Fachbereich Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Prof. Dr. Walter F. Tichy

Bachelorarbeit

Implementierung des SURE-Verfahrens zur automatischen Parallelisierung verschachtelter Schleifen in R-Programmen

Eingereicht von:
André Wengert

Betreut von:
Dr. Frank Padberg

4. September 2013

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Karlsruhe, den _____ _____

(Datum/*Date*) (Unterschrift/*Signature*)

Inhaltsverzeichnis

1. Einleitung

R ist eine Programmiersprache bzw. -umgebung für statistische Auswertungen und Grafiken [Tea]. Sie wird vor allem im Fachgebiet der Bioinformatik zur Berechnung und Analyse von genomischen Sequenzen [Kri09] eingesetzt. Die sequenzielle Abarbeitung dieser Aufgaben ist sehr zeitintensiv und R bietet keine native Unterstützung für Parallelisierung um den Prozess zu beschleunigen.
Zweck der R-Experimentierplattform ALCHEMY [Mir11] ist die automatische Parallelisierung von R-Programmen. Den Ansatz, den ALCHEMY für die Verarbeitung und Ausführung von R-Programmen verfolgt, wird in Abbildung 1 dargestellt:

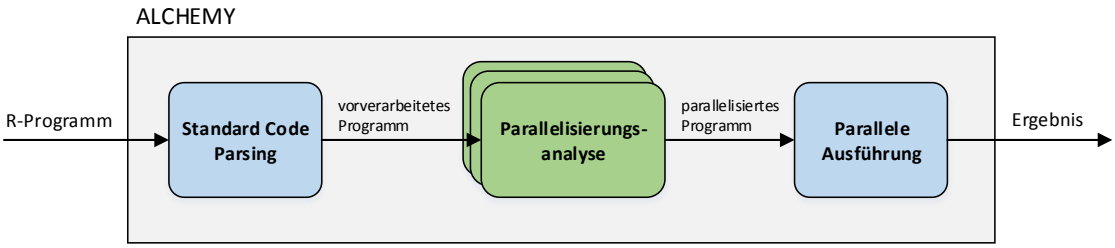


Abbildung 1: Arbeitsablauf der ALCHEMY-Plattform nach [Mir11]

Dieser Ansatz kommt vor allem Benutzern zu gute, die Experten ihrer Domäne sind, jedoch wenig Kenntnisse über Programmierung verfügen [PM12]. Für die Parallelisierungsanalyse wurden bereits einige Module, sog. PAMs, entwickelt.
Ziel dieser Arbeit ist die Implementierung des SURE-Verfahrens als PAM zur automatischen Parallelisierung verschachtelter Schleifen mit Matrixzugriffen. Ursprünglich definiert wurde dieses Verfahren durch Karp, Miller und Winograd in [KMW67]. Im Kontext der Schleifenparallelisierung wurde es von Allain Darte in [Dar98] vorgestellt. Wie Abbildung 2 zu entnehmen ist, spaltet sich das Verfahren in drei abgeschlossene Teilprozesse für eine „Pipeline-artige“ Abarbeitung auf.

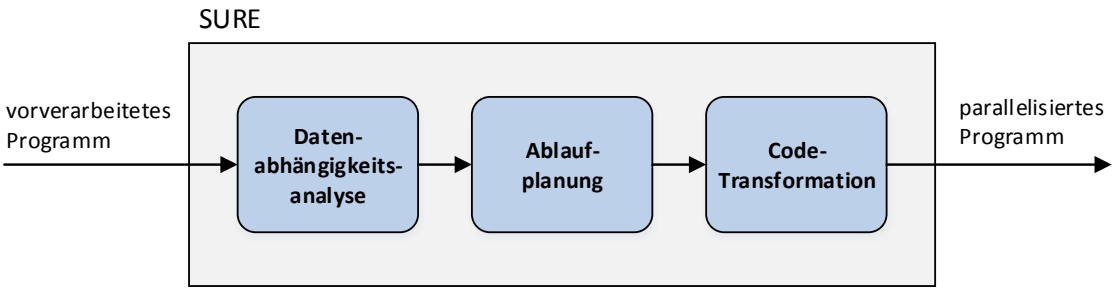


Abbildung 2: Ansatz des SURE-Verfahrens

Vorbereitend für die Analyse wird das R-Programm nach verschachtelten Schleifen mit Matrixzugriffen (SUREs) durchsucht. Im ersten Teil des Verfahrens wird der Schleifenrumpf des identifizierten Code-Stücks auf Datenabhängigkeiten untersucht. Die Ermittlung präziser Abhängigkeitsinformationen stellt das zentrale Problem der Parallelisierungsanalyse dar. Unter Berücksichtigung der gewonnenen Abhängigkeitsinformationen ist es nun möglich, Entscheidungen über eine parallele Ausführung der Schleifen zu treffen bzw. die Reihenfolge der Schleifeniterationen entsprechend umzuordnen. Abschließend wird aufgrund der gewonnenen Erkenntnisse das sequenzielle Code-Stück in ein paralleles transformiert. Das parallelisierte R-Programm ist nun durch ein in ALCHEMY eingebundenes Backend ausführbar. Es existieren bereits mehrere Algorithmen für die Parallelisierung von Schleifen, wie sie z.B. in [AK87], [WL91] oder [DV96b] beschrieben sind, die diesen Ansatz verfolgen. Es ist jedoch zu bemerken, dass sie über unterschiedliche Mächtigkeiten bzgl. der Erkennung von Parallelisierbarkeit verfügen. Für diese Arbeit wird der Algorithmus von Allen und Kennedy aus [AK87] implementiert.
Schließlich soll die Leistung des PAMs und des erzeugten, parallelisierten Codes in direktem Vergleich zu anderen, für die ALCHEMY-Plattform entwickelten PAMs evaluiert werden.

In dieser Arbeit wurden die folgenden Ergebnisse erzielt:

- *Es wurde ein ALCHEMY-Transmutator zur Parallelisierung verschachtelter Schleifen erzeugt.* Das `SURE`-Modul enthält alle durch das SURE-Verfahren definierten Komponenten: Mustererkennung, Datenabhängigkeitsanalyse, Ablaufplanung und Code-Transformation. `SURE` setzt sich aus 48 Java-Klassen mit ungefähr 4.800 Zeilen Programm-Code zusammen.

- *Das SURE-Verfahren wurde auf Basis des Algorithmus von Allen und Kennedy analysiert.* Die Stärken, als auch die Schwächen des Algorithmus wurden aufgezeigt und anhand von mehreren Testfällen belegt.

- *Die Leistung des parallelisierten Programm-Codes wurde evaluiert.* Gegenüber einer sequentiellen Ausführung wurden bereits für relativ kleine Eingabegrößen gute Beschleunigungen erreicht. Es wurden reelle Beispiele (z.B. Matrixmultiplikation), als auch künstliche verwendet. Die Leistungsermittlung auf einem Rechner mit 8 Kernen zeigte Beschleunigungen von bis zu 5,1 für 500M Matrixelemente. Tendenziell steigt dieser Wert für größere Eingaben.

- *Es wurden Optimierungsvorschläge für die Parallelisierung präsentiert und evaluiert.* Es stellte sich heraus, dass ein hoher Grad an Parallelität nicht automatisch zu einer höheren Beschleunigung führte. Die Optimierungen

wiesen für die getesteten Beispiele einen weiteren Leistungszuwachs von 26-80% im Vergleich zu dem regulären parallelen Programm auf. Ebenfalls wurde der *Break-Even-Point* für geringere Eingabegrößen erreicht.

2. Verwandte Arbeiten

2.1. R-Experimentierplattform ALCHEMY

ALCHEMY [Mir11], [PM12] ist eine Experimentierplattform für die Durchführung von Parallelisierungsanalysen und parallele Ausführung von R-Programmen. Eine bestmögliche Parallelisierung hängt dabei immer von der Eingabe ab. Deshalb unterstützt ALCHEMY die Anwendung und Kombinationen mehrerer Analysemethoden. Die Durchführung dieser Analysen übernehmen sog. *Parallel Analysis Modules* (PAMs). Anschließend wird die Eingabe in ein paralleles Programm transformiert und von einem wählbaren Backend ausgeführt. Die modulare Architektur gewährleistet das Experimentieren mit unterschiedlichen Konfigurationen für PAMs und Backends. Verdeutlicht wird dies in Abbildung 3.

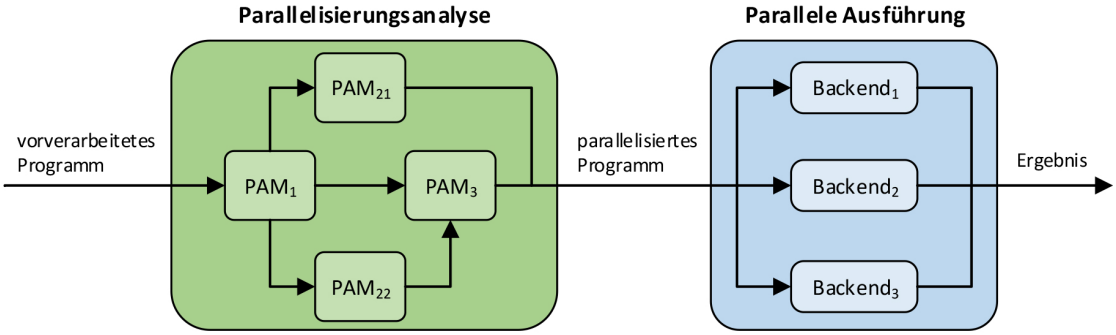


Abbildung 3: Vereinfachte Beispielkonfiguration für ALCHEMY

Ferner ermöglicht die Unterstützung durch ALCHEMY den Benutzern von R eine unkomplizierte Methode zur beschleunigten Ausführung derer Programme.
Dass von einem Benutzer keinerlei Anpassung des R-Programms erforderlich ist, soll ein typischer Parallelisierungsprozess in ALCHEMY verdeutlichen:

1. Der R-Interpreter aus ALCHEMY erhält über die interaktive Konsole oder aus einer Programmdatei ein R-Programm durch den Benutzer
2. Der R/ALCHEMY-Adapter fängt dieses ab und übersetzt es in die Zwischensprache AIR (siehe Kapitel 2.2)
3. Je nach Konfiguration wird die AIR nun durch eine Folge unterschiedlicher PAMs analysiert und transformiert. Teil dieser Konfiguration kann das zu entwickelnde `SURE`-Modul sein.
4. Das Ergebnis ist eine parallelisierte Version der AIR unter Verwendung von parallelen Skeletons (siehe Kapitel 2.4)

5. Ein Backend übernimmt dessen parallele Ausführung und präsentiert das Ergebnis dem Benutzer

2.2. Analysis Intermediate Representation (AIR)

Die *Analysis Intermediate Representation* (AIR) wurde in [PM12] speziell für die ALCHEMY-Plattform eingeführt und dient als Kommunikationssprache zwischen deren Modulen. AIR ist ein *Abstract Syntax Tree* (AST) und lässt sich als XML-Code veranschaulichen. Die Baumstruktur, welche in Abbildung 4 vereinfacht als Beispiel von Listing 1 präsentiert wird, soll dies verdeutlichen. Die entsprechende, vollständige AIR ist Anhang A zu entnehmen.

```
1 # ... Initialisiere
2 for (i in n) {
3   a[i,1] <- 0;
4 }
```

Listing 1: Einfaches Beispiel eines R-Programms

Im Gegensatz zu R's interner Repräsentation SEXPR [Tea] stellt sie eine Zwischensprache auf einer höheren Ebene dar und beinhaltet zudem parallele Skeletons als Ausdruck für Parallelismus in Programmen. Die Definition von AIR wurde durch die parallele Zwischensprache VCODE [BC90] inspiriert. Diese wurde für eine effiziente Darstellung paralleler Probleme im Zusammenhang mit Vektoroperationen entworfen. Die Ein- und Ausgabe eines jeden PAMs in ALCHEMY, also auch für das zu entwickelnde `SURE`-Modul, ist in AIR.

2.3. Parsergenerator ANTLR

Another Tool for Language Recognition (ANTLR), das sich mittlerweile in Version 4 [Par13] befindet, ist ein mächtiges, objektorientiertes Werkzeug für die Erzeugung von Parsern, Lexern und Tree-Parsern für LL(*)-Grammatiken. Es wird seit 1989 von Terence Parr an der Universität von San Francisco entwickelt und unterstützt u.a. Zielsprachen wie ActionScript, C++, C#, Java, JavaScript, Objective-C und sogar R, von denen sich einige jedoch noch in der Entwicklung befinden [Par12]. Ausgehend von der Beschreibung einer formalen Sprache, einer sog. "Grammatik", ist ANTLR in der Lage einen Parser zu generieren. Beim Parsen des Programm-Codes kann entsprechend der Grammatik automatisch ein Parsebaum erzeugt werden. Er dient als interne Repräsentation der Eingabe und zur weiteren Analyse und Verarbeitung durch den Rest der Anwendung. Abbildung 5 soll diesen Datenfluss in ANTLR schematisch darstellen.

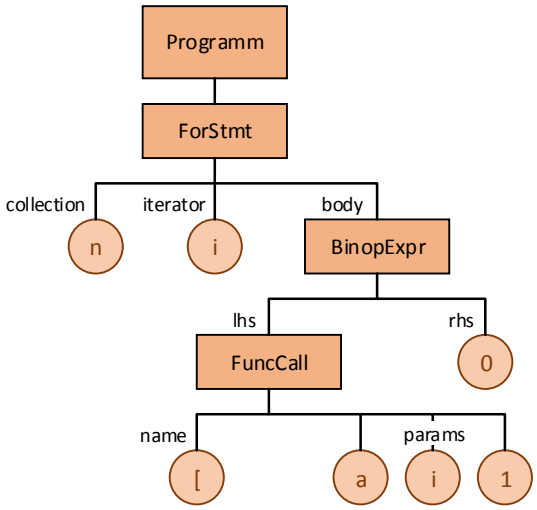


Abbildung 4: Baumdarstellung der AIR von Listing 1

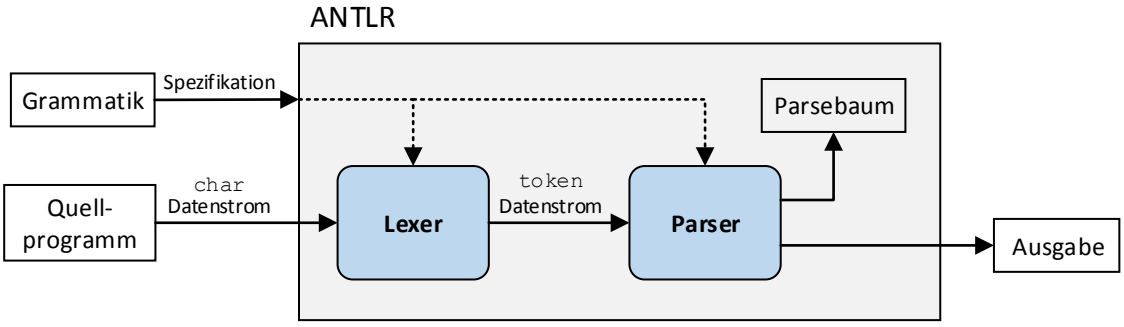


Abbildung 5: Schematischer Datenfluss in ANTLR

Die Arbeitsweise des Lexers und Parsers werden durch eine gemeinsame Grammatik-Spezifikation definiert. Der Lexer liest den `char`-Datenstrom des Programm-Codes und übersetzt diesen in einen `token`-Datenstrom. Der Parser erzeugt daraus für gewöhnlich einen Parsebaum, welcher nun von der Anwendung interpretiert werden kann.
Für die Implementierung des SURE-Verfahrens bietet sich die Verwendung von ANTLR als vorbehandelnder Schritt zur Erkennung von SUREs in R-Programmen, präziser in AIR-Programmen an.

