Institut für Parallele und Verteilte Systeme
Abteilung Simulation großer Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 138

Systemanalyse und Workflowverwaltung eines Frameworks für Cache-effiziente adaptive Simulation

Miriam Greis, Jessica Hackländer, Pascal Hirmer

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Schweitzer
Betreuer:	Dipl.-Inf. Oliver Meister
begonnen am:	01. Mai 2011
beendet am:	31. Oktober 2011
CR-Klassifikation:	F.2.1, G.1.8

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

Einleitung

Dieses Dokument stellt die Ausarbeitung der Fachstudie „Systemanalyse und Workflowverwaltung eines Frameworks für cache-effiziente adaptive Simulation“ dar, welche von Mai bis Oktober 2011 stattfand.

1 Aufgabenstellung

Die Abteilung „Simulation großer Systeme“ des Instituts für Parallele und Verteilte Systeme forscht derzeit an der Entwicklung eines Frameworks für eine effiziente Berechnung von fluiden Strömungen durch poröse Medien. Dabei wurde der Ansatz der Gitterberechnung nach Sierpinski gewählt und das Framework soweit funktionsfähig implementiert. Dieses System enthält jedoch noch Schwachstellen (siehe Kapitel 6), die mit Hilfe dieser Fachstudie behoben werden sollten.

Die Fachstudie ist in zwei Teile gegliedert. Der erste Teil umfasst die Analyse des gegebenen Frameworks mitsamt allen Hilfsmitteln, den eingesetzten Programmiersprachen und Werkzeugen. Des Weiteren sollte dabei der angewandte Simulationsprozess selbst mit den im gegebenen System verwendeten Konzepten und Methoden beschrieben werden, um die Problematik des Systems vollständig erfassen zu können. Hierbei sollte ein Nachbau des Systems in einer beliebigen Programmiersprache erfolgen.

Der zweite Teil der Fachstudie umfasst Lösungsansätze für die Problematiken des Systems anhand selbst entworfener Prototypen. Mögliche Lösungsansätze sind Präprozessor-Anweisungen, ein Python-Skript und die Erstellung einer Workflow-GUI. Dabei soll das bestehende System nicht ersetzt, sondern dessen Bedienung erleichtert werden.

Diese Lösungsansätze sollten untersucht, beschrieben und anschließend mit dem bestehenden System bzw. mit den anderen Lösungsansätzen verglichen werden. Das Ergebnis ist eine Empfehlung des bestmöglichen Lösungsansatzes.

Die meisten Informationen und Grafiken in diesem Dokument beruhen auf Gesprächen mit dem Betreuer Oliver Meister. Aufgrund dessen wurden nur wenige Quellen benötigt.

2 Gliederung

Dieses Dokument ist in 5 Kapitel gegliedert. Nach dem Einleitungskapitel befasst sich das darauffolgende zweite Kapitel mit der Analyse des gegebenen Systems. Hierbei werden die Methoden des Simulationsprozesses, die Einsatzgebiete und die eingesetzten Werkzeuge beschrieben. Am Ende dieses Kapitels wird die Problematik des bestehenden Systems geschildert und der Nachbau beschrieben. Das dritte Kapitel beschreibt die verschiedenen Ansätze zur Lösung der in Kapitel 6 beschriebenen Problematik. Die drei Lösungsansätze sind Präprozessor-Anweisungen, ein Python-Skript zur Codegenerierung und eine Workflow-GUI. Anschließend werden diese Lösungsansätze in Kapitel 11 bis Kapitel 14 bezüglich verschiedener Qualitäten wie Wartbarkeit, Einfachheit, Erlernbarkeit usw. verglichen. Aus diesen Informationen folgt im letzten Kapitel das Fazit, welches eine Empfehlung über die zu wählende Methode gibt.

Systemanalyse

3 Methoden des Simulationsprozesses

Dieses Kapitel soll einen kleinen Einblick in die verwendeten Methoden und numerischen sowie mathematischen Konzepte im Simulationsprozess geben. Dabei wird zuerst ein kurzer Überblick über eine häufig verwendete Methode zur Lösung partieller Differentialgleichungen, die Finite-Elemente-Methode, gegeben. Im Folgenden wird dann auf die Art des verwendeten Gitters, das Durchlaufen des Gitters und die Eigenschaften der Gitterelemente eingegangen. Der letzte Teil des Kapitels beschreibt dann ein allgemeines Vorgehen zur iterativen Lösung von Gleichungssystemen, bevor das gedämpfte Jacobi-Verfahren erklärt wird.

Dieses Kapitel orientiert sich stark an der Diplomarbeit „Speichereffiziente Algorithmen zum Lösen partieller Differentialgleichungen auf adaptiven Dreiecksgittern“ von Stefanie Schraufstetter (siehe [Scho6]) Die Diplomarbeit bietet noch sehr viel ausführlichere Informationen zu den hier beschriebenen Themen und wird als Hintergrundliteratur empfohlen. Alle Abbildungen in diesem Kapitel (außer Abbildung 6) wurden aus der Diplomarbeit übernommen.

3.1 Die Finite-Elemente-Methode

Die Lösungen partieller Differentialgleichungen werden oft mit der Finite-Elemente-Methode berechnet. Das zu berechnende Gebilde wird dabei in viele kleinere und einfacher zu berechnende Elemente geteilt. Aus den Lösungen für die einzelnen Elemente wird am Ende eine Gesamtlösung berechnet. Das Vorgehen bei der Implementierung der Finite-Elemente-Methode lässt sich grob in drei Schritten zusammenfassen:

1. Diskretisierung der Geometrie des Rechengebiets (Gittergenerierung)
2. Berechnung von Steifigkeitsmatrix und Lastvektor
3. Lösen des Gleichungssystems

Sowohl bei der Berechnung von Steifigkeitsmatrix und Lastvektor, als auch beim Lösen des Gleichungssystems gibt es zwei verschiedene Herangehensweisen. Bei knotenorientierten Verfahren werden alle Berechnungen für jeden Freiheitsgrad ausgeführt, während bei elementorientierten Verfahren die Berechnungen jeweils auf dem ganzen Element ausgeführt werden.

3.2 Gitterart

Es existieren viele verschiedene Gitterarten. Dabei kann zum Beispiel zwischen strukturierten und unstrukturierten Gittern unterschieden werden. Bei strukturierten Gittern sind die Knoten regelmäßig angeordnet, was dazu führt, dass die Nachbarschaftsbeziehungen normalerweise nicht extra gespeichert werden müssen. Unstrukturierte Gitter folgen dagegen keinerlei Regeln. Dadurch müssen jedoch die Nachbarschaftsbeziehungen gespeichert werden, was zu einem hohen Speicherbedarf führen kann.

Abbildung 1: Ein unstrukturiertes und ein rekursiv strukturiertes Dreiecksgitter

Abbildung 1 zeigt ein unstrukturiertes und ein rekursiv strukturiertes Dreiecksgitter. In der Simulation des Kunden wir ein rekursives Dreiecksgitter eingesetzt. Dieses besteht bei einer groben Initialisierung aus einem oder mehreren Dreiecken, welche dann nach und nach verfeinert werden können. Dafür wird immer ein Dreieck mit Hilfe von Bisektion in zwei kleinere Dreiecke aufgeteilt. Die neuen Dreiecke ergeben sich dann dadurch, dass von der Spitze zum Mittelpunkt der längsten Seite eine neue Kante gezogen wird. Entstehen bei der Teilung der Dreiecke hängende Knoten, so müssen auch die dort angrenzenden Dreiecke geteilt werden, so dass am Ende des Teilungsprozesses keine hängenden Knoten mehr vorhanden sind.

3.3 Gitterdurchlauf

Das Dreiecksgitter muss nun für die Berechnung durchlaufen werden. Dafür eignet sich das Prinzip der raumfüllenden Kurven. Für die Dreieckstraversierung wird die Sierpinski-Kurve ausgewählt. Für $n \to \infty$ füllt die Sierpinski-Kurve das Einheitsquadrat aus. Im Fall der Gittertraversierung reicht es jedoch, wenn sie jedes Element des Gitters, also jedes Dreieck einmal durchläuft. Damit genügt eine endliche Anzahl an Iterationen. Die Sierpinski-Kurve kann dabei mit der Dreieckserzeugung kombiniert werden:

Methode 1 (Konstruktion der n-ten Iteration der Sierpinski-Kurve)
Sei ein gleichschenkliges-rechtwinkliges Dreieck vom Level 0 gegeben. Die n-te Iteration der Sierpinski-Kurve ergibt sich dann wie folgt:

1. *Teile das gegebene Dreieck vom Level k - 1 in zwei kongruente Teildreiecke vom Level k.*
2. *Falls k = n: Verbinde die Mittelpunkte der beiden Teildreiecke. Falls k < n: Bestimme die raumfüllende Kurve für jedes der beiden Teildreiecke durch rekursive Aufrufe.*
3. *Verbinde die Kurven der beiden Teildreiecke.*

Abbildung 2 zeigt die ersten sechs Iterationen der Sierpinski-Kurve.

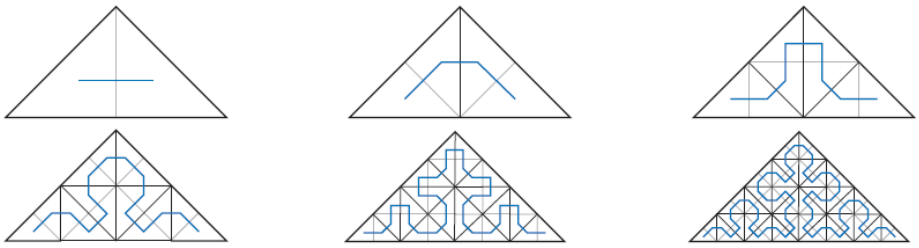

Abbildung 2: Die ersten sechs Iterationen der Sierpinski-Kurve

Das Durchlaufen des Gitters mit Hilfe der Sierpinski-Kurve entspricht einer Depth-First-Traversierung des Verfeinerungsbaumes (siehe Abbildung 3).

Abbildung 3: Die ersten sechs Iterationen der Sierpinski-Kurve

Dabei teilt die Sierpinski-Kurve das Dreieck in zwei Hälften auf. Diese werden durch die beiden Farben rot und grün charakterisiert. Die Aufteilung erfolgt nach der Seite der Sierpinski-Kurve, auf der die Knoten und Kanten liegen. Dies ist in Abbildung 4 zu sehen.

3.4 Eigenschaften der Gitterelemente

3.4.1 Kantenparameter

Die Kanten des Gitters können verschiedene Parameter besitzen. Hierbei unterscheidet man zwischen den Parametern neu und alt. Alte Kanten gehören zu einem benachbarten Dreieck, das bezüglich der Sierpinski-Ordnung vor dem aktuellen Dreieck liegt und somit bei einem Durchlauf bereits besucht wurde. Neue Kanten befinden sich zwischen Elementen, die bisher

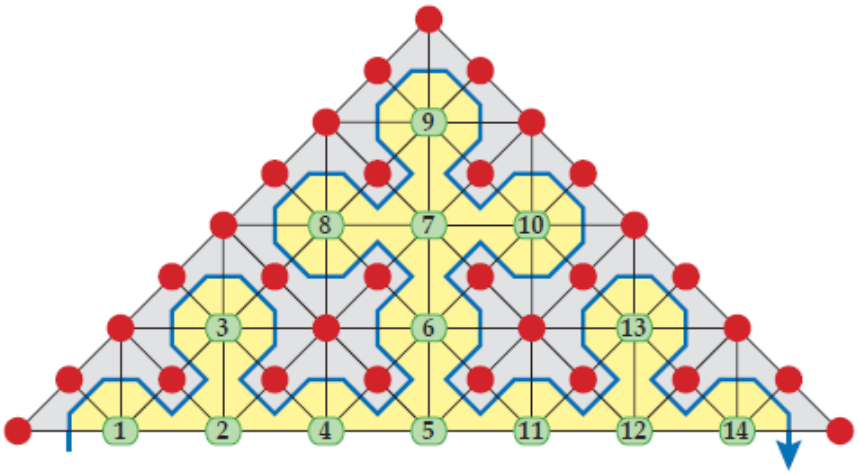

Abbildung 4: Die Sierpinski-Kurve teilt das Dreieck in zwei Hälften

noch nicht besucht wurden. Alle Kanten, die keine Randkanten sind, werden während der Traversierung zwei Mal besucht und wechseln dabei den Parameter von neu nach alt. Der Parameter für die Randkanten kann am Anfang frei gewählt werden.

3.4.2 Dreieckstypen

Ein Gitterelement ist ein Dreieck, das hier aus drei Knoten, drei Kanten und einer Zelle besteht. Dabei werden die von der Sierpinski-Kurve gekreuzten Kanten als Eintritts- und Austrittskante bezeichnet. Die dritte Kante des Dreiecks wird als durchgehende Kante bezeichnet.

Beim Durchlaufen der Dreiecke nach dem Sierpinski-Ansatz werden verschiedene Dreieckstypen unterschieden. Diese ergeben sich aus den Eintritts- und Austrittskanten der Sierpinski-Kurve. Folgende Dreieckstypen werden dabei betrachtet:

- **Typ H:** Der Eintritt der Sierpinski-Kurve erfolgt durch die Hypotenuse, der Austritt über eine Kathete des Dreiecks
- **Typ V:** Der Eintritt und der Austritt der Sierpinski-Kurve erfolgen über Katheten des Dreiecks
- **Typ K:** Der Eintritt der Sierpinski-Kurve erfolgt durch eine Kathete, der Austritt über die Hypotenuse des Dreiecks

Die verschiedenen Dreieckstypen sind in Abbildung 5 noch einmal dargestellt.

Den Dreiecken können nun wie den Kanten auch die Parameter alt oder neu zugewiesen werden. Dabei wird der Parameter der durchgehenden Kanten für das Dreieck übernommen,

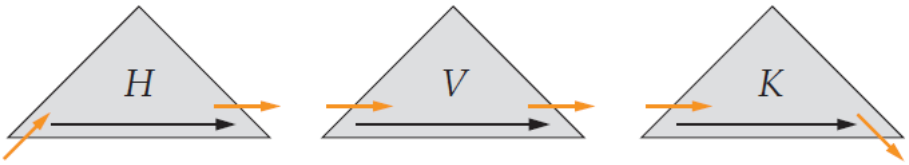

Abbildung 5: Die Dreieckstypen bei der Traversierung mit Hilfe der Sierpinski-Kurve

da die Eintrittskante immer den Parameter alt und die Austrittskante immer den Parameter neu besitzt. Dadurch ergeben sich dann Dreieckstypen mit dem Index n (new) und dem Index o (old).

3.5 Speicherstruktur während der Traversierung

Bei einer Untersuchung des Problems stellt sich heraus, dass Stapel als Speicherstruktur gut geeignet sind. Der Zugriff auf einen Stapel erfolgt nach dem „last in-first out“-Prinzip. Dabei sind zwei Operationen möglich:

- push: Ablegen eines Elementes oben auf dem Stapel
- pop: Entnehmen des obersten Elementes des Stapels

Bei Dreiecksgitter verwendet man in der Regel vier Stapel:

- Eingabestapel: enthält zu Beginn der Traversierung alle Daten
- grüner Stapel: Zwischenspeicher für grüne Knoten
- roter Stapel: Zwischenspeicher für rote Knoten
- Ausgabestapel: enthält am Ende der Traversierung alle Daten

Beim Durchlauf einer Traversierung können nun beim Lesen die Daten der Knoten auf der gemeinsamen Kante mit dem Vorgängerelement temporär gespeichert und so direkt von dort übernommen werden. Beim Schreiben können auch wieder die beiden Knoten, die auch zum Nachfolgerelement gehören so direkt weitergegeben werden. Der letzte verbleibende Knoten wird vom Eingabestapel oder von einem der Farbstapel gelesen. Dies ist davon abhängig, ob er während der Traversierung schon einmal besucht wurde. Beim Schreiben wird der Knoten auf den Ausgabestapel geschrieben, falls er bereits vollständig abgearbeitet ist, ansonsten wird er auf den zugehörigen Farbstapel geschrieben.

Auch die durchgehenden Kanten werden auf den Farbstapeln und dem Eingabe- und Ausgabestapel gespeichert, während die Eintritts- und Austrittskanten in einem Stream verwaltet werden.

Über die Wahl des richtigen Stapels entscheidet immer der Dreieckstyp. Falls der Dreieckstyp den Index n hat, werden die fehlenden Daten vom Eingabestapel gelesen, falls der Dreieckstyp den Index o hat, werden die entsprechenden Daten vom zugehörigen Farbstapel gelesen. Beim Schreiben werden die Daten in einem Dreieck mit dem Index n auf einen Farbstapel, in einem Dreieck mit dem Index o auf den Ausgabestapel geschrieben. Abbildung 6 zeigt noch einmal, auf welchen Stapel oder Stream die Knoten und Kanten abhängig vom Dreieckstyp gespeichert werden.

Die Farbe des Stapels wird jeweils durch die Farbe des Knotens bestimmt. Um die Farbe der Knoten zu bestimmen wird jedem Dreieck eine Referenzfarbe zugeordnet. Als Referenzfarbe eines Dreiecks wird die Farbe des Knotens, der gegenüber der Hypotenuse liegt, übernommen. Dadurch lässt sich für jeden Dreieckstyp auf die Farbe der anderen Knoten schließen. Abbildung 7 zeigt dies am Beispiel der Referenzfarbe Rot. Ausgehend von der Anfangstriangulation kann die Referenzfarbe der Dreiecke dann rekursiv bestimmt werden, da sie bei jeder Teilung der Dreiecke wechselt.

3.6 Zellorientierte iterative Lösungsverfahren

3.6.1 Allgemeine Vorgehensweise

Beim iterativen Lösen eines Gleichungssystems $Ax = b$ wird ausgehend von einem Startwert x_0 in jedem Iterationsschritt eine Korrektur Δx_k berechnet und diese auf den aktuellen Näherungswert x_k addiert. Als neuen Wert erhält man dann $x_{k+1} = x_k + \Delta x_k$. Die Berechnung des Korrekturwertes erfolgt dabei meistens mit Hilfe des Residuums $b - Ax_k$.

Bei der in Abschnitt 3.1 vorgestellten Finite-Elemente-Methode ergibt sich das zu lösende Gleichungssystem aus den Elementbeiträgen. Auf Basis der Elementgleichungssysteme können elementweise Beiträge zum Residuumsvektor berechnet werden. Da es auch Operationen gibt, die mit Knotenwerten arbeiten, müssen allerdings drei Arten von Operationen unterschieden werden:

- Elementweise Operationen: Durchführung der Berechnung in jedem Element
- Nodale Operationen: Durchführung für jeden Freiheitsgrad, erfolgen also für jeden Knoten einmal
- Skalare Operationen: Durchführung von Rechnungen mit skalaren Werten

Die elementweisen Operationen werden während der Traversierung der Elemente in jedem Element durchgeführt.

