genauerer Erklärung bedürfen. Dies ist zum einen GET_UNUSED_TASK. Dieser gibt ein $i \in \{1, \ldots, n\}$ zurück für einen Task t_i, welcher derzeit keine Heuristik ausführt.

Der zweite Aufruf ist SHOULD_FORK : $i, v_1, \ldots, v_n \mapsto \{true, false\}$ mit $i \in \{1, \ldots, n\}$ und $\forall j \in \{1, \ldots, n\} : v_j \in \mathbb{N}$. Dieser Aufruf kapselt die Entscheidung darüber, ob eine Verzweigung via Fork stattfinden sollte oder nicht. Dabei wird zum einen die Obergrenze n für die Zahl der gleichzeitig auszuführenden Heuristik-Tasks durchgesetzt; zum anderen wird geprüft, ob das Parallelisieren der Berechnung im jeweiligen Kontext strategisch

```
1  Async parallel-leak t_control
2
3  Constants:
4      H = (Q, Σ, δ, q_0): HRU−Modell mit Operationen A und Rechten R
5      r ∈ R: Recht
6      ŝ ∈ ℕ: Obergrenze für Schrittzahl
7
8  Variables:
9      s ∈ ℕ: Schrittzahl
10     L ⊆ Q: gefundene Zustände mit Ausbreitungen
11     v_i ∈ ℕ ∀i ∈ {1, ..., n}: Zustandsbewertung für Task i
12     j ∈ {1, ..., n}: lokale Variable, Nummer eines Tasks
13
14 Input:
15     msg = nil
16 Action if t_control ∈ N_0:
17     s ← 1
18     L ← ∅
19     ∀i ∈ {1, ..., n} : v_i ← 0
20     j ← GET_UNUSED_TASK()
21     send start(H, r, q_0) to (t_control, t_j) ∈ Out(t_control)
22     send dequeue() to (t_control, t_queue) ∈ Out(t_control)
23
24 Input:
25     msg = item(t_i, node(q, v)) from (t_queue, t_control) ∈ In(t_control)
26 Action:
27     if UNSAFE(q_0, q, r) then
28         L ← L ∪ {q}
29     end if
30     v_i ← v
31     if (s + 2 ≤ ŝ) ∧ (SHOULD_FORK(i, v_1, ..., v_n)) then
32         s ← s + 2
33         j ← GET_UNUSED_TASK()
34         send start(H, r, q) to (t_control, t_j) ∈ Out(t_control)
35         send do_step(nil) to (t_control, t_i) ∈ Out(t_control)
36     else if s + 1 ≤ ŝ then
37         s ← s + 1
38         send do_step(q) to (t_control, t_i) ∈ Out(t_control)
39     end if
40     send dequeue() to (t_control, t_queue) ∈ Out(t_control)
```

Listing 4.3: Der Koordinator als verteilter, asynchroner Algorithmus.

sinnvoll ist. i bezeichnet den Index desjenigen Tasks, der zur Parallelisierung zur Verfügung steht; die v_j sind die aktuellen Zustandsbewertungen der einzelnen Heuristiken. Wird eine Parallelisierung empfohlen, wird „true“, andernfalls „false“ zurückgegeben.

Initialisierung (Zeilen 14–22). Der Task t_{control} ist in N_0, läuft also spontan an (Zeile 14). Er initialisiert dann Variablen und startet den initialen Heuristik-Task über die entsprechende „start“-Botschaft (Zeile 21). Zudem fordert er t_{queue} auf, den ersten eingereihten Zustand zuzustellen.

Zentrale Koordinatorroutine (Zeilen 24–40). Die Botschaft, welche der Koordinator verarbeiten kann, ist die „item“-Botschaft der Queue (Zeile 24). Sie trägt die Benachrichtigung node(q, v) über die Berechnung eines Knotens $q \in Q$ mit Bewertung $v \in \mathbb{N}$ in sich, welche ursprünglich von Knoten t_i stammte.

Enthält q eine gesuchte Rechteausbreitung, wird er zu L hinzugefügt, was der Ausgabe-Operation in des Algorithmus 4.1 entspricht. Anschließend wird die Zustandsbewertung im entsprechenden v_i vermerkt.

Nachfolgend ruft der Koordinator in Zeile 31 eine Entscheidung der Parallelisierungsstrategie ab und prüft die Höchstschrittzahl. Dies entspricht im Fork/Join-Modell des Algorithmus (Listing 4.1) dem FORK-Aufruf. Ist die Höchstschrittzahl erreicht, erhält kein Task ein weiteres Mandat für einen Berechnungsschritt, allerdings wird die Queue weiterhin abgefragt, um alle bereits begonnenen Algorithmusschritte abschließen zu können.

Ist die Höchstschrittzahl nicht erreicht, hängt das weitere Vorgehen vom Ergebnis des SHOULD_FORK-Aufrufs in Zeile 31 ab. Gibt dieser „false“ zurück, wird keine Verzweigung durchgeführt. Der Koordinator sendet t_i dann über die „do_step“-Botschaft die Aufforderung zu, einen weiteren Berechnungsschritt durchzuführen. Dabei erhält t_i den Zustand q als Parameter und kann ihn daher als „erkundet“ markieren.

Retourniert die Parallelisierungsstrategie „true“, verlangt sie eine Verzweigung der Berechnungen. Dann wird in Zeile 34 ein neuer, ungenutzter Task gestartet, welcher q als Wurzelknoten seines Teilbaums erhält. Auch t_i erhält via „do_step“ die Aufforderung zu einer weiteren Berechnung, diesmal allerdings ohne die Übergabe von q. Dies hat zur Folge, dass q *nicht* durch t_i als „erkundet“ markiert und daher aus dem von t_i bearbeiteten Suchbaum entfernt wird.

In jedem Fall wird der Schrittzähler um die Zahl der beauftragten Berechnungsschritte erhöht: im Verzweigungsfall um zwei, bei linearer Fortführung der Berechnungen um eins.

Zuletzt informiert der Koordinator t_{queue} darüber, dass wieder ein Element der Warteschlange zugestellt werden soll.

Terminierung. Der Algorithmus terminiert, sobald kein Task mehr arbeitet und die Queue leer ist.

4.3 Strategien zur Parallelitätskontrolle

Der im vorhergehenden Abschnitt vorgestellte *Parallel-Leak*-Algorithmus lässt strategische Entscheidungen zur Taskkoordination offen. Ziel dieses Abschnittes ist es, diese Lücke zu schließen und eine erste simple, aber funktionsfähige Strategie zu entwickeln. Dabei werden eingangs einige Annahmen über die Analyseumgebung getroffen, aus welchen dann die Strategie selbst entwickelt wird.

4.3.1 Ziele und Annahmen

Wie bereits im Konzept (Abschnitt 4.1) benannt, können die Fragen nach dem Höchstgrad der Parallelität, dem Verzweigungs- sowie dem Ersetzungskriterium sehr unterschiedlich beantwortet werden. Entscheidende Faktoren sind hier in allen Unbekannten zu finden:

- **Die eingesetzte Systemumgebung.** Insbesondere spielt hier eine Rolle, wie viele Tasks das System tatsächlich parallel abarbeiten kann; zudem sollten spezielle Leistungsmerkmale beachtet werden.
- **Die verwendete Heuristik.** Vom Verhalten der Heuristik auf dem Suchbaum hängen sämtliche Entscheidungen ab. Hier ist vorallem darauf zu achten, inwiefern (un)günstige Wege tatsächlich anhand der Zustandsbewertung vorhergesagt werden können und welchen Einfluss das Einschränken der Zustandsauswahl auf die Qualität der Ergebnisse hat.
- **Das eingesetzte Modell.** Eine Strategie, welche gute Ergebnisse erzielen soll, muss ebenfalls auf das konkrete HRU-Modell abgestimmt sein. Insbesondere die Beschaffenheit des Suchbaums kann hier als Kriterium dienen; so bieten beispielsweise breitere Suchbäume auch die Möglichkeit, schnell viele parallel ablaufende Tasks zu erzeugen.

Da die Zielstellung der Arbeit keine konkreten Vorgaben zu diesen Einflussfaktoren macht, sollen im folgenden lediglich einfache und möglichst praxisnahe Annahmen zum Verhalten der Heuristik auf dem jeweils eingegebenen Modell getroffen werden.

Diese sind:

1. Die Zustandsbewertung eines Heuristik-Tasks ist proportional zu der Wahrscheinlichkeit, dass dieser Task eine Rechteausbreitung findet.
2. Die Zustandsbewertungen eines Heuristik-Tasks wachsen bei zunehmender Schrittzahl.
3. Eine Verzweigung eines Tasks bringt für beide entstehenden Zweige keine Verschlechterung der Zustandsbewertung.
4. Auf lange Sicht benötigt ein Heuristik-Task eine große Auswahl an bekannten Knoten, um qualitativ gute Ergebnisse zu erzeugen.

4.3.2 Strategieentwicklung

Anhand der oben genannten Annahmen sollen die einzelnen strategischen Entscheidungen nachfolgend behandelt werden.

Höchstgrad der Parallelität. Die Höchstzahl der einzusetzenden Tasks orientiert sich insbesondere am Modell, an der Heuristik sowie an der Systemumgebung. Für einfache Computersysteme gilt hier jedoch, dass die Zahl der zur Verfügung stehenden Prozessoren / Kerne meist sowieso schon sehr stark einschränkt.

Da die Annahmen keine Anhaltspunkte für die Beantwortung der Frage bieten, soll der Höchstgrad der Parallelität in einer Strategieimplementierung vom Benutzer konfigurierbar sein.

Verzweigungskriterium. Aus den gemachten Annahmen, insbesondere den Punkten 2 und 3, folgt, dass es keine explizit *schlechten* Zeitpunkte für eine Verzweigung gibt. Folglich kann verzweigt werden, sobald die Möglichkeit hierzu besteht – die Einhaltung der Höchstzahl an zu verwendenden Tasks dient hier also als notwendiges Kriterium.

Bleibt noch die Frage nach dem hinreichenden Kriterium: *welcher* Heuristik-Task soll verzweigt werden? Hier legen die Annahmen, insbesondere Punkt 1, nahe, dass der Task mit der zum jeweiligen Zeitpunkt besten Zustandsbewertung am wahrscheinlichsten zu einer Rechteausbreitung führt. Er soll daher verzweigt werden.

Ersetzungskriterium. Um die Frage zu beantworten, wann ein Heuristik-Task besonders ungeeignet ist, sodass seine Abarbeitung abgebrochen werden sollte, lässt sich zunächst ein ähnliches, hinreichendes Kriterium wie oben formulieren: Punkt 1 lässt hier erkennen, dass diejenige Heuristik mit der schlechtesten Zustandsbewertung am unwahrscheinlichsten zu einer Rechteausbreitung führt.

Doch gilt es hier zu bedenken, dass die Strategie bei der Betrachtung des Ersetzungskriteriums rückblickend auch ältere Werte zur Entscheidungsfindung nutzen kann. Immerhin wachsen die Werte gemäß Punkt 2, wodurch es möglich wird, dass ein Task einen anderen bezüglich der Bewertung „überholt“. Entsprechend kann hier der Durchschnitt über die jeweils n letzten Zustandsbewertungen als Entscheidungsgrundlage dienen.

Bezüglich des notwendigen Kriteriums steht zu bedenken, dass eine Heuristik (wie in Punkt 4 beschrieben) eine gewisse Auswahl an bekannten Knoten benötigt, um gute Ergebnisse zu erzeugen. Häufiger Abbruch laufender Heuristik-Tasks führt jedoch zu häufigen Verzweigungen, welche letztlich für eine sehr kurze Laufzeit der Tasks sorgen – und damit für eine sehr kleine Zustandsauswahl.

Daher muss vermieden werden, Heuristiken zu oft zu ersetzen. Als notwendiges Kriterium der hier beschriebenen Strategie soll daher eine „Gnadenfrist“ dienen, eine konfigurierbare Zahl m an Berechnungsschritten. Die Semantik: innerhalb von m Schritten nach Start eines Heuristik-Task wird dieser Task nicht abgebrochen.

Die so beschriebene Strategie wird im Rahmen der Arbeit implementiert, Details finden sich im Implementierungskapitel in Abschnitt 5.3.

4.4 Architektur

Die in den vorangegangenen Abschnitten gezeigten Konzepte und Algorithmen verdichtet der vorliegende Abschnitt zu einer Architektur. Vorgestellt werden dabei die wesentlichen Neuerungen im Vergleich zum bisherigen Stand von *I-Graphoscope*. Am Ende des Abschnittes steht ein Gesamtüberblick über Statik und Dynamik der neuen Simulationsumgebung.

Auf Grundlage der hier gezeigten Informationen wird Kapitel 5 dann den Feinentwurf sowie die resultierende Implementierung darstellen.

4.4.1 Kontrollkomponente

Aus dem Konzept in Abschnitt 4.1 geht hervor, dass eine separate Komponente die Safety-Analyse kontrollieren soll. Zum bisherigen Stand der Entwicklung von *I-Graphoscope* ist hierfür die Klasse `IGraphSimulator` zuständig (siehe Abschnitt 2.2.2): sie lädt die Heuristik, initialisiert alle nötigen Daten und bietet an ihrer Schnittstelle auch Methoden zur Steuerung des Vorgangs durch den Benutzer (Pause, Abbruch etc.). Die Platzierung

der `IGraphSimulator`-Klasse im Paket `core` spricht für ihre kontrollierende Aufgabe. Sie ist also prädestiniert zur Steuerung des Analysevorgangs.

Dennoch verfügt die Klasse über eine zweite Verantwortlichkeit, welche hier aus den im Konzept genannten Gründen getrennt werden sollte: dies ist die Berechnung von Zustandsübergängen und die Verwaltung derjenigen Datenstrukturen, die es ermöglichen, bereits errechnete Zustände abzufragen. Um Anforderung 3.4 (Erweiterbarkeit) zu berücksichtigen und insbesondere der damit verbundenen Notwendigkeit der Wartbarkeit genüge zu tun, werden diese Verantwortlichkeiten in die separate Klasse `HruAutomaton` ausgelagert.

Ebenfalls getrennt implementiert wird der eigentliche Koordinator (siehe Task t_{control} in Abschnitt 4.2). Diese Entscheidung lässt sich mit Anforderungen 3.4 (Erweiterbarkeit) und 3.7 (Korrektheit) begründen: da der Koordinator in einem eigenen Thread ablaufen wird, lassen sich seine Datenstrukturen strukturell von denen des Simulators trennen, was den Nachweis der Korrektheit erleichtert. Die Trennung der Verantwortlichkeiten sorgt für die geforderte Erweiterbarkeit. Es entsteht die `Coordinator`-Klasse, die von `Thread` erbt. Um die Klasse `IGraphSimulator` als zentrale Schnittstelle etablieren zu können, sollte die Klasse `Coordinator` nur dieser bekannt sein. Das erleichtert spätere Änderungen in der Simulationsarchitektur.

4.4.2 Heuristik

Für die Implementierung der Heuristik-Tasks des verteilten Algorithmus wird eine Klasse benötigt, die den jeweils ablaufenden Thread repräsentiert, also beispielsweise von `Thread` erbt.

Die `Heuristic`-Klasse erfüllt diese Anforderung bereits. In Erweiterung zum bisherigen Stand von *I-Graphoscope* kommen ihr allerdings neue Funktionen zu. Dies sind:

- die Kommunikation zum Koordinator,
- die Möglichkeit des gleichzeitigen Einsatzes mehrerer, paralleler `Heuristic`-Threads, sowie
- die Verwendung der `HruAutomaton`-Objekte.

In einer Heuristik-Implementierung müssen diese Punkte berücksichtigt werden. Detailliertere Anforderungen an solche Implementierungen nennt das Implementierungskapitel in Abschnitt 5.2.1.

4.4.3 Berechnungen am HRU-Modell

Wie oben erwähnt, findet die Berechnung von Zustandsübergängen sowie die Aufbewahrung zugehöriger Informationen in einer separaten Klasse, der `HruAutomaton`, statt.