2.4. Parallele Skeletons

Algorithmische oder parallele *Skeletons* (deu. Skelette), wie sie unter anderem von Murray Cole in [Col91] definiert werden, sind Muster für die parallele Program-

mierung. Ähnlich wie die wohlbekannten "Entwurfsmuster" in Bezug auf Software-Entwicklung repräsentieren sie Lösungsvorlagen für allgemeine Problemstellungen der parallelen Programmierung.
Für die Parallelisierung von (verschachtelten) Schleifen wird das Skeleton `DOPAR` verwendet. `DOPAR`, wie in Abbildung 6 veranschaulicht, ordnet eine parallele Ausführung aller Schleifeniterationen an:

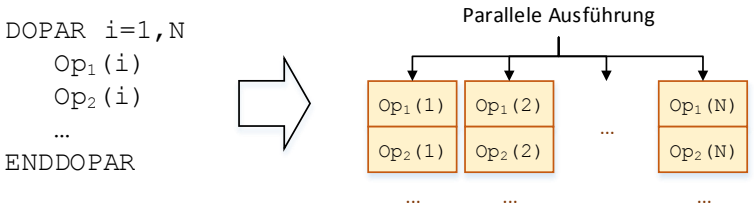


Abbildung 6: DOPAR Skeleton

Eine Liste samt verständlicher Erläuterung weiterer algorithmischer Skeletons, wie z. B. `MAP`, `SCAN` oder `ZIPW` kann [Col] entnommen werden.

2.5. Backends zur Ausführung von parallelisiertem R-Code

Die parallele Ausführung von R-Code ist zu diesem Zeitpunkt lediglich mittels spezieller Bibliotheken möglich. Im Gegensatz zur Zielvorstellung der automatischen Parallelisierung sind R-Benutzer jedoch dazu gezwungen, eigenhändig parallele Befehle explizit in ihren Programmen zu deklarieren.
Zu den gängigsten unter diesen zählen die Implementierungen des Message Passing Interface (MPI), *Rmpi* (siehe [Yu02]) und der Parallel Virtual Machine (PVM), *RPVM* (siehe [NLR01]). Eine weitere beliebte Bibliothek stellt *R multicore* [Urb] dar, welche die Funktion `mclapply` als parallele Version von R's `lapply` anbietet. Dieser Ansatz fand auch ursprünglich Anwendung in einem ersten R/ALCHEMY-Backend. Eine Lösung für die parallele Ausführung von Schleifen bietet die Bibliothek *doMC*. Diese stellt Mechanismen zu parallelen Ausführung von `foreach`-Schleifen bereit und baut auf *R multicore* auf.
Hinsichtlich automatischer Parallelisierung ist ausschließlich die Bibliothek *pR* [MLS07] bekannt, welche eigenständig Datenabhängigkeiten analysiert und entsprechend Operationen mittels MPI an verschiedene Rechner im Netzwerk auslagert.
Zweckbestimmt für die ALCHEMY-Plattform befindet sich derzeit ein Backend [Kie13] in Entwicklung, welches die Bibliothek *Array Building Blocks* (ArBB) von Intel verwendet. Es unterstützt die von den PAMs verwendeten parallelen Skeletons `MAP`, `SCAN` und `ZIPW` und wird für die Kompatibilität mit dem `SURE`-

Modul um `DOPAR` erweitert. Die parallele Ausführung wird von der ArBB eigenen virtuellen Maschine automatisch für unterschiedliche Ziel-Hardware gehandhabt. Das grundlegende Vorgehen (siehe Abbildung 7) des Backends lässt sich wie folgt beschreiben:

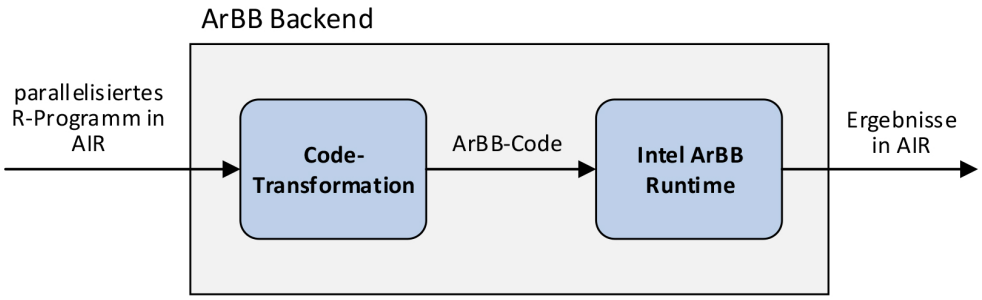


Abbildung 7: Parallele Ausführung durch das ArBB Backend laut [Kie13]

1. Das Backend erhält von der ALCHEMY-Plattform parallelisierten R-Code in AIR
2. Die Eingabe wird in Backend-Code (C++) übersetzt und kompiliert
3. Das Programm wird nun durch die Intel ArBB Runtime auf Multicore-Hardware ausgeführt
4. Die ermittelten Ergebnisse werden zurück in AIR übersetzt und an ALCHEMY zurück gegeben, wo sie schließlich dem R-Benutzer bzw. -Experimentierer präsentiert werden

Es ist geplant, die Evaluierung dieser Arbeit vollständig unter Verwendung des Intel ArBB Backends durchzuführen.

3. SURE-Verfahren

3.1. Überblick

Systems of Uniform Recurrence Equations (SURE) ist eine mathematische Spezifikation von Werten, die implizit als Ergebnisse von rekursiven Berechnungen beschrieben werden. Eingeführt wurde dieses Modell von Karp, Miller und Winograd [KMW67] mit dem Ziel

- Präzise Einsichten in die Organisation dieser Berechnungen zu erlangen
- Möglichkeiten herauszufinden, wie besagte Werte explizit durch eine angebrachte Umsortierung berechnet werden können

Ein Beispiel für ein einfaches SURE im Kontext verschachtelter Schleifen mit Matrixzugriffen ist in Listing 2 zu sehen:

```
1 # ... Initialisiere
2 for (i in 1:n) {
3   for (j in 1:n) {
4     a[i,j] = a[i-1,j-1] + a[i,j-1];
5   }
6 }
```

Listing 2: R-Beispiel eines SURE

Die formale Definition von SURE und eine Demonstration, inwieweit die Arbeiten von Karp, Miller und Winograd Einfluss auf die Entwicklung im Zusammenhang mit automatischer Parallelisierung hatten, ist [Dar98] zu entnehmen.
Laut [PM12] berechnet das SURE-Verfahren aus den Indexzugriffen im Schleifenrumpf eine kompakte Darstellung der Datenabhängigkeiten. Mittels linearer Programmierung werden zyklische Abhängigkeiten ermittelt, die wiederum Grundlage für die Berechnung eines Ablaufplans für die Operationen im Schleifenrumpf sind. Schließlich wird mit Hilfe dieses Ablaufplans eine parallele Version des Schleifennests unter Verwendung des `DOPAR` Skeletons berechnet. Die Parallelisierungsanalyse mit `SURE` ist in mehrere Schritte unterteilt, die idealtypisch jeweils eine wiederverwertbare Ausgabe generieren (vergleiche Abbildung 2). Speziell für die Parallelisierung verschachtelter Schleifen wurden bereits mehrere Algorithmen - im Folgenden nur noch als „SURE-Algorithmen“ bezeichnet - mit unterschiedlichen Ansätzen entworfen:

- Allen und Kennedy [AK87]: Dekomposition des Abhängigkeitsgraphen in *Strongly Connected Components* (SCCs) mittels Graphenalgorithmen

- Wolf und Lam [WL91]: Automatisch generierte, unimodulare Schleifentransformationen durch Matrixberechnungen
- Darte und Vivien [DV96b]: Ermittlung multidimensionaler Ablaufpläne mittels linearer Programmierung

Für diese Arbeit wird der (SURE-)Algorithmus von Allen und Kennedy aus [AK87] implementiert. In den folgenden Unterkapiteln werden die einzelnen Teilschritte des SURE-Verfahrens in Anlehnung dessen noch einmal genauer betrachtet.

3.2. Vorverarbeitung

Bevor das SURE-Verfahren angewendet werden kann, muss die Eingabe, in diesem Fall ein R-Programm (in AIR), durch eine Reihe von vorverarbeitenden Schritten angepasst werden. An dieser Stelle ist auch zu bemerken, dass eine berechenbare[1] Eingabe vorausgesetzt wird. In erster Linie dient die Vorverarbeitung zur Erkennung von SURE(s) im Programm-Code und der Extraktion dieser erkannten Code-Stücke, sofern vorhanden.

```
1  # ... Einige non-SURE Operationen
2
3  for (i in 1:n) {
4    for (j in 1:n) {
5      for (k in 1:n) {
6        a[i,j,k] <- a[i-1,i+j,k] * a[i,j,k] + b[i,i-1,k];
7        b[i,j,k] <- b[i,i-k,k] * a[i-1,j,k] + 42;
8      }
9    }
10 }
11
12 # Weitere non-SURE Operationen ...
```

Listing 3: Erkanntes SURE in einem R-Programm

Für eine korrekte Erkennung ist an erster Stelle zu definieren, wie genau das zu erkennende Muster auszusehen hat, damit eine Anwendung des Algorithmus von Allen und Kennedy durchführbar ist:

1. Die Zellenwerte einer oder mehrerer Matrizen werden Zelle für Zelle modifiziert
2. Adressierung der Matrixzellen durch Indexvariablen verschachtelter Schleifen

[1] Berechenbarkeitsanalyse ist nicht Teil dieser Arbeit

Diese Definition schließt das Auftreten von Sonderfällen, wie z.B. eingeschobene Operationen zwischen den Schleifen (siehe Listing 4) mit ein.

```
1 # ... Initialisiere
2 for (i in 1:n) {
3   a[i,] = 1
4   for (j in 1:n) {
5     a[i,j] = a[i-1,j-1] + a[i,j-1];
6   }
7 }
```

Listing 4: Sonderfall "Eingeschobene Operation"

Eine Verwendung des SURE-Algorithmus aus [WL91] würde weitere (Struktur verändernde) Vorverarbeitungsschritte voraussetzen, nämlich das Aufspalten des SURE in atomare URE. Eingeschobene Operationen müssten ebenfalls separat berücksichtigt werden.
Unter Zuhilfenahme eines Parsers (siehe Kapitel 2.3) und einer dem Muster entsprechenden Grammatik, wird die Eingabe untersucht und jegliche Vorkommen als separates Code-Stück zurück geliefert.

3.3. Datenabhängigkeitsanalyse

Die Optimierung von sequentiellen Programmen für eine parallele Ausführung basiert ausschließlich auf verschiedenen Verfahren hinsichtlich der Bestimmung von Datenabhängigkeiten. Die Datenabhängigkeitsanalyse identifiziert Beziehungen zwischen Operationen mit Speicherzugriffen. Sie dient somit als nötige Informationsquelle für eine zulässige Code-Transformationen. Eine sehr ausführliche Beschreibung des Abhängigkeitsproblems und unterschiedliche Lösungsansätze können [Kyr07] entnommen werden.
In Anbetracht der Problemstellung werden im Folgenden Datenabhängigkeiten im Rumpf von `for`-Schleifen analysiert. Vorerst werden diese entsprechend der Definitionen und Erläuterungen aus [DV95] und [Dar98] beschrieben.

Datenabhängigkeiten existieren zwischen Operationen (engl. Statements) im Schleifenrumpf. Zwei Statements S_1 und S_2 werden als abhängig bezeichnet, falls beide auf den selben Speicherort zugreifen und mindestens einer dieser Zugriffe schreibend ist. Man schreibt $S_i(I) \Longrightarrow S_j(J)$, falls eine Operation S_j der (Schleifen-)Iteration J von Operation S_i der Iteration I abhängig ist. Je nach Anordnung entsprechen diese einer

True Dependence (deu. Echte Abhängigkeit), falls Daten nach einer Schreiboperation gelesen werden (z.B. S_1 schreibend und S_2 lesend)

Anti-Dependence (deu. Gegenabhängigkeit), falls Daten nach einer Leseoperation modifiziert werden (z.B. S_1 lesend und S_2 schreibend)

Output Dependence (deu. Ausgabeabhängigkeit), falls Daten mehrfach modifiziert werden (z.B. S_1 und S_2 schreibend)

Datenabhängigkeiten in Schleifen lassen sich durch einen gerichteten, azyklischen Graphen, den *Expanded Dependence Graph* (EDG) darstellen. In einem solchen Graphen stellen Operationen[2] die Knoten dar und die Abhängigkeiten zwischen diesen sind als Kanten realisiert (siehe Abbildung 8):

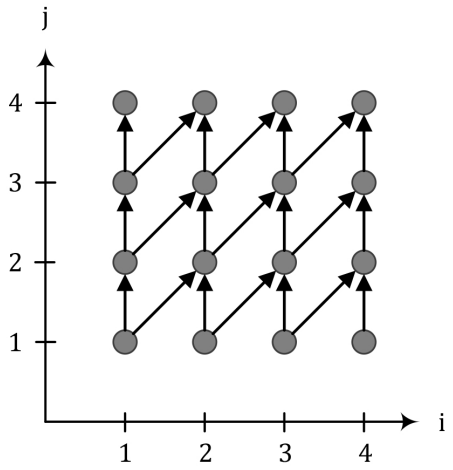


Abbildung 8: EDG zu Listing 2

"Das Erkennen von Parallelisierbarkeit bedeutet das Finden von Antiketten im EDG" [DV95]. Existiert also eine Teilmenge an Knoten im EDG, für die keine zwei ihrer Elemente durch einen gerichteten Kantenzug verbunden sind so ist ein gewisser Grad an Parallelisierbarkeit vorhanden. Leider ist eine Verwendung von EDGs als Eingabe für SURE-Algorithmen zu ineffizient. Die Generierung von EDGs beansprucht zu viel Zeit und die Graphen werden zu groß. [Dar98] zeigt, dass in manchen Fällen eine präzise Datenabhängigkeitsanalyse zwar möglich, oft aber gar nicht benötigt ist. Aus diesem Grund wird eine gekürzte Form eines approximierten EDGs, der *Reduced Dependence Graph* (RDG), verwendet. Im Gegensatz zum EDG stellen lediglich die Statements im Schleifenrumpf die Knoten des Graphen dar. Wie Abbildung 9 verdeutlicht, sind die Kanten (Abhängigkeiten)

[2] Instanzen eines Statements

entsprechend der gewählten Approximation gewichtet. Ferner ist zu vermerken, dass der RDG auch zyklisch sein kann.

Abbildung 9: RDG zu Listing 2 mit Abhängigkeitsgraden (links) und Richtungsvektoren (rechts)

Generell können Algorithmen nicht zwischen EDGs unterscheiden, welche durch den gleichen RDG approximiert werden. Die Qualität der Datenabhängigkeitsanalyse ist folglich durch die Genauigkeit und Darstellung der gewählten Approximation definiert.
Die meisten SURE-Algorithmen berechnen für die Datenabhängigkeitsanalyse einen Satz aus Distanzvektoren $(J - I)$, das sog. *Distance Set*:

$$E_{i,j} = \{(J - I) | S_i(I) \Longrightarrow S_j(J)\}$$

Die unterschiedlichen Repräsentationen für das *Distance Set* stellen nun die besagten Approximationen der Kantengewichte im RDG dar. Für den Algorithmus von Allen und Kennedy wird das *Distance Set* durch einen einzigen Wert l repräsentiert, den "Abhängigkeitsgrad", für welchen gilt:

$l = \infty$ Falls $S_i(I) \Longrightarrow S_j(J)$ und mit Distanzvektor $(J - I) = 0$ (*loop independent*)

$1 \leq l \leq n$ Falls $S_i(I) \Longrightarrow S_j(J)$ und die erste Komponente ungleich Null von $(J - I)$ die l-te Komponente ist. n entspricht der Tiefe des Schleifennests (*loop carried dependence*)

Ein RDG mit Abhängigkeitsgraden (eng. *Level of Dependence*) als Kantenbeschreibung wird *Reduced Leveled Dependence Graph* (RLDG) genannt. Dieser stellt die Eingabe des Algorithmus von Allen und Kennedy dar (siehe Abbildung 10):

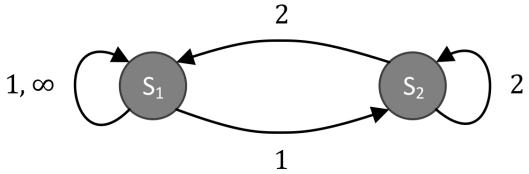


Abbildung 10: RLDG von Listing 3

Ein weiteres klassisches Beispiel für die Repräsentation eines *Distance Sets* sind Richtungsvektoren (eingeführt in [Lam74]). Aus Gründen der Irrelevanz für diese Arbeit wird diese Repräsentation ausschließlich in Anhang B erläutert.