Die nodalen Operationen können an mehreren Stellen durchgeführt werden. Dabei ist zu beachten, dass sie auf keinen Fall mehrmals ausgeführt werden dürfen. Dabei kann es auch sein, dass gewisse nodale Operationen vor den elementweisen oder nach den elementweisen Operationen erfolgen müssen. Die Berechnung der Knotendaten sollte also an zwei Stellen während der Traversierung stattfinden:

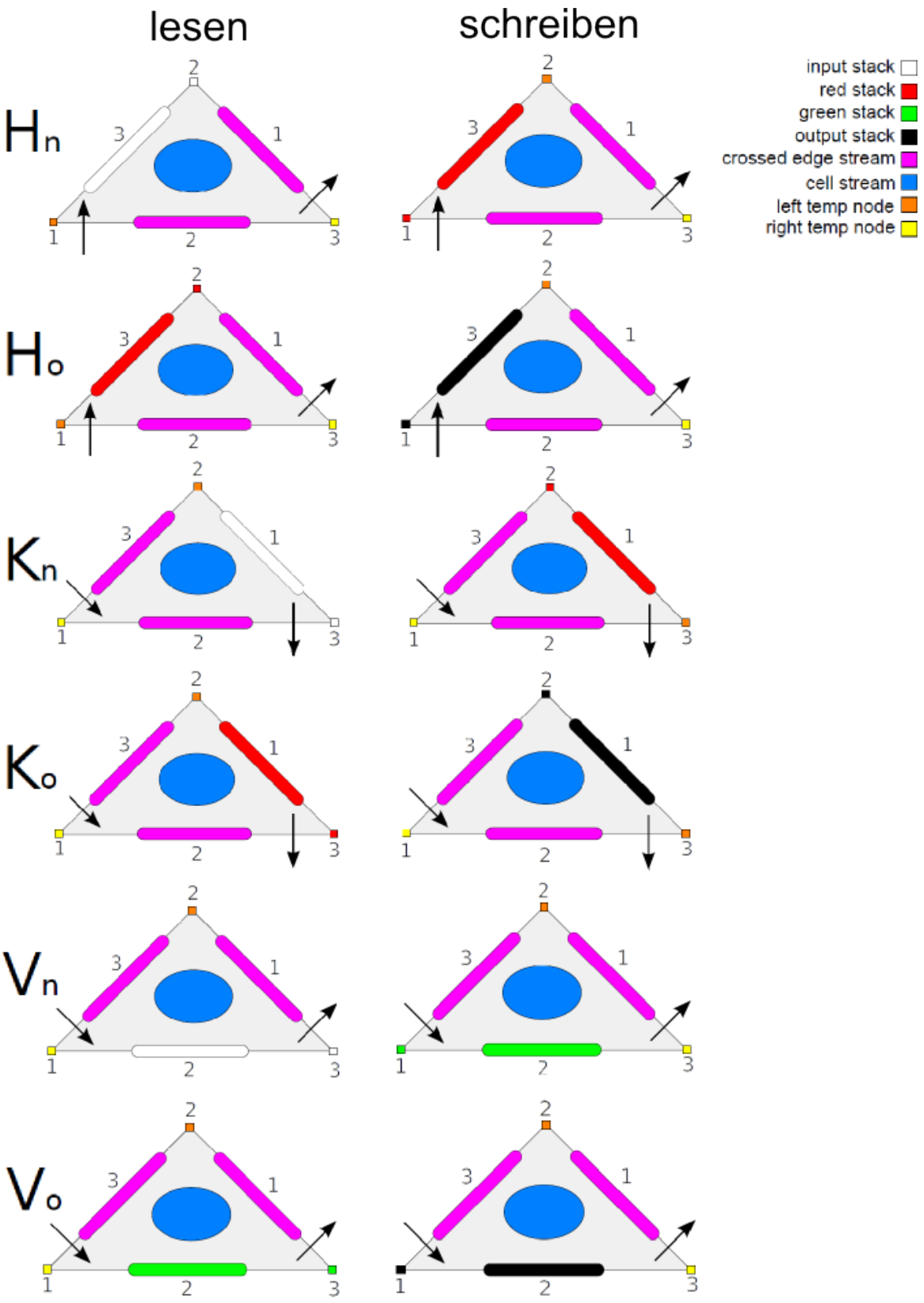

Abbildung 6: Unterscheidung des Speicherorts von Knoten und Kanten nach Dreieckstyp

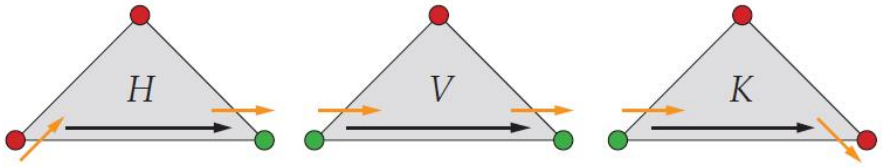

Abbildung 7: Beispiel der Farbverteilung in Dreiecken mit der Referenzfarbe rot

- beim Lesen des Knotens vom Eingabestapel
- beim Schreiben des Knotens auf den Ausgabestapel

Damit ist sichergestellt, dass die Daten des Knotens vor der ersten elementweisen Operationen, die ihn betrifft, aktualisiert werden und auch nach der letzten elementweisen Operation, die ihn betrifft, noch einmal aktualisiert werden können.

Skalare Operationen werden zwischen Gitterdurchläufen durchgeführt.

3.6.2 Gedämpftes Jacobi-Verfahren

Ausgehend von dem zu lösenden linearen Gleichungssystem $Ax = b$, benötigt man für das Jacobi-Verfahren auch noch die Diagonalmatrix $D = diag(A)$. Dabei ist $\omega \in]0,2[$ der Relaxationsparameter für den gewöhnlich $\omega \leqslant 1$ gilt.

Algorithmus 0.1 Gedämpftes Jacobi-Verfahren - knotenorientiert

wähle Startvektor x_0
für $k = 0, 1, 2, ...$
$r_k = D^{-1}(b - Ax_k)$
$x_{k+1} = x_k + \omega \cdot r_k$

Algorithmus 0.2 Gedämpftes Jacobi-Verfahren - elementorientiert

Initialisiere x mit Startwerten
für $k = 0, 1, 2, ...$
Gitterdurchlauf:

1. Knoten-Update:	$r_i \leftarrow 0.0$
	$d_i \leftarrow 0.0$
Element-Update:	$r_e \leftarrow r_e + b_e - A_e x_e$
	$d_e \leftarrow d_e + diag(A_e)$
2. Knoten-Update:	$x_i \leftarrow x_i + \omega \cdot r_i / d_i$

Normalerweise wird das Jacobi-Verfahren wie in Algorithmus 0.1 gezeigt knotenorientiert ausgeführt. Da das Jacobi-Verfahren aber ein Mehrschrittverfahren ist und deshalb keine bestimmte Reihenfolge bei der Bearbeitung der Freiheitsgrade einzuhalten ist, kann der Algorithmus auch elementorientiert formuliert werden. Eine Iteration entspricht dann genau einem Gitterdurchlauf. Die elementorientierte Variante des Algorithmus zeigt der Algorithmus 0.2. Variable d wurde eingeführt, da die globalen Diagonalelemente während der Traversierung extra berechnet werden müssen. Sie liegen nicht vor, da ja nur auf den Elementmatrizen gearbeitet wird. Die Variablen r und d werden immer am Anfang des Gitterdurchlaufs mit 0 initialisiert und brauchen damit nicht auf dem Ein- und Ausgabestapel abgelegt werden. Die Variable x dagegen wird während der Gitterdurchläufe immer nur aktualisiert und sollte somit auch vom Eingabestapel gelesen und am Ende auf den Ausgabestapel geschrieben werden.

4 Einsatzgebiet des Systems

Dieses Kapitel beschreibt das bestehende System und sein Einsatzgebiet.

4.1 Aufgabe des Systems

Durch die Parallelisierung von Plattformen und die wachsende Performancedifferenz zwischen CPU und Speichergeschwindigkeit in den letzten Jahren ist die Anpassung von Simulationscode auf die neue parallele Hardware ein wichtiges Thema geworden. Für gitterbasierte Löser von partiellen Differentialgleichungen haben sich bereits Lösungsansätze mit strukturierten Gittern als erfolgreich erwiesen.
Das Hauptziel des bestehenden Systems ist es, dieses Ergebnis nun auf die Hardware anzupassen und die komplizierten Traversierungsalgorithmen einem größeren Spektrum von Benutzern und Anwendungen zugänglich zu machen. Dafür wurde ein Schichtendesign entwickelt, welches aus drei verschiedenen Ebenen besteht und damit die Komplexität der zugrunde liegenden Datenstrukturen und Algorithmen vor dem Anwender verbergen soll. Abbildung 8 veranschaulicht die verschiedenen Ebenen.

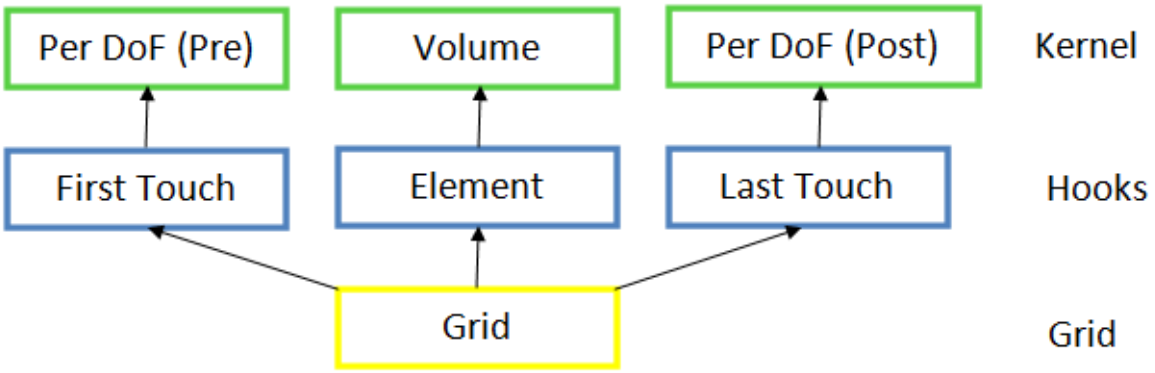

Abbildung 8: 3-Schichten-Modell

Auf der obersten Ebene befindet sich das Kernel-Layer als Frontend. Diese Ebene soll Benutzern die schnelle Integration von Szenarien ohne großes Wissen über die zugrunde liegenden Gitterprozesse ermöglichen.
Eine Ebene darunter befindet sich das Hook-Layer. Hier finden Aufrufe des Kernel-Layers statt und es werden Optimierungen implementiert, die direkten Gitterdatenzugriff verwenden. Des Weiteren können mit Hilfe des Hook-Layers element- und kantenbasierte Operatoren implementiert werden.
Auf der untersten Ebene befindet sich das Grid-Layer. Es enthält die Implementierungen von allen gitter- und datenrelevanten Basisfunktionalitäten sowie alle stack- und streambasierten Speichertransfers.

Während in Standardtraversierungen die Eingabe- und Ausgabestreams durch ihre fixen Gesamtgrößen zu einer einzigen Arraydatenstruktur zusammengefasst werden können, ist dies für die Verfeinerungs- bzw. Vergröberungstraversierungen nicht möglich, da sich die Gesamtgröße der Streams hier verändert. Daher wurde hier eine Arraystruktur für adaptive Traversierungen gewählt, bei der zwar die Menge an benötigtem Speicher vorübergehend verdoppelt wird, aber dennoch eine bessere Performance erzielt wird, als bei Stacks mit dynamischer Größe.
Für die adaptive Verfeinerung tritt jedoch das Problem auf, dass mehrere Elemente gleichzeitig geupdated werden müssen. Dadurch müssen normalerweise einige Schreiboperationen auf Stacks bzw. Streams hinausgeschoben werden, bis alle Ausgabeelemente verarbeitet wurden. Um dies zu umgehen wurde ein Element-Ringspeicher verwendet, der temporäre Daten speichert bis sie zurück auf die Stacks bzw. Streams geschrieben werden können.

4.2 Einsatzgebiet

Eine der Zielanwendungen dieses Systems ist die Berechnung der Strömungen von flüssigen Stoffen durch poröse Medien. Dies wird in Abbildung 9 veranschaulicht. Abbildung 10 zeigt den Ablauf der zugehörigen Traversierungen.
Zunächst werden die Anfangsbedingungen gesetzt. Danach wird die Permeabilität berechnet. Im dritten Schritt wird die Druckgleichung gelöst und anschließend im vierten Schritt die Geschwindigkeit berechnet. Danach erfolgt der erste Output über Paraview.
Im sechsten Schritt wird das lineare Gleichungssystem gelöst und erneut die Geschwindigkeit berechnet. Anschließend erfolgt der Transportschritt, indem die neue Sättigung berechnet wird. Dann wird die Permeabilität neu berechnet und es erfolgt wieder ein Output über Paraview. Die Schritte 6 bis 10 können beliebig oft wiederholt werden.

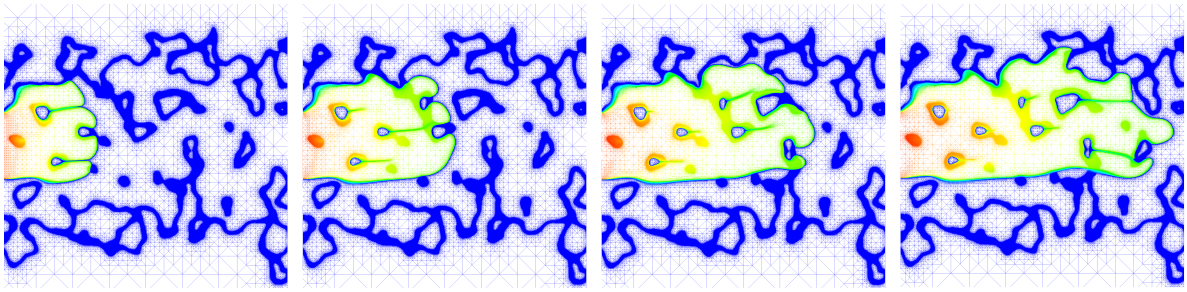

Abbildung 9: Bewegung eines flüssigen Stoffes durch ein poröses Medium

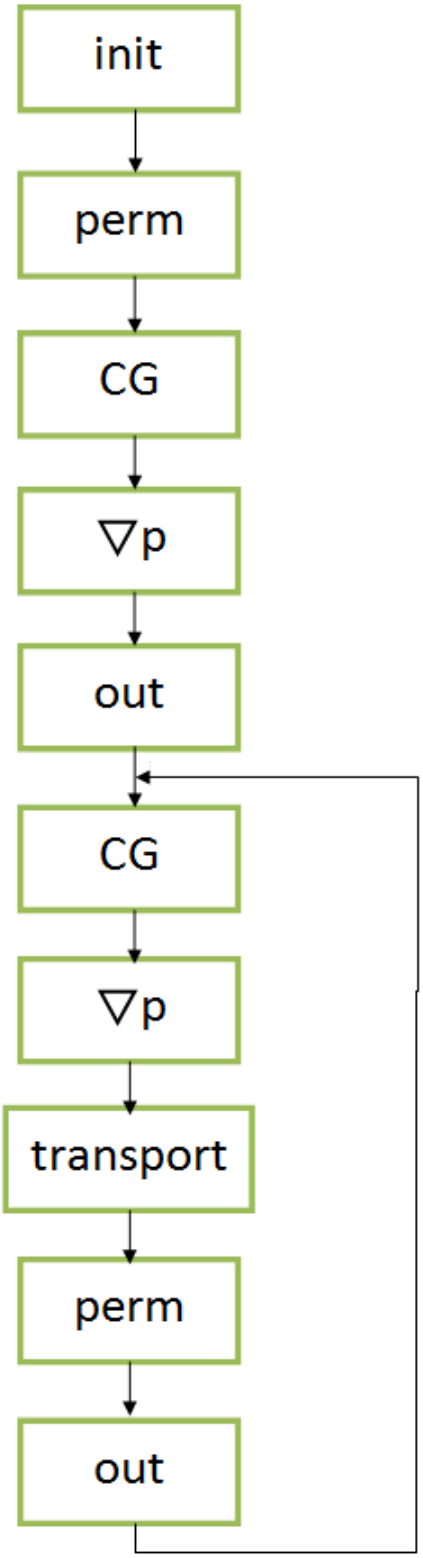

Abbildung 10: Berechnung der Bewegung eines flüssigen Stoffes durch ein poröses Medium

Um ein poröses Medium zu simulieren, wurde der Berechnungsbereich in einen frei fließenden Bereich mit teilweise undurchlässigem Material aufgeteilt. Durch die Diskretisierung auf einem feinen Gitter und weitere Verfeinerung der Transitionen erhält man ein Ausgangsnetz von ungefähr 1.4 Millionen Zellen.
Für die Analyse der seriellen Performance der Traversierungen wurde ein Shared-Memory-Compute-Server mit Intel-Xeon-E7540-Prozessoren und ein Laptop mit einem Intel-Core-i7-Prozessor verwendet.

Eine weitere Zielanwendung sind die Hitzegleichungen. In Abbildung 11 wird die kreisförmige Bewegung eines Laserstrahls auf einer porösen Metallplatte veranschaulicht. Abbildung 12 zeigt den Ablauf der Berechnung der Hitzegleichungen. Zuerst werden die Anfangsbedingungen gesetzt. Dann kann im nächsten Schritt beliebig verfeinert werden. Danach werden mit der Methode von Heun Zeitschritt 1 und Zeitschritt 2 berechnet. Darauf folgt der Output. Die Schritte 3 bis 5 können beliebig oft wiederholt werden.