Die darin gespeicherten Informationen sind insbesondere solche, anhand derer sich beliebige, bereits erkundete Zustände des Suchbaums rekonstruieren lassen. Der wechselseitige Ausschluss beim Zugriff auf diese Daten zwischen den verschiedenen Heuristik-Threads wird hier *by design* gewährleistet: jeder `Heuristic`-Instanz ist eine Instanz der `HruAutomaton`-Klasse zugeordnet, die nur solche Daten enthält, welche zum Such(teil)baum des Heuristik-Threads gehören.

Um später, ausgehend von der grafischen Oberfläche, Zustände aus allen errechneten Bereichen des Suchbaumes rekonstruieren zu können, sind die `HruAutomaton`-Objekte baumartig miteinander verbunden. Dies ist aufgrund von Satz 4.5 möglich. Um den

wechselseitigen Ausschluss zwischen der GUI und den Heuristik-Threads zu realisieren, darf die Nutzeroberfläche genau dann zugreifen, wenn die Heuristik nicht arbeitet, das heißt, wenn die Simulation pausiert oder beendet ist.

4.4.4 Parallelisierungsstrategie

Die vom Koordinator abgefragte Parallelisierungsstrategie, welche im *Parallel-Leak*-Algorithmus durch die FORK- beziehungsweise SHOULD_FORK-Aufrufe gekapselt wird, kann sehr unterschiedliche Entscheidungsalgorithmen verwenden; die Einflussfaktoren stellte Abschnitt 4.3 dar.

Aus diesem Grund wird eine getrennte Klasse geschaffen, die `ParallelismStrategy`, welche als abstrakte Oberklasse für Strategieimplementierungen dient. Diese Arbeit wird bereits einige konkrete Strategien implementieren; nähere Informationen bietet Abschnitt 5.3.

4.4.5 Gesamtarchitektur der Simulationsumgebung

Aus den oben beschriebenen Konzepten ergibt sich die nachfolgend dargestellte Gesamtarchitektur der Simulationsumgebung. Aufgezeigt werden die statische und die dynamische Sicht auf die enthaltenen Komponenten.

Statik. Abbildung 4.1 stellt die beteiligten Objekte und ihre Beziehungen zueinander dar. Die Grafik ist dabei als ein Schnappschuss zur Laufzeit zu verstehen.

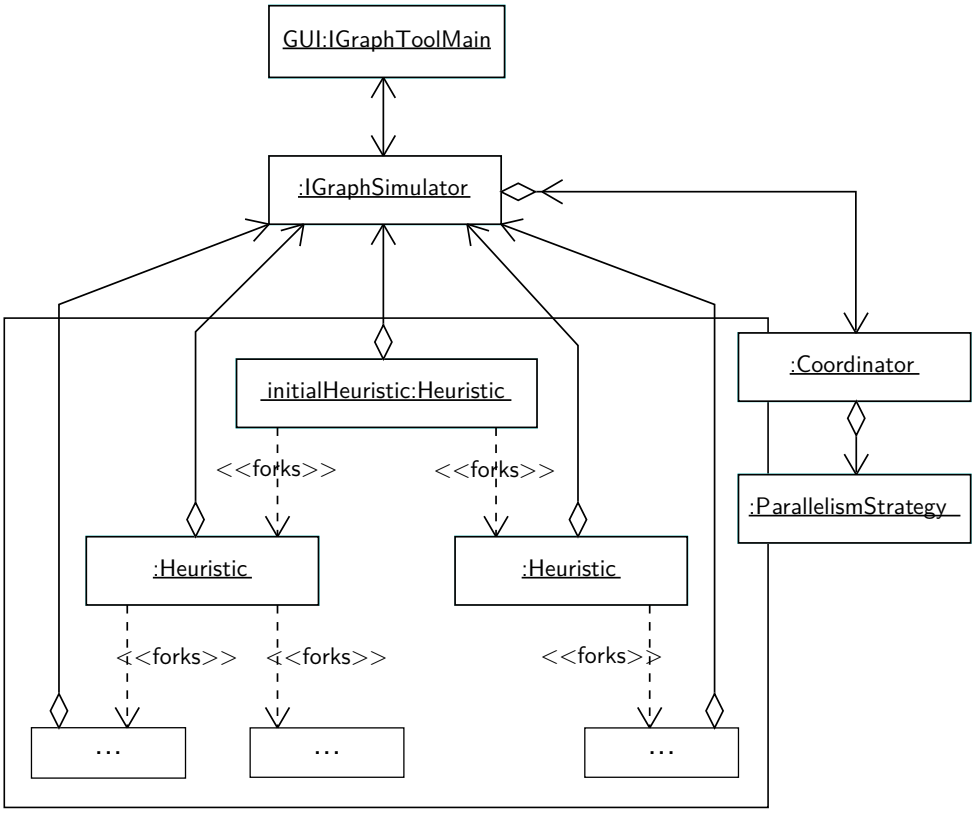


Abbildung 4.1: An der Safety-Analyse beteiligte Objekte im UML-Objektdiagramm. Dargestellt sind neben der Simulationsumgebung mehrere, zur Laufzeit erzeugte Heuristikobjekte und deren Beziehungen.

Wie bisher (siehe Abschnitt 2.2.2) wird die grafische Nutzeroberfläche durch die `IGraphToolMain`-Klasse repräsentiert, welche die Simulation über die Klasse `IGraphSimulator` beeinflussen kann.

Ausgehend vom Simulator laufen nun mehrere `Heuristic`-Objekte ab, welche den Heuristik-Tasks im *Parallel-Leak*-Algorithmus (Abschnitt 4.2) entsprechen und jeweils einen separaten Thread repräsentieren. Durch diejenige Relation, welche durch die FORK()-Operation induziert wird, lassen sich die einzelnen Heuristik-Objekte wie in Satz 4.5 baumartig anordnen.

Nicht in der Grafik dargestellt sind die oben beschriebenen `HruAutomaton`-Instanzen, von denen je eine das Aggregat eines Heuristik-Objektes ist.

Der, wie oben begründet, getrennt angelegte Koordinator wird ausgehend vom Simulator erzeugt und repräsentiert ebenfalls einen eigenen Thread. Zu der Entscheidung, an welcher Stelle die Berechnungen verzweigt werden sollen, befragt er ein `ParallelismStrategy`-Objekt.

Dynamik. Aufgrund der aufgezeigten Konzepte entsteht eine grundlegend neue Dynamik zwischen denjenigen Komponenten, die an einer Safety-Analyse beteiligt sind. Sie soll anhand eines Kollaborationsdiagramms in Abbildung 4.2 veranschaulicht werden.

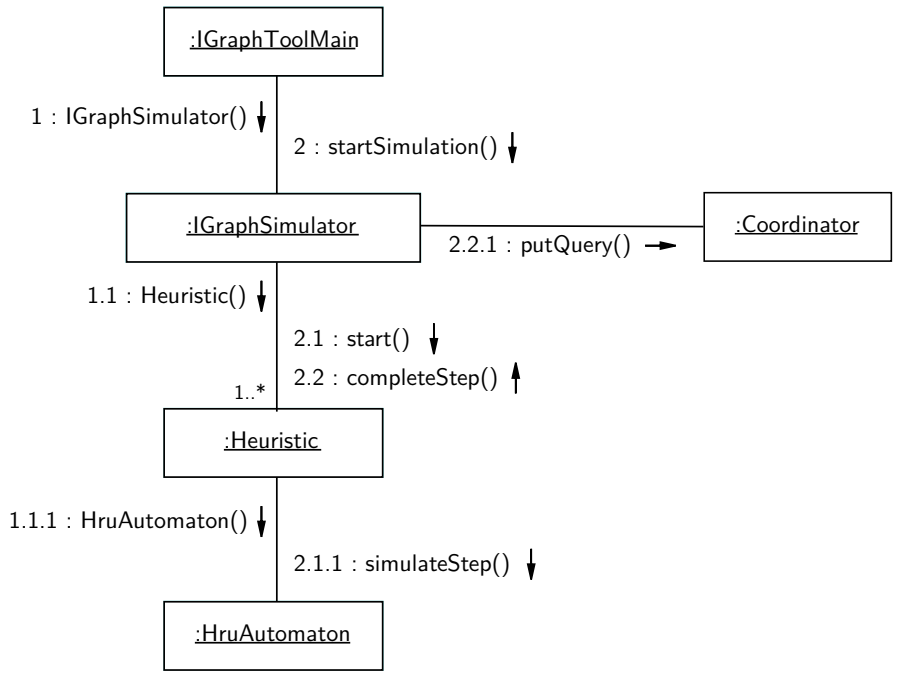


Abbildung 4.2: Dynamik während der Safety-Analyse nach den Architekturänderungen. dargestellt als UML-Kollaborationsdiagramm. Die Botschaften nach Botschaft 1 gehören zum Initialisierungsvorgang, jene nach Botschaft 2 bilden Start und Durchführung der Safety-Analyse ab.

Zum Vergleich sei auf das Kollaborationsdiagramm in Abbildung 2.2 verwiesen. Die Rollen und Abhängigkeiten der Komponenten sind, als Folge der neuen Konzepte, nun klarer strukturiert. Zwar werden die Kommunikationswege länger, allerdings geschieht dies nur für solche Methodenaufrufe, die sich vergleichsweise selten ereignen. Die häufig genutzten Verbindungen zwischen `Heuristic` und `HruAutomaton` bzw. `Heuristic` und `IGraphSimulator` bleiben kurz und damit performant, was mit Anforderung 3.3 übereinstimmt.

4.5 Zusammenfassung

Das vorliegende Kapitel „Entwurf“ erarbeitete aus den im Kapitel 3 (Problemanalyse) gewonnenen Informationen einen Softwareentwurf.

Hierbei wurde zuerst ein allgemeines Konzept erschlossen, das Struktur und Verhalten der an der parallelisierten Safety-Analyse beteiligten Komponenten umreißt.

Aus diesem Konzept ging dann einerseits der *Parallel-Leak*-Algorithmus hervor, dessen Leistung darin besteht, Informationen zur Steuerung der verteilten Analyse zwischen den beteiligten Threads auszutauschen. Zu dem Algorithmus wurden Evaluierbarkeit und Korrektheit überprüft.

Andererseits stellte das Kapitel, basierend auf Konzept und *Parallel-Leak*-Algorithmus, eine neue Architektur der *I-Graphoscope*-Simulationsumgebung vor. Wichtige Elemente sind hier die `HruAutomaton`-Klasse, welche Berechnungen im Zusammenhang mit dem jeweiligen HRU-Modell vornimmt, sowie die `Coordinator`-Klasse, die für die Koordination der ablaufenden Threads zuständig ist.

Zur späteren Implementierung stellte das Kapitel auch einen Entwurf für eine erste, einfache und allgemein gehaltene Parallelisierungsstrategie vor.

KAPITEL 5

Implementierung

Zum Umfang der Arbeit gehört die Anfertigung einer Implementierung, welche den beschriebenen Entwurf in ein Programm umsetzt und so *I-Graphoscope* um die in Anforderung 3.1 beschriebene Funktion der Nutzung mehrerer Prozessoren erweitert.

Dieses Kapitel dokumentiert wesentliche Aspekte der Implementierung. Es beschreibt hierzu den Feinentwurf (Abschnitt 5.1), in welchem ausgewählte Klassen, Schnittstellen und Methoden sowie deren Semantik erläutert sind. Anschließend führt Abschnitt 5.2 Details zur Implementierung des *Parallel-Leak*-Algorithmus aus; er enthält auch einen Beweis zur korrekten Synchronisation des Quelltextes. Der letzte Abschnitt des Kapitels (Abschnitt 5.3) gibt einen Überblick über die implementierten Parallelisierungsstrategien.

5.1 Feinentwurf

Dieser Abschnitt dokumentiert die Implementierung der in Abschnitt 4.4 beschriebenen Architektur. Ziel ist es dabei, zu jeder relevanten Klasse deren Schnittstelle, Funktion und Semantik nahezubringen. Der Abschnitt stellt dazu Klassendiagramme vor und erläutert Detaillösungen und weitere Entwurfsentscheidungen.

Aus Gründen der Übersichtlichkeit wird auf die Darstellung von Attributen und nichtöffentlichen Methoden verzichtet. Derartige Details können in der *Javadoc*-Quelltextdokumentation nachgelesen werden.

5.1.1 `HruAutomaton`

Im Zusammenhang mit dem jeweiligen HRU-Modell müssen Berechnungen durchgeführt und Informationen gespeichert werden, um Zustandsübergänge auszuführen und bereits erkundete Knoten rekonstruieren zu können. Die Implementierung dieser Funktionen ist bei [Rin09] dokumentiert und erfolgt dort in der `IGraphSimulator`-Klasse. Abweichend dazu sieht die Architektur der vorliegenden Arbeit, wie in Abschnitt 4.4 erläutert, die separate Klasse `HruAutomaton` vor. Das Klassendiagramm findet sich in Abbildung 5.1.

Der innere Zustand einer `HruAutomaton`-Instanz umfasst insbesondere einen ausgezeichneten Knoten, welcher als aktueller Zustand des HRU-Automaten gilt, und von dem aus Operationen ausgeführt werden. Die Semantik der meisten Methoden der Klasse ist identisch mit der Beschreibung in [Rin09].

Eine Instanz der `HruAutomaton`-Klasse ist je einer Heuristik zugeordnet und enthält ausschließlich Informationen über denjenigen Teil des Suchbaums, der von der entsprechenden Heuristik bearbeitet wird. Wie in 4.4.3 erläutert, sind die `HruAutomaton`-Objekte untereinander in einer Baumstruktur verknüpft. Zur Konstruktion des Baums werden alle Objekte außer der ersten Instanz durch die `createInstanceFromHere`-Methode erzeugt.

Die Operationen einer `HruAutomaton`-Instanz unterstützen die rekursive Abfrage von Informationen aus Kind-Instanzen (im Sinne der Baumstruktur). Um Codeduplizierung

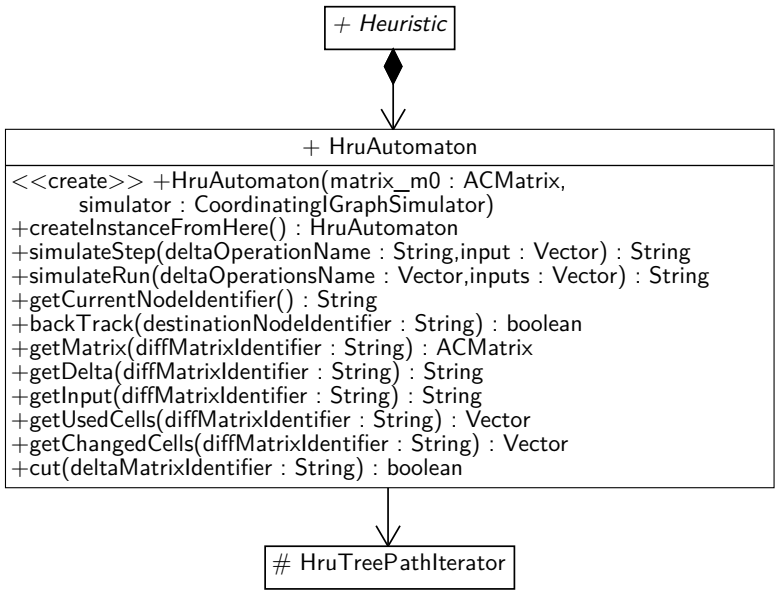


Abbildung 5.1: Die Klasse `HruAutomaton`, dargestellt im UML-Klassendiagramm.

zu vermeiden und die Erweiterbarkeit der Software zu wahren, wurde dieses Verhalten in einer inneren Iterator-Klasse gekapselt, welche einen Weg durch den Baum durchlaufen kann.

Während der Analyse jedoch darf ein Heuristik-Thread keine Knoten abfragen, welche außerhalb des ihm zugewiesenen Teilbaums liegen. Daher muss der Wurzelknoten eines jeden Teilbaums vorberechnet im Speicher abgelegt werden, während seine anderen Knoten (wie bei [Rin09] erläutert) schrittweise aus der Wurzel erzeugt werden können. Es ist zu beachten, dass dieses Zwischenspeichern von Knoten die Laufzeit- auf Kosten der Speichereffizienz erhöht.