Datenabhängigkeitstests

In [KA02] beschreiben K. Kennedy und J.R. Allen Variablen in Schleifenstrukturen:

Index Eine Variable zur Iteration einer Schleife. Für gewöhnlich werden (Schleifen-)Indizes für die Indizierung von Matrizen verwendet

Subscript Ein Index in Bezug auf eine Matrixreferenz ist ein Subscript (Matrixindex). Beispielsweise besitzt die Matrix `m[i, j]` zwei Subscripts.

Die größte Herausforderung bei der Datenabhängigkeitsanalyse besteht darin, zwei Subscripts auf Gleichheit zu überprüfen. Die Anzahl von unterschiedlichen Indizes in einer Matrix m wird die *Komplexität* von Matrix m genannt. Im Folgenden sind die drei existieren Komplexitätsgrade aufgelistet:

ZIV Eine *zero index variable* einer Matrixreferenz verwendet **keinen** Index in einem Subscript-Paar.

SIV Eine *single index variable* verwendet **einen** Index in einem Subscript-Paar.

MIV Eine *multiple index variable* tritt auf, falls eine Matrixreferenz **mehrere** Indizes in einem Subscript-Paar verwendet.

Vorausgesetzt, ein Statement $m[1,\ 2i,\ j] \leftarrow m[42,\ i,\ k-1]$ befindet sich im Rumpf eines 3-dimensionalen Schleifennests, so lässt sich feststellen, dass

- das erste Subscript-Paar $(1,\ 42)$ eine ZIV enthält,
- das zweite Subscript-Paar $(2i,\ i)$ eine SIV enthält und
- das dritte Subscript-Paar $(j,\ k-1)$ eine MIV enthält

Es existieren viele unterschiedliche Datenabhängigkeitstests, welche sich auf einen jeweiligen Komplexitätsgrad spezialisiert haben. In [JS03] werden einige davon aufgelistet und erläutert. Ferner wird die Behauptung aufgestellt, dass "Datenabhängigkeitstests über das Potential vermögen Abhängigkeit zu erkennen, obwohl diese nicht existiert. Niemals wird jedoch eine Unabhängigkeit zwischen Subscripts deklariert, falls eine Abhängigkeit existiert."

3.4. Ablaufplanung

Im vorhergehenden Schritt des SURE-Verfahrens wurden die nötigen Abhängigkeitsinformationen in Form eines RDGs approximiert. Ziel dieses Schrittes ist es nun die Statements im Schleifenrumpf mittels der gewonnenen Informationen derart umzuordnen, sodass eine parallele Ausführung ermöglicht wird. Die neue Anordnung wird in einem Ablaufplan vereint. Um solch eine neue Anordnung zu definieren, existieren laut [DRV01] mehrere Ansätze:

1. Elementare Schleifentransformationen, wie *loop distribution*[3] (wie sie in [AK87] verwendet wird)
2. Lineare Änderung auf Basis der Iterationsdomäne durch eine unimodulare Transformation des Iterationsvektors (wie in [WL91])
3. Definieren eines d-dimensionalen Ablaufplans durch eine affine Transformation des Iterationsraums von $\mathbb{Z}^n$ in $\mathbb{Z}^d$ (wie in [DV96b])

Inwiefern der hier verwendete Algorithmus von Allen und Kennedy Gebrauch von elementaren Schleifentransformationen macht und wie dessen generelle Arbeitsweise aussieht, wird im weiteren Verlauf näher betrachtet.

Algorithmus von Allen und Kennedy

Aus [AK87] geht hervor, dass der Algorithmus ursprünglich entworfen wurde, um Programme mit sequentiellen Schleifenoperationen derart zu transformieren, sodass mehrere Schleifenoperationen simultan ausgeführt werde können ("Schleifenvektorisierung"). Dieser Entwurf wurde dann weiter verfeinert. Dadurch wurde es möglich, die Anzahl an parallelen Schleifen zu maximieren und gleichzeitig die Anzahl an Synchronisationen zu minimieren.
Der Algorithmus macht dabei Gebrauch von den aus der Graphentheorie stammenden *Strongly Connected Components* (SCCs). Laut [NSs94] wird ein gerichteter (Teil-)Graph G als *strongly connected* definiert, falls von jedem Knoten in G ein Pfad zu allen anderen Knoten führt. Die SCCs von G sind demzufolge eine disjunkte Menge von *strongly connected* Teilgraphen aus G.
Die Idee des Algorithmus basiert auf den folgenden Fakten, die an dieser Stelle noch einmal als Zusammenfassung von [DV96a] und den Erkenntnissen aus Kapitel 3.3 verdeutlicht werden:

[3] Aufspalten der Operationen im Schleifenrumpf in mehrere Schleifen mit selbem Indexbereich

1. Eine Schleife der Tiefe i ist **parallel**, falls keine *loop carried dependence* besteht. D.h. es darf keine Abhängigkeit im RLDG mit dem Abhängigkeitsgrad gleich i, bezüglich einem Statement im Schleifenrumpf, existieren.
   Diese Schleife kann dementsprechend mit "forall"[4] markiert werden

2. Alle Iterationen eines Statements S_1 können **vor** einer beliebigen Iteration eines Statements S_2 durchgeführt werden, falls im RLDG keine Abhängigkeit von S_2 nach S_1 existiert.
   Dies ermöglicht eine Extraktion der SCCs des RLDGs mittels *loop distribution* um diese unabhängig auf Parallelisierbarkeit zu untersuchen

Für den in [DV95] definierten, rekursiven Algorithmus `Allen-Kennedy(`G, k`)` mit initialem Aufruf `Allen-Kennedy(`G, `1)` gilt:

G	sei der aus der Datenabhängigkeitsanalyse berechnete RLDG und
k	die Indexvariable der aktuellen Schleifentiefe

Ferner sei:

N	die Dimension der Schleifenverschachtelung,
$G(k)$	der Teilgraph von G, dessen Kanten i mit Abhängigkeitsgrad $c_i < k$ entfernt wurden und
G_j	die SCCs von $G(k)$

Wie in Abbildung 11 veranschaulicht, wird in jedem rekursiven Schritt in Abhängigkeit der aktuellen Schleifentiefe ($k = 1$ entspricht hierbei der äußersten Schleife) zuerst $G(k)$ berechnet. Nun wird dieser modifizierte RLDG in seine SCCs G_j zerlegt. Anschließend werden die SCCs topologisch sortiert um deren ursprüngliche Ordnung beizubehalten. Entsprechend dieser Sortierung, werden alle G_j (bzw. die Statements, welche von G_j umschlossen sind) in separate Schleifennester der aktuellen Tiefe aufgespalten (*loop distribution*). Die Schleife mit Tiefe k eines jeden G_j wird als parallel ausführbar ("forall") markiert, falls G_j keine Kante mit Abhängigkeitsgrad gleich k besitzt. Schließlich leitet der rekursive Aufruf den Prozess für alle SCCs auf der nächste Schleifenebene ein.

Ein detailliertes Beispiel einer vollständigen Ausführung des SURE-Algorithmus auf Basis des RLDGs aus Abbildung 10 ist in Anhang C zu finden. Der Ablaufplan umfasst alle relevanten Informationen, die durch den SURE-Algorithmus ausgewertet werden.

[4] andere Bezeichnung für DOPAR

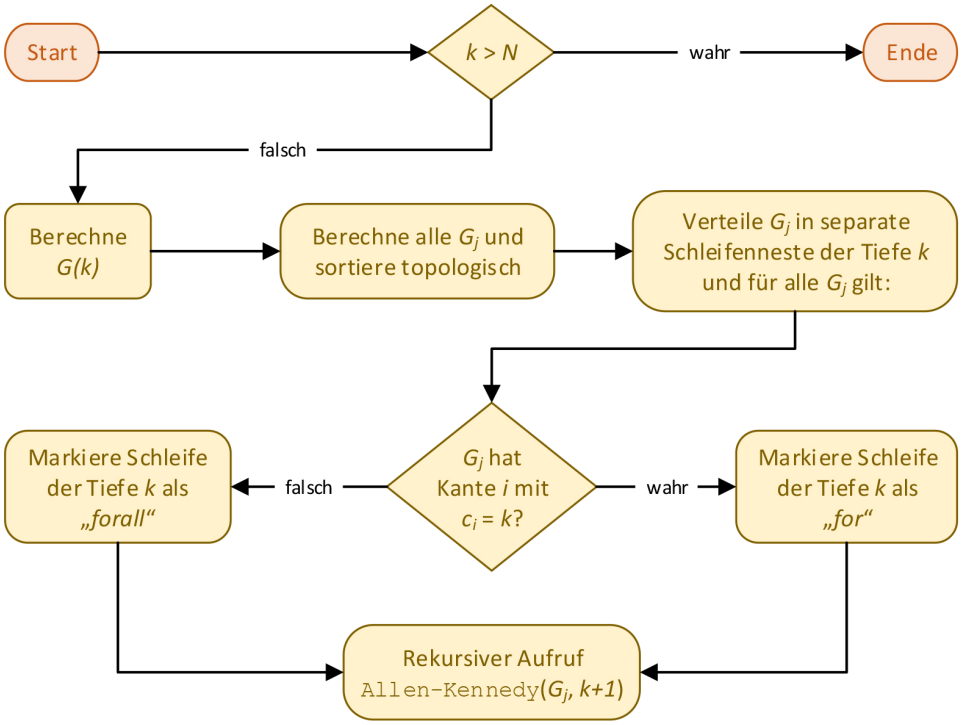


Abbildung 11: Arbeitsschritte eines Aufrufs von `Allen-Kennedy`(G, k)

3.5. Transformation

Im letzten Schritt des SURE-Verfahrens wird das analysierte Code-Stück (AIR) in Abhängigkeit des Ablaufplans transformiert. Durch diese Umwandlung wird eine parallele Ausführung mit dem gewählten Backend ermöglicht. Hierfür wird das in Kapitel 2.4 vorgestellte parallele Skeleton `DOPAR` verwendet.
Listing 5 stellt den parallelisierten Pseudo-Code des Listings 3 mit “forall“-Markierungen dar.

```
# ... Einige non-SURE Operationen

for (i in 1:n) {
  for (j in 1:n) {
    forall (k in 1:n) {
      b[i,j,k] <- b[i,i-k,k] * a[i-1,j,k] + 42;
    }
  }
  forall (j in 1:n) {
    forall (k in 1:n) {
      a[i,j,k] <- a[i-1,i+j,k] * a[i,j,k] + b[i,i-1,k];
    }
  }
}

# Weitere non-SURE Operationen ...
```

Listing 5: Parallelisierter Pseudo-Code von Listing 3

3.6. Parallele Ausführung

Wie man dem Laufbeispiel entnehmen kann, wurde der ursprüngliche R-Programm-Code

1. auf SURE-Vorkommen **untersucht**
2. auf Abhängigkeiten zwischen Matrixzugriffen **geprüft**
3. entsprechend eines Ablaufplans **umgeordnet**
4. unter Einbezug des parallelen Programmiermusters `DOPAR` **transformiert**

Der Programmcode ist nun in einem ausführbaren Zustand. Ein beliebiges, `DOPAR`-kompatibles Backend übernimmt dessen Ausführung und präsentiert das Ergebnis dem Benutzer.

4. Anforderungsanalyse

In Kapitel 3 wurde das grundlegende Vorgehen des SURE-Verfahrens beschrieben, welches zentrale Anforderungen an das zu entwickelnde `SURE`-Modul impliziert. An dieser Stelle werden nun die zentralen Anwendungsfälle (AFs) und Datenstrukturen, die durch `SURE` realisiert werden sollen, definiert. In den weiteren Unterkapiteln wird eine objektorientierte Analyse für alle Anwendungsfälle ausgearbeitet.

Anwendungsfälle

Die von `SURE` realisierten Anwendungsfälle sind in Abbildung 12 zusammengefasst:

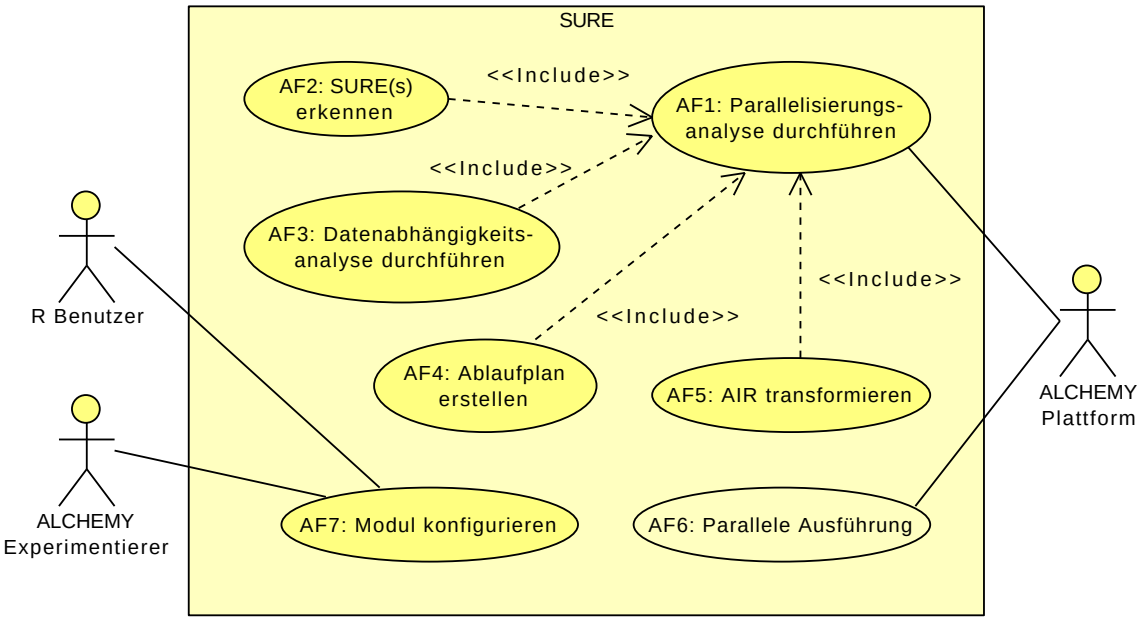


Abbildung 12: Anwendungsfälle für SURE

In erster Linie schreibt das SURE-Verfahren die Durchführung einer Parallelisierungsanalyse (AF1) vor. Aus Kapitel 3.1 ist bekannt, dass das Verfahren mehrere Teilschritte umfasst:

- Die Programmstruktur eines SURE in AIR-Programmen muss erkannt werden (AF2)
- Ein SURE muss auf Datenabhängigkeiten zwischen Matrixzugriffen untersucht werden (AF3)

- Es muss ein Ablaufplan für die Matrixzuweisungen (im Schleifenrumpf) erzeugt werden (AF4)
- Das ursprüngliche AIR-Programm wird unter Einbezug des `DOPAR`-Skeletons transformiert (AF5)

Wie aus Kapitel 3.6 hervorgeht, wird das parallelisierte AIR-Programm vom `DOPAR`-kompatiblen `ArBB`-Backend ausgeführt (AF6). Hinsichtlich der Erweiterbarkeit um weitere SURE-Algorithmen, soll das Modul konfigurierbar sein (AF7).

In Abhängigkeit der gewählten Programmiersprache Java für die ALCHEMY-Implementierung, ist `SURE` ebenfalls in dieser zu implementieren.

Datenstrukturen

Aus Kapitel 3 ist bekannt, dass die Parallelisierungsanalyse einem Pipeline-Schema folgt. Die zentralen Datenstrukturen lassen sich deshalb in Ein- und Ausgaben gliedern.