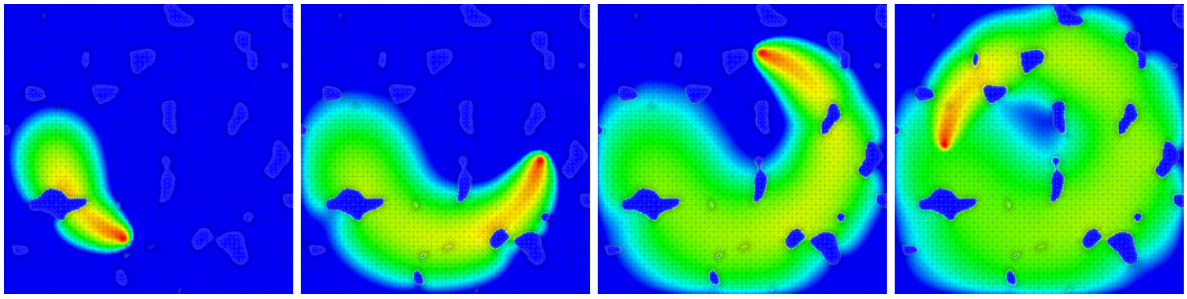

Abbildung 11: Bewegung eines Lasers auf einer porösen Metallplatte

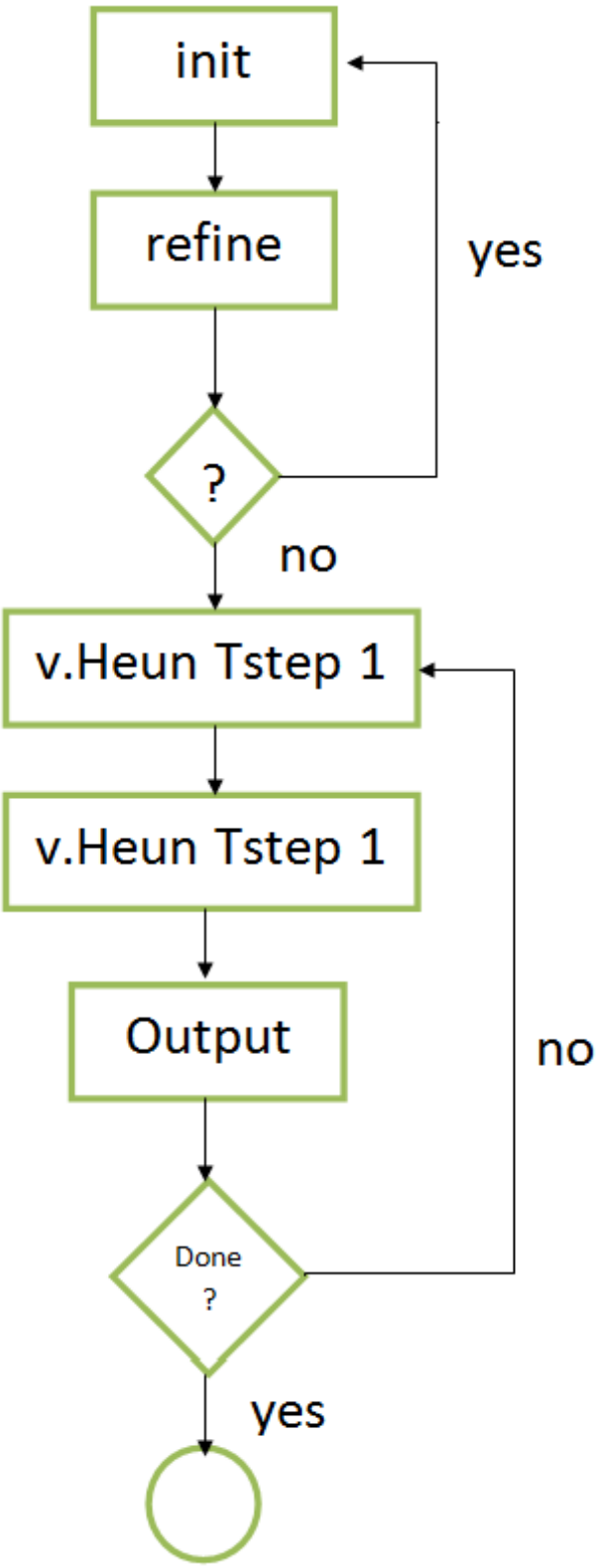

Abbildung 12: Berechnung der Hitzegleichungen

4.3 Ergebnisse

Die Berechnung von Strömungen von flüssigen Stoffen in porösen Medien ist rechnerisch sehr kostengünstig. Der Großteil der Ausführungszeit wird deshalb auch bei sehr vielen zu verarbeitenden Elementen für den Speicherzugriff aufgebracht. Die Speicheranforderungen mit adaptiver Verfeinerung sind nahezu doppelt so hoch, wie dies bei einer nicht-adaptiven Traversierung mit 1.4 Millionen Zellen der Fall ist.
Ein ganzer Durchlauf der Simulation, vom Zeitpunkt des Eintritts des flüssigen Stoffes in das poröse Medium, bis zu dem Zeitpunkt, zu dem der flüssige Stoff das poröse Medium komplett ausgefüllt hat, hat auf dem Testsystem annähernd 11 Stunden gedauert. Den Großteil dieser Zeit verbraucht der lineare Löser, obwohl normalerweise nur wenige Iterationen in einem Schritt ausgeführt werden. Es hat sich dennoch herausgestellt, dass die Ausführungszeit durch das Wechseln von einem Jacobi-Löser zu einem CG(Konjugierte-Gradienten)-Löser und zu einem Jacobi-preconditioned-CG-Löser um 52% reduziert werden konnte.

5 Eingesetzte Werkzeuge

Im Folgenden werden die eingesetzten Werkzeuge des bestehenden Systems aufgeführt.

5.1 Programmiersprache (FORTRAN)

Das System ist vollständig in der Programmiersprache FORTRAN 77/2003 implementiert, da diese Sprache für die effiziente Berechnung numerischer Probleme konzipiert wurde. Ob der Einsatz dieser Sprache zukünftig noch sinnvoll ist, wurde noch nicht evaluiert.

5.2 Präprozessoren

Da der Code des bestehenden Systems zusammengebaut werden muss und Codeteile nur manchmal ausgeführt werden sollen, wird mit Präprozessoren gearbeitet. Im bestehenden System wird konkret ein C-Präprozessor eingesetzt. Dafür wird der C-Compiler ccp verwendet.

5.3 ParaView

Für die Anzeige der generierten Daten muss ein grafisches Werkzeug eingesetzt werden, da die Anzeige nicht direkt im Programm möglich ist. Hier wird bisher ParaView verwendet. Jedoch werden die Daten nicht im einfachen Legacy-Format gespeichert, sondern in einem XML-Format, da dieses unter anderem schneller ist.

6 Problematik des bestehenden Systems

In diesem Kapitel wird die Problematik des in den vorigen Kapiteln beschriebenen Systems geschildert. Dabei sind einige Schwachstellen aufgefallen, die im Folgenden erläutert werden.

6.1 Die Programmiersprache FORTRAN 77 / 2003

Das gegebene System ist in der Programmiersprache FORTRAN 77 / 2003 implementiert. Mit dieser Sprache ist es zwar möglich Berechnungen sehr effizient auszuführen, jedoch bietet sie keine Objektorientierung und somit keine Vererbung an. Bei der Erstellung einer bestimmten Gittertraversierung steht der Benutzer nun vor dem Problem, dass die einzelnen Programmkomponenten zusammengefügt werden müssen. Bisher wird dies mit Präprozessor-Anweisungen gelöst. Weitere Möglichkeiten die Objektorientierung zu simulieren werden in dieser Fachstudie untersucht.

Ein weiteres Problem mit FORTRAN ist die Effizienz bei der Codeausführung. Zwar können Berechnungen sehr schnell ausgeführt werden, jedoch ist das Durchlaufen des FORTRAN-Sourcecodes eher langsam.

6.2 Bedienung des Systems

Wie im Abschnitt davor angedeutet ist die derzeitige Bedienung des bestehenden Systems alles andere als trivial. Der Benutzer benötigt Kenntnisse über die Methoden des Systems und über Präprozessor-Anweisungen. Für einen Benutzer, der nicht aus dem Bereich der Softwareentwicklung stammt (z.B. ein Physiker, der Berechnungen ausführen will) ist es nur schwer bedienbar.

6.3 Verfügbarkeit der Analyseergebnisse

Während der Analyse können mit dem bestehenden System zwar ohne weiteres Analyseergebnisse generiert werden, dies jedoch in einem Dateiformat, das keinen direkten Einblick in die Analyseergebnisse erlaubt. Hierfür muss das Zusatztool ParaView aufgerufen werden. Dieser zusätzliche Schritt soll zukünftig vermieden werden (z.B. mit dem Ansatz der Workflow-GUI).

6.4 Erweiterbarkeit

Die Erweiterbarkeit des bestehenden Systems ist eingeschränkt, da beim Hinzufügen neuer Methoden auch die entsprechenden Präprozessor-Anweisungen eingebaut und überarbeitet werden müssen. Dies sorgt für zusätzlichen Aufwand, der beispielsweise mit der Verwendung eines Python-Skriptes behoben werden könnte.

6.5 Zusammensetzen der Methoden

Das Zusammensetzen der Methoden um Code einer bestimmten Gittertraversierung zu erhalten geschieht derzeit mittels Präprozessor-Anweisungen. Die Nachteile bzw. Probleme bei dieser Methode sind in Kapitel 8 nachzulesen.

7 Nachbau des bestehenden Systems

Wie im Einführungskapitel beschrieben, sollte im Rahmen dieser Fachstudie ein nicht-funktionaler Nachbau des bestehenden Systems erfolgen, um die Problematik nachvollziehen zu können und auf Basis des Nachbaus Prototypen für die Lösungsansätze zu erstellen. Der Nachbau erfolgte in der Programmiersprache Java 1.6.

Hierbei wurde das grobe Konzept des bestehenden Systems mit den wichtigsten Methoden und Datentypen nachgebaut.

7.1 Datenstrukturen und Typen

Im Klassendiagramm in Abbildung 13 ist die Klassenstruktur des Nachbaus abgebildet.

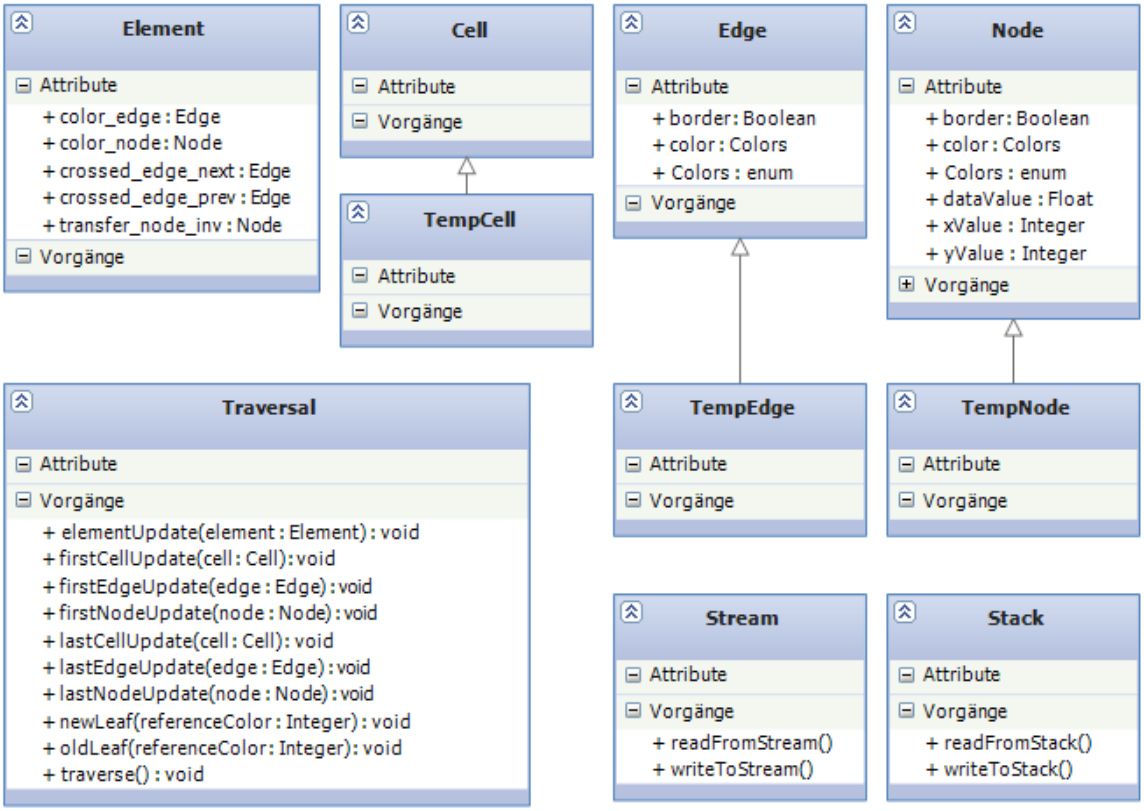

Abbildung 13: Klassendiagramm des Nachbaus

Folgende Typen wurden in Form von Klassen implementiert.

- Knoten, Kanten, Zellen
- Temporäre Daten für Knoten, Kanten und Zellen

- Element, welches den konkreten Typ von Knoten und Kanten während einer Traversierung beschreibt

Stacks und Streams
Die genutzten Stacks und Streams wurden durch generische Klassen simuliert, welche sowohl Kanten, Knoten als auch Zellen aufnehmen können. Durch die simulierten Stacks und Streams war der Nachbau selbstverständlich nicht-funktional.

7.2 Methoden

Methode für die Interaktion auf einem altem Element

Diese Methode bekommt eine Referenzfarbe übergeben und führt nun diverse Operationen auf Kanten, Knoten und Zellen durch.

Als Erstes werden die Knoten und Kanten von den Stacks gelesen und falls temporäre Daten vom Benutzer gewünscht sind, auch diese. Danach wird die Zelle vom Stream gelesen.

Anschließend werden First-Touch-, Last-Touch- und Element-Operatoren auf diesen Elementen in einer bestimmten Reihenfolge durchgeführt:

```
elementUpdate(element);
lastNodeUpdate(element.transfer_node);

//falls temporäre Daten vorhanden
lastNodeUpdate(tempNode);

lastEdgeUpdate(element.color_edge);

//falls temporäre Daten vorhanden
lastEdgeUpdate(tempEdge);

lastCellUpdate(element.cell);

//falls temporäre Daten vorhanden
lastCellUpdate(tempCell);
```

Anschließend werden die Elemente wieder auf die entsprechenden Stacks oder Streams geschrieben. Die Reihenfolge der Lese-und Schreiboperationen auf den Streams bzw. Stacks ist in Kapitel 3 nachzulesen.

Methode für die Interaktion auf einem neuen Element

Diese Methode ist analog zur Methode für die Interaktion auf einem alten Knoten aufgebaut, lediglich die Aufrufe sind verschieden:

```
firstNodeUpdate(element.color_node);

//falls temporäre Daten vorhanden
firstNodeUpdate(tempNode);
```

```
firstEdgeUpdate(element.color_edge);

//falls temporäre Daten vorhanden
firstEdgeUpdate(tempEdge);

firstCellUpdate(element.cell);

//falls temporäre Daten vorhanden
firstCellUpdate(tempCell);

elementUpdate(element);
```

Traversierungs-Methode
Diese Methode führt je nach Element entweder die Methode für alte oder neue Elemente aus.

Mögliche Lösungsansätze

8 Präprozessor-Anweisungen

Dieses Kapitel behandelt den Lösungsansatz mit Präprozessor-Anweisungen. Dabei wird zunächst am Beispiel eines C-Präprozessors beschrieben, wie Präprozessoren im Allgemeinen funktionieren. Anschließend wird darauf eingegangen, welche Einstellungen der Benutzer für die Traversierung vornehmen kann und wie der Präprozessor ausgeführt wird und die konfigurierte Traversierung zusammenstellt.

8.1 Funktionalität

8.1.1 Präprozessoren

Ein Präprozessor ist im Grunde genommen ein Textersetzungsprogramm. Er dient dazu, Eingabedaten zu konvertieren bevor sie an ein anderes Programm weitergegeben werden, damit dieses sie verarbeiten kann. Das passiert zum Beispiel direkt bevor ein Programm kompiliert wird. Es gibt jedoch auch die Möglichkeit einen Präprozessor separat auszuführen, wie in diesem Lösungsansatz.

Die Funktionen eines Präprozessors umfassen unter anderem:

- Das Definieren von Konstanten
- Das Auslassen von Textblöcken durch if-Bedingungen
- Das Einfügen von Dateiinhalten

Für die oben genannten Funktionen gibt es sogenannte Präprozessor-Direktiven [MSD], mit denen die Funktionen in den Code eingebaut werden können:

`#define` definiert eine Konstante, z.B. `#define X`

`#include` fügt eine Quelldatei ein, z.B. `#include 'Jacobi_Traversal.java'`

`#if` beschreibt ein Codestück, das ausgeführt wird, wenn eine bestimmte Bedingung zutrifft, z.B.

```
#if X>0
<Codestück>
#endif
```

`#else` beschreibt ein Codestück, das ausgeführt wird, wenn eine vorhergegangene if-Bedingung nicht erfüllt wurde, z.B.

```
X=0
#if X>0
<Codestück das nicht ausgeführt wird>
#else
<Codestück, das ausgeführt wird>
#endif
```

`#elif` beschreibt ein Codestück, das ausgeführt wird, wenn eine vorhergegangene if-Bedingung nicht zugetroffen hat und eine gegebene Bedingung erfüllt wird, z.B.

```
X=0
#if X>0
<Codestück das nicht ausgeführt wird>
#elif X<0
<Codestück, das nicht ausgeführt wird>
#elif X=0
<Codestück, das ausgeführt wird>
#endif
```

`#ifdef` beschreibt ein Codestück, das ausgeführt wird, wenn eine bestimmte Konstante definiert ist, z.B.

```
#define X
#ifdef X
<Codestück>
#endif
```

`#ifndef` beschreibt ein Codestück, das ausgeführt wird, wenn eine bestimmte Konstante nicht definiert ist, z.B.

```
#define X
#ifndef Y
<Codestück>
#endif
```

`#undef` macht das Definieren einer Konstante rückgängig, bzw. löscht diese, z.B.

```
#define X
```

```
#ifdef X
<Codestück das ausgeführt wird>
#endif
#undef X
#ifdef X
<Codestück das nicht ausgeführt wird>
#endif
```

`#endif` ist der Abschluss einer if-Bedingung.

8.1.2 Lösungsansatz

Das Ziel dieses Lösungsansatzes ist es die Problematik aus Kapitel 6 mit Hilfe von Präprozessor-Anweisungen zu umgehen. Die ursprüngliche Funktionalität soll dabei unverändert bleiben.

Der Benutzer soll folgende Einstellungen vornehmen können:

- **First-Touch-/Last-Touch- und Element-Operatoren**
  Der Benutzer soll die First-Touch-/Last-Touch- und Element-Operatoren selbst definieren können.

Der Programmierer soll zusätzlich noch Folgendes vornehmen können:

- **Knoten, Kanten und Zellen**
  Der Programmierer soll auswählen können, ob er Knoten, Kanten und Zellen in der Traversierung haben möchte und diese entsprechend ein- bzw. ausschalten können.
- **Temporäre Daten**
  Der Programmierer soll auswählen können, ob er zusätzlich einen temporären Speicher für Knoten, Kanten und Zellen haben möchte.

8.2 Umsetzung

Im Lösungsansatz mit Präprozessor-Anweisungen kann der Benutzer die obigen Einstellungen in der Präprozessor-Klasse vornehmen. Sie enthält die Definitionen der Konstanten und Operatoren. In unserem Beispiel heißt die Präprozessor-Klasse *TraversierungPP.jpp*. Dabei handelt es sich um eine Java-Klasse mit C-Präprozessor-Anweisungen, die nachher mit einem C-Präprozessor übersetzt werden soll. Wie diese Klasse in unserem Beispiel aussieht kann im Anhang nachgelesen werden.