Da `HruAutomaton`-Objekte das fachliche Modell umsetzen, wird die Klasse im Paket `model` platziert.

5.1.2 `Heuristic`

Die abstrakte Basisklasse für alle Heuristiken wird bei [Rin09] erläutert. Die dort entwickelte Schnittstelle ist nahezu identisch zu der `Heuristic`-Schnittstelle der vorliegenden Arbeit. Das Klassendiagramm wird in Abbildung 5.2 dargestellt.

Die zwei wesentlichen Neuerungen sind, äquivalent zu den Erläuterungen der Architektur in 4.4.2:

- **Gleichzeitiger Einsatz mehrerer, paralleler Heuristik-Threads.** Diese ergeben, wie bereits oben beschrieben, eine Baumstruktur. Zur Wahrung der Konsistenz werden alle Heuristik-Objekte außer der ersten Instanz durch die `createInstance`-Methode erzeugt, die Methode setzt dabei die in Abschnitt 4.2 beschriebene Semantik für ein „Fork“ um.
- **Verwendung der `HruAutomaton`-Objekte.** Jede `Heuristic`-Instanz kapselt ein `HruAutomaton`-Objekt. Der jeweilige Automat wird vom Konstruktor beziehungsweise von der `createInstance`-Methode instantiiert, sodass sich Heuristikimplementierungen hierum im Regelfall[1] nicht mehr kümmern müssen.

[1] Regelfall meint hier: solange keine besonderen Anforderungen der Heuristik vorliegen.

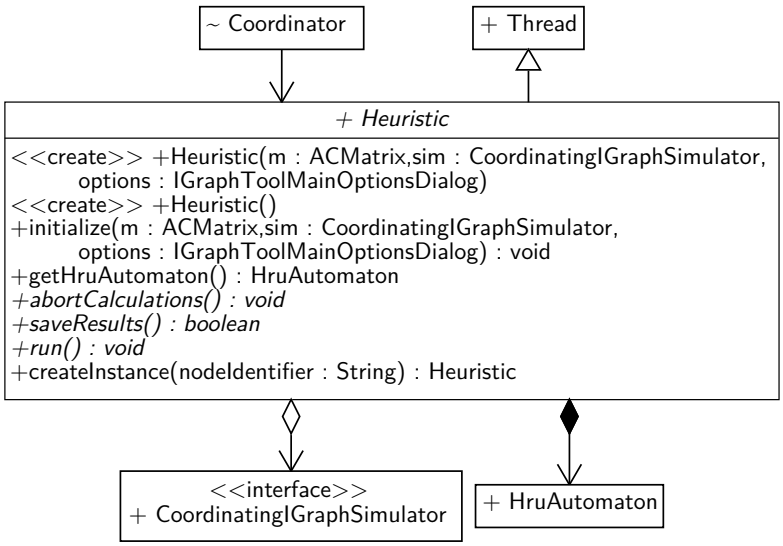


Abbildung 5.2: Die Klasse `Heuristic`, dargestellt im UML-Klassendiagramm.

Um Signale und Ergebnisse an die Simulationsumgebung senden oder Informationen abrufen zu können, hat eine `Heuristic`-Instanz Zugriff auf eine Implementierung der `CoordinatingIGraphSimulator`-Schnittstelle, welche in Abschnitt 5.1.5 beschrieben wird.

Zur Initialisierung ihres inneren Zustandes sollten abgeleitete Klassen die `initialize`-Methode oder die einzelnen Konstruktoren überschreiben.

5.1.3 `ParallelismStrategy`

Die abstrakte Basisklasse für Parallelisierungsstrategien ist in Abbildung 5.3 dargestellt. Sie wird vom Koordinator aufgerufen, um Informationen über die ablaufenden Heuristik-Threads zu übergeben und Strategieentscheidungen abzufragen.

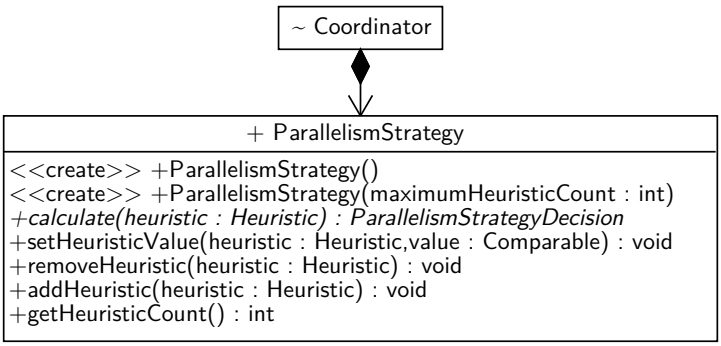


Abbildung 5.3: Die Klasse `ParallelismStrategy`, dargestellt im UML-Klassendiagramm.

Die grundlegende Verwaltung der bekannten Heuristiken wird durch die Klasse bereits implementiert: eine Hashtabelle bildet die bekannten `Heuristic`-Instanzen auf deren zuletzt bekannten heuristischen Werte ab.

Die `calculate`-Methode ist abstrakt und muss durch eine Kindklasse implementiert

werden. Sie wird vom Koordinator aufgerufen, um zur übergebenen Heuristik-Instanz eine Entscheidung zu erhalten, ob diese ihre Berechnungen verzweigen, abbrechen oder schlicht fortsetzen soll.

5.1.4 `Coordinator`

Der Koordinator wird, wie in Abschnitt 4.4.1 begründet, in einer eigenen Klasse implementiert. Er nimmt Benachrichtigungen darüber entgegen, dass ein Heuristik-Thread einen neuen Knoten berechnet hat, führt eine Liste gefundener unsicherer Zustände, kapselt die Fork/Join-Strategie und prüft auf die Terminierungsbedingungen.

Das Klassendiagramm ist in Abbildung 5.4 dargestellt.

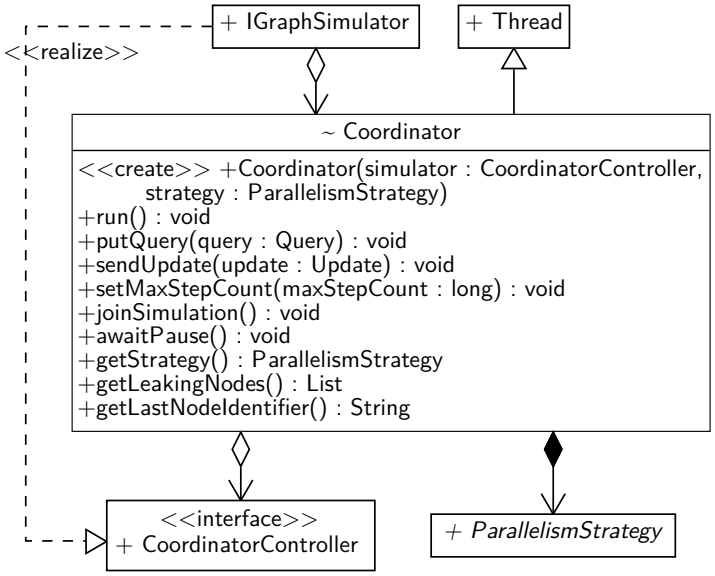


Abbildung 5.4: Die Klasse `Coordinator`, dargestellt im UML-Klassendiagramm.

Die zentrale Koordinatorroutine, wie sie auch in Abschnitt 4.2 (*Parallel-Leak*-Algorithmus) beschrieben ist, wird im `Coordinator` in der `run`-Methode implementiert. Hier wird auch die zugeordnete `ParallelismStrategy`-Instanz mit Informationen versorgt und konsultiert. Eine Reihe nicht-öffentlicher Methoden kapseln besondere Aufgaben; eine solche Methode ist `isTerminationConditionMet`, welche entscheidet, ob die Terminierungsbedingung der Simulation erfüllt ist.

Dem Benachrichtigungsmechanismus ist, wie im *Parallel-Leak*-Algorithmus modelliert, eine Warteschlange vorgeschaltet, welche eine faire Serialisierungsreihenfolge der verschiedenen Heuristik-Tasks in der Koordinatorroutine zusichert. Die `putQuery`-Methode reiht ein neues Anfrage-Objekt in diese Warteschlange ein, es enthält alle von der Heuristik an den Koordinator übergebenen Informationen und dient später auch zum Übertragen der Rückantwort, welche gemäß dem *Parallel-Leak*-Algorithmus „Fork“, „Join“ oder „None“ sein kann.

Es existiert eine zweite Art von Anfragen an den Koordinator, sie werden hier „Updates“ genannt. Über sie empfängt die `Coordinator`-Instanz die Information, dass ein Heuristik-Thread sich beendet und keine weiteren Knoten mehr berechnen wird. Da das Wissen über solch ein „Join“ entscheidend für die Koordinatorstrategie ist, werden diese Updates priorisiert abgearbeitet. Sie werden dem Koordinator mit der `sendUpdate`-Methode übergeben.

Um Signale und Ergebnisse an die Simulationsumgebung senden oder Informationen abrufen zu können, hat eine `Coordinator`-Instanz Zugriff auf eine Implementierung der `CoordinatorController`-Schnittstelle, welche in Abschnitt 5.1.5 beschrieben wird.

Die Simulationsumgebung selbst nutzt die Methoden der `Coordinator`-Klasse, um Informationen abzufragen oder auf bestimmte Zustände der beteiligten Threads zu warten: so kann sie warten, bis alle Heuristik-Threads pausiert (`awaitPause`) oder beendet (`joinSimulation`) sind.

5.1.5 `IGraphSimulator`

Das `IGraphSimulator`-Objekt dient als Kontrollinstanz für die gesamte Simulation und steht daher an zentraler Stelle in der Architektur. Es ist verantwortlich für die Initialisierung der Simulation, bietet der grafischen Oberfläche Methoden zur Steuerung des Analysevorgangs, nimmt Anfragen und Ergebnismeldungen der Heuristik-Threads entgegen, um diese dann an den Koordinator weiterzuleiten, und bietet sowohl den Heuristik-Threads als auch der grafischen Oberfläche Informationen an.

Der Grund, all diese Verantwortlichkeiten in einer Klasse zusammenzuführen, liegt – neben deren Rolle als Kontrollinstanz – in der gemeinsamen Datenbasis, welche für viele der Aufgaben erforderlich ist. Um dennoch den Anforderungen der Erweiterbarkeit und Korrektheit gerecht zu werden, wurden die Sichten der unterschiedlichen, beteiligten Objekte auf die `IGraphSimulator`-Klasse in verschiedenen Interfaces spezifiziert.

Diese Interfaces, welche von der `IGraphSimulator` realisiert werden, sind im Klassendiagramm in Abbildung 5.5 dargestellt.

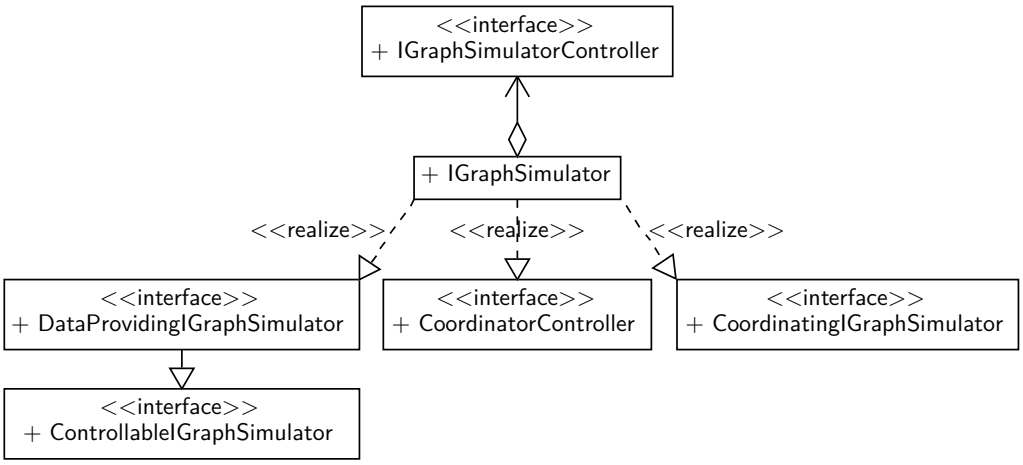


Abbildung 5.5: Die Schnittstellen des `IGraphSimulator`, dargestellt im UML-Klassendiagramm.

Im folgenden wird kurz auf die einzelnen Schnittstellen eingegangen.

Sicht der GUI. Die grafische Nutzeroberfläche greift auf ein Objekt der `IGraphSimulator`-Klasse zu, um die Simulation zu steuern oder Ergebnisse abzufragen. Für diese beiden Aufgaben stehen die Interface-Klassen `ControllableIGraphSimulator` sowie `DataProvidingIGraphSimulator` zur Verfügung, welche im Anhang in Abbildung A.1 als Klassendiagramm dargestellt sind.

Die Methoden des `ControllableIGraphSimulator` gliedern sich in drei Gruppen. Erstens sind dies Möglichkeiten, die Simulation zu starten, zu pausieren, wieder anlaufen zu lassen oder abzubrechen. Zweitens besteht die Möglichkeit, den Status der Simulation abzufragen und so zu prüfen, ob sie gestartet, pausiert, abgebrochen oder abgeschlossen ist. Drittens können Parameter gesetzt werden; dies ist hier die erlaubte Höchstschrittzahl, nach welcher der Vorgang beendet werden soll.

Der `DataProvidingIGraphSimulator` bietet zusätzlich Möglichkeiten, Informationen zu einzelnen Knoten anhand deren Identifikationsnummer abzufragen oder aber die zuletzt gesehene Knoten-Identifikation zu erhalten. Auf diese Art und Weise kann die grafische Oberfläche dem Nutzer Ergebnisse präsentieren.

Sicht der Heuristik. Eine Heuristik benötigt eine `IGraphSimulator`-Instanz, um ihr Ergebnisse und Signale zu senden oder Daten von ihr abzufragen. Diese Aufgaben werden in der Schnittstelle `CoordinatingIGraphSimulator` zusammengefasst (Klassendiagramm im Anhang in Abbildung A.2).

Hervorzuheben sind hier folgende Operationen:

- **Abschluss eines Simulationsschrittes.** Eine Heuristik meldet jeden errechneten Zustand gemeinsam mit zugehörigen Informationen an die Simulationsumgebung. In Erweiterung zum *Parallel-Leak*-Algorithmus gehört zu diesen Informationen auch ein Flag, das angibt, ob der gemeldete Zustand eine gesuchte Rechteausbreitung enthält: eine Heuristik kann dies mit den ihr zur Verfügung stehenden Informationen am effizientesten entscheiden.

  In der vorliegenden Architektur reicht die `IGraphSimulator`-Instanz die Information über den Abschluss des Simulationsschritts an das assoziierte `Coordinator`-Objekt weiter.

- **Beendingung des Threads.** Stellt eine Heuristik ihre Tätigkeit ein („Join“), signalisiert sie dies der Simulationsumgebung. Nach einem solchen Signal darf sie keine Berechnungsschritte mehr ausführen.

- **Fehlerausgabe.** Über die Schnittstelle ist es einer Heuristik möglich, dem Benutzer Fehlermeldungen zu präsentieren. Die Implementierung sorgt hier dafür, dass der Simulationsvorgang nicht unterbrochen wird, bis der Benutzer die Meldung bestätigt.

- **Abbruch des Simulationsvorgangs.** Eine Heuristik kann den Abbruch der gesamten Analyse veranlassen; dies kann beispielsweise infolge schwerwiegender Fehler sinnvoll sein.

Sicht des Koordinators. Der Koordinator fragt von der Simulationsumgebung den Zustand der Simulation ab und sendet selbst Zustandsinformationen oder Signale. Hierfür hat er Zugriff auf die `CoordinatorController`-Schnittstelle, welche durch den `IGraphSimulator` implementiert wird (Klassendiagramm im Anhang in Abbildung A.3).

Hervorgehoben werden sollen hier die Möglichkeiten, die Simulation abzubrechen oder (bei Zutreffen einer Terminierungsbedingung) erfolgreich zu beenden, eine Fehlermeldung zu übertragen, sowie den aktuellen Stand des Schrittzählers mitzuteilen.