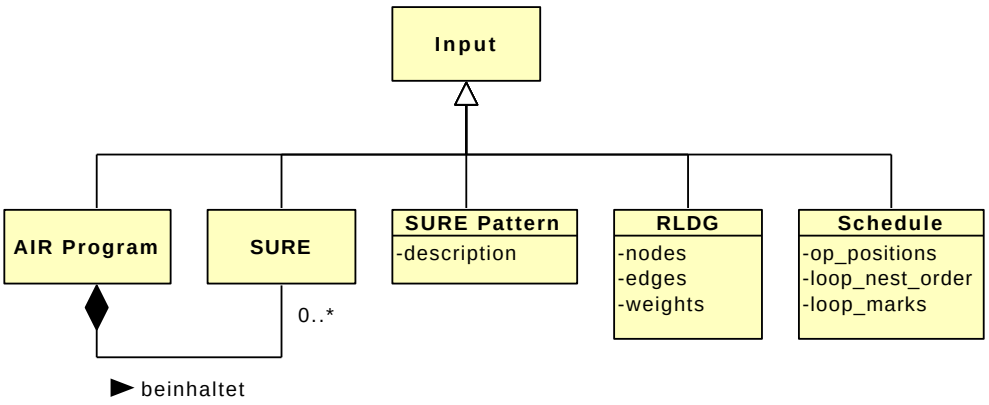


Abbildung 13: Eingaben von `SURE`

Die Ein- und Ausgaben der einzelnen Verarbeitungsschritte sind wie folgt definiert (siehe Abbildungen 13 und 14):

AIR Program Das zu parallelisierende R-Programm als AIR. Ein Paar bestehend aus einer *AIR Expression* und der *AIR Environment* (siehe [Pfa12]). Ist Eingabe für AF2

SURE Pattern Repräsentiert das Muster für die Erkennung von verschachtelten Schleifen mit Matrixmodifikationen (siehe Kapitel 3.2). Ist ebenfalls Eingabe für AF2

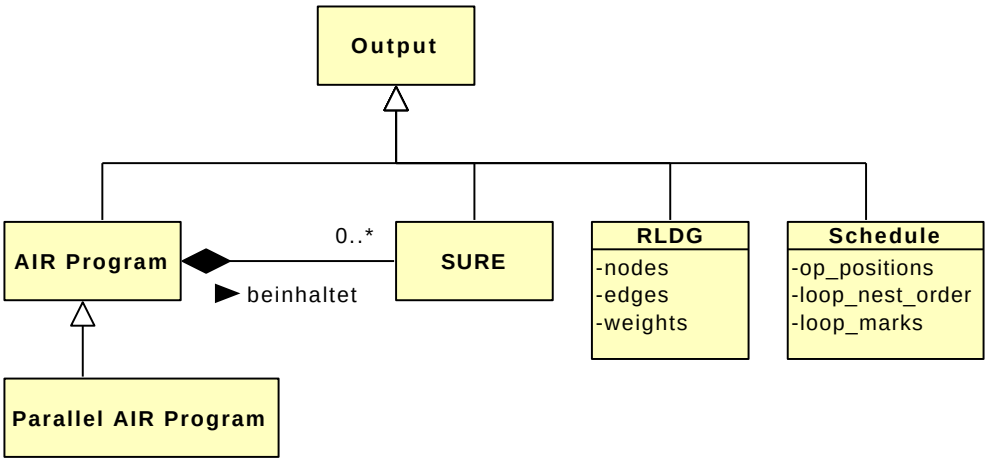


Abbildung 14: Ausgaben von `SURE`

SURE Ein AIR Code-Stück aus einem *AIR Program*, welches dem *SURE Pattern* entspricht. Ist Ausgabe von AF2 und gleichzeitig Eingabe für AF3

RLDG Entspricht dem in Kapitel 3.3 vorgestellten "Reduced Leveled Dependency Graph". Stellt Operationen im Schleifenrumpf als Knoten und Datenabhängigkeiten als gewichtete Kanten dar. Ist Ausgabe von AF3 und gleichzeitig Eingabe für AF4

Schedule Präziser Ablaufplan für die Operationen eines *SURE*. Umfasst die in Kapitel 3.4 vorgegebenen Informationen für ein parallel ausführbares *SURE*. Ausgabe von AF4 und gleichzeitig Eingabe für AF5

Parallel AIR Program Ein *AIR Program*, dessen *SURE* parallelisiert wurden. Ist durch das `ArBB`-Backend parallel ausführbar. Ausgabe von AF5

In den weiteren Unterkapiteln werden alle genannten AFs noch einmal (entsprechend des Verfahrens aus [RK99]) detailliert analysiert. Ziel ist es, die einzelnen Anwendungsfälle in ihre relevanten Bestandteile zu zerlegen und die verschiedenen Aspekte durch vereinfachte Kollaborationsdiagramme näher zu untersuchen. Die Kollaborationsdiagramme verwenden dabei sogenannte "Analyseklassen", die in Tabelle 1 kurz beschrieben sind:

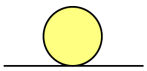
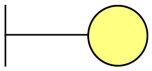
Entitätsklasse		Repräsentiert Objekte aus dem Domänenmodell, welche persistente Informationen halten.
Kontrollklasse		Für die Steuerung und Koordination zwischen Entitäts- oder anderen Kontrollobjekten zuständig.
Grenzklasse		Stellt eine Schnittstelle im System dar.

Tabelle 1: Bedeutung der verwendeten Analyseklassen

Für eine anschaulichere Darstellung sind die Analyseklassen verschieden eingefärbt:

Gelb Zu erzeugende Artefakte

Orange Bereits vorhandene bzw. externe Artefakte, wie z.B. Bibliotheken

4.1. AF1: Parallelisierungsanalyse durchführen

Jede Parallelisierungsanalyse wird durch ALCHEMY eingeleitet. Falls die *Transmutation Configuration* (siehe Kapitel 2.1) eine Analyse mit `SURE` vorsieht, leitet das *Transmutation Interface* eine Anfrage an den *SURE Controller*. Der *SURE Controller* initiiert dann die Analyse. Dieser Anwendungsfall ist, wie in Kapitel 3 beschrieben, in Abbildung 15 dargestellt.

Die folgenden Analyseklassen realisieren den Anwendungsfall:

AIR Set Eine Liste von *AIR Programs*

SURE Set Eine Liste von *SUREs*

SURE Controller Erhält ein *AIR Set* als Eingabe, steuert die Parallelisierung der beinhaltenden *AIR Programs* und gibt das transformierte *AIR Set* aus. Ferner bildet der *SURE Controller* das Bindeglied aller im SURE-Verfahren inbegriffenen Teilschritte und kommuniziert die verwendeten und entstandenen Artefakte mit den anderen Klassen

SURE Detector Steuert die Erkennung jeglicher Vorkommen von SURE(s) in einem *AIR Program*. Liefert dabei immer ein *SURE Set* zurück oder leitet einen vorzeitigen Abbruch der Analyse ein, falls kein *SURE Set* erzeugt werden kann (entsprechend Kapitel 3.2)

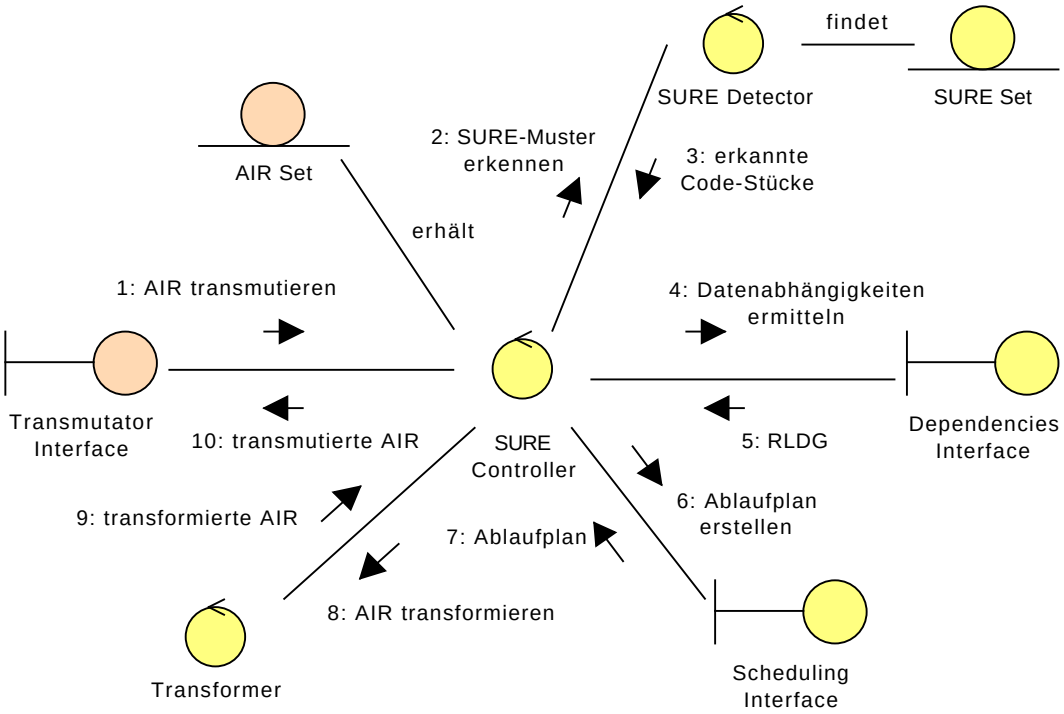


Abbildung 15: Kollaborationsdiagramm für AF1: "Parallelisierungsanalyse durchführen"

Dependencies Interface Erzeugt eine Repräsentation der Datenabhängigkeiten entsprechend des verwendeten SURE-Algorithmus. In diesem Fall ist dies ein *RLDG*, der abschließend als Ausgabe zurück an den *SURE Controller* gesendet wird (entsprechend Kapitel 3.3)

Scheduling Interface Analysiert die zu Grunde liegende Repräsentation der Datenabhängigkeiten (hier: den *RLDG*) und erzeugt einen *Schedule* (entsprechend Kapitel 3.4). Die Inhalte des *Schedules* werden gemäß des verwendeten SURE-Algorithmus (hier: Allen und Kennedy) erstellt (siehe Kapitel 4.4)

Transformer Wandelt entsprechend den Vorgaben des *Schedules* das *AIR Program* in ein *Parallel AIR Program* um. Die Parallelisierung für das aktuelle *SURE* ist danach beendet (entsprechend Kapitel 3.5)

4.2. AF2: SURE(s) erkennen

Unabhängig vom verwendeten SURE-Algorithmus wird der *SURE Detector* verwendet um das *AIR Program* auf das *SURE Pattern* zu untersuchen und erkannte *SUREs* auszugeben. Gestaltet sich die Suche als erfolglos, so wird die Analyse

abgebrochen und der Parallelisierungsprozess beendet.

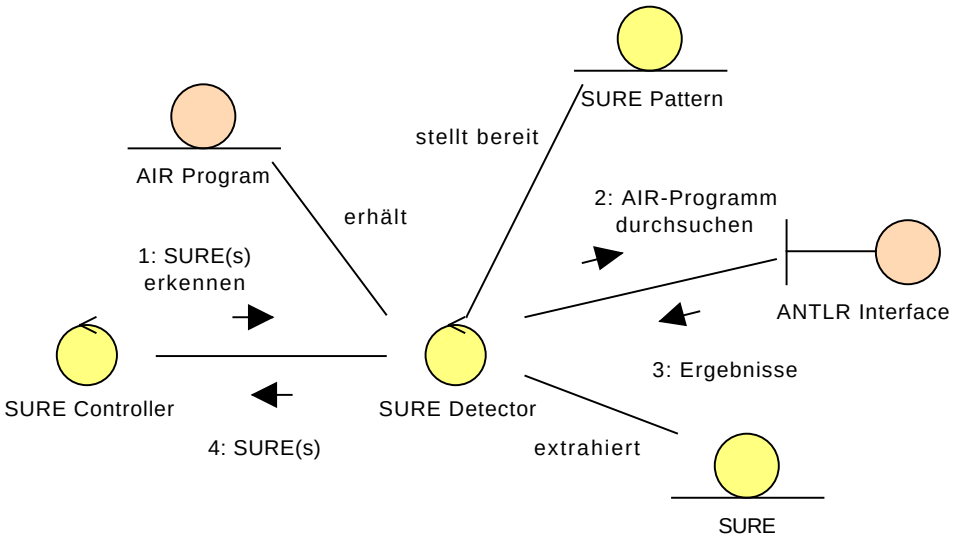


Abbildung 16: Kollaborationsdiagramm für AF2: "SURE erkennen"

Einzige neue Analyseklasse:

ANTLR Interface Die Schnittstelle für den Parsergenerator ANTLR (siehe Kapitel 2.3). Identifiziert vorhandene *SURE* in einem *AIR Program*. Verwendet das bereit gestellte *SURE Pattern*

4.3. AF3: Datenabhängigkeitsanalyse durchführen

Der *Dependency Analyzer* delegiert die Erzeugung einer entsprechenden Repräsentation der Datenabhängigkeiten an einen entsprechenden Graphengenerator. In diesem Fall werden der Reihe nach die *SURE* aus dem *SURE Set* entnommen und an den *RLDG Generator* überreicht. Wurde ein anderer SURE-Algorithmus ausgewählt werden die *SURE* an den *RDG Generator* überreicht. Nachdem ein *RLDG* erzeugt wurde, wird dieser zurück an den *Dependency Analyzer* gesendet und mit dem nächsten *SURE* fortgefahren. Die Datenabhängigkeitsanalyse ist beendet, wenn für alle Elemente aus dem *SURE Set* ein zugehöriger *RLDG* generiert wurde.

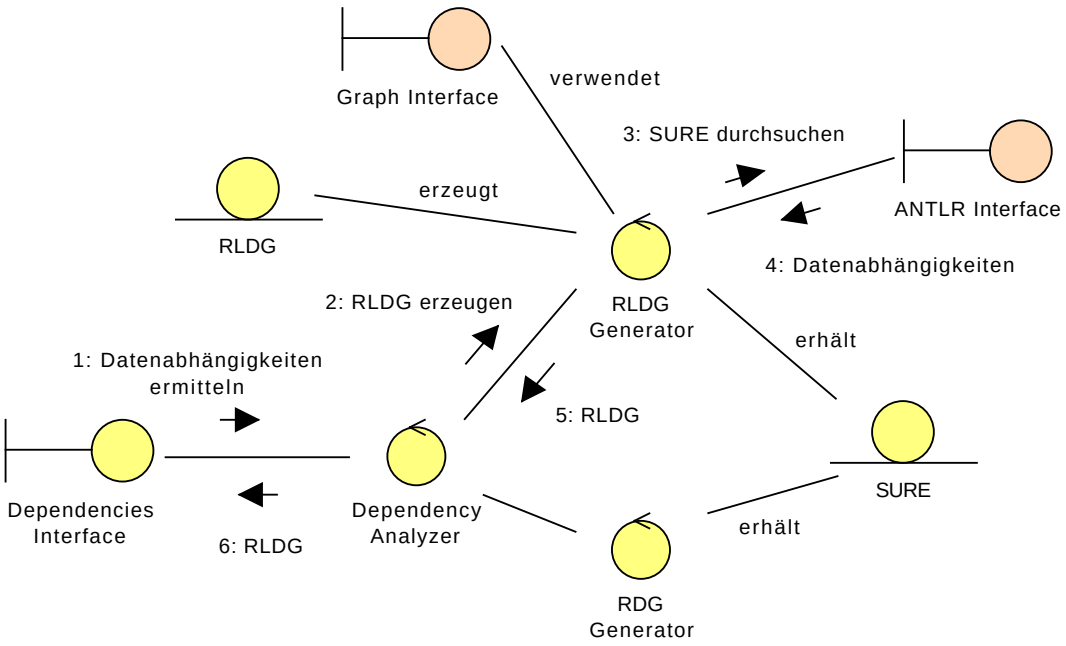


Abbildung 17: Kollaborationsdiagramm für AF3: "Datenabhängigkeits-analyse durchführen"

Die neu hinzugekommenen Analyseklassen sind:

Dependency Analyzer Steuert die Erzeugung von Abhängigkeitsgraphen abhängend von dem gewählten SURE-Algorithmus. Erhält ein *SURE Set* als Eingabe

RLDG Generator Erzeugt aus einem *SURE* einen *RLDG*. Verwendet dabei das *ANTLR Interface* um ein *SURE* auf die nötigen Datenabhängigkeiten zu durchsuchen. Das *Graph Interface* stellt die nötige Funktionalität bereit.

RDG Generator[5] Erzeugt aus einem *SURE* einen "Reduced Dependency Graph". Wird nur verwendet, falls nicht der Algorithmus von Allen und Kennedy ausgewählt wurde

Graph Interface Bietet alle Datenstrukturen und Funktionalitäten für die Erzeugung und Verarbeitung von Graphen (inkl. *RLDGs*)

4.4. AF4: Ablaufplan erstellen

Dieser Anwendungsfall beschreibt die Erstellung eines *Schedules* mit Hilfe des gewählten SURE-Algorithmus. Ein SURE-Algorithmus arbeitet auf Basis eines Abhängigkeitsgraphen (siehe Kapitel 3.4). Der *Operation Scheduler* analysiert die

Abhängigkeiten im Graphen, wie in Abbildung 11 bereits veranschaulicht wurde. Produziert wird ein *Schedule*, der folgende Informationen enthält:

1. Die **Zugehörigkeit** einer Operationen zu einem Schleifennest
2. Die sequentielle **Anordnung** der aufgeteilten Schleifennester
3. Die **Markierungen**, ob eine Schleife parallel ausgeführt werden kann oder nicht

Diese Informationen gehen aus dem Ansatz der Schleifentransformation *loop distribution* (siehe Kapitel 3.4) und der Beschreibung des Algorithmus von Allen und Kennedy hervor.