8.2.1 Einstellungen für die Traversierung vornehmen (aus Programmierersicht)

Knoten, Kanten und Zellen ein- bzw. ausschalten

Für Knoten, Kanten und Zellen sind drei Konstanten vorgesehen. In unserem Beispiel heißen diese `NODE`, `EDGE` und `CELL`. Sie geben an, ob die entsprechenden Codeteile für Knoten, Kanten und Zellen mitkompiliert werden sollen oder nicht. Dafür werden die Präprozessor-Anweisungen `#define` und `#undef` verwendet.
Mit `#define` wird die entsprechende Konstante explizit definiert. Das bedeutet, dass der entsprechende Codeteil mitkompiliert wird. Mit `#undef` wird eine Konstante explizit *nicht* definiert, bzw. gelöscht. Dass heißt, der entsprechende Codeteil wird ausgelassen. In diesem Fall hätte es den gleichen Effekt, die entsprechende Zeile mit `define` einfach heraus zu löschen und leer zu lassen. Die Verwendung von `#undef` dient hier nur dazu undefinierte Variablen trotzdem sichtbar im Code stehen zu haben, damit der Benutzer jederzeit sieht, welche möglichen Variablen es gibt, die er ein- bzw. ausschalten kann.
Das Definieren der Konstanten findet am Anfang der Präprozessor-Klasse statt. Das folgende Beispiel kompiliert nur die Codeteile für Knoten und Kanten, Zellen werden nicht beachtet:

```
#define NODE
#define EDGE
#undef CELL
```

oder

```
#define NODE
#define EDGE
```

Temporäre Daten ein- bzw. ausschalten

Analog zum Ein- und Ausschalten von Knoten, Kanten und Zellen gibt es auch eine Konstante für das Ein- und Ausschalten von temporären Daten. In unserem Beispiel heißt diese Konstante `TEMP`. Wird diese Konstante mit `#define` definiert, werden die Teile des Codes für temporäre Daten von Knoten, Kanten und Zellen kompiliert, sofern die entsprechenden Variablen für Knoten, Kanten und Zellen auch definiert sind.
Das folgende Beispiel kompiliert die temporären Daten für Knoten und Zellen:

```
#define NODE
#define CELL
#undef EDGE
#define TEMP
```

8.2.2 Einstellungen für die Traversierung vornehmen (aus Benutzersicht)

First-Touch-/Last-Touch- und Element-Operatoren definieren

Für die First-Touch-/Last-Touch- und Element-Operatoren werden zunächst Konstanten definiert, die beim Ausführen des Präprozessors in der Quelldatei durch die zugehörigen Funktionsrümpfe ersetzt werden. Die Funktionsrümpfe für die First-Touch-/Last-Touch- und Element-Operatoren befinden sich nach den Konstanten in der Präprozessor-Klasse und sind zunächst leer. In diese Funktionsrümpfe kann der Benutzer den Code für die Operatoren frei eintragen. Diese Operatoren werden später durch den Präprozessor an der entsprechenden Stelle in der Quelldatei eingefügt bzw. ersetzt.
In unserer Präprozessor-Klasse sehen die Funktionsrümpfe wie folgt aus:

```
 #define ELEMENT_OP ElementUpdate


public void ElementUpdate(Element element){
<Vom Benutzer eingegebener Code>
}
```

8.2.3 Ausführen des Präprozessors

Wenn der Benutzer alle Einstellungen für die Traversierung in der Präprozessor-Klasse vorgenommen hat, kann er den Code ersetzen lassen. Dazu wird zunächst ein Präprozessor benötigt. Das kann z.B. *gcc* oder der *C/C++-Compiler von Microsoft* sein. Diesen kann der Benutzer über eine Kommandozeile ausführen lassen. Für die genannten Beispiele wären das:

`gcc -E` bzw. `g++ -E` beim gcc

bzw.

`cl /P` beim Microsoft C/C++-Compiler.

Der Compiler setzt dann entsprechend den vom Benutzer vorgenommenen Einstellungen aus den Codestücken den endgültigen Code für die Traversierung zusammen.
In unserem Beispiel haben wir den C/C++-Compiler von Microsoft verwendet. Da Präprozessoren im Grunde genommen reine Textersetzungsprogramme sind und deshalb im Allgemeinen nicht von der Programmiersprache abhängig sind, ist es kein Problem einen C-Präprozessor zu verwenden um eine Java-Klasse zu übersetzen.
Da unsere Präprozessor-Klasse 'TraversierungPP.java' heißt, lautet die einzugebende

Kommandozeile `cl /P TraversierungPP.java`.

Wenn der Benutzer nun den Präprozessor ausführt, wird die Präprozessor-Klasse durchlaufen. Dabei werden die Konstanten und Operatoren definiert. Anschließend wird die Quelldatei durchlaufen, welche durch `#include` am Ende in die Präprozessor-Klasse eingebunden wurde. Entsprechend der Definitionen der Konstanten wird diese Datei durchlaufen und die entsprechenden Codeteile zusammengesetzt, bzw. die Operatoren durch die vom Benutzer definierten Funktionsrümpfe ersetzt.
In unserem Beispiel erhält der Benutzer am Ende eine Datei mit Java-Code für eine Traversierung mit der Jacobi-Methode. Diese kann nun von einem Java-Compiler übersetzt und ausgeführt werden.

9 Python-Skripte

In diesem Kapitel wird der Lösungsansatz eines Python-Skriptes betrachtet. Es wird hierbei auf die besonderen Eigenschaften von Skriptsprachen, die Programmiersprache Python und auf die Funktionalität und Umsetzung des verfassten Python-Skriptes dieser Fachstudie eingegangen.

Die Beschreibung der Funktionalität gliedert sich in zwei Teile. Im ersten Teil wird die Bedienung des Skriptes aus der Perspektive des Nutzers geschildert, im zweiten die Umsetzung des Skriptes aus der Programmiererperspektive.

9.1 Eigenschaften von Skriptsprachen

Als Skripte werden kleine, überschaubare, eigenständige Programme bezeichnet, bei denen ein hoher Wert auf Einfachheit und Übersichtlichkeit gelegt wird. Auf den Deklarationszwang von Variablen wird meist verzichtet. Dadurch können kleine Programme sehr schnell erstellt werden. Typische Merkmale einer Skriptsprache sind implizit deklarierte Variablen, dynamische Typisierung, automatische Speicherverwaltung und das keine getrennte Übersetzungsphase existiert (sie werden interpretiert). Der Aufruf eines Skriptes erfolgt über einen Kommandozeilenbefehl, hierbei können Ausführungsparameter übergeben werden. Skripte werden meist dazu verwendet mit anderssprachigen Programmen zu interagieren (Programmmodifikation, Programmaufrufe).

9.2 Die Programmiersprache Python

Die Programmiersprache Python gehört zu den höheren Programmiersprachen. Sie unterstützt sowohl objektorientierte, aspektorientierte, als auch funktionale Programmierung. Da es sich bei Python um eine sehr dynamische Sprache handelt, wird sie oft als Skriptsprache verwendet. Die Syntax von Python ist reduziert und auf Übersichtlichkeit optimiert, die Anzahl der Schlüsselworte gering gehalten. Die String-Operationen von Python sind sehr vielfältig und trotzdem einfach zu verstehen.

Python bietet die Möglichkeit Programme aus anderen Programmiersprachen als Module einzubetten. Eine Besonderheit dieser Sprache ist die Strukturierung durch Einrückung, womit gewährleistet ist, dass der Code übersichtlich gestaltet wird. [Pyt]

9.3 Bedienung und Funktionalität des Prototypen

Das Ziel dieses Python-Skriptes ist es, die in Kapitel 6 beschriebene Problematik zu beheben und die geforderte Funktionalität weiterhin zu gewährleisten (siehe auch Kapitel 1). Das Python-Skript dient der Generierung von Quellcode für die Ausführung einer Gittertraversierung. Je nach Einstellung des Benutzers, setzt es entsprechende Programmteile

baukastenartig zusammen und liefert letztendlich den angepassten Code für eine beliebige Gittertraversierung.

9.3.1 Konfigurieren des Skripts

Vor der Ausführung des Skriptes hat der Benutzer die Möglichkeit dieses für eine bestimmte Gittertraversierung anzupassen. Dies geschieht in einer Konfigurationsdatei, deren vollständiger Dateiname bei der Ausführung des Skriptes als Parameter übergeben wird. In dieser werden sowohl die Einstellungen des Skriptes festgelegt, als auch eigener Code eingefügt. Die Konfigurationsdatei lässt sich mit einem beliebigen Texteditor erstellen, öffnen und ändern. Der Aufruf des Skriptes mit der angepassten Konfigurationsdatei (für die Beispieldatei config.cfg und dem Skriptnamen python_cg) sieht wie folgt aus:

python python_cg.py config.cfg

Folgende Einstellungsmöglichkeiten existieren in der Konfigurationsdatei (genauere Erklärungen hierzu in Kapitel 3):

- Knoten, Kanten und Zellen:
  Der Nutzer hat die Möglichkeit Knoten, Kanten oder Zellen für die Traversierung ein- bzw. auszuschalten. Je nach Wahl, werden diese dann im generierten Code beachtet, oder nicht.
- Temporäre Daten:
  Der Nutzer hat die Möglichkeit die Traversierung mit oder ohne temporären Daten durchzuführen. Temporäre Daten existieren für Knoten, Kanten und Zellen. Sind diese aktiviert, so wird ein weiterer temporärer Speicher genutzt und die benötigten Aufrufe dementsprechend im generierten Code eingefügt.
- First-Touch-, Last-Touch-, und Element-Operator:
  Da das Skript sehr generisch ist und somit für jegliche Traversierung Code generiert, muss der Benutzer die ausgeführten Berechnungen und Operationen auf den Knoten, Kanten und Zellen selbst angeben. Dabei muss der Benutzer den First-Touch-, den Last-Touch- und den Element-Operator definieren, welche Berechnungen auf den Knoten, Kanten und Zellen ausführen. Der Code dieser Operatoren kann direkt in die Konfigurationsdatei geschrieben werden und wird vom Skript an der entsprechenden Stelle im Code eingefügt und aufgerufen.

9.3.2 Sonderfälle

Bei einer Traversierung gibt es nicht nur den Fall, dass ein Element (Kante, Knoten oder Zelle) als „alt“ oder „neu“ klassifiziert wird, es gibt auch diverse Sonderfälle. Zu diesen gehören:

- Randknoten

- Erste Zelle des Gitters
- First / Lasttouch Events

Da die Methoden für diese Sonderfälle eine sehr große Ähnlichkeit besitzen, werden sie vom Skript baukastenartig aus vordefinierten Aufrufen zusammengesetzt und eingefügt (siehe Abschnitt Umsetzung).

9.3.3 Ausführen des Skriptes

Für das Ausführen des Python-Skriptes ist mindestens die Python-Version 2.7.1 Voraussetzung. Es wird über den Kommandozeilenbefehl *python „Skriptname.py“ „Dateipfad der Konfigurationsdatei“* ausgeführt. Dadurch lässt sich das Skript problemlos in ein Makefile integrieren

9.3.4 Ablauf des Skriptes / Funktionalität aus Nutzersicht

Im Folgenden wird die Funktionalität des Skriptes anhand des Ablaufs geschildert.
Ist das Skript konfiguriert und sind die Last- bzw. First-Touch Operatoren definiert, kann das Skript über den oben genannten Kommandozeilenbefehl ausgeführt werden. Anfangs wird entweder eine neue „leere“ Codedatei angelegt oder, falls diese schon besteht, wird die bestehende Datei geleert. Anschließend liest das Skript die Informationen aus der Konfigurationsdatei ein und speichert diese temporär ab. Nun werden die je nach Einstellung benötigten Typdeklarationen, in welchen der Aufbau von Knoten, Kanten oder Zellen definiert ist, in die Codedatei geschrieben. Danach werden die Methoden für die alten bzw. neuen Elemente der Traversierung generiert (siehe Kapitel 3 (alte neue Knoten / Kanten)). Hierbei wird eine Codedatei ausgelesen, die diese Methoden für eine vollständige Traversierung (mit Knoten, Kanten, Zellen und temporären Daten) enthält. Auf dem ausgelesenen Code werden nun Filteroperationen ausgeführt, die je nach Einstellung Kanten, Knoten, Zellen oder temporäre Daten herausfiltern. Der gefilterte Code wird dann in die angelegte Datei geschrieben. Am Ende werden die Operatoren und die Methoden für die Sonderfälle in die Datei geschrieben und das Skript beendet.

Der Gesamtablauf des Skriptes noch einmal zusammengefasst:

- Ein- / Ausschalten von Kanten, Knoten, Zellen und temporären Daten durch den Benutzer
- Einfügen der Operatoren durch den Benutzer
- Ausführen des Skriptes
- Automatische Generierung und Befüllen der Codedatei

Für den Benutzer ist der Ablauf des Skriptes weitestgehend unsichtbar, nach der Ausführung sind keinerlei Benutzereingaben notwendig.

9.4 Umsetzung

In diesem Abschnitt wird eine mögliche Umsetzung des Skriptes aus Programmiersicht geschildert. Der Prototyp des Skriptes wurde in der Pythonversion 2.7.1 verfasst und ist somit auf die Mittel und Möglichkeiten dieser Version beschränkt.

Anlegen der Datei für den generierten Code

Das Anlegen einer leeren Codedatei ist in Python sehr simpel, durch den Befehl *Variable = open("Dateiname", "w")* wird eine neue Datei angelegt und durch eine Variable repräsentiert, auf welcher die Schreib- und Leseoperationen ausgeführt werden. Der Parameter „w" gibt der Datei eine Schreibberechtigung.

Auslesen der Konfigurationsdatei

Für das Auslesen der Konfigurationsdatei wird der Dateiname, welcher beim Aufruf des Skriptes übergeben wird, mit Hilfe des *sys* - Moduls eingelesen und die Konfigurationsdatei zum Lesen geöffnet.

```
for eachArg in sys.argv: \\iteriere über die bei der Ausführung übergebenen Argumente
       y = eachArg
       config = open(y, 'r') \\öffne die Konfigurationsdatei mit Leserechten
```

Mit Hilfe einer for-Schleife über die Zeilen der Konfigurationsdatei werden die Einstellungen für das Ein- und Ausschalten der Knoten, Kanten und Zellen mit dem Python-Befehl *readLine()* aus der Konfigurationsdatei ausgelesen und in Variablen abgespeichert. Eine Alternative wäre es, diese Einstellungen als Parameter beim Aufruf des Skriptes statt der Konfigurationsdatei zu übergeben. Beispielsweise könnte ein Aufruf dann wie folgt aussehen: *python „Skriptname" (temp)nodes (temp)edges (temp)cells*, dies würde den Aufruf des Skriptes aber komplizierter gestalten und es müsste eine Anleitung hierfür existieren. Die Konfigurationsdatei hingegen ist selbsterklärend. Das Auslesen der Operatoren funktioniert ähnlich wie das Auslesen der Einstellungen. Das Python-Skript liest die Konfigurationsdatei Zeile für Zeile mittels einer Schleife ein und speichert die Informationen ab einem eingefügten Startzeichen, bis ein weiteres Zeichen für das Ende der zu lesenden Information erreicht wird.

Im Python-Skript:

```
for line in config:   //iteriere über alle Zeilen
  if "start" in line:
    for line in config:
      temp += config.readline() //speichere die Zeilen bis das Endzeichen erreicht ist
      if "end" in line:
        break //verlasse die Schleife wenn das Endzeichen erreicht ist
```

In der Konfigurationsdatei:

```
//start #Ab diesem Punkt einlesen und speichern
Code des Operators
Code des Operators
...
//end #Ab diesem Punkt das Einlesen unterbrechen
```

Auslesen und Einfügen der Typdeklarationen

Das Auslesen der Typdeklarationen für Kanten, Knoten und Zellen kann sehr einfach gestalten werden. Da diese meist in separaten Klassen (bzw. Codedateien) liegen, können sie nach dem Öffnen der Dateien mit der *read()* Funktion von Python ausgelesen und direkt in die generierte Codedatei geschrieben werden. Sind mehrere Typen in einer Codedatei definiert, so kann wie beim Auslesen der Operatoren ein Start- und Endzeichen für das Einlesen des Codes hinzugefügt werden.

Auslesen der benötigten Methoden

Das Auslesen der Methoden erfolgt analog zum Auslesen der Operatoren aus der Konfigurationsdatei mittels einem Start- bzw. Endzeichens.

Filteroperationen

Für das Filtern der Methoden werden die von Python bereitgestellten String Operationen genutzt. Mit Hilfe der Python-Funktionen *readline* und *continue* kann eine Zeile (bzw. beliebig viele darunter liegende Zeilen) ausgelesen oder übersprungen werden.
Im folgenden Beispiel wird beschrieben, wie Kanten und die Aufrufe des temporären Speichers herausgefiltert werden; für Zellen und Knoten funktioniert dies analog. Die Variablen edge und temp_edge sind dabei die ausgelesenen Einstellungen aus der Konfigurationsdatei.

```
for line in code_file: //auslesen der Codedatei Zeile für Zeile
  temp = code_file.readline(); //speichern des Zeileninhalts
  if edge == 'n' and 'edge' in temp: //Zeile überspringen falls sie Aufrufe für Kanten enthält
    continue
  if temp_egde == 'n' and 'temp_edge' in temp: //Zeile überspringen falls sie Aufrufe für
      temporäre Daten enthält
    continue
  generatedCode.write(temp) //schreiben des gefilterten Codes
```

Einfügen der Methoden für die Sonderfälle

Da die Methoden der Sonderfälle, wie oben bereits erwähnt, eine sehr hohe Ähnlichkeit aufweisen, lassen sich diese aus vordefinierten Bausteinen zusammensetzen. Alle Aufrufe, die in diesen Methoden benötigt werden, können in ein Array geschrieben werden.

Anschließend können die Methoden aus diesen Arrayeinträgen zusammengestellt werden. Diese Vorgehensweise ist sehr sinnvoll, da die Methoden beinahe dieselben Aufrufe enthalten und so kein redundanter Code im Skript vorhanden ist.

Zusammenfügen der generierten Bausteine

Das Zusammenfügen der ausgelesenen Codestücke geschieht mit der *write()* Operation von Python in einer festgelegten Reihenfolge.

10 Workflow-GUI

Als möglicher Lösungsansatz für das System des Kunden kommt auch eine Workflow-GUI in Frage. Hierfür wurde ein Prototyp einer Workflow-GUI in der Programmiersprache Java entwickelt. Dieser soll eine Vorstellung vermitteln, wie man die gewünschte Funktionalität mit Hilfe einer grafischen Oberfläche bedienen kann. Der Ursprungsgedanke hierbei ist, dass eine grafische Oberfläche meist intuitiv zu bedienen ist und der Benutzer den dahinterstehenden Code nicht kennen und nicht selbst verändern muss. Das Zusammenbauen des Codes kann im Hintergrund geschehen ohne, dass der Benutzer etwas davon erfährt.

In diesem Abschnitt wird zuerst kurz die Umsetzung des Prototyps beschrieben, um danach den Funktionsumfang und seine Benutzung zu erläutern. Abschließend werden weiterführende Ideen ausgearbeitet, die mit einer Workflow-GUI umsetzbar sind, bisher aber nicht im Prototypen enthalten sind.

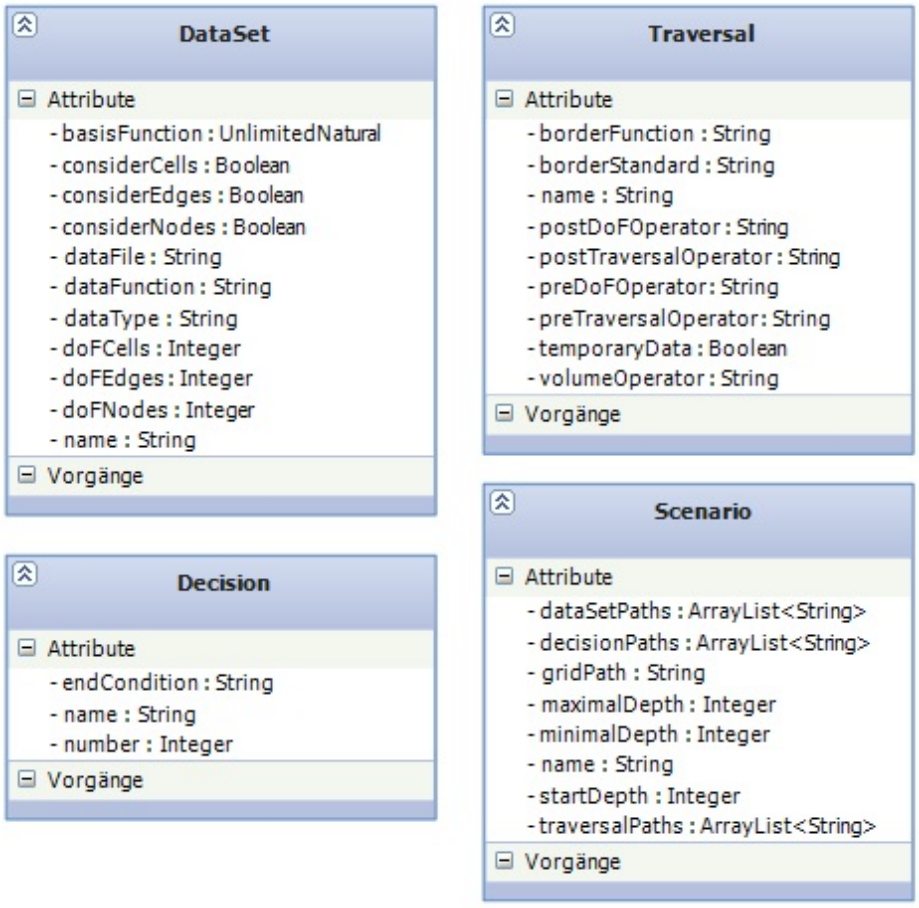

Abbildung 14: Datenmodell der Prototyps

10.1 Umsetzung des Prototyps

Der Prototyp für die Workflow-GUI wurde in der Programmiersprache Java mit Hilfe von Swing und AWT entwickelt. Dabei wurde ein kleines Datenmodell für die GUI angelegt. Es beinhaltet die Klassen `Scenario`, `DataSet`, `Traversal` und `Decision`. Dies sind die wichtigsten Bausteine des Prototyps. Das Klassendiagramm in Abbildung 14 zeigt dieses Datenmodell. Alle vorhandenen Variablen können mit Hilfe von get- und set-Methoden ausgelesen und gesetzt werden. Für die Listen in der Klasse `Scenario` existieren außerdem noch jeweils eine add- und eine remove-Methode.