5.1.6 Schnittstelle der GUI

Auch aus der `IGraphToolMain`-Klasse, welche die Schnittstelle zwischen Simulationsumgebung und grafischer Nutzeroberfläche bildet, wurde ein Interface herausgearbeitet. Das `IGraphSimulatorController` (Klassendiagramm im Anhang in Abbildung A.4) dient dem Simulator als Callback-Interface und erlaubt es, verschiedene Signale und Botschaften zu verschicken.

Im Gegensatz zu früheren Entwürfen der Schnittstelle ist die korrekte Funktionsweise der Simulationskomponenten dabei nicht von einer korrekten Implementierung des Interfaces abhängig.

Die Nutzeroberfläche bietet mit dem `IGraphSimulatorController` folgende Funktionen:

- **Benachrichtigung über Simulationszustand.** Die Simulationsumgebung informiert die Oberfläche darüber, ob die Simulation ablaufend, pausiert, abgebrochen oder abgeschlossen ist. Zur Semantik der letzten beiden Zustände muss erwähnt werden, dass eine Simulation auf zwei Arten „abgeschlossen“ sein kann: erfolgreich oder nicht erfolgreich, je nach Wert des „abgebrochen“-Flags.

- **Übergabe von Fehlermeldungen.** Die Simulationsumgebung kann die Nutzeroberfläche dazu veranlassen, dem Benutzer eine Fehlermeldung zu präsentieren. Dieser Aufruf kann blockieren, bis der Nutzer die Fehlermeldung bestätigt hat.

- **Übermitteln eines neuen Schrittzählerwertes.** Sobald der Analysevorgang einen Berechnungsschritt begonnen hat, teilt sie der Oberfläche die Gesamtzahl der begonnenen Schritte mit.

- **Ausführen von Aktionen im GUI-Thread.** Um Berechnungsvorgänge nicht unnötig zu verzögern, beispielsweise mit dem Warten auf Benutzerantworten oder dem Neuzeichnen der grafischen Oberfläche, bietet die Schnittstelle eine Methode, um `Runnable`-Objekte im Thread der grafischen Oberfläche auszuführen.

- **Abfragen der Programmkonfiguration.** Eine Methode der Schnittstelle retourniert die Konfigurationsinformationen von *I-Graphoscope*, sodass Einstellungen des Benutzers auch an die Simulationsumgebung übergeben werden können.

5.2 Implementierung des *Parallel-Leak*-Algorithmus

Der *Parallel-Leak*-Algorithmus, der den Ablauf einer parallelisierten HRU-Safety-Analyse beschreibt, wird im Rahmen dieser Arbeit implementiert. Dieser Abschnitt geht hierzu auf die Besonderheiten ein, vergleicht den formalen Algorithmus mit der dazugehörigen Implementierung und zeigt deren korrekte Synchronisation.

Abschnitt 5.2.1 nennt Anforderungen an die Implementierung einer Heuristik, die mit der hier entworfenen Software zusammenarbeiten soll. Der folgende Abschnitt 5.2.2 beschäftigt sich beispielhaft mit der Implementierung der Kommunikation zwischen Heuristik und Koordinator. Dort wird auch der Nachweis über die korrekte Synchronisation des Codes erbracht; ein Nachweis zu einem weiteren Teil der Implementierung findet sich im Anhang B.

5.2.1 Anforderungen an Heuristik-Implementierungen

Um mit dem in dieser Arbeit beschriebenen Parallelisierungs-Framework ausführbar zu sein, muss eine Heuristik konkrete Anforderungen erfüllen. Diese lassen sich auf unterschiedliche Ebenen aufteilen, insbesondere auf Modell- sowie auf Implementierungsebene.

Die Anforderungen auf Modellebene beschreibt Abschnitt 4.2.1, die Funktionsweise einer Heuristik ist der Darstellung des *Parallel-Leak*-Algorithmus im selben Teil der Arbeit zu entnehmen.

In Hinblick auf die Implementierungsebene muss eine Heuristik für ein korrektes Funktionieren des Frameworks zum einen das Modell korrekt implementieren. Zum anderen muss sie vor allem folgenden weiteren Anforderungen gerecht werden:

- **Aufruf der `completeStep`-Methode.** Hat eine Heuristik einen Berechnungsschritt erfolgreich abgeschlossen, ist dieser über die genannte Methode der `IGraphSimulator`-Instanz zu melden. Die Methode retourniert die Entscheidung der Parallelisierungsstrategie; dies ist einer der Werte `fork`, `join` oder `none`. (Siehe auch: Abschnitt 5.1.5, „Sicht der Heuristik“)

- **Korrektes Umsetzen der Parallelisierungsstrategie.** Im Fall einer `fork`-Entscheidung ist dafür zu sorgen, dass der berechnete Knoten nicht vom selben Thread weiterverwendet wird. Die Verzweigung selbst wurde bereits vom Koordinator ausgeführt, hierfür bietet die `Heuristic`-Klasse die `createInstance`-Methode. Lautet die Entscheidung `join`, darf der Heuristik-Thread keinen weiteren Berechnungsschritt ausführen, sondern sollte sich schnellstmöglich beenden. Gab die `completeStep`-Methode ein `none` zurück, ist der soeben erkundete Knoten in den weiteren Berechnungen mit einzubeziehen.

- **Beenden eines Heuristik-Threads.** Wird ein Heuristik-Thread beendet, ist die `IGraphSimulator`-Instanz hierüber mithilfe der `heuristicJoin`-Methode zu informieren. Dies gilt auch, wenn das Beenden aufgrund einer Strategieentscheidung geschieht. Nach einem Aufruf von `heuristicJoin` gilt ebenfalls, dass kein Berechnungsschritt mehr ausgeführt werden darf.

Für weitere Informationen zur Implementierung der Heuristik sei auf den Entwurf in den Abschnitten 4.4.2 und 5.1.2, sowie auf deren Sicht auf die kontrollierende `IGraphSimulator`-Klasse in Abschnitt 5.1.5 verwiesen.

5.2.2 Kommunikation zwischen Heuristik und Koordinator

Die im Rahmen der Arbeit entstandene Implementierung nutzt an vielen Stellen Synchronisationsmechanismen, um den *Parallel-Leak*-Algorithmus zu implementieren. Der verbleibende Teil des Abschnitts erläutert beispielhaft die Kommunikation zwischen Heuristik und Koordinator dort, wo ein Heuristik-Thread einen erfolgreich abgeschlossenen Berechnungsschritt meldet. Die hier umgesetzten Synchronisationsmechanismen sind für die Implementierung typisch. Zugleich handelt es sich bei dem Beispiel um ein zentrales Konzept im Algorithmus.

Bezug zum Modell. Im verteilten Algorithmus ist die Funktion repräsentiert durch Zeile 23 in Listing 4.2 sowie Zeilen 22, 40 und 24f. (Empfang der Botschaft) bzw. 35 und 38 (Senden der Antwort) in Listing 4.3. Dort schickt der Heuristik-Task eine Botschaft $\text{node}(q', v)$ an eine Warteschlange, welche dann vom Koordinator-Task abgerufen wird.

Der verteilte Algorithmus nutzt eine Queue, um Fairness zwischen den wartenden Heuristiken zu garantieren. Dies ist auch in der Implementierung nötig, da die an dieser Stelle geeigneten Java-Synchronisationsmechanismen keine Garantien zur Fairness geben.

Implementierung. Beteiligte Threads sind ein `Heuristic`-Thread sowie der Thread einer `Coordinator`-Instanz[2]. Die `Coordinator`-Klasse enthält eine Warteschlange[3], die Anfragen von Heuristiken enthält. Die Warteschlange ist dafür ausgelegt, von mehreren Threads benutzt zu werden; die Entnehmen-Operation blockiert beispielsweise so lange, bis ein Element verfügbar ist [Sun08, BlockingQueue].

Eine Anfrage wird repräsentiert durch ein `Query`-Objekt. In diesem werden einerseits die Informationen der Heuristik gespeichert (im Modell also der erkundete Knoten q' und seine Bewertung v), andererseits legt darin der Koordinator seine Antwort ab.

Die Kooperation zwischen beiden Threads läuft nun wie folgt ab. Der Thread der Heuristik reiht seine Anfrage in die Warteschlange ein und *wartet* nun auf die Antwort. Hierzu nutzt er den Monitor-Syntax von Java (siehe Abschnitt 2.3.3), wie in Listing 5.1 dargestellt[4].

```
Query query = new Query(/* Informationen der Heuristik */);
queue.put(query);
synchronized(query) {
    while(!query.responseAvailable()) {
        query.wait();
    }
}
query.getResponse(); /* Abrufen der Antwort */
```

Listing 5.1: Senden der Anfrage und Erhalt der Antwort durch den Heuristik-Thread.

Das `Query`-Objekt dient hier gleichsam als Monitor: der Heuristik-Thread wartet, bis eine Antwort des Koordinators verfügbar ist und eine *Benachrichtigung* hierüber eingeht. Diese wird durch den Koordinator über die `notify`-Methode des `Query`-Objektes versendet, wie die nachfolgenden Absätze zeigen. Das `synchronized`-Konstrukt realisiert wechselseitigen Ausschluss über das `query`-Objekt, sorgt für die Speichersynchronisation (siehe Abschnitt 2.3.3) und verhindert, dass die Antwort (samt Benachrichtigung) zwischen dem Abfragen der Schleifenbedingung und dem Beginn des Wartens eintrifft.

Das Entgegennehmen, Verarbeiten und Beantworten der Anfrage durch den Koordinator wird in Listing 5.2 dargestellt.

Der Koordinator fragt ein Element aus der Warteschlange ab. Aufgrund der besonderen Semantik der eingesetzten Warteschlange [Sun08, BlockingQueue] blockiert diese Operation gegebenenfalls, bis ein Element verfügbar ist. Anschließend belegt er die Sperre auf das `query`-Objekt (diese wurde von der Heuristik für die Dauer der `wait`-Operation implizit freigegeben), verarbeitet die Anfrage und *benachrichtigt* den Heuristik-Thread mithilfe der `notify`-Methode.

[2]Der Quelltext zu den hier dargelegten Abläufen findet sich sämtlich in der `Coordinator`-Klasse, insbesondere in den Methoden `putQuery` und `run`.

[3]Das Feld, das die Warteschlange referenziert, ist `waitingHeuristics`.

[4]Es ist zu beachten, dass die gezeigten Quelltexte eine vereinfachte Darstellung der tatsächlichen Implementierung sind.

```
Query query = queue.take();
synchronized(query) {
    query.setResponse(processQuery(query));
    query.notify();
}
```

Listing 5.2: Erhalt der Anfrage und Senden der Antwort durch den Koordinator.

Korrekte Synchronisation. Um sicherzustellen, dass die oben dargestellten Zeilen unter allen Umständen korrekt ausgeführt werden, müssen sie durch die *Happens-Before*-Relation korrekt geordnet sein (siehe Abschnitt 2.3.4, „Formales Speichermodell“). Das Schreiben der Anfrage muss in jedem Fall vor deren Verarbeitung geschehen; ebenso muss das Schreiben der Antwort vor dem Lesen derselben stattfinden.

Satz 5.1 (Korrekte Synchronisation der Kommunikation).
Für den in 5.2.2 dargestellten Quellcode gilt: `new Query()` $\curvearrowright$ `processQuery(query)` $\curvearrowright$ `query.setResponse()` $\curvearrowright$ `query.getResponse()`.

Beweis. Zur Anfrage ist festzustellen: es gilt `new Query()` $\curvearrowright$ `processQuery(query)`, da diese über Einreihung und Entnahme in/aus der Warteschlange synchronisiert sind [Sun08, BlockingQueue].

Dass weiterhin `processQuery(query)` laut *Happens-Before*-Relation vor `query.setResponse()` angeordnet ist, folgt aus der Programmordnung.

Bleibt zu zeigen: `query.setResponse()` $\curvearrowright$ `query.getResponse()`. Betrachte hierzu die beiden `synchronized`-Blöcke: der Block in Listing 5.1 wartet auf eine Antwort, während der Block in Listing 5.2 diese erzeugt. Sie seien im folgenden b_{wait} und b_{notify} genannt.

Da beide Blöcke bezüglich des selben Objekts synchronisieren, gilt entweder $b_{\text{wait}} \curvearrowright b_{\text{notify}}$ oder $b_{\text{notify}} \curvearrowright b_{\text{wait}}$.

Fall 1: Sei nun $b_{\text{notify}} \curvearrowright b_{\text{wait}}$. Dann sind die Operationen der Blöcke laut Definition 2.9 in Reihenfolge gebracht, und es gilt `query.setResponse()` $\curvearrowright$ `query.responseAvailable()`. Weiterhin gilt laut Programmordnung `query.responseAvailable()` $\curvearrowright$ `query.getResponse()`. Es folgt `query.setResponse()` $\curvearrowright$ `query.getResponse()`, was zu zeigen war.

Fall 2: Es ist laut Fallunterscheidung $b_{\text{wait}} \curvearrowright b_{\text{notify}}$. Dann wurde die Sperre von b_{wait} über die `wait`-Methode freigegeben und der Heuristik-Thread in einen Wartezustand versetzt (siehe Abschnitt 2.3.3). Es gilt `query.wait()` $\curvearrowright$ `query.setResponse()` $\curvearrowright$ `query.notify()`.

Die `notify`-Operation befreit den wartenden Heuristik-Thread aus seiner Wartesituation. Durch den wechselseitigen Ausschluss kann dieser erst fortfahren, wenn b_{notify} abgearbeitet ist, indem er erneut die Sperre bezüglich dem *query*-Objekt besetzt. Laut Definition 2.9 sind die Operationen dann in Reihenfolge gebracht, und es gilt: `query.setResponse` $\curvearrowright$ `query.notify()` $\curvearrowright$ `query.getResponse()`, was zu zeigen war.

Es folgt die Behauptung. □

5.3 Strategieimplementierungen

Neben der in Abschnitt 4.3 entworfenen Strategie wurden noch einige weitere, einfache Strategien entwickelt, welche nachfolgend beschrieben werden.

Die einzusetzende Strategie und ihre Parameter können im Konfigurationsdialog der *I-Graphoscope*-Software eingestellt werden. Die Klassen finden sich im Paket `igraph-tool.model.parallelism`.

5.3.1 „No Fork“-Strategie

Diese Strategie, implementiert in der `NoForkStrategy`-Klasse, entscheidet nie auf eine Verzweigung, und somit auch nicht auf das Abbrechen einer laufenden Heuristik.

In der Konsequenz sorgt diese Strategie für eine einfache, nicht-parallele Berechnung der Safety-Analyse. Selbstverständlich arbeitet der Koordinator dennoch gemäß dem *Parallel-Leak*-Algorithmus.

5.3.2 „Bounded Fork“-Strategie

Als Parameter wird der in der `BoundedForkStrategy`-Klasse implementierten Strategie eine Obergrenze an auszuführenden Tasks übergeben. Die Strategie wird genau dann eine Verzweigung verlangen, wenn diese Obergrenze nicht erreicht ist. Auf „Join“ entscheidet sie nie.

Die Wirkung ist eine breite Suche auf dem Zustandsbaum. In Modellen, die nur wenige (ausführbare) Operationen bieten, wird dies jedoch dazu führen, dass die ersten Heuristik-Tasks sämtliche ihnen zur Verfügung stehenden Berechnungsmöglichkeiten an verzweigte Tasks übergeben und so ihren Teil des Suchbaums erschöpfen.

5.3.3 „Merciful Join“-Strategie

Diese Strategie, implementiert in der `MercifulJoinStrategy`-Klasse, setzt die in 4.3 entworfenen Konzepte um. Konfigurierbar ist hier ebenfalls eine Höchstzahl an gleichzeitig einzusetzenden Tasks, sowie die „Gnadenfrist“, im Programm „Mercy Step Count“ bezeichnet: nur Tasks, welche bereits so viele Berechnungsschritte abgeschlossen haben, können abgebrochen werden.

Da die Strategie anhand allgemeiner Konzepte entworfen wurde, setzt sie bezüglich der Zustandsbewertungen keinen konkreten Datentyp voraus, sondern verlangt lediglich dessen Vergleichbarkeit. Um zu entscheiden, welcher Heuristik-Task die schlechtesten Ergebnisse liefert, verwendet sie daher ein *relatives* Gütekriterium, und wählt diejenige Heuristik, deren Zustandsbewertungen im Durchschnitt am kleinsten gewesen sind.