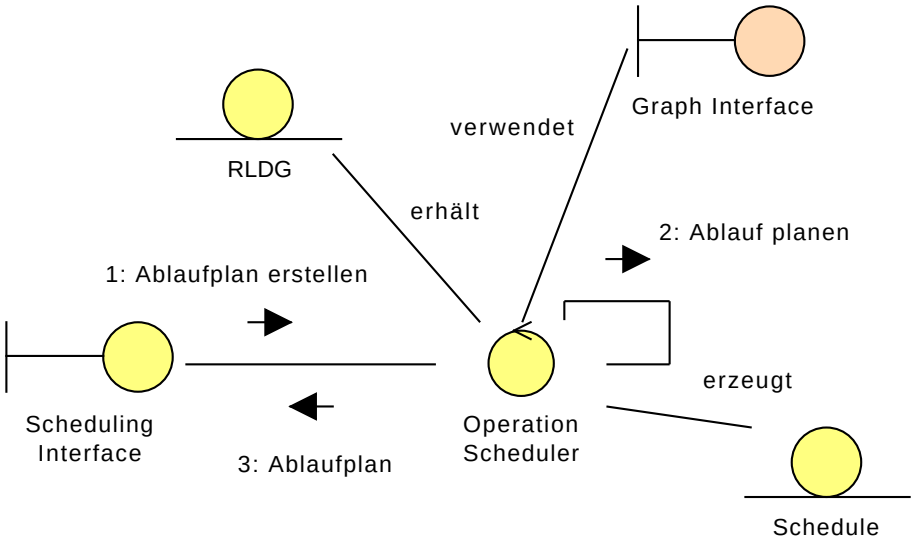


Abbildung 18: Kollaborationsdiagramm für AF4: "Ablaufplan erstellen"

Operation Scheduler Realisiert den in Kapitel 3.4 vorgestellten Algorithmus von Allen und Kennedy

4.5. AF5: AIR transformieren

Der *Transformer* wandelt das *AIR Program* in ein *Parallel AIR Program*. Hierfür wird der in AF4 beschriebene *Schedule* verwendet.

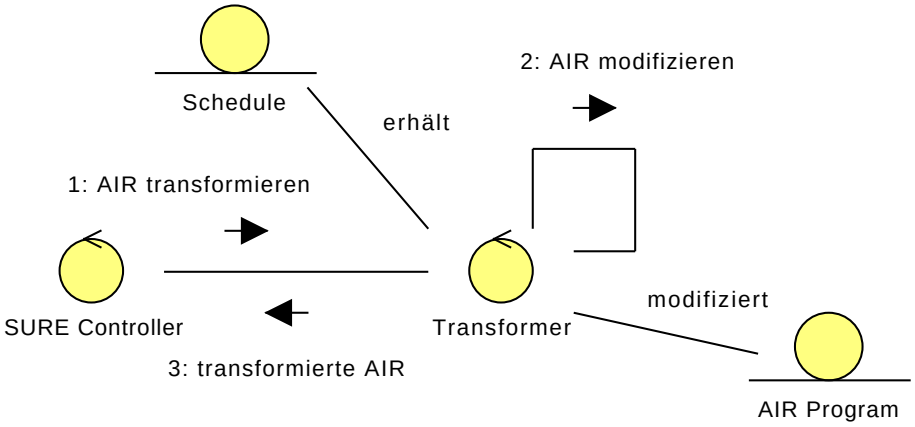


Abbildung 19: Kollaborationsdiagramm für AF5: "AIR transformieren"

Für Schleifen, die parallel ausgeführt werden sollen, wird ausschließlich das `DOPAR`-Skeleton verwendet. Parallele Skeletons sind als *AIR Expressions*, präziser als `SkeletonExpr`, zu realisieren:

```
<SkeletonExpr name="DOPAR">
  <params>
    <param name="#paramname#">
      #AIRExpr#
    </param>
    <param name="#paramname#">
      #AIRExpr#
    </param>
    ...

  </params>
</ SkeletonExpr>
```

Die `SkeletonExpr` wird mit "DOPAR" bezeichnet und besitzt eine Liste an Parametern, die sich wieder auf andere *AIR Expressions* beziehen.
Der konzeptionelle Transformationsprozess ist in Abbildung 20 dargestellt.

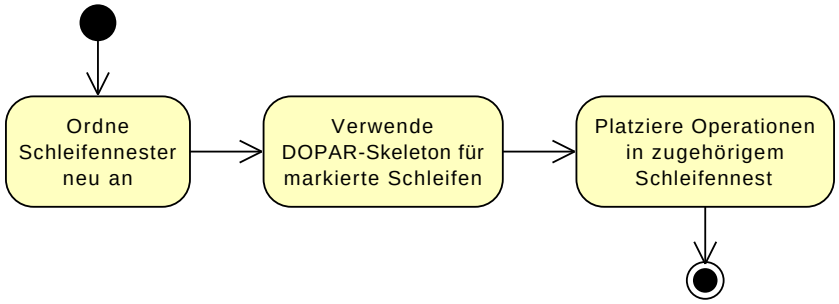


Abbildung 20: Transformation des *AIR Programs*

4.6. AF6: Parallele Ausführung

Nachdem die Parallelisierungsanalyse mit `SURE` abgeschlossen ist, wird die parallele Ausführung mit dem `ArBB`-Backend eingeleitet. Das `ArBB`-Backend kommuniziert, wie das `SURE`-Modul, mit dem *Transmutator Interface* und muss in der *Transmutation Configuration* definiert sein.

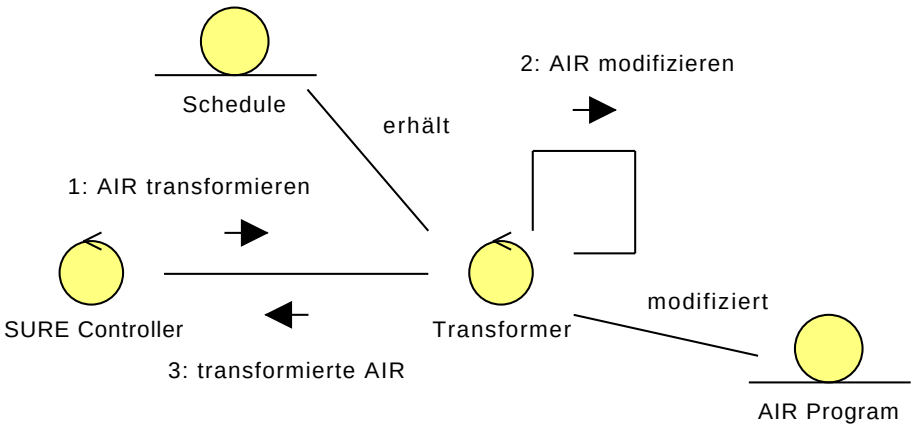


Abbildung 21: Kollaborationsdiagramm für AF6: "Parallele Ausführung"

Die neu hinzugekommenen Analyseklassen sind:

Transmutation Controller Entscheidet gemäß der *Transmutation Configuration* welche Module auf einem *AIR Set* ausgeführt werden und delegiert dessen Weiterverarbeitung entsprechend (siehe [Mir11])

ArBB Backend Zuständig für die Ausführung von parallelisierten *AIR Programs*.

4.7. AF7: SURE konfigurieren

Es existieren mehrere Algorithmen, die für das SURE-Verfahren in Frage kommen. Für zukünftige Arbeiten mit dem Ziel, weitere SURE-Algorithmen zu implementieren, wird deshalb sichergestellt, dass der zu verwendende SURE-Algorithmus von einem Benutzer bzw. Experimentierer ausgewählt werden kann. Da viele Bearbeitungsschritte der Parallelisierungsanalyse von dem verwendeten SURE-Algorithmus abhängen, ist es nicht nötig, eine dynamische Abfrage der Konfiguration zu gewährleisten. Folglich ist der Beginn des Parallelisierungsprozesses mit `SURE` der sinnvollste Zeitpunkt.

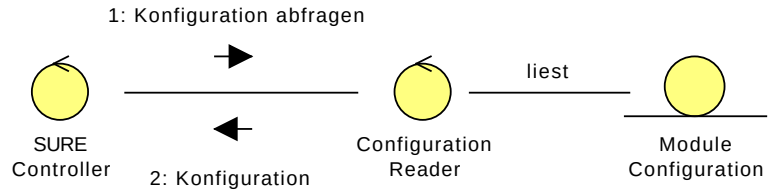


Abbildung 22: Kollaborationsdiagramm für AF7: SURE konfigurieren“

Eine Konfiguration des Moduls setzt sich aus den folgenden, in Abbildung 22 dargestellten Schritten zusammen:

1. Der *SURE Controller* leitet eine Abfrage der Konfiguration durch den *Configuration Reader* ein
2. Der *Configuration Reader* liest die aktuellen Einstellungen aus *Module Configuration* und sendet diese an den *SURE Controller* zurück

Die neu hinzugekommenen Analyseklassen sind:

Configuration Reader Ausschließlich für das Auslesen von Konfigurationsinformationen zuständig

Module Configuration Textdatei, welche alle nötigen Konfigurationsinformationen bereitstellt

5. Entwurf und Implementierung

In diesem Kapitel wird die Struktur des zu entwickelnden `SURE`-PAMs beschrieben. Dabei werden die in Kapitel 4 ausgearbeiteten funktionalen, als auch nichtfunktionalen Anforderungen berücksichtigt und realisiert.
Ähnlich wie im vorhergehenden Kapitel sind die Klassen verschieden eingefärbt:

Gelb Zu erzeugende Artefakte

Orange Bereits vorhandene bzw. externe Artefakte, wie z.B. Bibliotheken

Weiß Unimplementierte Artefakte

Grau Java-fremde Dateien

5.1. ALCHEMY Schnittstelle

Das `SURE`-Modul ist als ein sog. *Transmutator* in die R-Experimentierplattform ALCHEMY implementiert (siehe Abbildung 23). In [Mir11] sind Transmutatoren als spezielle Komponenten definiert, welche für die Transformation von R-Programm-Code zuständig sind.

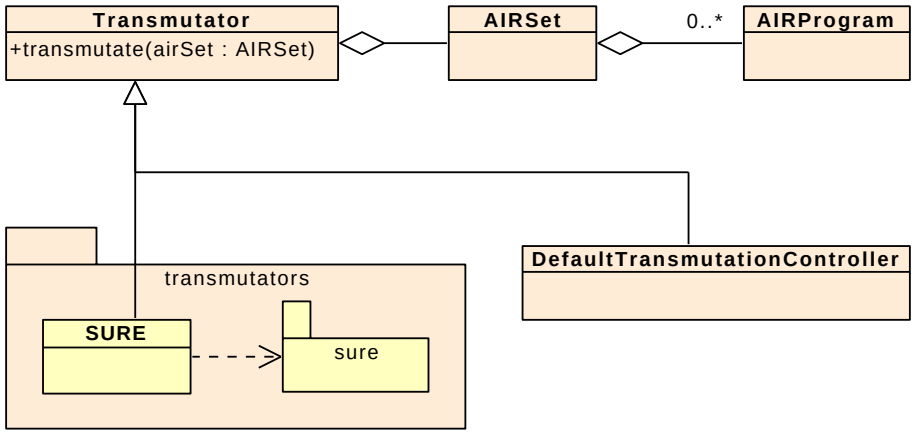


Abbildung 23: Eingliederung in ALCHEMY

Wie in Kapitel 4.1 beschrieben, stellt die Eingabe eines Transmutators ein `AIRSet` dar. Die Transmutationslogik in ALCHEMY realisiert die `DefaultTransmutationController`-Klasse. Die `SURE`-Klasse "implementiert"[6] die Transmutationsschnittstelle `Transmutator`. Hierdurch wird eine Kommunikation zwischen `SURE` und

[6] Erweitert `Transmutator`-Klasse

ALCHEMY ermöglicht. Die Parallelisierungsanalyse mit `SURE` wird schließlich durch die `transmutate()`-Methode initiiert.
`SURE` lässt sich durch einen Eintrag in der ALCHEMY-Konfigurationsdatei in den generellen (ALCHEMY-)Transformationsprozess einplanen.

5.2. Systemarchitektur

Abbildung 24 ist zu entnehmen, dass die `SURE`-PAM in mehrere Unterpakete bzw. -komponenten aufgeteilt wurde. Die `pipeline`-Komponente initiiert und koordiniert die vier Teilschritte des SURE-Verfahrens. Repräsentiert werden diese durch die Komponenten `detector`, `dpanalyzer`, `opscheduler` und `transformer`.

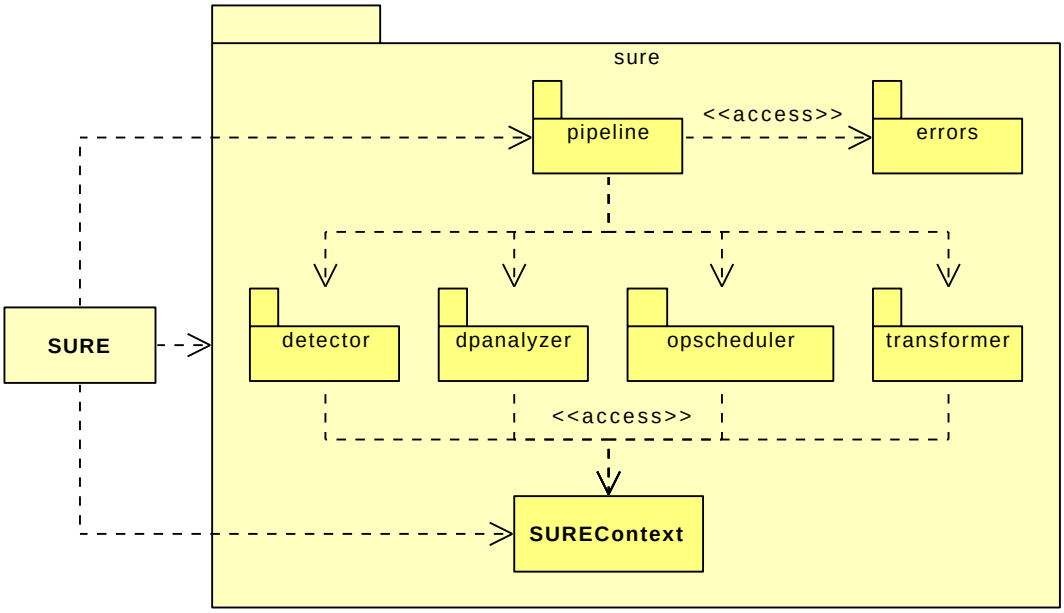


Abbildung 24: Paketdiagramm: Struktureller Überblick

Die Systemarchitektur ist so konzipiert, dass ein Parallelisierungsprozess einfach durch zusätzliche Komponenten erweitert werden kann. Dies wird eingeschränkt nötig sein, falls weitere SURE-Algorithmen implementiert werden sollen.

5.3. Konfiguration von SURE

Bevor die Parallelisierungsanalyse mit `SURE` beginnt, wird die SURE-Konfigurationsdatei ausgelesen. Das entsprechende Klassendiagramm wird durch Abbildung 25 repräsentiert.