Aufbauend auf diesem Datenmodell wurde die Oberfläche des Prototyps konzipiert. Dabei wurde als Grundgerüst ein Fenster und ein darin liegendes TabbedPane verwendet. Für die Repräsentation der einzelnen Datenelemente in der grafischen Oberfläche wurden einzelne Panels erstellt. Diese leiten sich jeweils von einer Basisklasse namens `MainPanel` ab. In Abbildung 15 sind die Klassen und ihre Beziehungen dargestellt.

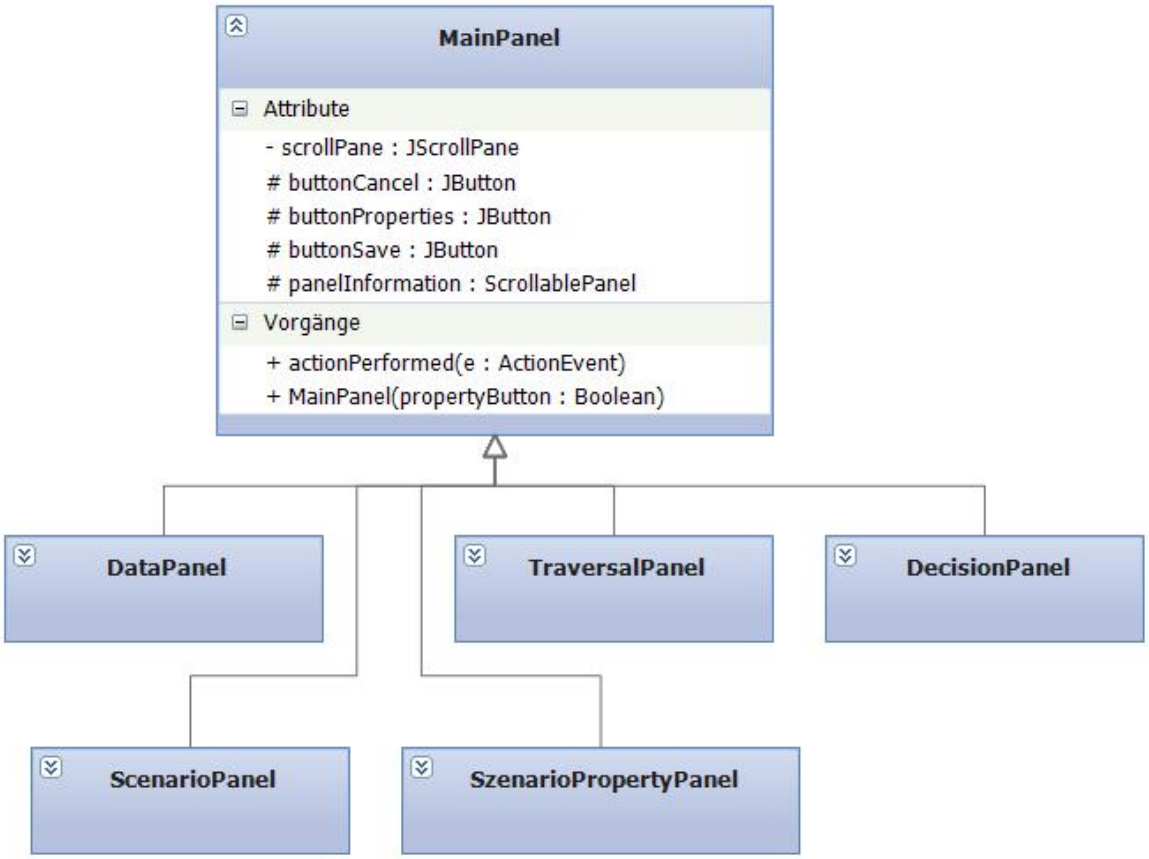

Abbildung 15: MainPanel und untergeordnete Klassen

`MainPanel` stellt hierbei ein Panel für den Inhalt, so wie die Hauptbuttons für die Aktionen Speichern und Abbrechen, so wie einen Button zum Öffnen der Einstellungen zur Verfügung. Dabei wird auch schon eine generische Aktion für das Abbrechen implementiert. Alle Datenelemente leiten sich aus `MainPanel` ab und können selbst nun ihre Funktionalität implementieren. Dabei bieten alle Elemente eine Funktion an, die ihre GUI-Komponenten initialisiert. Zusätzlich dazu kann natürlich ein existierendes Datenelement in die Oberfläche geladen werden, so dass danach die Felder die existierenden Daten anzeigen. Für das Speichern können alle Daten aus den Feldern ausgelesen werden und in ein Objekt gespeichert werden. Davor wird (falls notwendig) überprüft, ob der Benutzer alle nötigen Eingaben getätigt hat. Alle Panels implementieren auch `java.awt.event.ActionListener` um die Eingaben des Benutzers und den Klick auf die verschiedenen Buttons zu verarbeiten. Einzelne Datenelemente enthalten noch spezifische Funktionalität, auf die hier jedoch nicht weiter eingegangen wird.

Zusätzlich zu den bisher genannten Klassen wurde noch eine Toolbar und eigene Listmodels implementiert. Außerdem existiert noch die Klasse `ScenarioRunPanel`, welche ein eigenes Panel für die Ausführung der Szenarien bereitstellt. Da diese Klasse von den anderen abweicht, leitet sie sich nicht aus `MainPanel` ab. Die Gestaltung der Oberfläche wird im nächsten Kapitel auch noch einmal deutlich, da dort die Bedienung des Prototyps erläutert wird und Screenshots der Oberfläche eingefügt sind.

10.2 Bedienung des Prototyps

Startet der Benutzer den Prototyp mit einem Doppelklick auf das Programmicon, so öffnet sich ein leeres Hauptfenster des Prototyps (siehe Abbildung 16). Dieses Fenster ist der Ausgangspunkt für alle weiteren Prototypfunktionen.

Wie in Abbildung 17 gezeigt, kann der Benutzer in der Toolbar eine der drei Hauptaktionen „Bestehendes Szenario laden“, „Neues Szenario anlegen“ oder „Szenario durchführen“ auswählen.

Wählt der Benutzer nun eine der drei Tätigkeiten aus, so wird ein neues Inhaltsfenster im bisher leeren Bereich angezeigt. Dieses dient als Hauptfenster für alle weiteren Aktionen. Die Workflow-GUI baut also, wie die drei Hauptaktionen schon zeigen, auf einem Szenario-Konzept auf. Dieses wird in den folgenden Unterkapiteln näher erläutert.

10.2.1 Szenario

Die Szenarien bilden die Hauptbestandteile des Prototyps. Ein Szenario besteht dabei aus Datensätzen, Traversierungen und Entscheidungsknoten, die letztendlich zu einem ausführbaren Szenario kombiniert werden können.

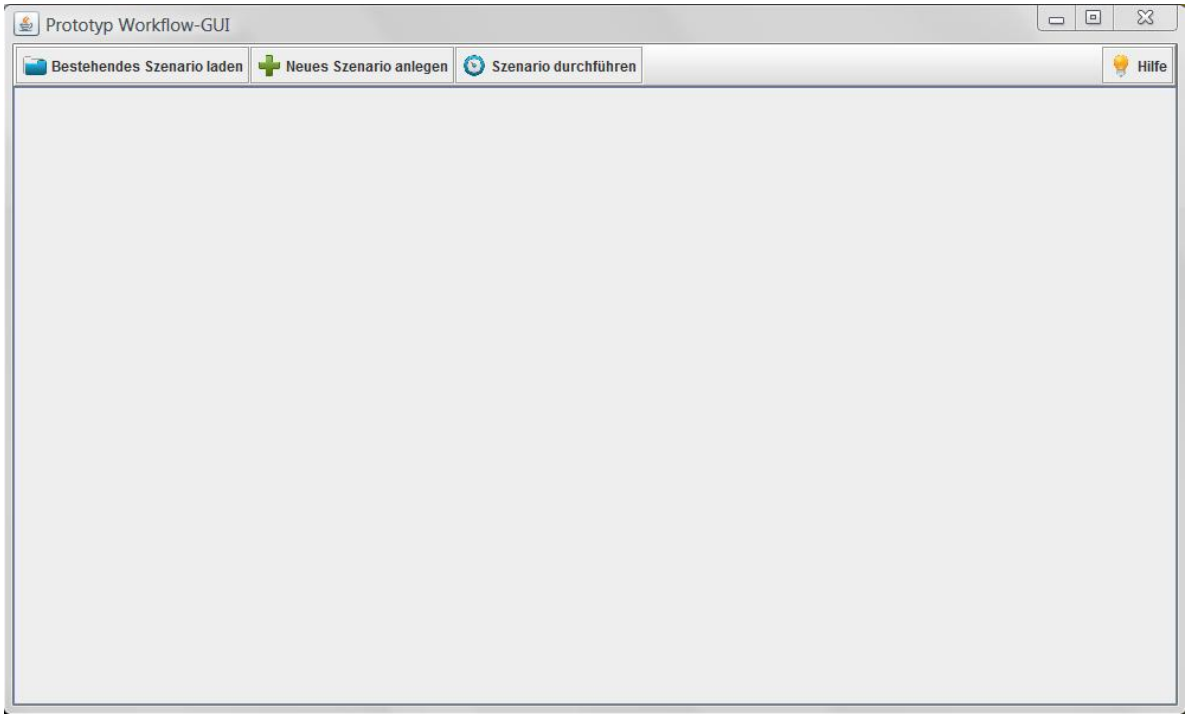

Abbildung 16: Hauptfenster der Prototyps

Abbildung 17: Toolbar des Prototyps

Szenario anlegen und laden

Möchte der Benutzer ein bestehendes Szenario laden, kann er einfach ein vorher gespeichertes Szenario mit Hilfe eines Dateiauswahldialogs auswählen. Dieses wird dann geöffnet. Eventuell vorhandene Daten werden nun durch den Prototyp geladen und angezeigt.

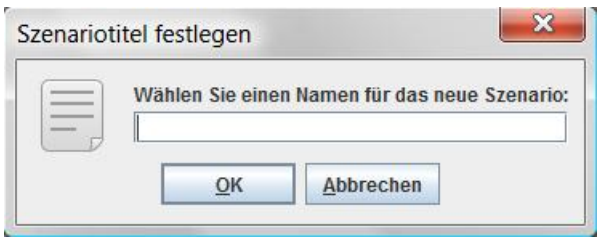

Abbildung 18: Szenariotitel festlegen

Der Benutzer kann auch ein neues Szenario anlegen. Dafür muss er, wie in Abbildung 18 dargestellt, einen Namen für das neue Szenario eingeben. In beiden Fällen stehen ihm danach alle Interaktionsmöglichkeiten zur Verfügung.

Zu einem Szenario gehört jeweils eine Liste an Traversierungen, Datensätzen und Entscheidungsknoten. Wird ein neues Szenario angelegt, sind diese Listen erst einmal leer (siehe Abbildung 19).

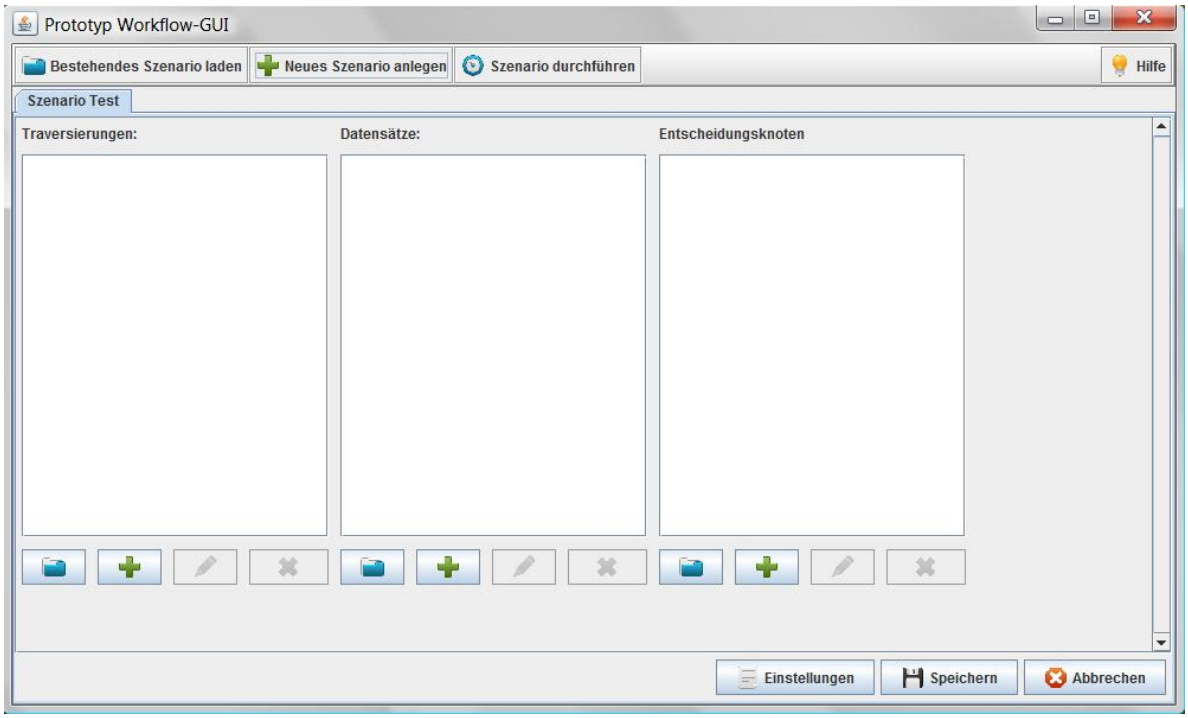

Abbildung 19: leere Szenarioübersicht

Der Benutzer kann nun Traversierungen, Datensätze und Entscheidungsknoten komplett neu anlegen oder aus dem Dateisystem laden. Dabei können diese bereits in anderen Szenarien verwendet werden. Sind bereits Einträge in den Listen vorhanden, kann er diese auch bearbeiten oder aus dem Szenario entfernen. Beim Anlegen oder Bearbeiten von Elementen öffnet sich jeweils ein neuer Tab in der Tableiste. Der Benutzer kann dann auch zwischen allen Tabs hin- und herspringen und mehrere Elemente zeitgleich geöffnet haben.

Einstellungen

Klickt der Benutzer auf den Button „Einstellungen“ können verschiedene Einstellungen zu einem Szenario ausgewählt werden (siehe Abbildung 20). Hierbei können zum Beispiel die grundlegenden Einstellungen des Gitters wie minimale und maximale Tiefe, so wie die Starttiefe angegeben werden.

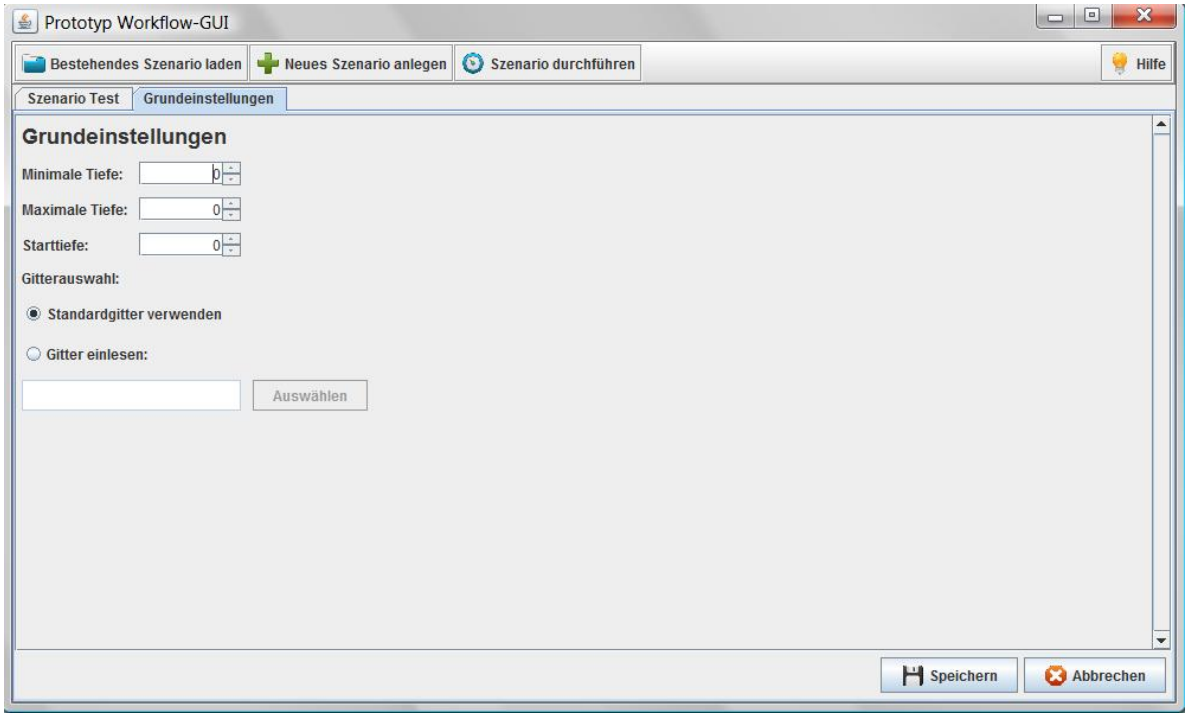

Abbildung 20: Einstellungen des Szenarios ändern

Für das Gitter wird immer vorausgesetzt, dass ein normales Standardgitter verwendet wird. Im Prototyp kann der Benutzer aber auch ein Gitter in einem bestimmten Dateiformat hochladen, so dass nicht mit einem Standardgitter gearbeitet werden muss.

Szenario speichern und schließen

Klickt der Benutzer auf den Button „Speichern“, wird das komplette Szenario mit allen hinzugefügten Traversierungen, Datensätzen und Entscheidungsknoten gespeichert. Dafür müssen jedoch alle anderen Tabs mit Traversierungen und Datensätzen geschlossen und somit gespeichert sein.

Der Benutzer kann ein Szenario auch schließen ohne zu speichern. Dafür klickt er auf den Button „Abbrechen“. Damit wird das komplette Szenario inklusive aller zugehörigen Tabs geschlossen.