5.4 Zusammenfassung

Das obige Kapitel beschrieb wesentliche Aspekte der Implementierung.

Im Feinentwurf wurden relevante Klassen, Schnittstellen und Methoden dargelegt. Dabei wurden UML-Klassendiagramme gezeigt und zentrale Fähigkeiten sowie Besonderheiten erläutert.

Der nachfolgende Abschnitt präsentierte einige Details zur Implementierung des im Entwurf erarbeiteten *Parallel-Leak*-Algorithmus. Er beschrieb dabei Anforderungen an künftige Heuristik-Implementierungen, welche sich wesentlich auf das Umsetzen des Heuristik-Modells sowie auf eine korrekte Kommunikation zum Koordinator beziehen. Außerdem brachte der Abschnitt beispielhaft die Umsetzung der Kommunikation zwischen Heuristik-Threads und Koordinator nahe, und bewies im Zuge dessen die korrekte Synchronisation des Quelltextes.

Ein letzter Abschnitt beschrieb implementierte Parallelisierungsstrategien. Dies sind eine nicht parallelisierende Strategie, eine, die bis zu einer konfigurierbaren Obergrenze Heuristik-Threads erzeugt, und eine, die Heuristik-Tasks mithilfe einer „Gnadenfrist“ abbricht.

KAPITEL 6

Evaluierung

Die im Rahmen der Arbeit entstandene und im vorhergehenden Kapitel beschriebene Implementierung soll nun evaluiert werden. Dies muss sich an den Zielen der Arbeit orientieren.

Das grundsätzliche Ziel war es, die *Performanz* der HRU-Safety-Analyse zu erhöhen, genauer: es dem ablaufenden Analyseverfahren zu ermöglichen, mehr Knoten pro Zeiteinheit zu bearbeiten. Der Weg bestand in der Parallelisierung – hierdurch kann jedoch zusätzlicher *Overhead* entstehen. Die Performanzsteigerung wirkt sich insbesondere auf die Breite und/oder Tiefe des erkundeten Suchbaumes aus, das heißt: die *Reichweite* der Heuristik wird erhöht.

All dies untersucht das vorliegende Kapitel. Es erläutert hierzu zuerst die eingesetzten Messmethoden (Abschnitt 6.1) und bestimmt einige Metriken (Abschnitt 6.2). Anschließend legt es die Ergebnisse der Messungen dar (Abschnitt 6.3). Die nachfolgende Diskussion (Abschnitt 6.4) wird sich nicht allein auf die Resultate der Messungen beziehen, sondern die Ergebnisse der Arbeit darüber hinaus bewerten. Ein abschließender Ausblick (Abschnitt 6.5) zeigt Möglichkeiten zur Weiterentwicklung auf.

6.1 Messmethodik

Zur Nachvollziehbarkeit der Messungen werden im folgenden die Methoden beschrieben, nach denen die im Kapitel präsentierten Messungen durchgeführt wurden.

6.1.1 Systemumgebung

Die Messungen führt ein Computersystem mit folgender Konfiguration durch:

- ein *Intel Core 2 Duo T9300* (2,49 GHz) mit zwei Prozessorkernen, 6 MiB Level-2-Cache und 3 GiB Hauptspeicher,
- das Betriebssystem *Microsoft Windows XP* mit Service Pack 3, sowie
- als virtuelle Maschine die Laufzeitumgebung des *Sun JDK* Version 1.6.0_11 mit einer Heapgröße von 1,5 GiB.

Das System wurde während der Messungen nicht benutzt; neben der Entwicklungsumgebung liefen keine weiteren Anwendungen.

6.1.2 Messmethodik

Da die Analysedurchläufe teilweise viel Zeit in Anspruch nehmen ist die Frage, wie die Ergebnisse ausgewertet werden können, ein kritischer Aspekt bei der Erarbeitung der Messmethoden. Denn um zur Verfügung stehende Metriken detailliert analysieren und

auswählen zu können, bedarf es einiger Flexibilität bei der Auswertung der Simulationsdaten.

Deshalb wurde als Messmethode ein Logging-Ansatz gewählt: statt jede Metrik einzeln zur Laufzeit berechnen zu lassen, werden an ausgewählten Stellen im Quelltext leichtgewichtige Methodenaufrufe eingeführt, welche *Ereignisse* sowie dazugehörige Informationen (insbesondere einen Zeitstempel) in ein Logbuch schreiben. Um die Belastung durch Ein-/Ausgabeoperation gering zu halten, werden diese gepuffert (Puffergröße 10 KiB).

Die Logs können dann im nachhinein anhand von kleinen Programmen oder Skripten ohne zusätzlichen Zeitaufwand ausgewertet und in grafische Darstellungen überführt werden.

Logging-Mechanismus. Zur Übersichtlichkeit wurde das bereits in *I-Graphoscope* verwendete *Apache Logging Services Project* [Apa07] weiter genutzt: es bietet die benötigte Funktionalität und wurde auf Performanz optimiert. Bei der Java-Bibliothek handelt es sich um *log4j* Version 1.2.15.

Um die Wartbarkeit des Quelltextes zu erhalten und das Logging transparent zu realisieren, wurden die entsprechenden Aufrufe auf Apaches Logging-Bibliothek in die Klassen `EventLogger` und `Event` ausgelagert, welche sich im Paket `core.log` befinden.

Die `EventLogger`-Klasse realisiert das Singleton-Muster und kapselt die Initialisierung der benötigten Klassen. Die `Event`-Klasse dient als abstrakte Basisklasse für konkret eingesetzte Ereignistypen. Einem `Event`-Objekt werden im Konstruktor die aufzuzeichnenden Daten übergeben, das tatsächliche Zusammenstellen der Ausgabe erfolgt jedoch erst, wenn das Log tatsächlich aufgezeichnet wird.

Die Performanzeinbußen durch das Verfahren sind gering, da die Granularität des Loggings für aussagekräftige Ergebnisse nicht sonderlich hoch sein muss; in jeder Sekunde werden oft nur weniger als 10 Ereignisse protokolliert.

6.2 Metriken

Nachfolgend werden Metriken erarbeitet, um die eingangs beschriebenen Gegenstände evaluieren zu können.

Genannt werden neben den eigentlichen Metriken auch das eingegebene Modell und die verwendete Strategie. Als Heuristik dient jeweils (wie in Abschnitt 3.2) der *Leak-Search*-Algorithmus [Joh09].

Alle verwendeten Zeitangaben meinen die Echtzeit (und nicht etwa die virtuelle Threadzeit).

6.2.1 Performanz

Die Performanzeigenschaften der Implementierung sollen an realitätsnahen Modellen gemessen werden; daher wird für beide im folgenden vorgestellten Metriken das HRU-Modell des „`/home/user1`“-Verzeichnisses eines realen Linux-Systems verwendet, wie in Abschnitt 3.2.1 beschrieben.

Durchsatz. Die naheliegendste Performanzmetrik ist die Zahl der Berechnungsschritte, welche der gesamte Analysevorgang pro Minute durchführen kann („Durchsatz“). Als Berechnungsschritt zählt dabei ein Schritt im Sinne des *Parallel-Leak*-Algorithmus. Im

Optimalfall erzielt der Einsatz von Parallelität gegenüber der linearen Ausführung für n Prozessoren/Kerne den n-fachen Durchsatz.

Einflüsse hierauf können insbesondere auch die eingesetzten Strategien, einschließlich der eingegebenen Parameter erzielen. Es ist zu erwarten, dass die linear ausführende „No Fork“-Strategie im Vergleich zu einer „Bounded Fork“-Strategie mit maximal zwei Threads lediglich den halben Durchsatz erreicht. Eine weitere Erhöhung der Thread-Obergrenze sollte auf der vorliegenden Zwei-Kern-Maschine theoretisch den Durchsatz senken. Praktisch können jedoch die Cache-Eigenschaften der `HruAutomaton`-Klasse (siehe Abschnitt 5.1.1) auch dann noch begrenzt positive Auswirkungen zeigen.

Auch die „Merciful Join“-Strategie soll hier zum Einsatz kommen. Erste Praxistests lassen auf der Zwei-Kern-Maschine eine Threadbeschränkung von 4 als sinnvollen Kompromiss aus Performanz und Konflikten erscheinen. Als „Mercy Step Count“ werden Werte zwischen 1 und der Hälfte der Gesamtschrittzahl konfiguriert. Eine sehr hohe Gnadenfrist sollte dabei das Verhalten der „Merciful Join“-Strategie dem der „Bounded Fork“-Strategie angleichen.

Schrittzeit. Die Performanz jedes einzelnen Threads soll anhand der durchschnittlichen Zeit gemessen werden, die ein Thread für einen Berechnungsschritt benötigt. Einfluss auf diesen Wert haben die folgenden Faktoren:

- das verwendete Modell,
- die Tiefe des zu berechnenden Knotens im Baum, sowie
- die Auslastung des Computersystems.

Die Darstellung des letzten Faktors ist wünschenswert. Das Modell scheidet als Einflussfaktor aus, da stets das oben beschriebene Linux-System eingegeben wird. Die Tiefe im Suchbaum jedoch kann die Werte gerade bei hohen Schrittzahlen und seltenen Threadabbrüchen beeinflussen, was entsprechend zu berücksichtigen ist.

Als Eingaben für die Simulationen gelten die oben genannten. Es ist zu erwarten, dass Analysen, die weit mehr Threads einsetzen als Prozessoren vorhanden sind, eine weit höhere Schrittzeit benötigen.

6.2.2 Overhead

Bei der parallelisierten Safety-Analyse entsteht Overhead im Sinne zusätzlicher Laufzeitkosten durch Synchronisation und Konkurrenz zwischen den Threads. Konkurrenz tritt dabei einerseits auf, sobald mehr Threads abgearbeitet werden sollen als Prozessoren zur Verfügung stehen, und andererseits, sobald mehrere Heuristik-Threads versuchen gleichzeitig auf den Koordinator zuzugreifen.

Aussagen zur Konkurrenz um den Prozessor können bereits anhand der Performanzmetriken im vorhergehenden Abschnitt getroffen werden. Bezüglich der Konkurrenz um den Koordinator kann einerseits die durchschnittliche Warteschlangenlänge herangezogen werden, andererseits kann beobachtet werden, wie lange ein Thread durchschnittlich rechnet, und wie lange er (auf die Antwort des Koordinators) wartet.

Als Eingaben dienen die selben Werte wie in Abschnitt 6.2.1.

Warteschlangenlänge. Die hier vorgeschlagene Metrik ist der Durchschnitt über die Länge der Koordinatorwarteschlange, *bevor* der Koordinator ein Element (und damit die Referenz auf einen Heuristik-Thread) entnimmt.

Für die „No Fork“-Strategie sollte dies 1,0 sein, da hier stets vor Entnahme eines neuen Elements genau ein Thread auf die Antwort des Koordinators wartet. Aufgrund der kurzen Rechenzeit des Koordinators sollte die Warteschlange auch für Strategien, die pro Prozessor nur ein oder höchstens zwei Threads einsetzen, Werte zwischen 1 und 1,5 annehmen. Höhere Threadzahlen hingegen sollten häufigere Wartesituationen zur Folge haben.

Verhältnis von Rechen- und Wartezeit. Wird betrachtet, zu welchem Anteil seiner Gesamtlaufzeit jeder Thread durchschnittlich rechnet, und zu welchem Teil er durchschnittlich wartet, zeigt sich, wie viel zur Verfügung stehende Rechenzeit durch das Warten verloren geht.

Es liegt die Vermutung nahe, dass die Wartezeit vor allem von der Zahl der Threads abhängig ist, die um den Koordinator konkurrieren.

6.2.3 Qualität

Um etwas über die Qualität der Analyse aussagen zu können, kann zum einen die Reichweite der Analyse betrachtet werden, also die Breite oder Tiefe des erkundeten Teils des Suchbaumes. Andererseits kann betrachtet werden, nach wie vielen Schritten die Heuristik die erste Rechteausbreitung findet.

Um beides zu messen wurde ein Modell konstruiert, dessen Rechteausbreitung erst verhältnismäßig tief im Baum liegt. Parallelisierte Analysen sollten hier einen Vorteil haben, eine höhere Reichweite erzielen und damit die Ausbreitung nach einer geringeren Schrittzahl finden.

Zu beachten ist, dass hier ein Vergleich der Schrittzahlen schnell täuschen kann: wenn eine Analyse mit n Threads und eine Analyse mit $2n$ Threads jeweils nur die selbe Schrittzahl zur Verfügung haben, wird die $2n$-Suche nur dann tiefer vordringen können, wenn ihre Strategie tatsächlich auf eine große Baumtiefe abzielt.

Das kaskadierende Modell. Das für die Analyse verwendete Modell ist so beschaffen, dass die gesuchte Rechteausbreitung für ein festes $d \in \mathbb{N}$ erst nach Anwendung von mindestens d Operationen erzeugt werden kann.

Hierzu stehen eine Reihe anwendungsspezifischer Operationen zur Verfügung, welche es für eine Matrixzelle ermöglichen, nach und nach alle möglichen Kombinationen der vorhandenen Rechte zu setzen. Diejenige Operation, welche die Rechteausbreitung erzeugt, setzt dann eine Kombination voraus, welche nur nach mindestens $d - 1$ Schritten entstehen kann.

Zu diesem Modell genügt eine Zugriffssteuerungsmatrix mit nur einer Zelle.

Die eingesetzte Metrik. Für unterschiedliche Zieltiefen wird betrachtet, nach wie vielen Schritten und nach welcher Zeit eine Analyse die gesuchte Rechteausbreitung findet. Während die nicht parallelisierte Suche einen Vorteil bei Ausbreitungen in geringen Baumtiefen haben sollte, da solche Stellen leicht zu finden sind, und parallelisierte Verfahren dennoch durch die vielen Threads mehr Schritte ausführen, sollte sich die Parallelität bei tieferen Zielen lohnen.

6.3 Messergebnisse

6.3.1 Durchsatz

Die ersten Messungen zeigen den Durchsatz der Gesamtanalyse, wie in 6.2.1 erläutert. Die Ergebnisse sind im Balkendiagramm in Abbildung 6.1 zu sehen.

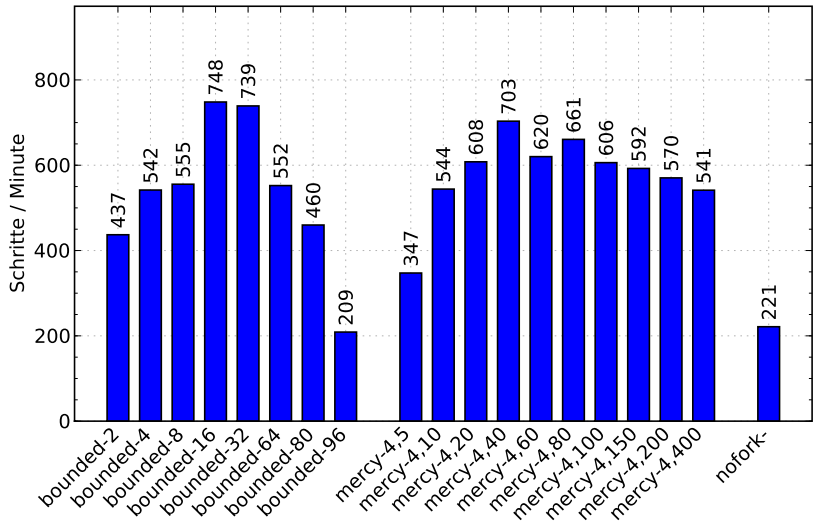


Abbildung 6.1: Durchsatz der Analysen am Modell eines Linux-Systems mit je 1000 Schritten. Die Namen bezeichnen jeweils: `bounded-X` die „Bounded Fork"-Strategie mit X Threads; `mercy-X,Y` die „Merciful Join"-Strategie mit X Threads und einem Mercy Step Count von Y sowie `no fork` die „No Fork"-Strategie.