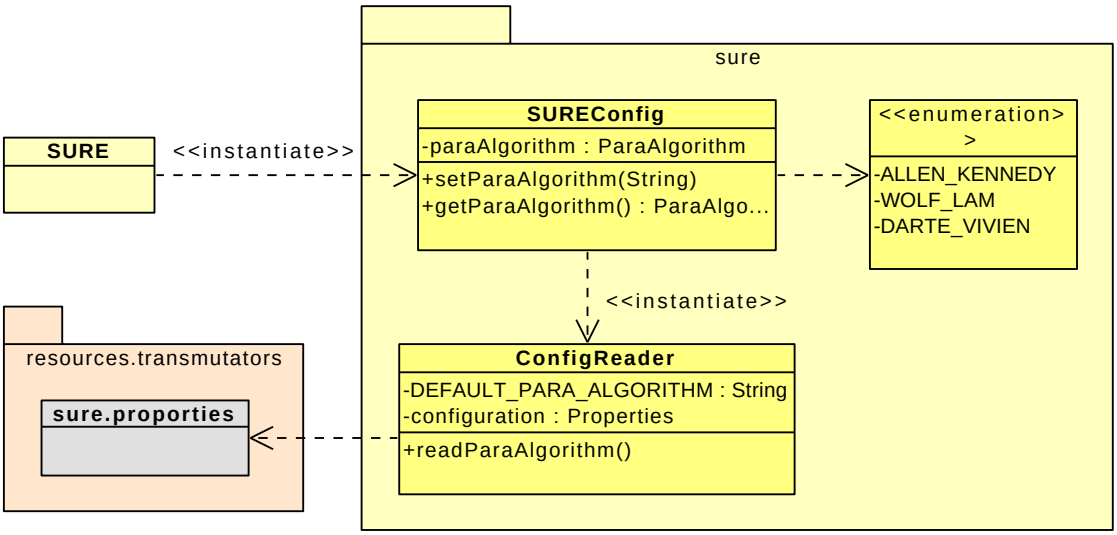


Abbildung 25: Klassendiagramm: SURE-Konfiguration

Für den Benutzer (bzw. ALCHEMY-Experimentierer) besteht die Möglichkeit die Konfiguration anhand der `sure.properties`-Datei zu manipulieren. Bisher kann lediglich der zu verwendende SURE-Algorithmus eingestellt werden. Dabei stehen folgende Optionen zur Verfügung:

- Algorithmus von „Allen und Kennedy“ (Standardeinstellung)
- Algorithmus von „Wolf und Lam“ (Implementierung steht in Aussicht)
- Algorithmus von „Darte und Vivien“ (Implementierung steht in Aussicht)

Das konzeptionelle Aussehen der `sure.properties`-Datei ist wie folgt:

```
para_algorithm     = ( allen_kennedy | wolf_lam | darte_vivien )
some_other_option  = ...
```

Dabei bezeichnet $x \mid y$ eine Auswahl zwischen x und y. Ist die Konfigurationsdatei unlesbar oder nicht vorhanden, so wird die Standardeinstellung verwendet.
Die Klasse `SUREConfig` verwendet einen `ConfigReader` für das Auslesen der relevanten Informationen.

5.4. SURE Pipeline

Der Parallelisierungsprozess des SURE-Verfahrens ist als Pipeline (`Pipe`) in Anlehnung an das Muster „Pipes and Filters“ (siehe Abbildung 26) realisiert:

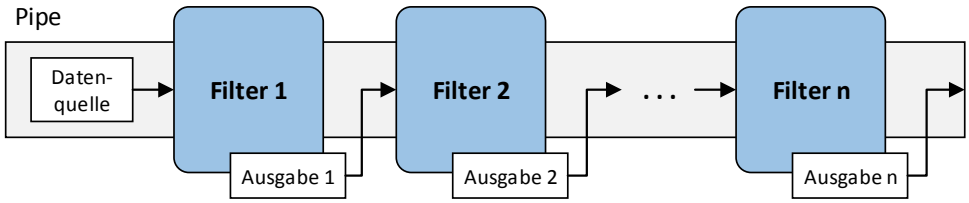


Abbildung 26: Architekturmuster „Pipes and Filters“

Laut [BMR98] ist das angestrebte Ziel ein Modul in mehrere sequentiell verarbeitende Phasen zu unterteilen. Jede Phase entspricht einem `Filter`. Ein `Filter` ist eine Ausführungseinheit, die Eingabe- zu Ausgabedaten verarbeitet. Die Ausgabe des einen `Filters` entspricht der Eingabe des nächsten.
Das Muster bietet sich aufgrund der aufeinander aufbauenden Teilschritte an. Eine Zwischenspeicherung der jeweiligen Ausgaben ist nicht nötig, aber möglich. Es wird eine hohe Flexibilität durch Austausch- und Rekombinierbarkeit von `Filtern` gewährleistet. Dies steht vor allem in Bezug auf die Wiederverwendbarkeit für andere SURE-Algorithmen.
Das „Pipes and Filters“-Muster ist eine Vereinigung der Muster „Chain of Responsibility“ und „Command“ sehr ähnlich. Jedoch unterscheiden sich Ein- und Ausgabe typ nicht von der initialen Eingabe. Im Fall des SURE-Verfahrens werden jedoch grundverschiedene Ein- und Ausgabetypen verarbeitet.

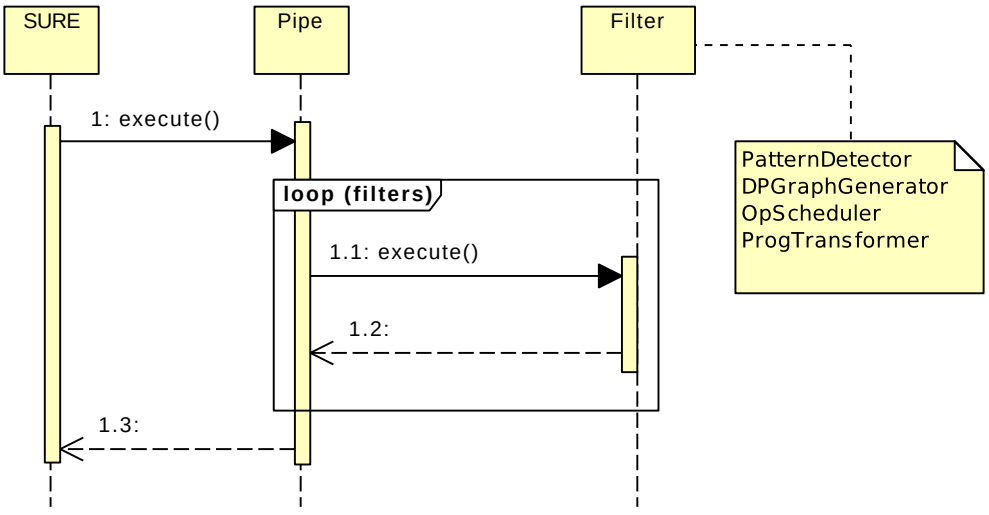


Abbildung 27: Sequenzdiagramm: `Pipe.execute()`

Die Pipeline wird innerhalb der `transmute()`-Methode der Transmutatorklasse `SURE` initialisiert. Das Sequenzdiagramm in Abbildung 28 umfasst alle relevanten Funktionsaufrufe. Die Pipeline-Ausführung wird mittels, der in `Pipe` definierten, `execute()`-Methode eingeleitet (siehe Abbildung 27).

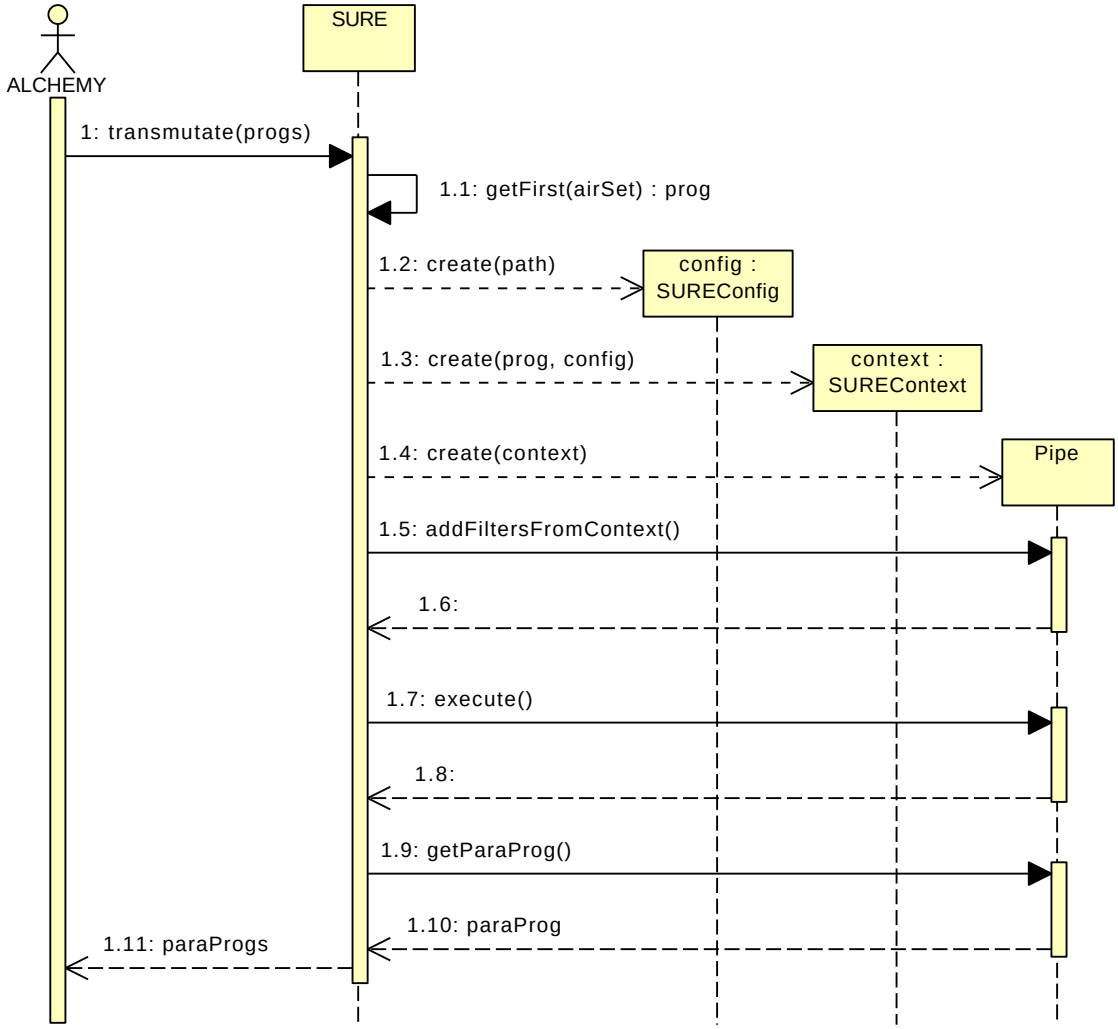


Abbildung 28: Sequenzdiagramm: `SURE.transmutate()`

Wie Abbildung 29 zu entnehmen ist, setzt sich die Pipeline aus der Klasse `Pipe` und der Schnittstelle `Filter` zusammen. Die `Pipe`-Klasse regelt den Datenfluss und koordiniert die anzuwendenden `Filter`. Über die `addFiltersFromContext()`-Methode wird in Abhängigkeit des `SURE`**`Contexts`** entschieden welche `Filter` hinzugefügt werden. Die `SURE`**`Context`**-Klasse ist die geteilte Datengrundlage aller `Filter`. Wird ein `Filter` instanziiert, wird der `SURE`**`Context`** mitübergeben.

Die Entscheidung für einen geteilten Kontext wurde aufgrund folgender Argumente getroffen:

- Es sind keine komplexen Datenkonvertierungen für unterschiedliche Ein- und Ausgaben nötig
- Übersichtliche `Filter`-Schnittstelle
- Geringere Fehleranfälligkeit

Ebenfalls Teil der Pipeline ist die Fehlerbehandlung. Fehler treten hauptsächlich während der Ausführung eines `Filters` auf und werden in einer Liste gesammelt. Vor jeder `Filter`-Ausführung wird auf Fehler überprüft und eine Fehlerbehandlung durchgeführt. Der Grad der auftretenden Fehler bestimmt über Abbruch oder Fortsetzung der Parallelisierungsanalyse.

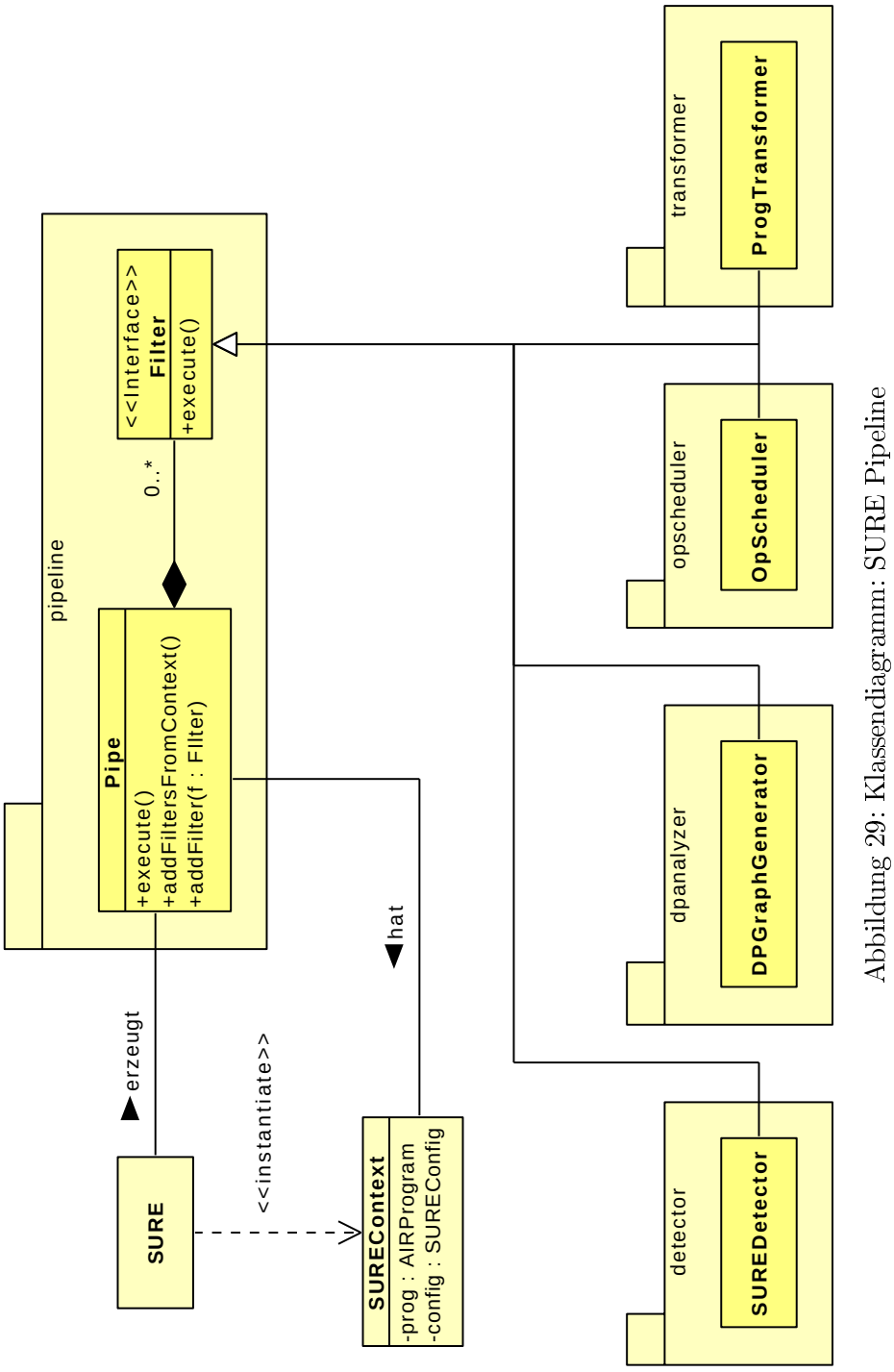


Abbildung 29: Klassendiagramm: SURE Pipeline

5.5. ANTLR Integration

In Kapitel 2.3 wird ein grober Überblick der Arbeitsweise von ANTLR vorgestellt. ANTLR wird von `SURE`-Komponenten `detector` und `dpanalyzer` verwendet. Einerseits für die Erkennung von SUREs und andererseits als Teil der Datenabhängigkeitsanalyse. In Abbildung 31 ist das Zusammenspiel zwischen `SURE` und der verwendeten ANTLR-Klassen dargestellt.

Die Grammatikdatei `AIRGrammar.g4` enthält die Spezifikation für die Erzeugung eines Lexers (`AIRGrammarLexer`) und Parsers (`AIRGrammarParser`). Zusätzlich lässt sich nach Belieben die Klasse `AIRGrammarListener` erzeugen. Als Datenquelle dient die XML-Repräsentation eines `AIRprograms` (bzw. eines `AIRprogram`-Ausschnitts).