Szenario durchführen

Ein Szenario kann nun auch durchgeführt werden. Dafür muss der Benutzer jedoch erst noch einige Informationen angeben. Ein Datensatz muss ausgewählt werden und die Reihenfolge der Traversierungen und Entscheidungsknoten muss festgelegt werden. Dafür

eignet sich eine Flussdiagrammdarstellung. Diese ist im Prototyp nicht implementiert, gehört jedoch zum kompletten Ablauf dazu und wird deshalb auch in diesem Kapitel erklärt und beschrieben. In der Flussdiagrammdarstellung kann der Benutzer die angelegten Bausteine (Datensätze, Traversierungen und Entscheidungsknoten) mit Drag & Drop zu einem Diagramm zusammenbauen und so ganz einfach die Ausführungsreihenfolge der Szenariobausteine bestimmen. Der Prototyp besitzt hier bisher, wie in Abbildung 21 dargestellt, nur ein Feld zum Laden eines Szenarios und einen Fortschrittsbalken, sowie einen „Start“- und „Abbrechen“-Button mit dem die Ausführung gestartet und abgebrochen werden kann.

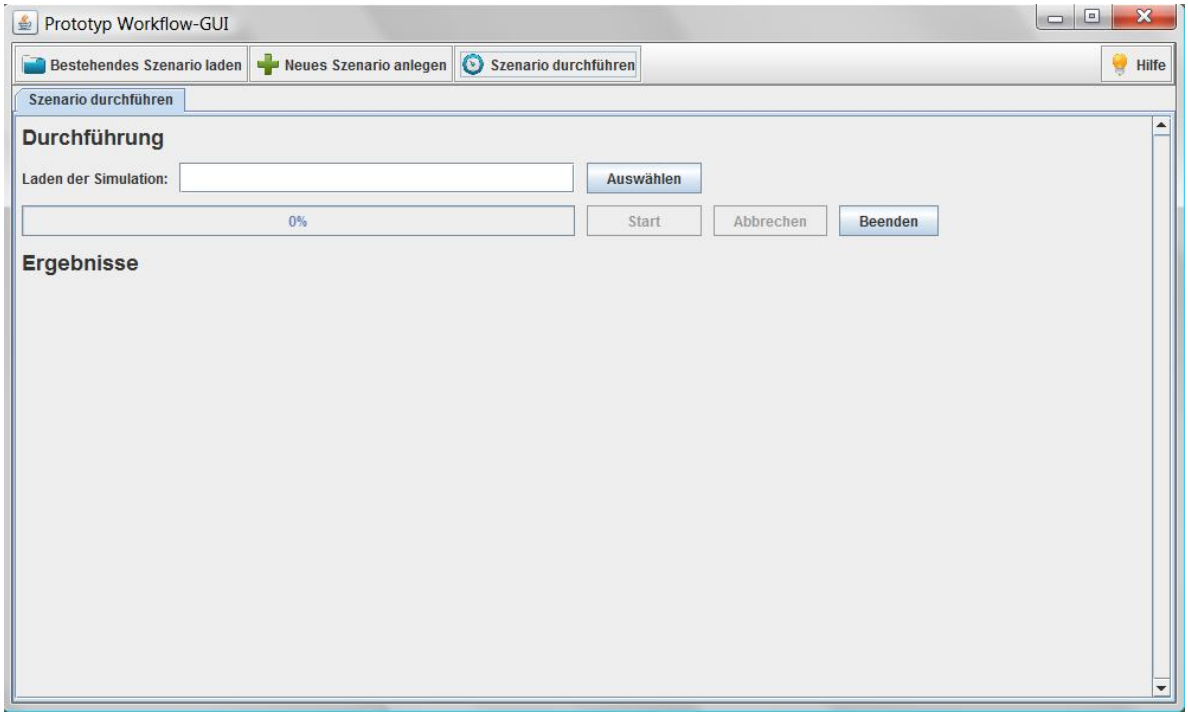

Abbildung 21: Durchführung eines Szenarios

Der Prototyp sieht außerdem noch einen Teil des Fensters für die Anzeige der Ergebnisse vor. Hier kann ein Ergebnisbild angezeigt werden. Merkt der Benutzer dadurch, dass die Ausführung nicht in seinem Sinne verläuft, kann er die Ausführung abbrechen, die Ausführungsreihenfolge korrigieren oder ändern oder das komplette Szenario anpassen und die Ausführung danach noch einmal neustarten.

10.2.2 Traversierung

Eine Traversierung kann, wie schon oben erwähnt, geladen oder neu angelegt werden. Ist sie vorhanden, kann sie auch bearbeitet oder aus dem Szenario entfernt werden. Eine Traversierung kann dabei generell zu mehreren oder auch zu keinem Szenario gehören.

Beim Anlegen und Bearbeiten einer Traversierung muss der Benutzer alle erforderlichen Informationen angeben. Das Panel zum Eingeben der Informationen für eine Traversierung ist in Abbildung 22 und Abbildung 23 dargestellt.

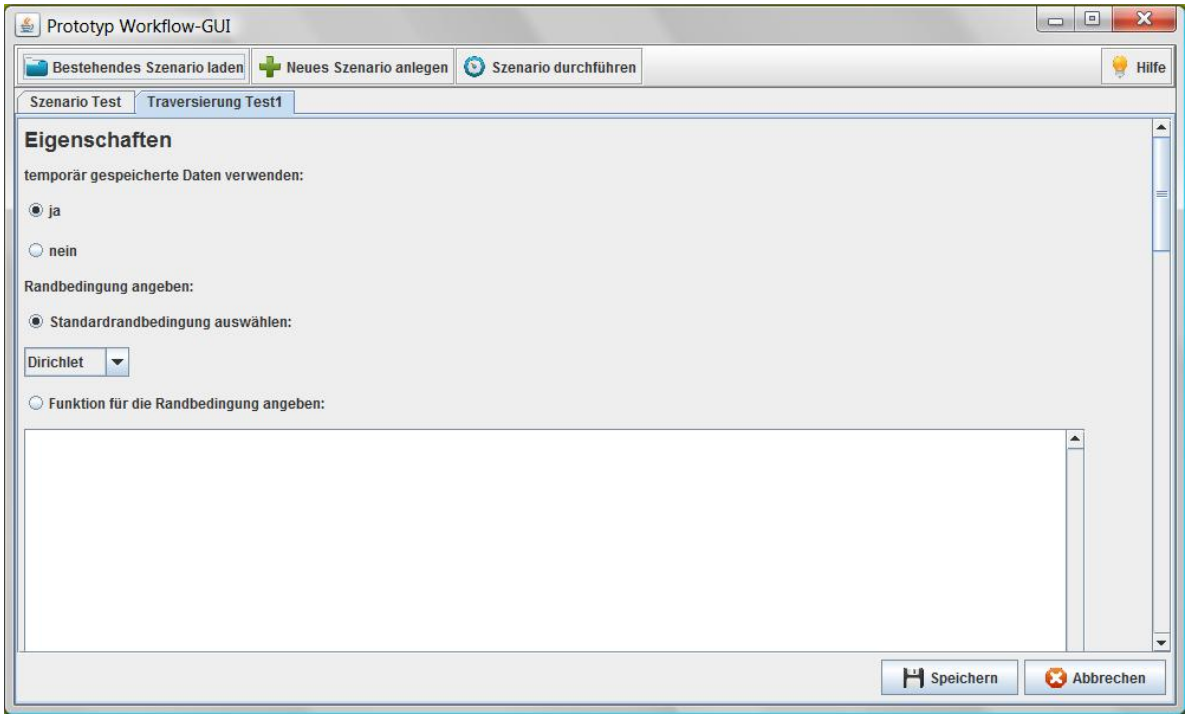

Abbildung 22: Anlegen und bearbeiten einer Traversierung

Zuerst kann der Benutzer angeben, ob Daten während der Traversierung temporär gespeichert werden sollen. Hier kann durch die Workflow-GUI bereits eine sinnvolle Default-Einstellung vorgegeben werden. In diesem Fall ist die Antwort „ja“ vorausgewählt. Als nächstes kann der Benutzer nun eine Randbedingung für die Traversierung wählen. Hierbei bietet der Prototyp in einer Kombinationsbox eine Reihe von Standardrandbedingungen an. Dies sind die Dirichlet-, Neumann-, und die periodischen Randbedingungen. Der Benutzer kann jedoch auch auswählen, dass er eine eigene Funktion für die Randbedingung angeben möchte und diese in ein Textfeld eingeben. Dabei muss der Benutzer diese Funktion selbst programmieren. Zu guter Letzt kann der Benutzer noch die Update-Funktionen für eine Traversierung angeben. Auch hierfür stehen ihm im Prototyp Textfelder zur

Verfügung. Der Benutzer kann in diesen Textfeldern Funktionen für den Volume-Operator, den Pre-DoF-Operator, den Post-DoF-Operator, den Pre-Traversal-Operator und den Post-Traversal-Operator festlegen.

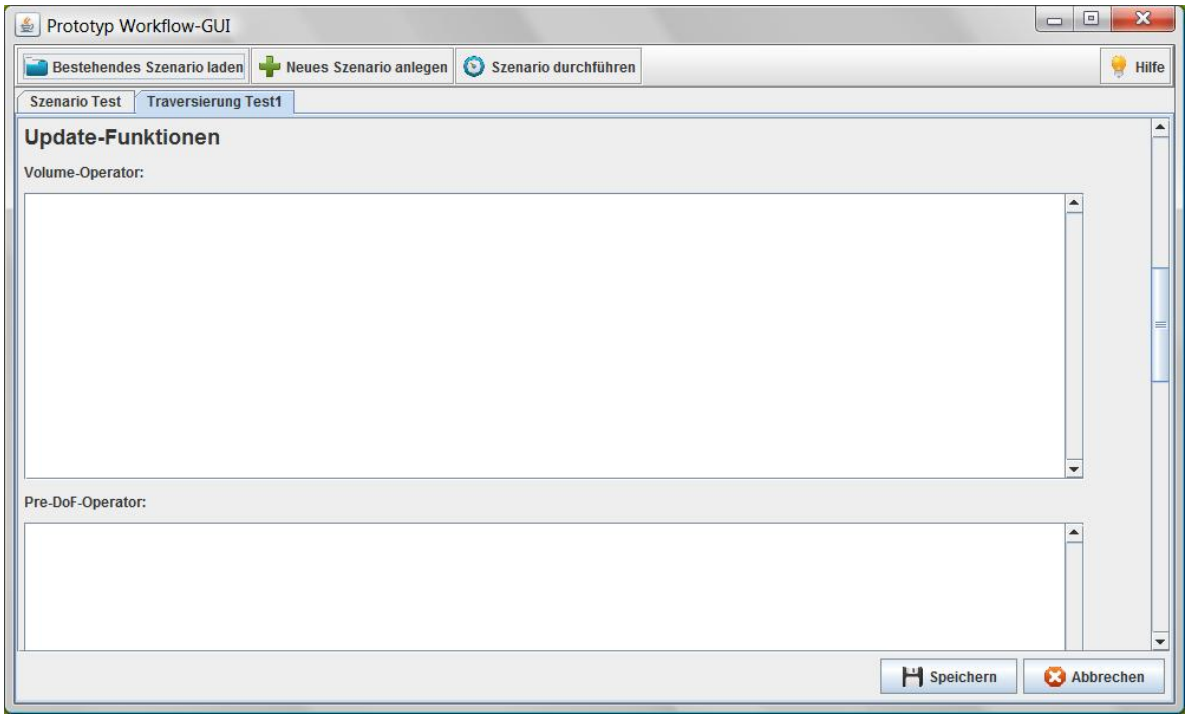

Abbildung 23: Anlegen und bearbeiten einer Traversierung

Hat der Benutzer alle Angaben gemacht, die er für seine Traversierung braucht, kann er diese speichern. Damit kann sie auch szenarioübergreifend verwendet werden. Klickt der Benutzer auf „Abbrechen“ kann er alle Änderungen oder das neue Anlegen der Traversierung verwerfen.

10.2.3 Datensatz

Ein Datensatz kann, wie schon oben erwähnt, geladen oder neu angelegt werden. Ist er vorhanden, kann er auch bearbeitet oder aus dem Szenario entfernt werden. Ein Datensatz kann zu mehreren oder auch zu keinem Szenario zugeordnet sein.

Beim Anlegen und Bearbeiten eines Datensatzes muss der Benutzer alle erforderlichen Informationen angeben. Das Panel zum Eingeben der Informationen für einen Datensatz ist in Abbildung 24 und Abbildung 25 dargestellt.

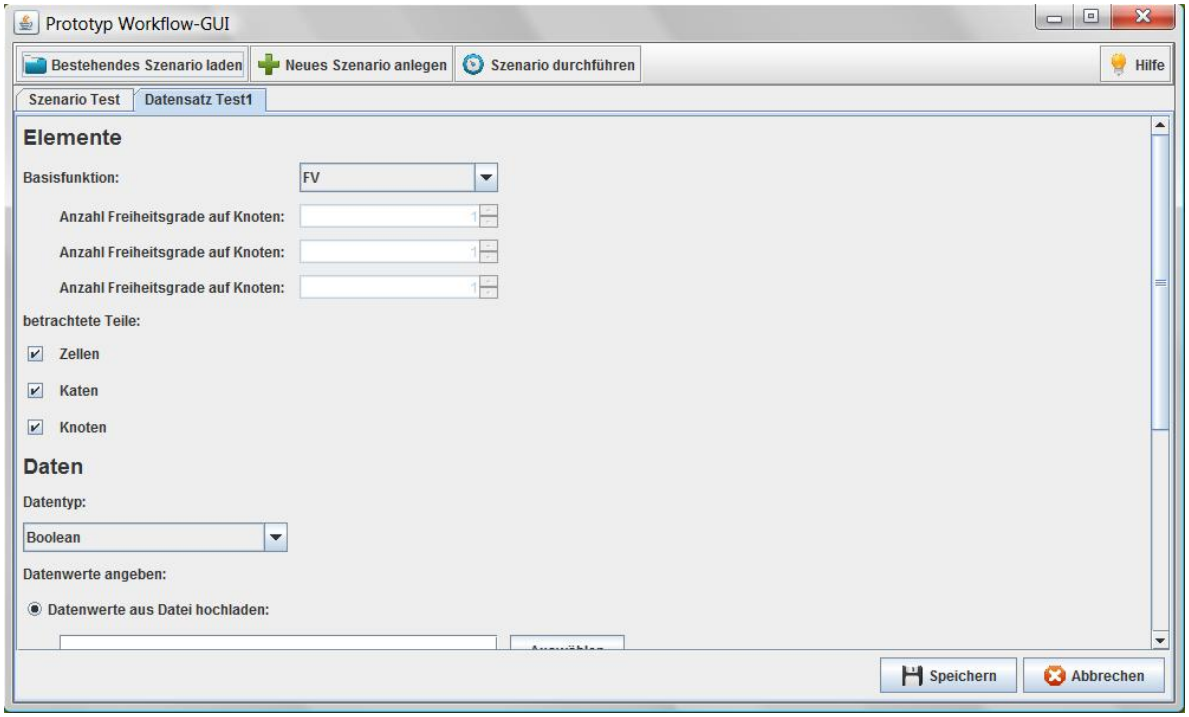

Abbildung 24: Erste Auswahl für den Datensatz

Der Benutzer kann nun zuerst eine Basisfunktion für den Datensatz wählen. Momentan hat er dabei die Auswahl zwischen „FV“, „FV dual“, „FEM Langrange-Basis“ und „FEM hierarchische Basis“. Wählt der Benutzer „Andere“, so kann er selbst die Freiheitsgrade von Knoten, Kanten und Zellen bestimmen. Zusätzlich dazu muss der Benutzer festlegen, welche Teile des Gitters überhaupt betrachtet werden sollen. Zellen, Kanten und Knoten können durch ihn deaktiviert werden. Auch hier kann die Workflow-GUI eine sinnvolle und leicht ersichtliche Default-Einstellung anzeigen. In diesem Fall sind alle Elemente zuerst einmal aktiviert. Um mit den Datenwerten sinnvoll rechnen zu können, muss der Benutzer nun noch den Datentyp für die Datenwerte festlegen und diese angeben. Hierbei bietet der Prototyp in einer Kombinationsbox momentan die Datentypen Boolean, Integer, Real und Double an. Beim Eingeben von Datenwerten kann der Benutzer entweder selbst eine Funktion programmieren, die die Datenwerte berechnet oder eine Datei mit Datenwerten einlesen.

Hat der Benutzer alle erforderlichen Angaben gemacht, kann er den Datensatz speichern, damit er auch szenarioübergreifend verwendet werden kann. Über den Button „Abbrechen“ kann er auch das Anlegen oder Bearbeiten verwerfen und somit seine Änderungen rückgängig machen.

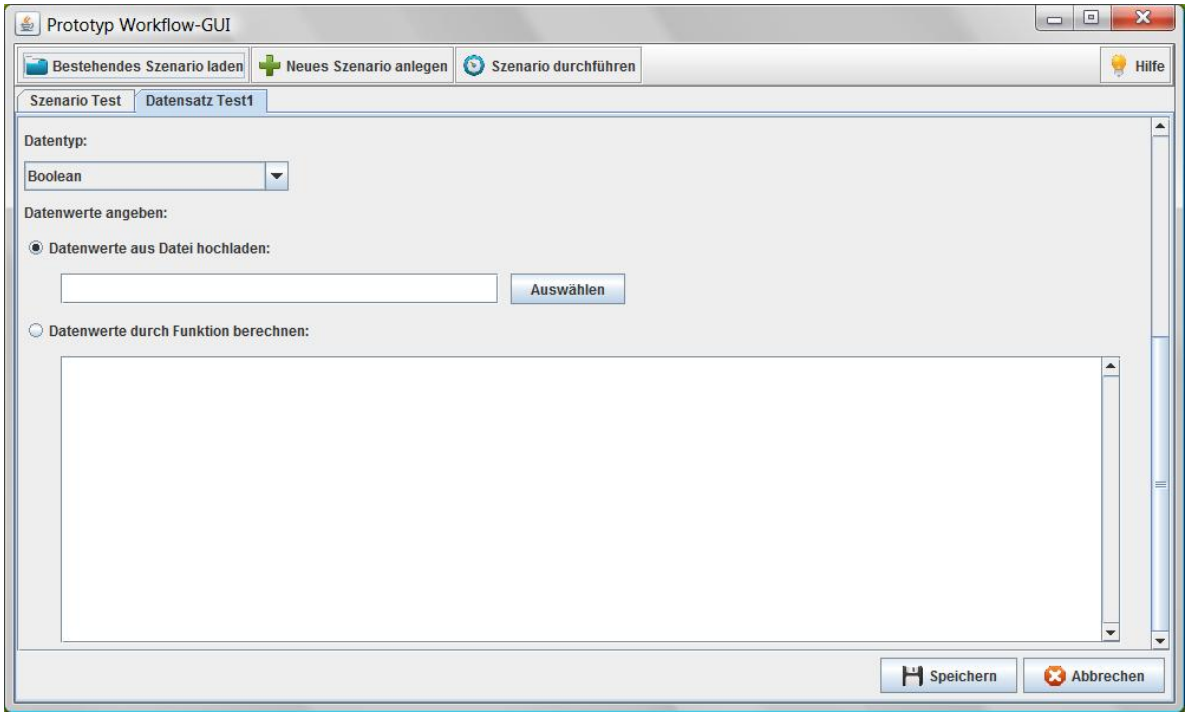

Abbildung 25: Auswahl zu den Datenwerten für einen Datensatz

10.2.4 Entscheidungsknoten

Ein Entscheidungsknoten kann, wie auch eine Traversierung und ein Datensatz, geladen oder neu angelegt werden. Ist er vorhanden, kann er auch bearbeitet oder aus dem Szenario entfernt werden. Ein Entscheidungsknoten kann zu mehreren oder auch zu keinem Szenario zugeordnet sein.