Die Versuche für eine „Bounded Fork"-Strategie mit 112 oder 128 Threads erzielten Schrittzeiten unter 9 Schritten pro Minute und wurden abgebrochen. Auch die Versuche mit der „Merciful Join"-Strategie mit einer Mercy Step Count unter 5 konnten nicht beendet werden, da hier der zur Verfügung stehende Heip-Speicher der virtuellen Maschine nicht ausreichte.

Die erste Vermutung, dass die „Bounded Fork"-Strategie mit 2 Threads den doppelten Durchsatz der „No Fork"-Strategie erzielt, bestätigt sich. Auch die positiven Effekte durch das Caching des `HruAutomaton` sind noch bei 80 Threads zu beobachten[1], danach fällt der Durchsatz stark ab.

Bezüglich den Ergebnissen zur „Merciful Join"-Strategie bestätigt sich, dass sich der Wert mit wachsendem Mercy Step Count dem der `bounded-4`-Strategie annähert. Zu niedrige Gnadenfristen zeigen hier ebenfalls negative Auswirkungen: die häufigen Abbrüche und Neustarts der Threads senken die Performanz erheblich. Der maximale Durchsatz wird bei einer Frist von 40 Schritten erzielt, wobei beim praktischen Einsatz auch qualitative Erwägungen einbezogen werden müssen.

[1] Zu beachten ist, dass ein hoher Durchsatz nichts über die tatsächliche Qualität der Ergebnisse aussagt.

6.3.2 Schrittzeit

Die Ergebnisse zur in Abschnitt 6.2.1 beschriebenen Metrik der durchschnittlichen Schrittzeit wird in Abbildung 6.2 dargestellt.

Die Schrittzeiten der mit zwei Threads ausgeführten Analyse sind fast identisch zu denen der „No Fork“-Analyse, was für wenig Konkurrenz zwischen den beiden Threads um den Koordinator spricht. Eine Verdopplung der Threadanzahl bis auf etwa 64 Threads führt zu einer Verdopplung der Schrittzeit, weitere Erhöhungen lassen die Schrittzeiten überlinear steigen.

Die abweichenden Verläufe von Schrittzeit und Durchsatz folgen aus den verschiedenen Messmethoden: während beim Durchsatz die parallelisierte Ausführung nicht beachtet wird, sondern der Berechnung lediglich die Gesamtlaufzeit der Analyse zu Grunde liegt, wird bei der Schrittzeit der Durchschnitt über die realen Laufzeiten aller Berechnungsschritte verwendet. Daher wirkt sich beim Betrachten des Durchsatzes der Gewinn der Parallelität stark aus, während bei der reinen Schrittzeit der Verlust durch die Parallelisierung deutlicher sichtbar wird.

In Bezug auf die „Merciful Join“-Strategie hat die Größe der Gnadenfrist einen nur geringen Einfluss auf die Schrittzeit; es zeigen sich die selben Verhältnisse wie beim Durchsatz.

6.3.3 Warteschlangenlänge

Abschnitt 6.2.2 beschrieb die Metrik der Warteschlangenlänge. Die zugehörigen Messungen werden nachfolgend präsentiert (siehe Abbildung 6.3).

Die Annahme, dass die `bounded-2`-Analyse ebenso wie die `nofork`-Messung eine durchschnittliche Warteschlangenlänge von 1,0 erzielen, bestätigt sich. Eine weitere Erhöhung der Threadzahl lässt die Länge, und damit die Zahl der Konkurrenzsituationen am Koordinator zuerst nur langsam ansteigen, später zunehmend stark: während eine Verdopplung der Schrittzahl von 4 auf 8 die durchschnittliche Warteschlangenlänge um nur 33 % erhöht, führt die Verdopplung von 32 auf 64 Threads bereits zu einer Verdopplung des Wertes.

Bei Betrachtung der Ergebnisse zur „Merciful Join“-Strategie zeigt sich erneut ein nur geringer Einfluss des Gnadenfrist-Parameters. Erstaunlich ist hier jedoch, dass bei sehr niedriger Einstellung des Wertes die durchschnittliche Warteschlangenlänge nahezu 1 ist. Dies liegt darin begründet, dass durch den häufigen Thread-Abbruch selten 4 Threads gleichzeitig aktiv sind.

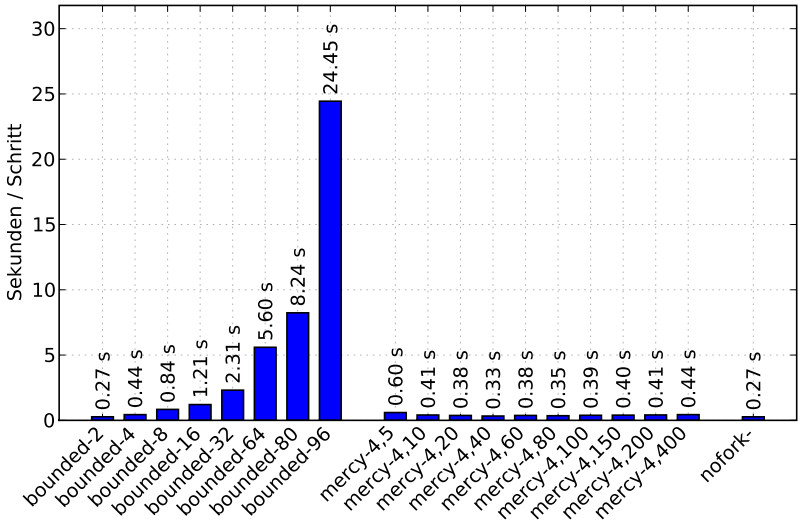


Abbildung 6.2: Durchschnittliche Zeit pro Berechnungsschritt am Modell eines Linux-Systems mit je 1000 Schritten. Die Namen bezeichnen jeweils: `bounded-X` die „Bounded Fork“-Strategie mit X Threads; `mercy-X,Y` die „Merciful Join“-Strategie mit X Threads und einem Mercy Step Count von Y sowie `no fork` die „No Fork“-Strategie.

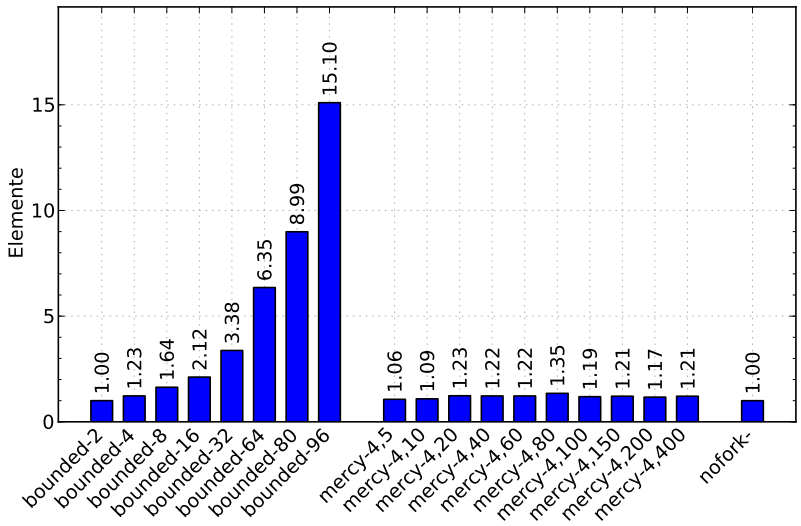


Abbildung 6.3: Durchschnittliche Warteschlangenlänge am Modell eines Linux-Systems mit je 1000 Schritten. Die Namen bezeichnen jeweils: `bounded-X` die „Bounded Fork“-Strategie mit X Threads; `mercy-X,Y` die „Merciful Join“-Strategie mit X Threads und einem Mercy Step Count von Y sowie `no fork` die „No Fork“-Strategie.

6.3.4 Verhältnis von Rechen- und Wartezeit

Ebenfalls in Abschnitt 6.2.2 wurde vorgeschlagen, das Verhältnis von Rechen- und Wartezeit pro Thread zu betrachten. Die Ergebnisse lassen sich in Abbildung 6.4 ablesen.

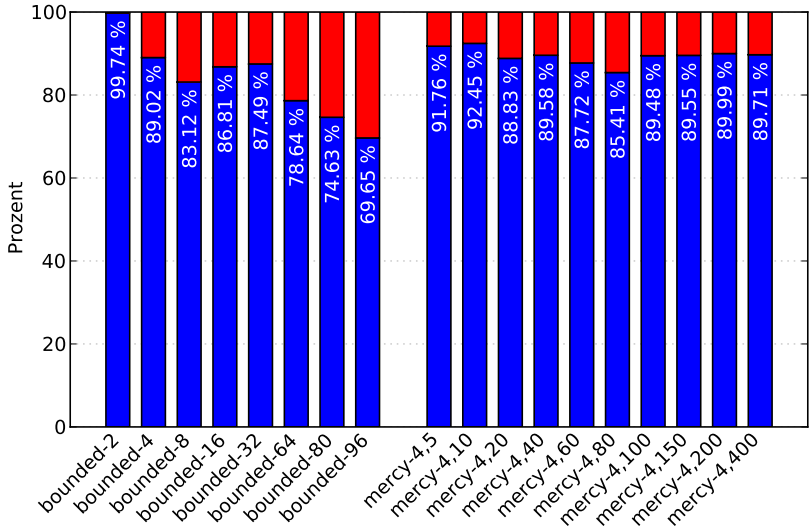


Abbildung 6.4: Durchschnittliche Laufzeit eines Threads, aufgegliedert in Rechen- und Wartezeit (blau bzw. rot). Die Prozentwerte entsprechen dem Anteil der Rechenan der Gesamtlaufzeit. Als Eingabe diente das Modell eines realen Linux-Systems mit je 1000 Schritten. Die Namen bezeichnen jeweils: `bounded-X` die „Bounded Fork"-Strategie mit X Threads und `mercy-X,Y` die „Merciful Join"-Strategie mit X Threads und einem Mercy Step Count von Y.

Die Ergebnisse sind vergleichbar zu der durchschnittlichen Warteschlangenlänge, auch wenn hier der Anteil der Wartezeit nicht so schnell ansteigt. Dies spricht für die kurze Bearbeitungszeit im Koordinator.

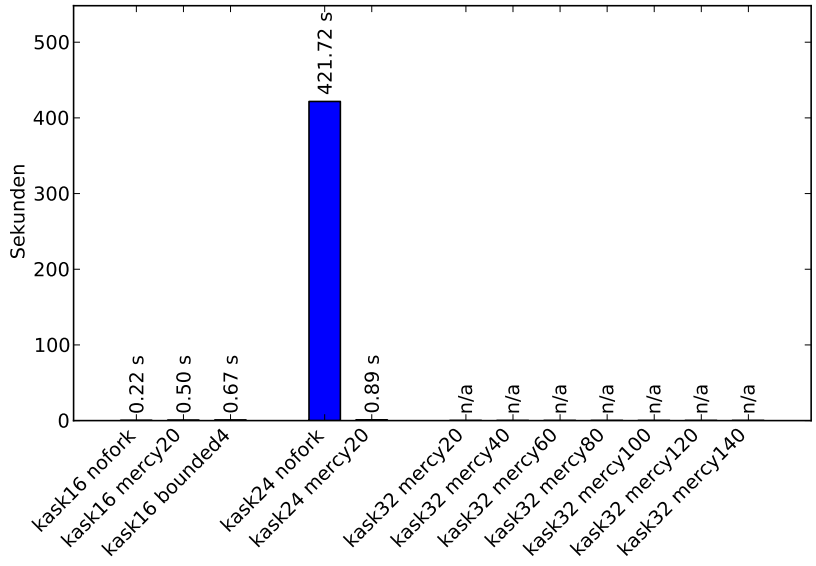
6.3.5 Reichweite der Analyse

Um die Qualität der Analyse zu messen, beschrieb Abschnitt 6.2.3 ein kaskadierendes HRU-Modell, in welchem die Rechteausbreitung erst ab einer bestimmten Zahl an Schritten entdeckt werden kann. Die Versuche wurden dabei mit 16, 24 und 32 Schritten durchgeführt. Wie viele Schritte und wieviel Zeit die jeweiligen Analysen tatsächlich bis zur Rechteausbreitung benötigten, ist in Abbildung 6.5 dargestellt.

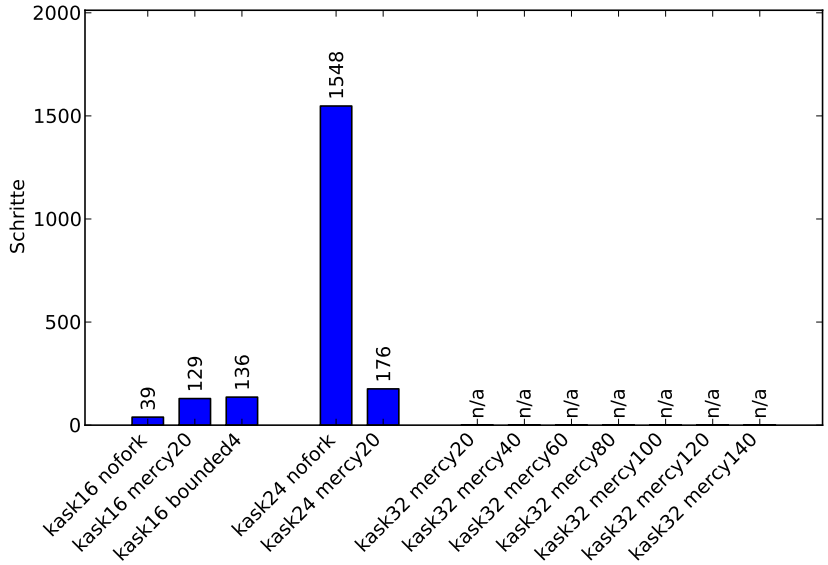
Die Ergebnisse zeigen, dass das Auffinden einer Rechteausbreitung mit einer minimalen Suchlänge von 16 Schritten für keine der Strategien ein Problem darstellt, obschon selbst die „No Fork"-Strategie hier mehr als die doppelte Zahl an Schritten benötigt. Analysen mit 4 Threads verwenden etwa die drei- bis vierfache Zahl an Schritten; hier entsteht also kein Gewinn durch die Parallelisierung.

Liegt die Rechteausbreitung nur doppelt so tief, kann sie also in mindestens 32 Schritten gefunden werden, kommt keine der untersuchten Strategien zu einem Ergebnis; alle Versuche, einschließlich dem mit einer „No Fork"-Strategie, scheitern früher oder später an einem Speicherüberlauf.

Ein Modell, bei dem die Rechteausbreitung in einer Suchbaumtiefe von 24 liegt, kann von allen Strategien gelöst werden, allerdings zeigen sich nun deutliche Unterschiede zwischen den Strategien. Während die lineare Analyse etwa 7 Minuten und genau 1548 Schritte benötigt, um die Ausbreitung zu finden, kommt die „Merciful Join"-Strategie



(a) Zeit bis zum Auffinden der Rechteausbreitung.



(b) Anzahl der Schritte bis zum Auffinden der Rechteausbreitung.

Abbildung 6.5: Dauer bis zum Auffinden der Rechteausbreitung am kaskadierenden HRU-Modell. `kaskD` steht dabei für eine Rechteausbreitung in mindestens D Schritten. Die eingesetzten Strategien sind `boundedX`, die „Bounded Fork“-Strategie mit X Threads, `mercyY`, die „Merciful Join“-Strategie mit 4 Threads und einem Mercy Step Count von Y, sowie `nofork`, die „No Fork“-Strategie.

bereits mit 176 Schritten und einer Zeit unter einer Sekunde zum Ergebnis. Um die Ursachen für diese Unterschiede zu finden, wird im folgenden betrachtet, welche Baumtiefen die beiden Strategien erreichen können.

Die entsprechende Darstellung findet sich in Abbildung 6.6.

Beim Vergleich der beiden Grafiken lässt sich ein entscheidender Vorteil der parallelisierten Analyse gut erkennen: mit beiden Strategien kann eine Tiefe von 20 schon nach 103 („No Fork“) bzw. 108 („Merciful Join“) Schritten erreicht werden. Doch während die parallelisierte Analyse schon nach wenigen weiteren Schritten einen Knoten in Tiefe 23 erreicht, benötigt die lineare Simulation hierfür noch einmal 158 Berechnungsschritte. Schließlich einen Knoten in Zieltiefe 24 zu erkunden, gelingt dem Algorithmus erst in Schritt 1543.