Die vereinfachten Parser-Regeln sind wie folgt:

`element` : $<$ *Name* `attribute`* $>$ `element`* $</$ *Name* $>$
| $<$ *Name* `attribute`* $/>$
`attribute` : *Name* $=$ *Value*

Name, *Value* und Sonderzeichen stehen dabei für Tokenbezeichner.

Die Klassen `AIRGrammarLexer`, `AIRGrammarParser`, `AIRGrammarListener` und `AIRGrammarBaseListener` werden automatisch auf Basis der `AIRGrammar.g4`-Datei erstellt. Ein `ANTLRInputStream` konvertiert die Datenquelle (z.B. eine Zeichenfolge) in ein für den `AIRGrammarLexer` lesbares Eingabeformat. `AIRGrammarLexer` wiederum interpretiert die Eingabe und erzeugt Tokens, die in einem `CommonTokenStream` hinterlegt werden. Schließlich generiert eine `Parser`-Instanz auf Basis des `CommonTokenStream` einen Parsebaum. Zugriff auf den Parsebaum wird über die Schnittstelle `ParseTree` bereitgestellt. Ein anschauliches Beispiel eines barsebaums ist Abbildung 30 zu entnehmen.

Nun ist es möglich mit Hilfe der `ParseTreeWalker`-Klasse den `ParseTree` via `walk()`-Methode zu durchlaufen. Dabei werden Ereignisse ausgelöst, die mittels einer Erweiterung der `AIRGrammarBaseListener`-Klasse abgehört werden können. Dieser Mechanismus wurde unter anderem für die Implementierung der Erkennungslogik für das SURE-Muster verwendet.

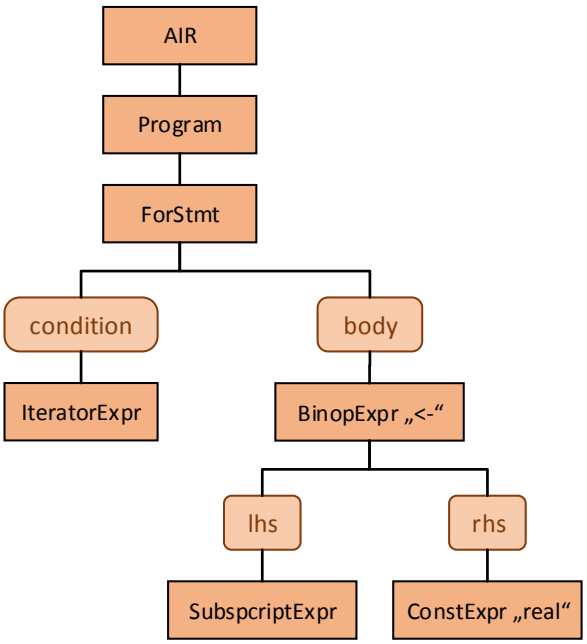


Abbildung 30: Vereinfachte Darstellung eines `ParseTrees`

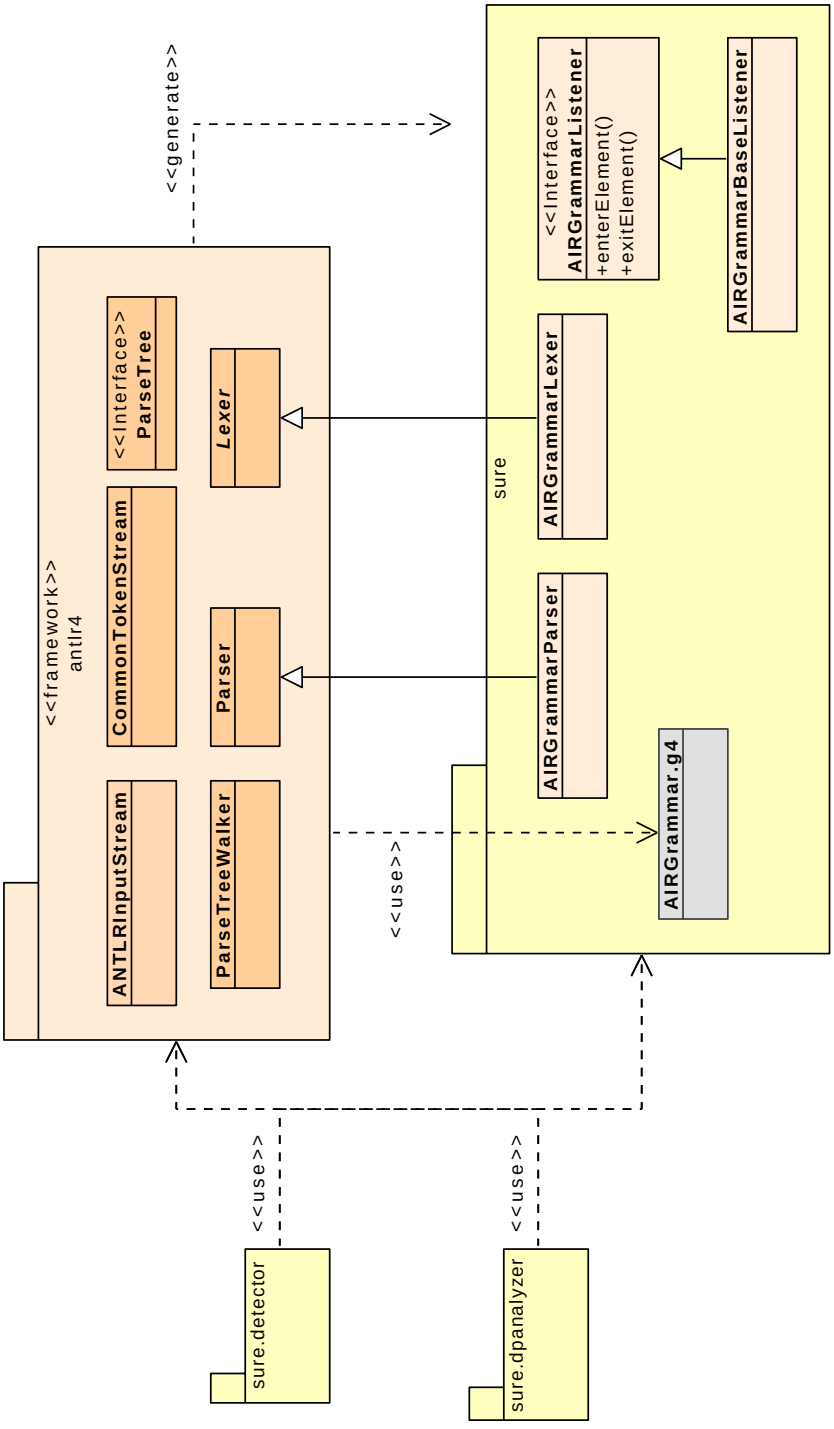


Abbildung 31: Klassendiagramm: ANTLR Integration im SURE-PAM

5.6. Komponenten des Parallelisierungsprozesses

Während das letzte Kapitel einen Überblick über die generelle Struktur und die Integration des Parsergenerators ANTLR bereitstellt, wird in diesem Kapitel nun der eigentliche Kernteil des Parallelisierungsprozesses mit dem SURE-Verfahren behandelt. Dabei wird in den jeweiligen Unterkapiteln detailliert auf die einzelnen Phasen eingegangen, welche durch den Algorithmus von Allen und Kennedy vorausgesetzt werden. Der allgemeine Datenfluss ist der Übersicht halber noch einmal in Abbildung 32 dargestellt:

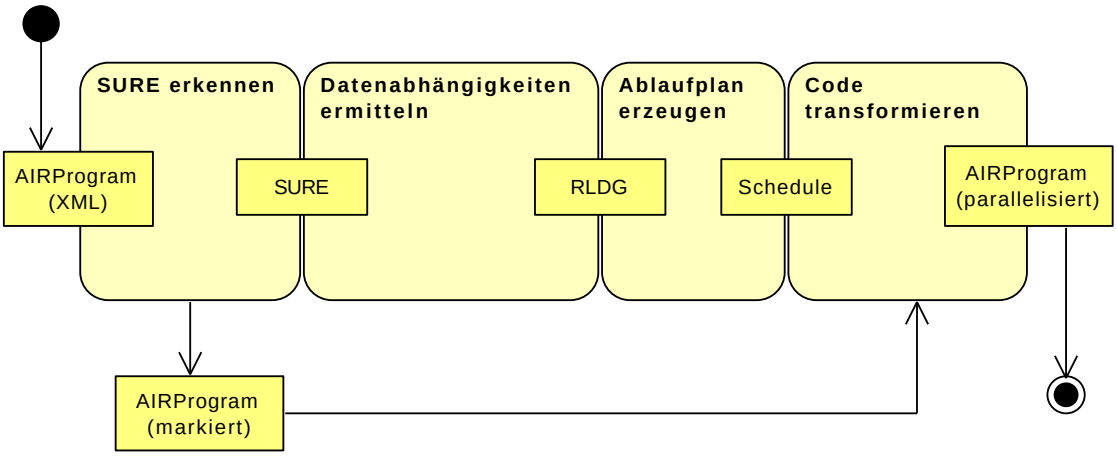


Abbildung 32: Datenfluss in `SURE`

5.6.1. SURE-Erkennung

Eingabe: `AIRprogram` in XML-Repräsentation

Aufgabe: Finden und Extrahieren aller SURE-Vorkommen

Ausgabe: Liste von SURE Code-Stücken und ein markiertes `AIRprogram`

Gestartet wird der Erkennungsprozess (wie alle abgeleiteten Klassen der `Filter`-Schnittstelle) durch die `execute()`-Methode der Controller-Klasse `PatternDetector` (siehe Abbildung 33).

Es werden so lange weitere Parsevorgänge via `inspect()`-Methode eingeleitet bis keine neuen SUREs mehr erkannt werden. Wird ein SURE erkannt, so wird dieses aus dem `AIRprogram` extrahiert und durch eine entsprechende Markierung ersetzt. Durch einen Aufruf der `inspect()`-Methode wird ferner die Zuhörerklasse `DetectProcessor` instanziiert. Dieser Vorgang ist in Abbildung 34 noch einmal genauer

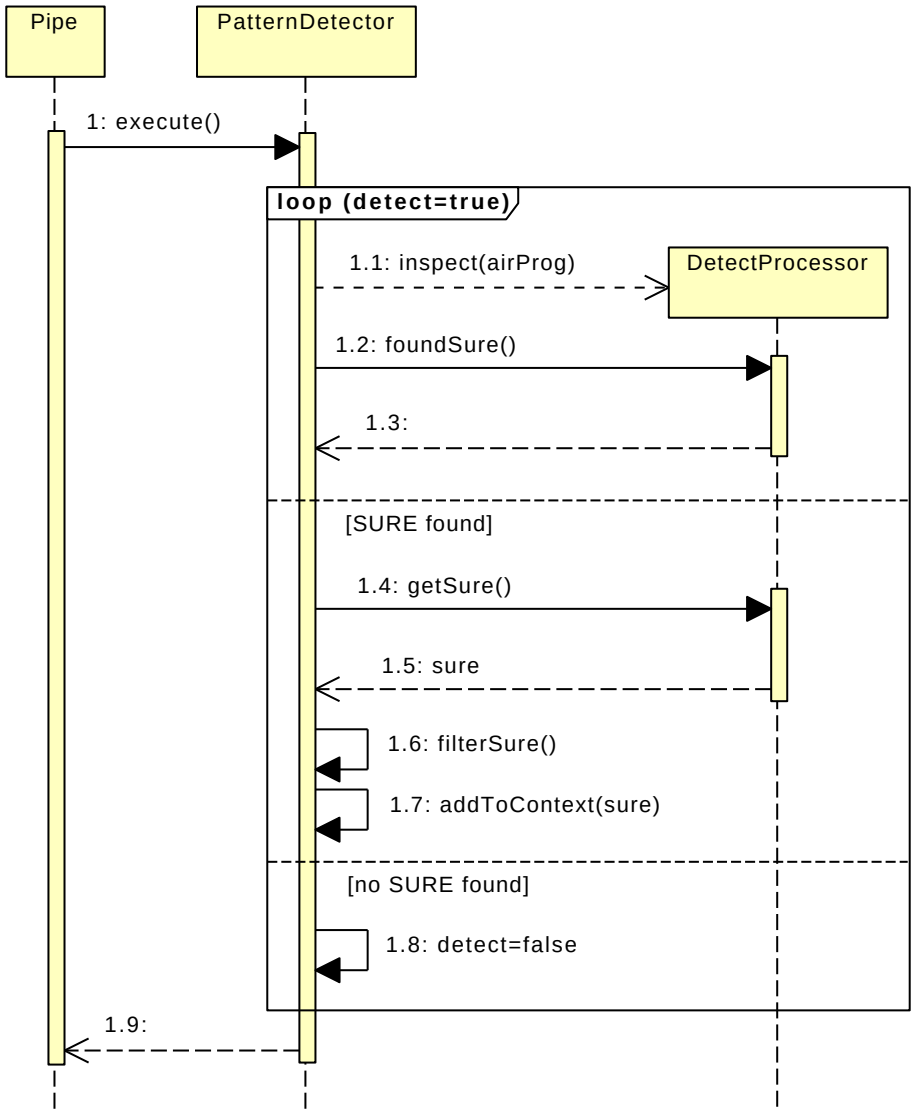


Abbildung 33: Sequenzdiagramm: `PatternDetector.execute()`

beschrieben.

Die `DetectProcessor`-Klasse erweitert die automatisch generierte Klasse `AIRGrammarBaseListener` und umfasst die Logik für die SURE-Erkennung. Während der Parsebaum durchlaufen wird, werden beim Betreten und Verlassen der Baumknoten Ereignisse ausgelöst. Definiert sind diese Baumknoten durch die Parser-Regeln. Die `DetectProcessor`-Klasse fängt die besagten Ereignisse durch überschreiben der `enterElement()`- und `exitElement()`-Methoden ab. An diesen Stellen werden Informationen über das eingegebene `AIRprogram` gesammelt. Sobald

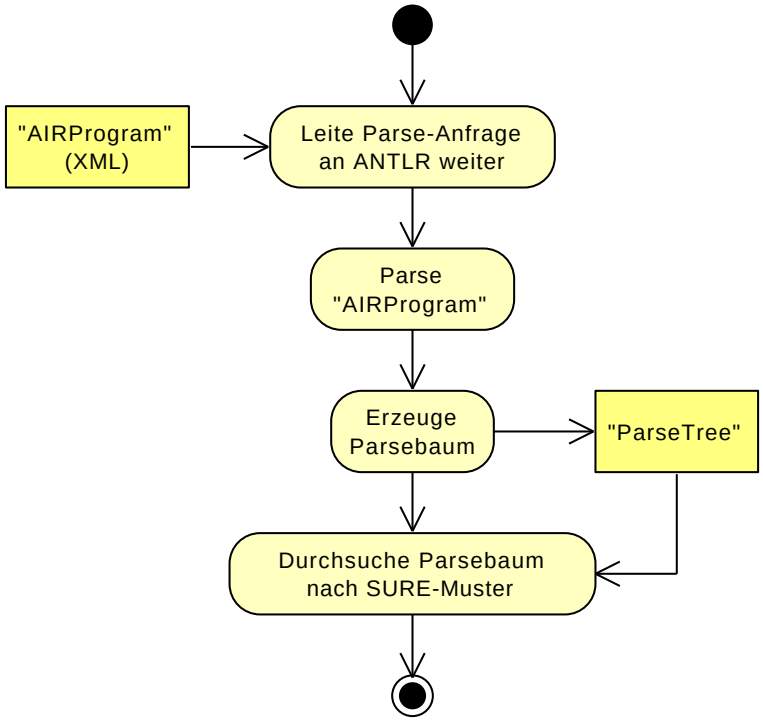


Abbildung 34: Aktivitätsdiagramm: `PatternDetector.inspect()`

eine `for`-Schleife erkannt wird, wird ein SURE-Vorkommen verdächtigt und dem entsprechend ein `SuspectedSure`-Objekt erzeugt. Dieses umfasst unter anderem den vollständigen Code-Block (inkl. Rumpf mit weiteren möglichen SUREs), einen Identifizierer und die Markierung `isCompatible`. Die Variable `isCompatible` stellt sicher, dass dieses "verdächtige SURE" nur Matrix-modifizierende Operationen enthält.
In Abbildung 35 ist dargestellt, wie die restlichen Voraussetzungen 1 und 2 aus Kapitel 3.2 in Folge geprüft werden.