Beim Anlegen oder Bearbeiten eines Entscheidungsknoten muss der Benutzer nur eine Abbruchbedingung für den Entscheidungsknoten angeben. Dies kann zum Beispiel eine feste Zahl sein (siehe dazu Abbildung 26). Der Entscheidungsknoten funktioniert dann ähnlich einer for-Schleife und springt so lange zurück, bis die angegebene Zahl erreicht ist. Eine andere Möglichkeit ist das Angeben einer Abbruchbedingung. Hier kann der Benutzer wieder selbst eine Funktion programmieren und so den Ausgang des Entscheidungsknoten bestimmen.

Auch der Entscheidungsknoten muss nun gespeichert werden, um ihn szenarioübergreifend zu verwenden. Über „Abbrechen“ kann der Benutzer auch hier seine Änderungen verwerfen oder das neue Anlegen des Entscheidungsknoten verhindern.

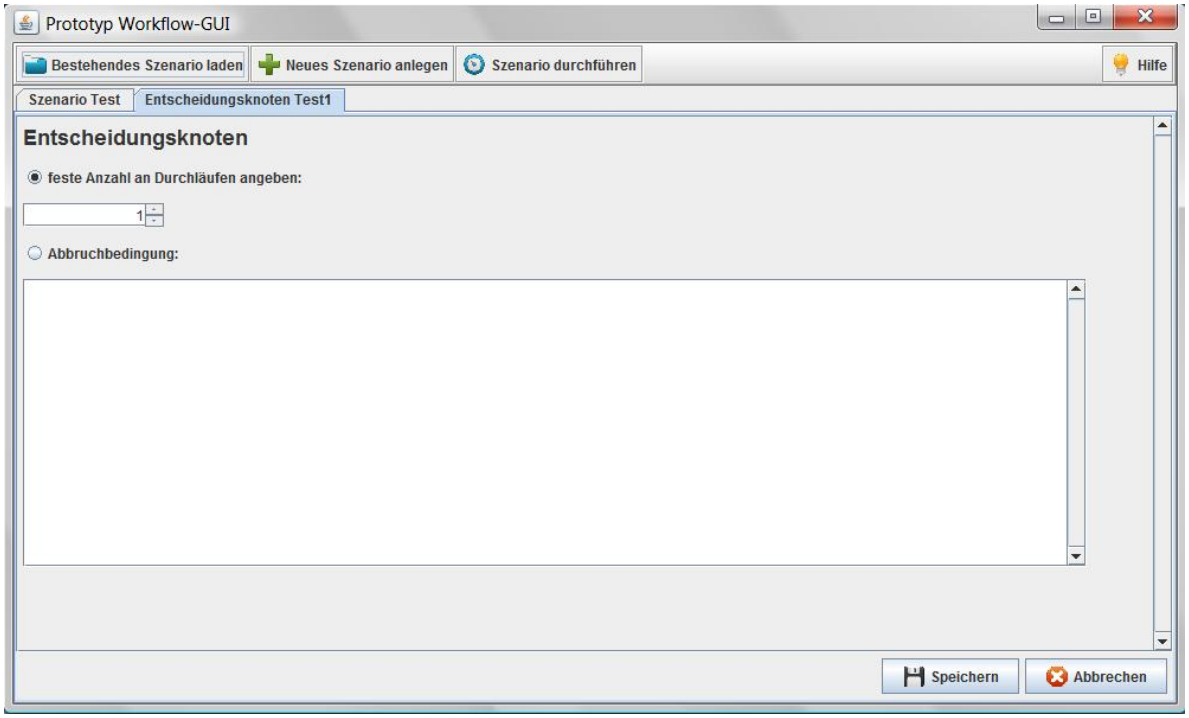

Abbildung 26: Anlegen und bearbeiten eines Entscheidungsknoten

10.3 Weiterführende Ideen

Bei der Programmierung des Prototyps sind einige Ideen und Konzepte aufgekommen, die in einer Workflow-GUI umgesetzt werden könnten. Diese wurden jedoch im Prototyp nicht realisiert. Dieser Abschnitt bietet nun einen kleinen Ausblick, wie man eine Workflow-GUI noch benutzerfreundlicher und besser gestalten könnte.

10.3.1 Workflow-GUI für Nicht-Informatiker

Aufbauend auf einer ersten Version der Workflow-GUI ließe sich einfach eine zweite neue Version erzeugen, die vor allem für Leute interessant sein könnte, die keinerlei Programmierkenntnisse besitzen. Wie bereits oben beschrieben müssen ja auch in der Workflow-GUI einige Funktionen selbst programmiert werden, da das Programm nicht in der Lage ist, alles vorzugeben. Die Durchführung von Szenarien könnte aber auch für andere Fachbereiche spannend sein.

In einer abgespeckten Version der Workflow-GUI könnte das Bearbeiten von Traversierungen, Datensätzen und Entscheidungsknoten nicht zur Verfügung stehen. Die bekannten und oft eingesetzten Traversierungen, Datensätze und Entscheidungsknoten könnten bereits angelegt

sein und immer zur Auswahl zur Verfügung stehen. Dafür könnte der Entwickler der jeweiligen Elemente eine kleine textuelle Beschreibung über die verwendeten Methoden verfassen, die die zweite Version der Workflow-GUI anzeigt. Jeder Benutzer kann nun seine eigenen Szenarien zusammenstellen und durchführen, ohne jemals irgendetwas programmieren zu müssen. Für die Durchführung genügt das mathematische Hintergrundwissen. Dabei könnten von Programmierern neu entwickelte Elemente immer auch an die anderen Personen weitergegeben werden, so dass diese eine möglichst große Auswahl an Traversierungen, Datensätzen und Entscheidungsknoten zur Verfügung haben.

10.3.2 Programmierung unterstützen

Der Prototyp bietet bisher für die Programmierung von Funktionen lediglich ein leeres Textfeld an. Dies ist nicht benutzerfreundlich. Hier sollte die Workflow-GUI Signatur und Rückgabetyp der Methoden bereits vorgeben, um eine sinnvolle Programmierung zu gewährleisten. Denkbar ist auch eine Toolbox mit verwendbaren Variablen oder Werten und ein unterstützendes Syntaxhighlighting.

Eine andere Möglichkeit zur Unterstützung der Programmierung wäre eine Funktion um Dateien einzulesen. Hier könnte die Workflow-GUI Dateien in einem bestimmten Format akzeptieren und diese als Funktionen einsetzen.

10.3.3 Funktionen als Standard festlegen

Der Prototyp bietet an mehreren Stellen Funktionen, die der Benutzer selbst programmieren kann. Hier wäre denkbar, dass diese Funktionen im Programm mit Hilfe eines Namens als Standardfunktion gespeichert werden können und dann beim nächsten Mal in einer Kombinationsbox zur Auswahl stehen. So könnten vorherige Implementierungen noch besser eingebunden werden und das Arbeiten mit dem Programm kann auch für Laien deutlich vereinfacht werden. Über einen Einstellungsdialog oder ähnliches kann der Benutzer dann die Standardeinstellungen festlegen.

10.3.4 Namen ändern

Der Prototyp sieht bisher nicht vor, dass die einmal gegebenen Namen für Szenarien, Traversierungen, Datensätze und Entscheidungsknoten noch einmal geändert werden können. Dies sollte eine Workflow-GUI jedoch unbedingt unterstützen. Über ein einfaches Textfeld oder einen editierbaren Tabheader könnte der Benutzer dann den Namen anpassen, falls er sich zum Beispiel beim ersten Eingeben vertippt hat oder sich grundlegendes an der Funktionalität eines Elementes geändert hat.

10.3.5 Bessere Fortschrittsanzeige

Die Fortschrittsanzeige im Prototyp ist bisher als Fortschrittsbalken dargestellt. Aufgrund der Entscheidungsknoten ist es jedoch sehr schwer hier eine sinnvolle und vor allem nachvollziehbare Fortschrittsanzeige zu implementieren.

Denkbar wäre die Einblendung des zusammengestellten Flussdiagramms, bei der der Benutzer farblich markiert sieht, woran das Programm gerade arbeitet. Zusätzlich dazu könnte zum Beispiel eingeblendet werden, wie oft welcher Rückgabewert bei einem Entscheidungsknoten ausgelöst wurde. Damit hätte die Fortschrittsanzeige auch noch statistischen Charakter und würde eventuell interessante Rückschlüsse über das zugrunde liegende Problem liefern.

10.3.6 Anzeige der Ergebnisse

Die Anzeige der Simulationsergebnisse ist im Prototyp nur angedeutet, jedoch nicht ausgeführt. Auch hier ergeben sich einige neue Möglichkeiten für eine Workflow-GUI. Der Benutzer könnte z.B. einstellen wie häufig oder wie oft er eine Rückmeldung über die momentane Ausführung des Szenarios erhalten möchte. Demnach könnte die GUI nach einem bestimmten Zeitintervall das angezeigte Ergebnis aktualisieren.

Des Weiteren wäre eine Play-Funktion wie bei einem Video denkbar, mit der der Benutzer zurück- oder vorspringen kann und sich die Sequenz der Bilder abspielen lassen kann. So könnte er den Verlauf der Ergebnisse gut beobachten. Natürlich ist es hier auch noch denkbar, viele weitreichende Einstellungsmöglichkeiten zum Anschauen und Untersuchen der Ergebnisse anzubieten.

Vergleich der Lösungsansätze

Bei der Beurteilung der Ansätze wird vorausgesetzt, dass der Benutzer oder Entwickler ausreichendes Hintergrundwissen über die Problematik und die notwendigen mathematischen Kenntnisse besitzt. Besitzt er dies nicht, wirkt sich das auf alle Lösungsansätze gleich aus.

Zur Bedienung des Programms zählt neben dem Ausführen des Skriptes das Definieren der Operatoren für die Simulation. Die Codierung beinhaltet das Ein-und Ausschalten von Kanten, Knoten und temporären Daten, sowie das vollständige Erstellen der Lösungsansätze, d.h. die Präprozessor-Anweisungen, das Python-Skript und die Workflow-GUI, jedoch nicht das Programmieren des bisherigen Systems, da alle Ansätze auf diesem basieren.

11 Mächtigkeit

11.1 Funktionsumfang

Präprozessor-Anweisungen und Python-Skript

Die Lösungsansätze mit Präprozessor-Anweisungen bzw. dem Python-Skript enthalten die folgenden Funktionen:

- Ein- und Ausschalten der Verwendung von Knoten, Kanten und Zellen
- Ein- und Ausschalten der Verwendung von temporären Daten
- Definieren der First-Touch-, Last-Touch- und Element-Operatoren
- Das Zusammensetzen der Traversierung aus obigen Einstellungen

Diese Funktionen werden genauer in Kapitel 8 beschrieben.

Workflow-GUI

Die Workflow-GUI unterstützt den Benutzer beim Arbeiten mit folgenden Funktionen:

- Einfaches Öffnen, Bearbeiten und Speichern von kompletten Szenarien bestehend aus Datensätzen, Traversierungen und Entscheidungsknoten
- Aufwandsfreie Mehrfachverwendung von Datensätzen, Traversierungen und Entscheidungsknoten in unterschiedlichen Szenarien

- Zusammenstellung von Szenarien durch Drag & Drop
- Durchführung von Szenarien mit Fortschritts- und Ergebnisanzeige

Der weitere Funktionsumfang der Workflow-GUI kann nur dahingehend beschrieben werden, dass die möglichen Aktionen als grafische Elemente in der Workflow-GUI vorgesehen sind. Die Workflow-GUI an sich implementiert diese Funktionen ja nicht sondern stellt nur die Oberfläche zur Auswahl zur Verfügung. Folgende Funktionen sind als grafische Elemente in der Workflow-GUI vorhanden:

- Ein- und Ausschalten der Verwendung von temporären Daten
- Auswählen verschiedener Randbedingungen
- Programmierung von Update-Funktionen
- Wählen verschiedener Basisfunktionen
- Betrachtung der Knoten, Kanten und Zellen festlegen
- Datentyp und Datenwerte angeben oder einlesen
- Abbruchbedingungen für Entscheidungsknoten festlegen
- Einstellung von Gitteroptionen

Bewertung

Im Großen und Ganzen ist die Grundfunktionalität (Ausschalten von Kanten/Knoten/Zellen und temporären Daten) der Lösungsansätze dieselbe. Die GUI hebt sich jedoch durch arbeits-vereinfachende Funktionen in diesem Punkt von den anderen Lösungsansätzen ab. Beispielsweise bietet sie eine Ergebnisanzeige, das Zusammenstellen von Szenarien mittels Drag & Drop oder einfaches Öffnen, Bearbeiten und Speichern von Szenarien.
Daher hat die Workflow-GUI im Bereich der Funktionalität deutliche Vorteile gegenüber den anderen Lösungsansätzen.

11.2 Umsetzbare Konzepte

Präprozessor-Anweisungen

Präprozessor-Anweisungen sind generell sprachenunabhängig. Die Auswahl der Verwendung von Knoten/Kanten/Zellen usw. beruht allein auf der Definition von Konstanten. Durch die beschränkte Anzahl an vorgegebenen Präprozessor-Anweisungen ist man in der Verwendung sehr eingeschränkt.

Python-Skript

Die Programmiersprache Python vereint mehrere Programmierparadigmen. Sie ist grundsätzlich imperativ, es ist aber auch möglich objektorientiert, aspektorientiert und

funktional zu programmieren. Durch die Python API kann ohne weiteres C-Code in das Skript eingebunden werden.

Python ist so konzipiert, dass ohne weiteres mit anderssprachigen Programmen interagiert werden kann, Python bietet beispielsweise die Möglichkeit diese auszuführen oder zu manipulieren.

Workflow-GUI

Die Programmiersprache für die Workflow-GUI kann im Grunde genommen beliebig gewählt werden. Deshalb stehen auch je nach Wahl der Programmiersprache andere Konzepte zur Verfügung. Dabei kann jedoch schon bei der Wahl der Programmiersprache darauf geachtet werden, dass alle benötigten Konzepte umsetzbar sind.

Bewertung

Im Bereich der umsetzbaren Konzepte sind Präprozessoren stark begrenzt, da diese lediglich Codestücke einfügen oder ersetzen können. Die Sprache Python hingegen bietet eine Vielzahl an Konzepten und ist eine vollwertige Hochsprache. Somit hat Python in diesem Punkt einen klaren Vorteil gegenüber den Präprozessor-Anweisungen.
Die Workflow-GUI ist hier schwer zu beurteilen, da diese in beliebigen Programmiersprachen erstellt werden kann, welche wiederum unterschiedliche Konzepte unterstützen.

12 Benutzerfreundlichkeit

12.1 Verständlichkeit und Einfachheit

Präprozessor-Anweisungen

Die Verwendung der Präprozessor-Anweisungen ist für jemanden mit Programmiererfahrung sehr einfach gehalten. Der Benutzer muss zunächst mittels `#define` und `#undef` die entsprechenden Konstanten definieren und den Code für die First-Touch-, Last-Touch- und Element-Operatoren in die entsprechenden Funktionsrümpfe eintragen. Anschließend kann der Präprozessor über einen einfachen Konsolenbefehl ausgeführt werden.
Für jemanden, der nicht aus dem Bereich Softwaretechnik oder Informatik stammt, ist die Bedienung dagegen ungeeignet, da sich der Benutzer zunächst damit auseinandersetzen muss, wie Präprozessor-Anweisungen funktionieren und des Weiteren Programmierkenntnisse benötigt, um die Funktionsrümpfe implementieren zu können.

Python-Skript

Die Benutzung des Python-Skriptes ist sehr einfach. Nach dem Anpassen der Konfigurationsdatei muss das Skript nur noch in der Konsole ausgeführt werden. Es ist des Weiteren auch möglich aus einem Python-Skript eine ausführbare (.exe) Datei zu erstellen, die der Benutzer nur noch ausführen muss. Benutzereingaben während der Skript-Ausführung sind nicht mehr nötig. Somit lässt sich das Skript auch ohne Weiteres in ein Makefile einbinden.

Die Konfigurationsdatei ist für einen Anwender, der sich mit Gittertraversierungen auskennt selbsterklärend. Der Aufruf des Skriptes geschieht über einen einfachen Kommandozeilenbefehl oder über eine ausführbare Datei.

Workflow-GUI

Der Umgang mit der Workflow-GUI ist unproblematisch. Durch Hinweise in Form von Labels, Tooltips oder Warnmeldungen kann der Benutzer immer gut informiert werden, wie das Programm bedient werden muss. Der Einsatz der typischen Bedienelemente mit ihren speziellen Eigenschaften gewährleisten, dass der Benutzer ohne Probleme mit diesen interagieren kann. Allerdings ist es wichtig, dass die Workflow-GUI eine gut konzipierte und übersichtliche Oberfläche hat.

Bewertung

Im Punkt Verständlichkeit und Einfachheit schneiden die Präprozessor-Anweisungen am schlechtesten ab, da diese nur für Menschen mit Programmiererfahrung einfach zu bedienen sind. Sobald am Code des Systems eine Änderung vorgenommen wird, müssen diese

unter Umständen angepasst werden. Des Weiteren muss der Code für die Operatoren von Hand verfasst werden. Dasselbe gilt auch für das Python-Skript. Dahingegen kann die Workflow-GUI so erstellt werden, dass sie auch ohne Programmiererfahrung Änderungen am Ablauf zulässt.
Somit hat die Workflow-GUI im Punkt „Verständlichkeit und Einfachheit“ einen klaren Vorteil.

12.2 Benötigte Einarbeitungszeit

12.2.1 Einarbeitungszeit in den Code

Präprozessor-Anweisungen

Die Präprozessor-Direktiven bestehen grundsätzlich aus sehr einfachen Anweisungen, die auch für einen Benutzer, der bisher nicht mit Präprozessoren zu tun hatte, einfach und schnell zu verstehen sind. Die Anweisungen orientieren sich an den aus anderen Programmiersprachen bekannten Konstrukten, wie beispielsweise einer if-Bedingung und bestehen meist nur aus einem Wort.
Da der Code der Traversierung durch die vielen Präprozessor-Anweisungen jedoch sehr unübersichtlich ist, wird die Einarbeitungszeit hier sehr erschwert. Mit zunehmender Menge der Präprozessor-Anweisungen wird die Übersichtlichkeit immer schlechter. Eventuell kann dem durch gute Kommentierung ein wenig entgegengewirkt werden.

Python-Skript

Die Einarbeitungszeit in den Code kann hier je nach Programmierstil stark variieren. Ist der Code gut kommentiert und mit sinnvollen Bezeichnern versehen, so kann die Einarbeitungszeit wie in jeder Programmiersprache sehr gering gehalten werden. Die einfache Syntax der Programmiersprache Python sorgt des Weiteren dafür, dass Python-Skripte leicht verstanden werden können. Durch die Strukturierung durch Einrückung wird außerdem für eine hohe Übersichtlichkeit gesorgt. Für eine Person mit Programmiererfahrung in einer beliebigen Programmiersprache ist die Einarbeitungszeit in Python Code gering, zumindest wenn das Skript übersichtlich gestaltet und gut kommentiert ist.