Die Ursache für das gute Abschneiden der „Merciful Join“-Strategie ist hier im Beenden des jeweils schlechtesten Threads nach schon kurzer Zeit zu sehen; dies treibt den Analysevorgang schnell in Bereiche mit guten heuristischen Bewertungen.

6.4 Diskussion

Die Ergebnisse dieser Arbeit sind insbesondere dann wertvoll, wenn die Methoden, die zu ihnen führten, kritisch diskutiert werden. Dies soll der vorliegende Abschnitt leisten. Hierzu werden zum einen die Messergebnisse in drei Thesen zusammengeführt (Abschnitt 6.4.1) und bewertet, zum anderen werden die verwandten Methoden kritisch betrachtet (Abschnitt 6.4.2).

6.4.1 Messergebnisse

Die Messungen in Abschnitt 6.3 anhand der ausgewählten Metriken lassen zusammenfassend folgende Thesen zu:

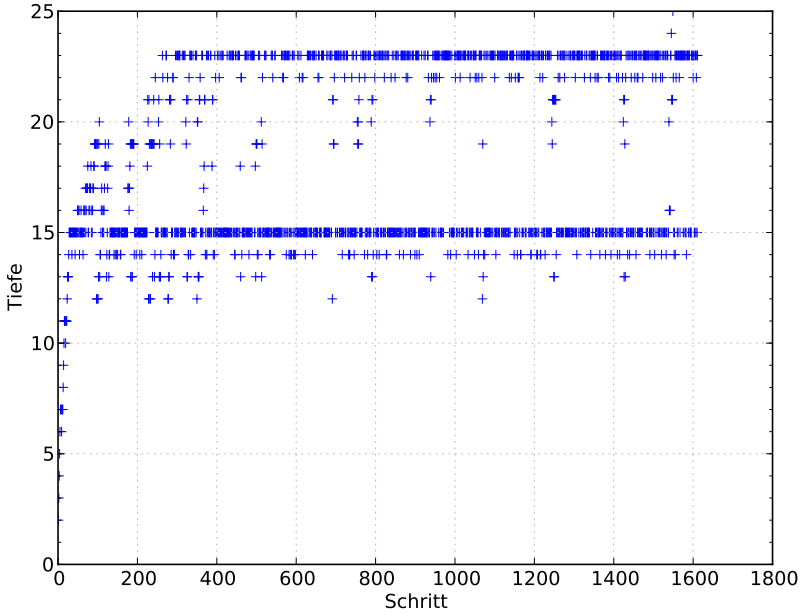
- **Erhöhung des Durchsatzes.** Die im Rahmen der Arbeit entstandene Implementierung zeigt, dass Parallelität den Durchsatz des Verfahrens auf einer Zwei-Kern-Maschine auf das Doppelte erhöhen kann; zusätzliche Mechanismen wie Caching erzielen weitere Verbesserungen.

  Wie sich die Laufzeiten mit zusätzlichen Prozessoren verhalten, bleibt zu untersuchen. Die Auswertung der Warteschlangenlänge lässt allerdings vermuten, dass die Konkurrenz der Threads um den Koordinator bei etwa 16 bis 32 Prozessoren zu einem stärkeren Einbruch der Wachstumsrate führt.

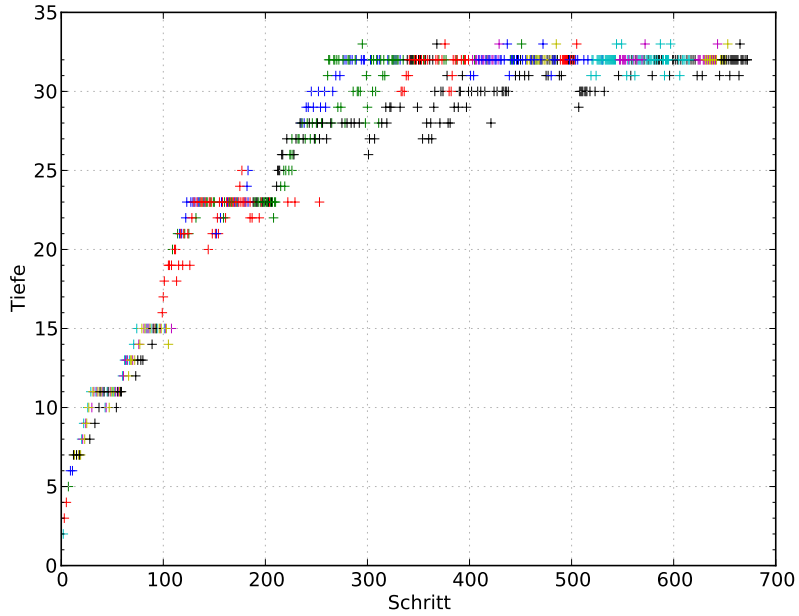
- **Erhöhung der Reichweite.** Durch die Möglichkeit der parallelen Analyse können in der selben Zeit weitere Teile des Suchbaumes erkundet werden. Je nach eingesetzter Heuristik und Strategie kann dies in die Tiefe oder die Breite des Baumes investiert werden.

- **Effiziente Nutzung der Rechenzeit.** Für das vorliegende System gilt: überschreitet die Zahl der höchstens einzusetzenden Threads die Zahl der verfügbaren Prozessoren nicht, nutzt die Analyse mehr als 99 % der Zeit für Berechnungen.

  Wie sich hier die Nutzung von mehr als zwei Prozessoren auswirkt, bleibt ebenfalls zu untersuchen. Die oben beschriebenen Effekte durch zunehmende Konkurrenz der Threads um den Koordinator treffen hier ebenso zu.



(a) Einsatz der „No Fork“-Strategie.



(b) Einsatz der „Merciful Join“-Strategie mit 4 Threads und einer Mercy Step Count von 20.

Abbildung 6.6: Baumtiefe der in den jeweiligen Schritten untersuchten Knoten am kaskadierenden HRU-Modell mit einer Rechteausbreitung nach mindestens 24 Schritten. Kreuze in unterschiedlichen Farben bezeichnen Schritte, welche von unterschiedlichen Heuristik-Threads berechnet wurden.

6.4.2 Methoden der Arbeit

Im folgenden werden die Methoden der Arbeit, wo nötig, kritisch bewertet. Ziel ist es, Schwächen aufzudecken und somit Unterstützung für die weitere Entwicklung und Nutzung der Software zu leisten.

Anforderungen an die Heuristik. Abschnitt 4.2 entwirft den *Parallel-Leak*-Algorithmus und macht dabei konkrete Annahmen einer Heuristik. Dies schränkt grundsätzlich ein. Für alternative heuristische Verfahren könnte es daher erforderlich sein, einen vollkommen anderen Algorithmus zur Parallelisierung der Analyse einzusetzen.

Begünstigend könnte hier wirken, dass die groben Architekturkonzepte weitgehend unabhängig vom konkreten Algorithmus entstanden sind und damit eine hohe Wiederverwendbarkeit aufweisen (vgl. Anforderung 3.4).

Diskrepanz zwischen formalem Algorithmus und Implementierung. Der *Parallel-Leak*-Algorithmus nutzte zur Modellierung der Heuristik eine Reihe von Operationen auf dem Heuristikzustand. Dieses Konzept wurde für die Implementierung nicht beibehalten: hier ist die Heuristik ein komplett eigener Thread, welcher Methoden anderer Klassen aufrufen kann. Dies bietet Raum für Fehler in künftigen Heuristiken und könnte auf einen Leser der Arbeit, welcher den Algorithmus mit dem Quelltext vergleicht, verwirrend wirken[2].

Caching-Funktion der `HruAutomaton`-Klasse. Die Arbeit entwarf die Berechnung der Automaten so, dass für jeden Thread der Wurzelknoten des zugehörigen Teilbaums gespeichert wird. Dies geht, gerade beim Einsatz vieler Threads und großer Modelle, auf Kosten des Speichers. Da im Rahmen der Arbeit keine Alternativen aufgezeigt werden, bleibt dies, bei Bedarf, Gegenstand zukünftiger Forschung.

Entwurf der „Merciful Join"-Strategie. Die im Rahmen der Arbeit hervorgebrachte Parallelisierungsstrategie macht keine konkreten Annahmen über Modell oder Heuristik. Dies lässt vermuten, dass hier noch viel Raum für Optimierungen besteht. Tatsächlich sollte eine Heuristik stets mit einer zugehörigen Parallelisierungsstrategie entwickelt werden, um ihre Möglichkeiten maximal auszuschöpfen.

Darüber hinaus steht zu bedenken, ob das Vorgehen, so schnell wie möglich bis auf die Obergrenze der zu verwendenen Threads zu verzweigen, in allen Situationen wirklich günstig ist. Hier könnte eine detailiertere Auswahl der zu verzweigenden Knoten, um beispielsweise die Baumerkundung in die Breite voranzutreiben, nützlich sein.

Messverfahren. Die Simulationsumgebung, welche bei den Messungen in Kapitel 6.3 zum Einsatz kam, schöpfte mit dem 2-Kern-Prozessor einen lediglich kleinen Teil der Möglichkeiten paralleler Systeme aus und lässt viel Raum für weitere Untersuchungen. Gerade das Verhalten in Systemen mit weit mehr als 4 Kernen könnte neue und entscheidende Erkentnisse über die Performanz der entstandenen Implementierung aufzeigen.

Auch die Messmethoden können verbessert werden, um aussagekräftigere Resultate zu erzielen: mithilfe eines Test-Frameworks für *I-Graphoscope* könnten weit mehr Parameter und Eingabekombinationen bei akzeptablem Zeitaufwand untersucht werden, um

[2] Für zukünftig zu entwickelnde Heuristiken sei dringend auf Abschnitt 5.2.1 verwiesen, welcher Anforderungen an Heuristikimplementierungen nennt.

so die genauen Verläufe von Metriken in Abhängigkeit der unterschiedlichen Parameter darstellen zu können.

Deterministische Analyse. Im Laufe des Entwurfs achtete die Arbeit stets auf deterministische Verfahren. Dennoch ist ein Teil der Analyseimplementierung nach wie vor abhängig vom Scheduler des Betriebssystems: je nach dessen Implementierung sowie der aktuellen Systemlast könnte dieser unterschiedliche Entscheidungen treffen, dadurch kann sich die Serialisierungsreihenfolge am zentralen Koordinator ändern – die Strategie würde dann andere Entscheidungen treffen.

Erste Messungen zeigen zwar, dass sich dies in der Praxis auf sonst unbelasteten Linux- und Windows-Systemen nicht auswirkt, dennoch bleibt der Scheduler als Einflussfaktor bestehen. Eine Möglichkeit, dies zu beheben, besteht darin, die Serialisierungsreihenfolge im Koordinator durch eine totale Ordnung über alle ablaufenden Threads festzusetzen. Diese Maßnahme ließe starke Einbußen bei den Wartezeiten der Heuristik-Threads, und damit auch bei der Gesamtperformanz erwarten.

6.5 Ausblick

Dieser letzte Abschnitt der Evaluierung präsentiert eine Liste der im Laufe der Arbeit entstandenen Ideen und Vorschläge für weitere Verbesserungen oder alternative Richtungen, in welche die Forschungsarbeit an *I-Graphoscope* vorangetrieben werden kann.

Gezielte Optimierung von Hot-Spot-Methoden. Im Rahmen einer Safety-Analyse werden bestimmte Methoden sehr häufig aufgerufen (vgl. Abschnitt 3.2). Ihre Implementierung auf Optimierungsmöglichkeiten hin zu untersuchen könnte die Performanz der Analyse weiter erhöhen.

Caching von Zuständen. Die Untersuchungen im Rahmen der Arbeit haben gezeigt, dass die Caching-Eigenschaft der `HruAutomaton`-Klasse entgegen den ursprünglichen Entwürfen die Laufzeit auf Kosten der Speichereffizienz senkt. Möglicherweise könnte dieses Prinzip weiter ausgebaut und verfeinert werden, um den Kompromiss aus der Nutzung beider Ressourcen problemorientierter zu definieren.

Abstimmen von Strategie und Heuristik. Wie schon im Abschnitt 6.4 angemerkt, bietet eine Strategie viel Raum für Optimierungen bezüglich der verwendeten Heuristik. Dies könnte die Performanz wie auch die Qualität der Ergebnisse weiter steigern.

Festgelegte Serialisierungsreihenfolge. Abschnitt 6.4 nennt die Möglichkeit einer festgelegten Serialisierungsreihenfolge der Heuristik-Threads im Koordinator. Dies beseitigt mögliche Einflüsse des Betriebssystem-Schedulers auf das Analyseergebnis auf Kosten der Performanz.

Implementierung der `cut`-Methode. Die bei [Rin09] entworfene `cut`-Methode der `HruAutomaton`-Klasse ist nicht implementiert. Sie könnte bei Problemen mit dem verfügbaren Heap-Speicher abhilfe schaffen, sollte jedoch stets so implementiert werden, dass gefundene Rechteausbreitungen jederzeit von der GUI aus nachvollzogen werden können.

Nutzerschnittstelle zur Auswahl gefundener Rechteausbreitungen. Die bisher verfügbare grafische Nutzeroberfläche wurde in der vorliegenden Arbeit kaum verändert. Bisher wird jedoch nur der Pfad zu *einer* Rechteausbreitung in der GUI angezeigt; hier wäre eine Schnittstelle zur Auswahl aus allen entdeckten Ausbreitungen sinnvoll. Die möglicherweise große Zahl solcher Zustände erfordert dabei ein ausgeklügeltes System zur Suche und Filterung, um die Safety-Analyse zielgerichtet vorantreiben zu können.

Test-Framework zur Batch-Ausführung von Analysen. Wie schon in Abschnitt 6.4 erwähnt, wäre ein Test-Framework für *I-Graphoscope* äußerst nützlich: so könnte eine beliebige Zahl an Safety-Analysen auf unterschiedlichen Eingaben oder mit unterschiedlichen Parametern durchgeführt werden, um detailiertere Auskunft über bestimmte Metriken zu erhalten.

6.6 Zusammenfassung

Im vorstehenden Kapitel wurden Methodik und Ergebnisse der Arbeit qualitativ und quantitativ bewertet. Dazu wurden eine Messmethodik sowie eine Reihe von Metriken entworfen, anhand derer Messungen an der Implementierung vorgenommen werden konnten. Anschließend wurden die Messergebnisse wie auch das Vorgehen der Arbeit diskutiert und Vorschläge zum weiteren Vorgehen erarbeitet.

Die bei der quantitativen Analyse verwandten Metriken dienten zur Bestimmung von Performanz, Overhead und Qualität der durch die Implementierung des *Parallel-Leak*-Algorithmus erzeugten Ergebnisse. Das zentrale Resultat: der Durchsatz einer Safety-Analyse auf einer Zwei-Kern-Maschine konnte durch die Parallelisierung verdoppelt werden, was eine Erhöhung der Reichweite zur Folge hat.

KAPITEL 7

Zusammenfassung

Diese Bachelorarbeit leistet die Parallelisierung eines Analyseverfahrens zum Nachweis der HRU-Safety an Modellen von Zugriffssteuerungssystemen. Ziel war es dabei, den Durchsatz der Analyse zu erhöhen, sodass diese in der Lage ist, in der selben Zeit eine größere Zahl an Knoten zu erkunden. Die Arbeit entwarf und implementierte zu diesem Zweck ein Konzept zur Parallelisierung der Safety-Analyse. Dieses sieht eine Verteilung der Berechnungen auf eine konfigurierbare Zahl an Threads vor.

Sie entwickelte den *Parallel-Leak*-Algorithmus, welcher die Kooperation zwischen den Threads des Analyseverfahrens einerseits und einem zentralen Koordinator andererseits beschreibt. Durch die verwandten Konzepte bleibt die Software skalierbar bezüglich der Zahl der verfügbaren Prozessoren / Prozessorkerne; diese ist lediglich abhängig von der Komplexität der Eingabedaten. Der zentrale Koordinator stellt dabei zwar einen Flaschenhals dar, doch lassen die Messungen vermuten, dass dies bis zu mindestens 16 Prozessoren kein größeres Problem darstellt.

Der Entwurf achtete darauf, die Erweiterbarkeit der Software zu gewährleisten, indem Verantwortlichkeiten klar getrennt und, wo nötig, über Interface-Klassen herausgearbeitet wurden. Er erlaubt es, anwendungsspezifische Parallelisierungsstrategien zu implementieren und kann mit einer großen Breite an möglichen Analyseverfahren zusammenarbeiten. Auch die Möglichkeit der Evaluierung der Software wurde vom Entwurf berücksichtigt. Die eingesetzten Algorithmen sind deterministisch, und die Ergebnisse paralleler Ausführung sind vergleichbar mit denen einer linearen Ausführung.

Auf technischer Ebene bildete die Software *I-Graphoscope* den Ausgangspunkt der Arbeit. Sie implementiert ein Verfahren zur HRU-Safety-Analyse und wurde durch die Arbeit so erweitert, dass sie hierzu den *Parallel-Leak*-Algorithmus verwendet. An der Software durchgeführte Messungen zeigen, dass der Durchsatz bei Verwendung von zwei Prozessoren so tatsächlich verdoppelt, und aufgrund von Caching-Effekten sogar verdreifacht werden konnte. Dies führt nachweislich dazu, dass in der selben Zeit größere Bereiche des der Analyse zugrunde liegenden Suchbaums erkundet werden können. Zu diesen Effekten trägt wesentlich die Implementierung des zentralen Koordinators bei, welcher Anfragen performant bearbeitet. Dadurch entstehen lediglich kurze Wartezeiten bei den ablaufenden Heuristik-Threads.

Durch neue Heuristiken und auf sie angepasste Parallelisierungsstrategien sollte sich das Laufzeitverhalten einer HRU-Safety-Analyse weiter verbessern lassen, was ein Ansatzpunkt für künftige Forschungsarbeit sein kann. Auch können die entwickelten Algorithmen so erweitert werden, dass Analysen auch auf verteilten Systemen wie beispielsweise Clustern ausführbar sind – auf diese Weise können noch größere Modelle berechnet, oder die Reichweite auf vorhandenen Modellen weiter vergrößert werden. Andererseits könnte die Parallelität dazu genutzt werden, verschiedene Heuristiken zur gleichen Zeit ablaufen zu lassen. Dies kann die Qualität der Ergebnisse, welche dem Anwender präsentiert werden, nochmals erhöhen.

Anhang

A Zusätzliche UML-Diagramme

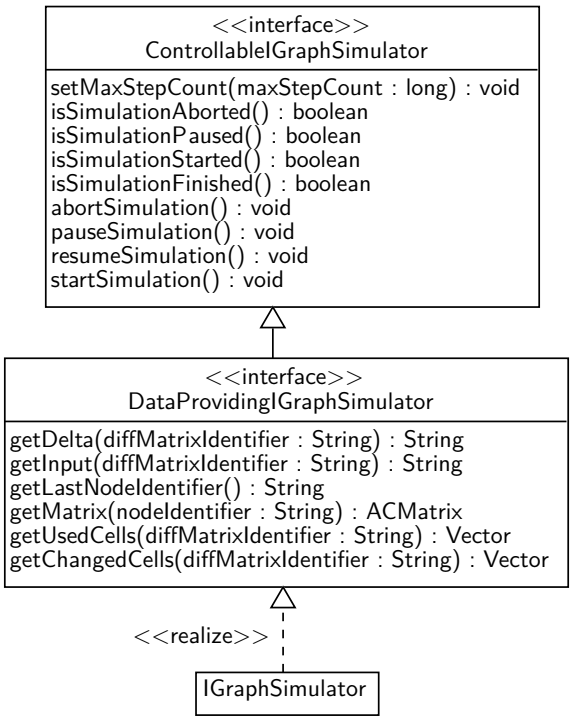


Abbildung A.1: Die Schnittstellen `ControllableIGraphSimulator` und `DataProvidingIGraphSimulator`, dargestellt im UML-Klassendiagramm. Siehe auch Abschnitt 5.1.5

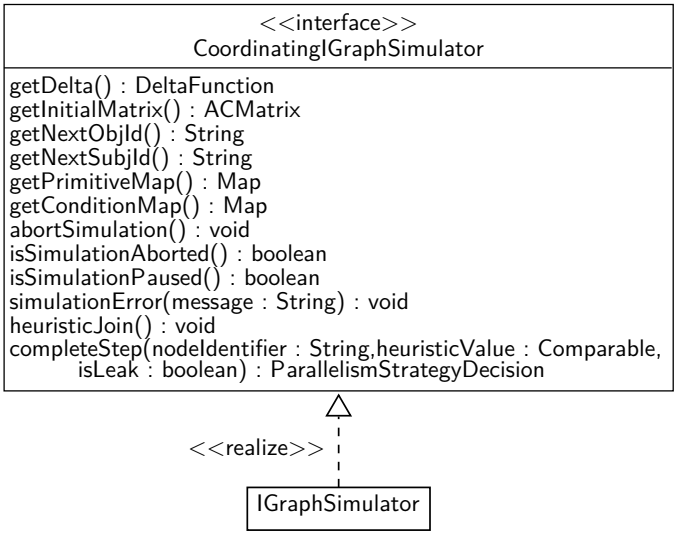


Abbildung A.2: Die Schnittstelle `CoordinatingIGraphSimulator`, dargestellt im UML-Klassendiagramm. Siehe auch Abschnitt 5.1.5

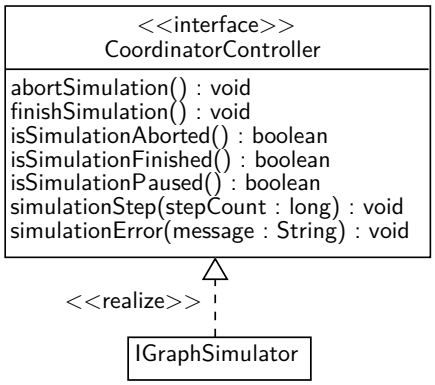


Abbildung A.3: Die Schnittstelle `CoordinatorController`, dargestellt im UML-Klassendiagramm. Siehe auch Abschnitt 5.1.5

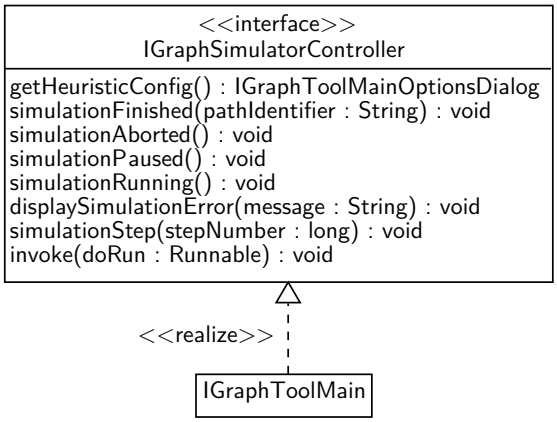


Abbildung A.4: Die Schnittstelle `IGraphSimulatorController`, dargestellt im UML-Klassendiagramm. Siehe auch Abschnitt 5.1.6

B Nachweis zur *Happens-Before*-Ordnung

Dieser Abschnitt enthält einen weiteren Nachweis zur *Happens-Before*-Ordnung. Der Beweis bezüglich der Kommunikation zwischen Koordinator und Heuristik wurde bereits in Satz 5.1 erbracht; nachfolgend wird gezeigt, dass der Eintritt in den Pause-Modus korrekt synchronisiert ist.

Das Pausieren der Simulation ist ein Vorgang, der durch den Benutzer veranlasst wird. Dieser betätigt eine entsprechende Schaltfläche in der Nutzeroberfläche, worauf hin die `pauseSimulation`-Methode in der jeweiligen `IGraphSimulator`-Instanz aufgerufen wird. Diese wartet dann, bis alle Threads pausiert sind, und meldet schließlich der GUI das erfolgreiche Pausieren.

Das Warten auf die Heuristik-Threads steht im Fokus dieses Abschnittes. Nötig wird es, weil im Pausenmodus auf Datenstrukturen zugegriffen wird, die im Simulationsvorgang bereits von den Threads benutzt werden und wechselseitigen Ausschluss erfordern (siehe Abschnitt 4.4.3).

Die eigentlichen Wartemechanismen implementiert die `Coordinator`-Klasse in ihrer Methode `awaitPause`. Die Grundidee basiert auf der Annahme, dass alle ablaufenden Heuristiken regelmäßig den Abschluss eines Zustandsübergangs melden und so in die Koordinator-Warteschlange eingereiht werden. Die `awaitPause` wartet also lediglich darauf, dass diese Warteschlange voll ist – dann ist die Ausführung aller Heuristik-Threads blockiert. Zugleich wird verhindert, dass der Koordinator im Pause-Modus Elemente aus der Warteschlange entnimmt.

Implementierung. Beteiligte Threads sind jener der Nutzeroberfläche, des Koordinators sowie die Heuristik-Threads. Nachfolgend wird zuerst der für diese Beteiligten relevante Code dargestellt, um anschließend den Beweis zu führen.

Nutzeroberfläche. Wird von der Nutzeroberfläche aus der Pause-Modus aktiviert, wird der in Listing B.1 dargestellte Code ausgeführt[1], welcher hauptsächlich in den oben genannten `pauseSimulation`- und `awaitPause`-Methoden zu finden ist.

```
pauseSimulation = true;

synchronized(queueFullMonitor) {
    while(!queueFull()) {
        queueFullMonitor.wait();
    }
}

simulationIsPaused();
```

Listing B.1: Aktivieren des Pause-Modus durch den GUI-Thread.

Die GUI setzt zuerst ein (im `IGraphSimulator`-Objekt gehaltenes) Flag und wartet dann, bis alle Heuristik-Threads pausiert sind, also die Warteschlange voll ist. Sollte dies noch nicht von anfang an der Fall sein, wird das `queueFullMonitor`-Objekt (das

[1] Es ist zu beachten, dass die gezeigten Quelltexte eine vereinfachte Darstellung der tatsächlichen Implementierung sind.

ein Attribut der `Coordinator`-Klasse ist) als Monitor eingesetzt, um zu warten, bis genügend Elemente in der Queue enthalten sind.

Der `simulationIsPaused`-Aufruf steht stellvertretend für alle Aktionen die nur dann ausgeführt werden dürfen, wenn die Simulation pausiert ist.

Heuristik-Threads. Damit die Nutzeroberfläche beim Warten auf das Pausieren aller Heuristik-Threads benachrichtigt werden kann, muss beim Einreihen einer Heuristik-Anfrage in die Warteschlange geprüft werden, ob diese nun voll ist. Der entsprechende Quelltext wird in Listing B.2 gezeigt. Die Repräsentation in den Quellen der Implementierung findet sich in der `putQuery`-Methode des Koordinators.

```
queue.put(query);

if(queueFull()) {
    synchronized(queueFullMonitor) {
        queueFullMonitor.notify();
    }
}
```

Listing B.2: Einreihen eines Heuristik-Threads in die Koordinator-Warteschlange durch einen Heuristik-Thread.

Nachdem sich der Heuristik-Thread in die Warteschlange eingereiht hat prüft er, ob diese voll ist, also genauso viele Elemente enthält, wie aktive Heuristik-Threads vorhanden sind. Ist dem so, sendet er eine Benachrichtigung an die (wartende) Nutzeroberfläche, indem er die `notify`-Methode des `queueFullMonitor`-Objekt aufruft.

Koordinator. Damit die Wartebedingung der Nutzeroberfläche erfüllt werden und die Warteschlange voll werden kann, darf der Koordinator bei aktivem Pause-Modus keine Elemente mehr entnehmen. Dies geschieht, indem er den Simulationszustand direkt vor der Entnahme des nächsten Elementes abfragt. Er wartet dann, bis die Simulation fortgesetzt wird; als Monitor dient hier die eigene Instanz.

Dieser Teil des Quelltextes wird der Einfachheit halber nicht dargestellt und ist nicht Teil des Beweises. Aufgrund seines geringen Umfangs ist er allerdings leicht zu validieren.

Korrekte Synchronisation. Um nachzuweisen, dass der oben gezeiget Quelltext unter allen Umständen korrekt ausgeführt wird, ist zu beweisen, dass die `simulationIsPaused`-Methode der Nutzeroberfläche erst dann ausgeführt wird, wenn alle Threads über die Koordinator-Warteschlange angehalten wurden.

Satz B.1 (Korrekte Synchronisation der Pausen-Aktivierung).
Für den in Abschnitt B gezeigten Quelltext gilt: `queueFull == true` $\curvearrowright$ `simulationIsPaused`.

Beweis. Wenn die Warteschlange bereits bei Eintritt in den Pause-Modus aktiv ist, wartet der GUI-Thread niemals und der Satz folgt aus der Programmordnung. Gelte also im folgenden, dass der GUI-Thread wartet.

Bei Aufruf der `wait`-Methode gibt der GUI-Thread das Lock auf `queueFullMonitor` frei, daher folgt `queueFullMonitor.wait` $\curvearrowright$ `queueFullMonitor.notify`.

Im folgenden wird angenommen, *dass* die Warteschlange irgendwann voll wird und daher die Bedingung `queueFull` erfüllt wird[2]. Dann geschieht das durch eine `queue.put`-Operation, die ausschließlich im dargestellten Quelltext ausgeführt wird.

Laut Programmordnung folgt `queue.put` $\curvearrowright$ `queueFull == true` sowie `queueFull == true` $\curvearrowright$ `queueFullMonitor.notify`. Durch die Benachrichtigung wird der GUI-Thread aus seiner Wartesituation befreit und erhält den Lock auf `queueFullMonitor`.

Es folgt aus der Programmordnung des GUI-Threads sowie Satz 2.9: `queueFull == true` $\curvearrowright$ `queueFullMonitor.notify` $\curvearrowright$ `simulationIsPaused`, was zu zeigen war. □

[2]Diese Annahme folgt aus der korrekten Implementierung der Warteschlange, welche hier vorausgesetzt wird, aus der korrekten Speichersynchronisation, die durch die Warteschlangenimplementierung gewährleistet wird [Sun08, BlockingQueue], sowie aus der korrekten Formulierung der Bedingung.

Literaturverzeichnis

[AFK09] Amthor, Peter ; Fischer, Anja ; Kühnhauser, Winfried E.: Analyse von Zugriffssteuerungssystemen. In: *D.A.CH Security 2009*, syssec, 2009, S. 49–61

[Amt08] Amthor, Peter: *Generierung von Informationsflussgraphen aus HRU-Modellen.* Studienjahresarbeit, Technische Universität Ilmenau, November 2008

[Apa07] Apache Software Foundation (Hrsg.): *Apache log4j 1.2.* `http://logging.apache.org/log4j/1.2/`: Apache Software Foundation, August 2007. – Besuch am 20.10.2009

[HRU76] Harrison, Michael A. ; Ruzzo, Walter L. ; Ullman, Jeffrey D.: Protection in Operating Systems. In: *Communications of the ACM*, ACM, August 1976, S. 461–471

[Joh09] John, Andreas: *Heuristische Safety-Analyse von HRU-Modellen.* Diplomarbeit, Technische Universität Ilmenau, März 2009

[Kü09] Kühnhauser, Winfried E.: *IT-Sicherheit in Verteilten Systemen.* Vorlesungsskript, Technische Universität Ilmenau, 2009

[Lea99] Lea, Doug: *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999

[MPA05] Manson, Jeremy ; Pugh, William ; Adve, Sarita V.: The Java Memory Model. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Long Beach, California, USA : ACM, Januar 2005, S. 378–391

[Rin09] Ring, Michael: *Simulation von dynamischen HRU-Sicherheitsmodellen.* Studienjahresarbeit, Technische Universität Ilmenau, 2009

[Sun08] Sun Microsystems Inc. (Hrsg.): *Java Platform, Standard Edition 6, API Specification.* `http://java.sun.com/javase/6/docs/api/`: Sun Microsystems Inc., 2008. – Besuch am 20.10.2009

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ilmenau, den 30. Oktober 2009

Felix Neumann