Workflow-GUI

Die Einarbeitungszeit in den Code hängt ganz von den bisherigen Erfahrungen des Entwicklers ab. Die Workflow-GUI kann im Grunde genommen in jeder beliebigen Programmiersprache implementiert werden. Besitzt der Entwickler bereits Erfahrungen mit dieser Programmiersprache und der Programmierung von GUIs, so sollte er keine Schwierigkeiten bei der Einarbeitung in den Code haben. Entscheidend ist natürlich auch, dass der

Code gut strukturiert ist und im besten Fall auch ein sinnvoller Styleguide eingehalten wurde.

Bewertung

Die Einarbeitungszeit in den Code des Python-Skriptes oder der Workflow-GUI hängt sehr stark von der Codequalität ab, d.h. Einrückung, Kommentierung, sinnvoll gewählten Bezeichnern usw. Ein Python-Skript hat den Vorteil, dass es durch Einrückung strukturiert wird, sodass der Ersteller eines Skriptes zur Einrückung gezwungen wird. Die Präprozessor-Anweisungen sind dagegen schwer nachvollziehbar. Da sich diese direkt im Code des Systems befinden wird dieser sehr unübersichtlich und die Einarbeitungszeit sehr hoch.
Somit haben Workflow-GUI und das Python-Skript in diesem Punkt Vorteile gegenüber den Präprozessor-Anweisungen.

12.2.2 Einarbeitungszeit für die Bedienung

Präprozessor-Anweisungen

Die eigentliche Bedienung besteht für den Benutzer allein darin den Code für die Operatoren einzutragen. Wenn der Benutzer bereits Programmierkenntnisse besitzt, ist die Einarbeitungszeit in die Bedienung also sehr gering gehalten.

Python-Skript

Die Bedienung des Skripts kann sehr benutzerfreundlich gestaltet werden. Der Aufruf des Skriptes sollte auch für einen unerfahrenen Benutzer keine Schwierigkeit darstellen. Die Konfigurationsdatei ist selbsterklärend, nach der Ausführung des Skriptes existieren keine Abfragen. Einzig das Definieren der Operatoren erfordert Programmiererfahrung in der jeweiligen Programmiersprache des bisherigen Systems.

Workflow-GUI

Die Einarbeitungszeit in die Bedienung der GUI ist gering, da der Benutzer die dahinter stehende Logik nicht verstehen muss. Er kann alle Einstellungen direkt in der Workflow-GUI treffen wobei das Zusammensetzen des Codes und das Einfügen des Codes komplett im Hintergrund passiert.

Bewertung

Da die Operatoren für die Berechnungen von Hand definiert werden müssen, führt dies zu einer erhöhten Einarbeitungszeit bei Bedienung der Lösungsansätze.
Im Gegensatz zu den anderen Lösungsansätzen kann die Workflow-GUI Hilfestellungen bei

der Bedienung bieten, wodurch die Einarbeitungszeit insgesamt geringer gehalten werden kann.

13 Erweiterbarkeit und Wartbarkeit

13.1 Abhängigkeiten der Lösungen

Präprozessor-Anweisungen

Der Ansatz mit Präprozessor-Anweisungen ist plattformunabhängig.
Für die Verwendung der Präprozessoren muss ein C-Präprozessor installiert sein.
Für die Codierung der Traversierung muss außerdem die Programmiersprache, in welcher sie geschrieben ist (in unserem Fall Java) installiert sein.
Des Weiteren ist ein Editor für das Konfigurieren der Traversierung von großem Vorteil.

Python-Skript

Die Programmiersprache Python ist ähnlich wie Java Betriebssystem-unabhängig, jedoch muss ein Python Interpreter für das Ausführen des Skriptes installiert sein und die Systemvariable muss richtig gesetzt werden.

Workflow-GUI

Je nach verwendeter Programmiersprache kann es zu unterschiedlichen Plattformabhängigkeiten kommen. Bei den meisten Sprachen werden jedoch die gängigen Plattformen unterstützt. Für die Ausführung der Workflow-GUI muss nur die GUI selbst vorhanden sein. Für das Implementieren ist es von Vorteil, wenn ein Editor für die entsprechende Programmiersprache zur Verfügung steht.

Bewertung

Bei Präprozessor-Anweisungen und beim Python-Skript existieren verschiedene Abhängigkeiten.
Diese sind bei der Nutzung einer Workflow-GUI meist vorinstalliert, daher hat diese in diesem Punkt leichte Vorteile.

13.2 Modularität der Lösungen

Präprozessor-Anweisungen

Die Modularität ist beim Lösungsansatz mit Präprozessoren gegeben. Über die Anweisung `#include Ḋateiname.java´` können in der Präprozessorklasse sehr einfach weitere Java-Dateien eingefügt werden. Allerdings ist es sehr umständlich die Präprozessor-Anweisungen in jede neu hinzugefügte Datei einzufügen.

Python-Skript

Sollte eine Änderung an der Hintergrundlogik erfolgen, so muss in den meisten Fällen auch das Skript angepasst werden. Der Aufwand für die Änderung am Skript ist aufgrund der geringen Skriptgröße jedoch relativ gering.

Workflow-GUI

Die Workflow-GUI und die Hintergrundlogik können bei einer GUI-Implementierung sehr gut getrennt werden. Es sollte auch darauf geachtet werden, dass hier nicht zu viele Abhängigkeiten entstehen und über definierte Interfaces kommuniziert wird. Neue Funktionen müssen dann auf beiden Seiten implementiert und am Ende verknüpft werden.

Bewertung

In diesem Punkt hat die Workflow-GUI den großen Vorteil nicht auf der Hintergrundlogik zu arbeiten. Es existiert also eine saubere Trennung von Code und GUI-Komponenten. Das Python-Skript und die Präprozessor-Anweisungen hingegen arbeiten direkt auf dem Code des Systems. Sobald sich dieser ändert, müssen auch das Skript und die Anweisungen angepasst werden. In der GUI müssen lediglich die Schnittstellen angepasst werden, sollten sich diese verändert haben.
Somit hat auch in diesem Punkt die Workflow-GUI Vorteile gegenüber den anderen Lösungsansätzen.

13.3 Vorausgesetztes Spezialwissen

Für alle Lösungsansätze wird vorausgesetzt, dass das bestehende System und die verwendete Programmiersprache bekannt sind. Falls dies nicht der Fall ist, hat dies die gleichen Auswirkungen auf die Lösungsansätze.

Präprozessor-Anweisungen

Für die Bedienung, sowie für die Codierung des Programms werden Grundkenntnisse über die Verwendung von Präprozessor-Anweisungen, sowie Kenntnisse über die verwendete Programmiersprache (in diesem Fall Java) zur Konfiguration der Funktionsrümpfe bzw. der Traversierung benötigt. Ansonsten ist keinerlei Spezialwissen nötig.

Python-Skript

Beim Benutzer wird kein Spezialwissen vorausgesetzt. Beim Programmierer wird lediglich das Wissen über die Python-Syntax und die Konzepte von Python vorausgesetzt.

Workflow-GUI

Für die Bedienung der Workflow-GUI wird keinerlei Spezialwissen vorausgesetzt.

Für die Codierung der GUI sollte Wissen über GUI-Programmierung und Entwurfsmuster im Allgemeinen vorhanden sein. Außerdem ist natürlich Wissen über die verwendete Programmiersprache hilfreich.

Bewertung

Für die Bedienung des Python-Skriptes und der Workflow-GUI wird kein Spezialwissen benötigt. Nur für die Bedienung der Präprozessor-Anweisungen sind Grundkenntnisse in diesem Bereich nötig. Bei der Codierung der Lösungsansätze ist jeweils das Wissen über die Programmiersprache vorausgesetzt. Insgesamt hat die Nutzung von Präprozessor-Anweisungen in diesem Punkt leichte Nachteile.

13.4 Lesbarkeit des Codes

Präprozessor-Anweisungen

Die Lesbarkeit wird stark dadurch eingeschränkt, dass der Code durch viele Präprozessor-Anweisungen schnell sehr unübersichtlich wird und es dadurch erschwert wird, sich im Code zurecht zu finden. Des Weiteren enthält der Code immer alle Codeteile, also auch diese, die nicht kompiliert werden. Der Benutzer muss also beim Lesen des Codes jedes Mal die `#ifdef`-Bedingungen im Code mit den `#define`-Anweisungen in der Präprozessor-Klasse vergleichen, um herauszufinden, welche Teile des Codes nachher ausgeführt werden. Die Übersichtlichkeit kann auch nur geringfügig durch entsprechende Absätze und Einrückungen der Präprozessor-Anweisungen sowie gute Dokumentation des Codes verbessert werden. So kann es sein, dass eine einfache Traversierung zunächst noch recht übersichtlich wirkt, mit zunehmender Zahl der Präprozessor-Anweisungen verschlechtert sich diese jedoch rapide.

Python-Skript

Da Python-Skripte durch Einrückung strukturiert werden, ist die Lesbarkeit von Python-Skripten im Allgemeinen sehr gut. Durch sinnvoll gewählte Bezeichner und ausreichende Kommentare kann diese des Weiteren noch weiter gesteigert werden.

Workflow-GUI

Die Lesbarkeit des Codes der Workflow-GUI sollte allgemein kein Problem darstellten. Natürlich ist die Lesbarkeit aber auch vom persönlichen Programmierstil, der Anzahl und

Formulierung der Kommentare, richtiger Einrückung und vielen anderen Faktoren abhängig.

Bewertung

Die Lesbarkeit des Codes wird durch die Präprozessor-Anweisungen stark beeinträchtigt. Somit haben das Python-Skript und die Workflow-GUI hier Vorteile. Zu beachten ist, dass die Lesbarkeit sehr stark von der Codequalität abhängt.

14 Aufwand

14.1 Übernahme von Altsystem

Präprozessor-Anweisungen

Das Altsystem kann für die Präprozessor-Anweisungen im Grunde genommen übernommen werden. Da der Präprozessor eigentlich nur ein Textersetzungsprogramm ist, kann man die Präprozessor-Anweisungen in das Altsystem übernehmen und dann den C-Präprozessor ausführen, welcher die Codestücke zu einem Programm in der Sprache des Altsystems zusammenfügt, das anschließend als solches ausgeführt werden kann. Lediglich die Präprozessorklasse muss dafür noch erstellt werden.

Python-Skript

Das Altsystem kann ohne Änderung weiterhin verwendet werden, das Python-Skript muss hierfür noch erstellt werden.

Workflow-GUI

Für die Workflow-GUI an sich, muss natürlich ein komplett neues Programm geschrieben werden. Die Funktionalität des Altsystems lässt sich jedoch sehr wahrscheinlich als Hintergrundlogik übernehmen. Hier müssen dann noch die Schnittstellen zwischen den beiden Programmteilen festgelegt werden.

Bewertung

Das Altsystem kann bei Präprozessor-Anweisungen problemlos übernommen werden. Das Python-Skript muss hingegen neu erstellt werden und die Schnittstellen zur Workflow-GUI müssen implementiert werden, dies kann je nach verwendeter Programmiersprache zu Problemen führen.
Somit haben die Präprozessor-Anweisungen in diesem Punkt klare Vorteile.

14.2 Programmieraufwand

Präprozessor-Anweisungen

Zunächst müssen Konstanten für die Verwendung von Knoten, Kanten, Zellen und temporären Daten, sowie leere Funktionsrümpfe für die First-Touch-, Last-Touch- und Element-Operatoren implementiert werden. Danach müssen die Funktionsrümpfe, sowie Konstanten in die Traversierung eingefügt werden und die Klasse mit der Traversierung muss in die Präprozessor-Klasse eingebunden werden.

Der Programmieraufwand hält sich hier in Grenzen.

Python-Skript

Der Programmieraufwand des Python-Skriptes ist gering. Das Auslesen und Filtern der Daten kann in unter 100 Zeilen Code implementiert werden. Der Gesamtaufwand für die Implementierung ist somit sehr gering, jedoch muss hier für einen ungeübten Python-Programmierer weiterer Aufwand für die Einarbeitung berücksichtigt werden.

Workflow-GUI

Der Programmieraufwand einer Workflow-GUI ist als hoch einzuschätzen. Einen funktionierenden Prototypen kann man zwar schon mit relativ wenig Aufwand erreichen, allerdings ist dieser dann natürlich nicht sehr ausgereift. Um wirklich alle Vorteile der GUI auszuschöpfen bedarf es einer sinnvoll strukturierten Oberfläche, die dann auch eine leichte Bedienung möglich macht. Auch Usability- und Design-Kriterien sollten hierbei beachtet werden. Wie bereits in Abschnitt 10.3 beschrieben, gibt es für den bereits implementierten Prototyp viele Erweiterungsmöglichkeiten, die jedoch komplizierter zu programmieren sind. Es besteht natürlich die Möglichkeit, erst einmal eine abgespeckte Version der Workflow-GUI zu programmieren und diese dann im Laufe der Zeit zu erweitern.

Bewertung

Der Aufwand ist beim Präprozessor und beim Python-Skript eher gering, bei der Workflow-GUI muss hingegen ein sehr großer Aufwand eingeplant werden, da die GUI-Erstellung und die Schnittstellen zur Hintergrundlogik sehr zeitaufwändig sind.

15 Überblick über alle Kriterien

In der folgenden Tabelle wird noch einmal ein Überblick über die in den vorherigen Abschnitten getroffen Bewertungen gegeben. Dabei wurden die Lösungsansätze bei jedem Kriterium auf einer Skala mit den Werten -, o und + eingeordnet.

Kriterium		**Präprozessor-Anweisungen**	**Python-Skript**	**Workflow-GUI**
Mächtigkeit	Funktionsumfang	o	o	+
	Umsetzbare Konzepte	-	+	o
	gesamt	-	+	+
Benutzerfreundlichkeit	Verständlichkeit und Einfachheit	o	o	+
	Einarbeitungszeit in den Code	-	o	o
	Einarbeitungszeit für die Bedienung	o	o	o
	gesamt	o	o	+
Erweiterbarkeit und Wartbarkeit	Abhängigkeiten der Lösung	o	o	+
	Modularität der Lösungen	-	-	+
	Vorausgesetztes Spezialwissen	-	o	o
	Lesbarkeit des Codes	-	o	o
	gesamt	-	o	+
Aufwand	Übernahme von Altsystem	+	o	-
	Programmieraufwand	o	+	-
	gesamt	o	o	-

Fazit und Empfehlung

In diesem Kapitel soll nun ein Fazit gezogen werden und eine Empfehlung für einen Lösungsansatz abgegeben werden. Nach dem Sammeln aller Informationen und der eigenen Auseinandersetzung mit den Lösungsansätzen kann auf dieser Basis entschieden werden, welcher der Lösungsansätze am besten geeignet ist.

Betrachtet man den Überblick der Lösungsansätze in Kapitel 15 fällt auf, dass der Präprozessor die schlechtesten Bewertungen bekommen hat. Dies liegt vor allem daran, dass der Präprozessor direkt auf dem Code arbeitet, was diesen sehr unübersichtlich und unhandlich macht. Des Weiteren ist für die Bedienung das Wissen über Präprozessoren zwingend erforderlich, welches nicht grundlegend bei einem Benutzer vorhanden oder zu erwarten ist.

Das Python-Skript schneidet bei der Bewertung insgesamt etwas besser ab, als die Präprozessor-Anweisungen und kann auch einige positive Punkte verbuchen. Der Vorteil des Python-Skriptes ist definitiv der geringe Programmieraufwand und die Integration in ein Makefile. Da Python eine Hochsprache ist, können vielfältige Konzepte eingesetzt werden und Programmierkenntnisse in anderen Sprachen können darauf angewendet werden.

Die Workflow-GUI schneidet aufgrund ihrer Bedienungsfreundlichkeit und großen Erweiterungsmöglichkeiten im Vergleich am besten ab. Das einzige Problem ist hier der große Programmieraufwand. Dieser lässt sich bei der Erstellung einer gut bedienbaren GUI nicht vermeiden. Trotz des hohen Aufwandes, denken wir, dass sich dieser lohnen würde, da die GUI schon mit wenig Aufwand auf ein ähnliches Level wie die anderen Ansätze gebracht werden kann. Der Mehraufwand ergibt sich somit nur für die vielen weiteren Vorteile. Wichtig hierbei ist die Definition sauberer Schnittstellen zwischen dem bestehenden System und der Workflow-GUI. Die Programmierung der Workflow-GUI kann sich auch an dem bereits bestehenden Prototyp orientieren, dabei wurden in kürzester Zeit die Grundfunktionalitäten als grafische Elemente angeboten. Die Weiterentwicklung der GUI kann dann auch über längere Zeit erfolgen, so dass sich der Aufwand gut über die Zeit verteilen lässt. Der größte Vorteil ist aus unserer Sicht die Tatsache, dass die Workflow-GUI auch dahingehend erweitert werden kann, dass sie komplett von Nicht-Informatikern bedient werden kann, ohne dass die Interaktionsmöglichkeiten zu weit eingeschränkt werden. Dies ist mit den anderen Konzepten gar nicht oder nur sehr beschränkt realisierbar.

Anhang

16 Präprozessorklasse

```
public class TraversierungPP {


#define NODE
#define EDGE
#define TEMP


#define ELEMENT_OP ElementUpdate


#define FT_CELL FirstCellUpdate
#define FT_NODE FirstNodeUpdate
#define FT_EDGE FirstEdgeUpdate


#define LT_CELL LastCellUpdate
#define LT_NODE LastNodeUpdate
#define LT_EDGE LastEdgeUpdate


#define FT_TEMP_CELL FirstTempCellUpdate
#define FT_TEMP_NODE FirstTempNodeUpdate
#define FT_TEMP_EDGE FirstTempEdgeUpdate


#define LT_TEMP_CELL LastTempCellUpdate
#define LT_TEMP_EDGE LastTempEdgeUpdate
#define LT_TEMP_NODE LastTempNodeUpdate


public void ElementUpdate(Element element){
}


#ifdef NODE
public void FirstNodeUpdate(Node node){
}
```

```
public void LastNodeUpdate(Node node){
}

#ifdef TEMP
public void FirstTempNodeUpdate(Node node, TempNode tempNode){
}

public void LastTempNodeUpdate(Node node, TempNode tempNode){
}
#endif
#endif

#ifdef EDGE
public void FirstEdgeUpdate(Edge edge){
}
public void LastEdgeUpdate(Edge edge){
}

#ifdef TEMP
public void FirstTempEdgeUpdate(Edge edge, TempEdge tempEdge){
}
public void LastTempEdgeUpdate(Edge edge, TempEdge tempEdge){
}
#endif
#endif

public void FirstCellUpdate(Cell cell){
}
public void LastCellUpdate(Cell cell){
}

#ifdef TEMP
public void FirstTempCellUpdate(Cell cell, TempCell tempCell){
}
public void LastTempCellUpdate(Cell cell, TempCell tempCell){
}
#endif

#include 'TraverseGrid_Jacobi.java'
}
```

Literaturverzeichnis

[MSD] MSDN. Preprocessor Directives. URL http://msdn.microsoft.com/en-us/library/3sxhs2ty%28v=vs.80%29.aspx. (Zitiert auf Seite 33)

[Pyt] Python. Python Summary. URL http://www.python.org/doc/essays/blurb/. (Zitiert auf Seite 39)

[Scho6] S. Schraufstetter. *Speichereffiziente Algorithmen zum Lösen partieller Differentialgleichungen auf adaptiven Dreiecksgittern*. Diplomarbeit, Technische Universität München, 2006. (Zitiert auf Seite 11)

Alle URLs wurden zuletzt am 27. Oktober 2011geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

_____

(Miriam Greis, Jessica Hackländer, Pascal Hirmer)