Bei einem Fund wird das kompatible SURE als `DefiniteSure` in `SUREContext` gespeichert. An dessen Stelle im `AIRprogram` wird eine sogenannte `MarkExpr` (*AIR Expression* aus ALCHEMY) eingefügt:

```
<MarkExpr name="sure1" \>
```

Das Attribut *name* dient als Identifikation (hier: für SUREs). Außer den extrahierten SUREs wird außerdem das markierte `AIRprogram` im `SUREcontext` hinterlegt.

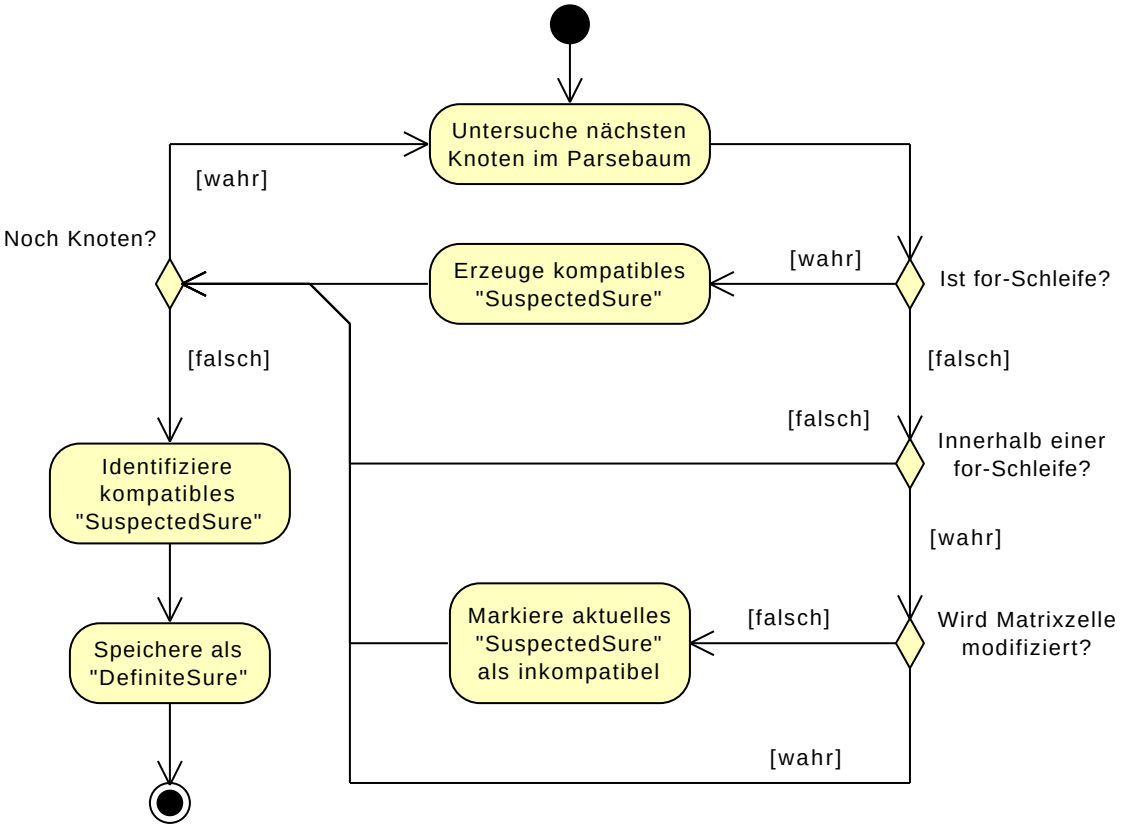


Abbildung 35: Aktivitätsdiagramm: `DetectProcessor.enterElement()`

Das Zusammenspiel aller genannten Klassen ist in Abbildung 36 veranschaulicht.

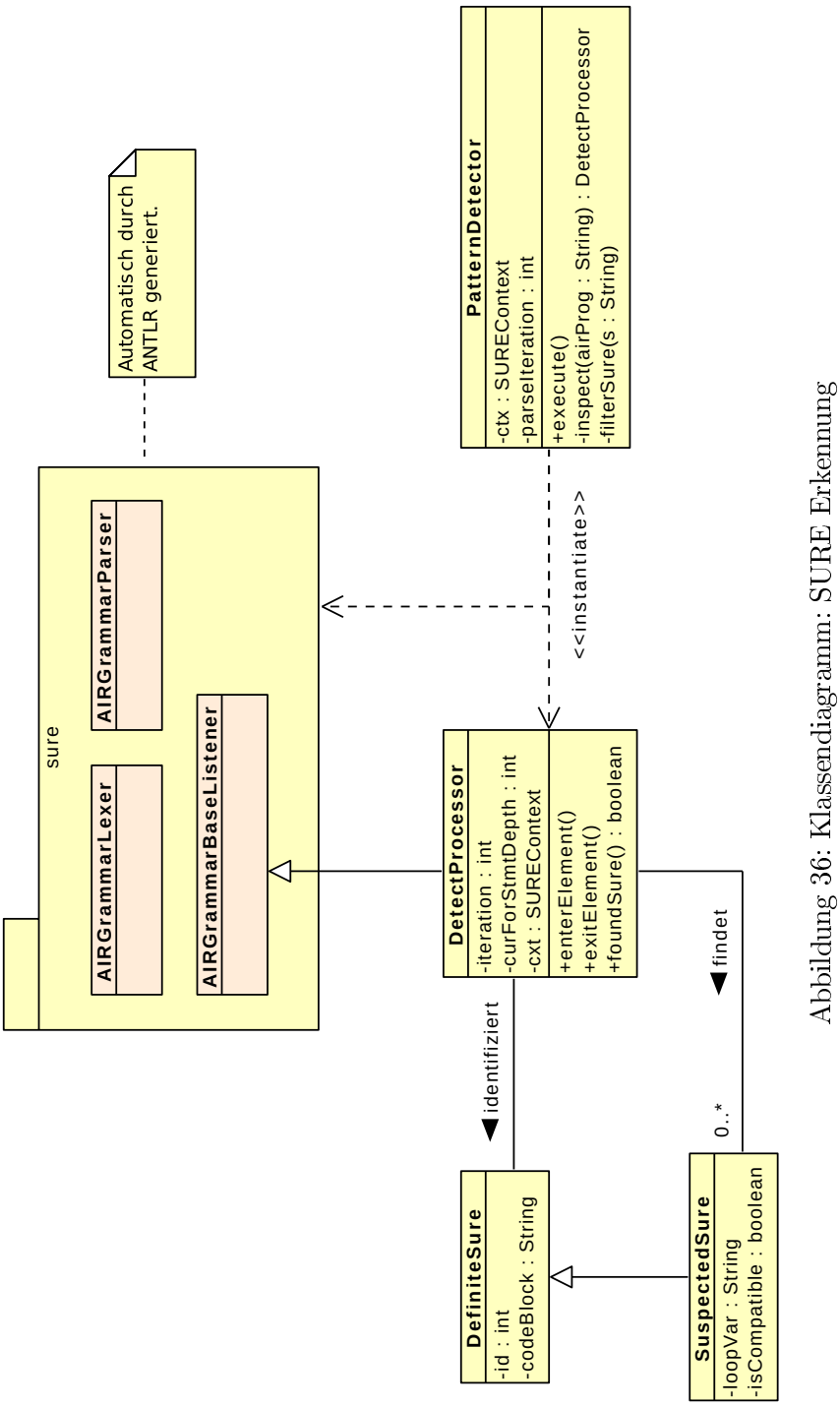


Abbildung 36: Klassendiagramm: SURE Erkennung

5.6.2. Datenabhängigkeitsanalyse

Eingabe: `DefiniteSure` in XML-Repräsentation

Aufgabe: Erzeugen eines RLDGs

Ausgabe: Ein `RLDG`

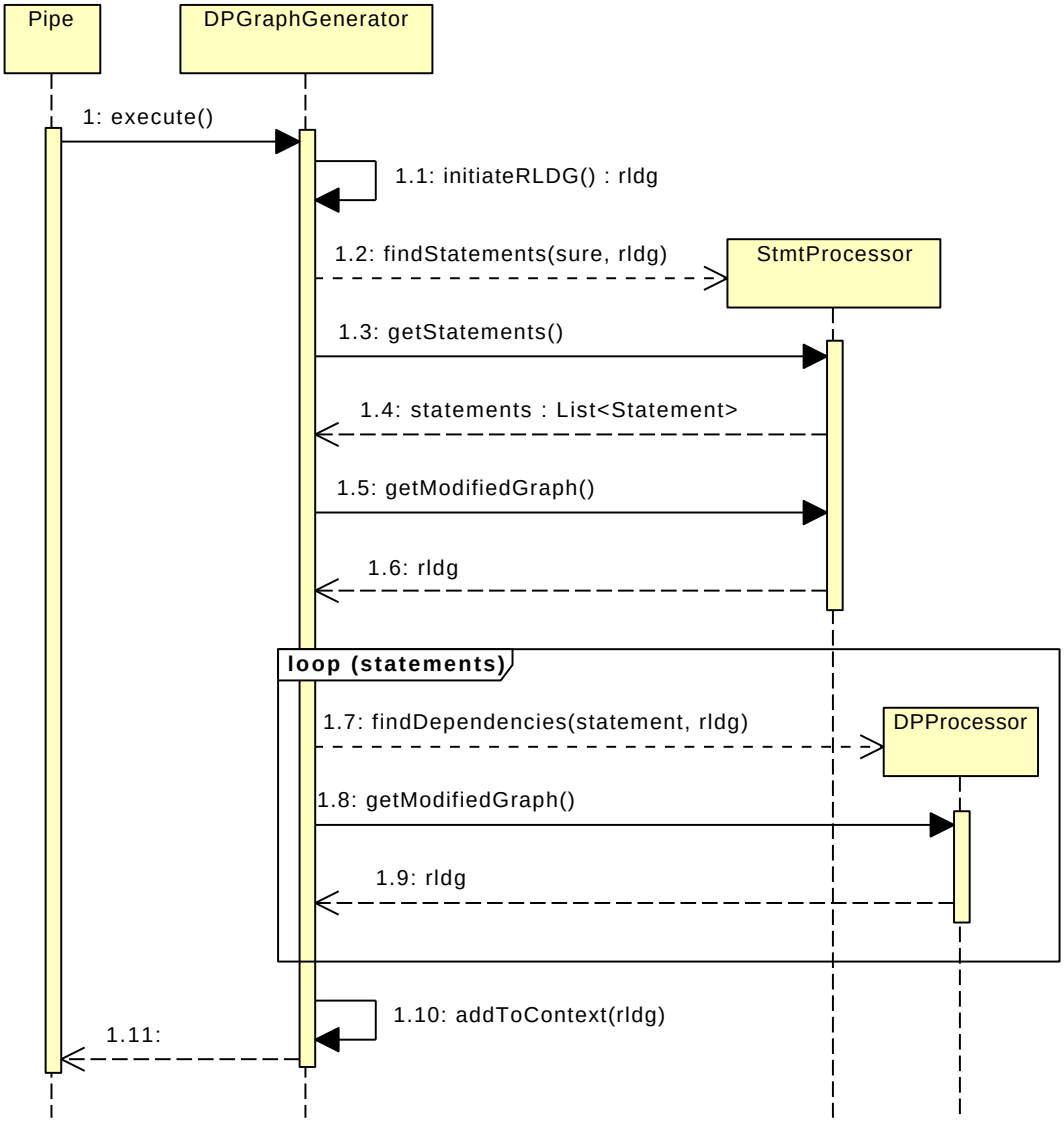


Abbildung 37: Sequenzdiagramm: `DPGraphGraphgenerator.execute()`

Die Ausführung der `execute()`-Methode (siehe Abbildung 37) spaltet den Erstellungsprozess des RLDGs in 2 Teile:

1. Hinzufügen der Knoten (Statements)
2. Hinzufügen der Kanten (Datenabhängigkeiten)

Diese entsprechen den Methoden `findStatements()` und `findDependencies()`. In beiden Fällen wird erneut auf ANTLR zurückgegriffen. Die `findStatements()`-Methode parst die XML-Repräsentation der `DefiniteSures` und extrahiert alle `Statements`. Anschließend werden diese als Knoten dem `RLDG` hinzugefügt. Dieser Vorgang wird durch das Aktivitätsdiagramm in Abbildung 38 dargestellt. Falls nicht der Algorithmus von Allen und Kennedy ausgewählt wurde, werden die `Statements` analog einem `RVDG` hinzugefügt.

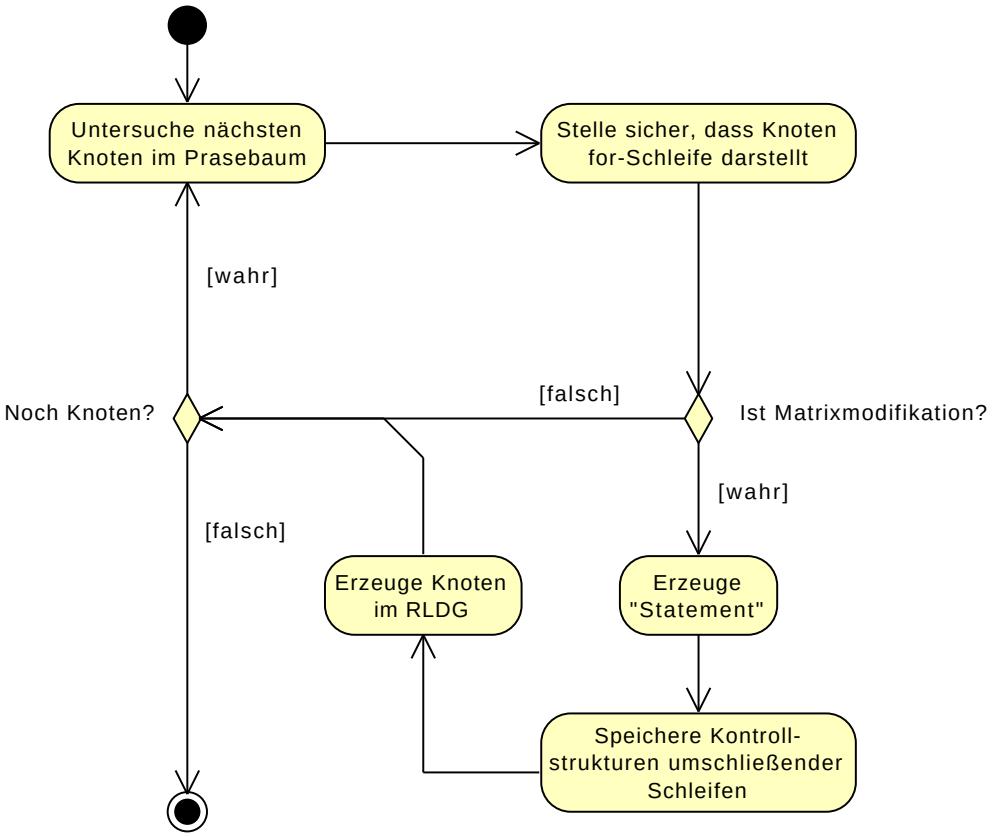


Abbildung 38: Aktivitätsdiagramm: `StmtProcessor.enterElement()`

Die `findDependencies()`-Methode wiederum parst die XML-Repräsentation der extrahierten `Statements`. Diese werden anhand der in Kapitel 3.3 erwähnten Datenabhängigkeitstests analysiert. Für das `SURE`-Modul wurde lediglich ein ZIV- und ein SIV-Test entprechend der Beschreibung aus [JS03] implementiert. Die aus

den Tests resultierenden Datenabhängigkeiten werden schließlich als Kanten dem `RLDG` hinzugefügt. Dieser Vorgang wird ebenfalls durch ein Aktivitätsdiagramm in Abbildung 40 zusammengefasst.

Die Klassen `StmtProcessor` und `DPProcessor` fungieren als Zuhörer mit der bereits erläuterten Erkennungslogik der Methoden `findStatements()` und `findDependencies()`. An dieser Stelle wird erneut zwischen den SURE-Algorithmen unterschieden. Der `DPLevelProcessor` berechnet Abhängigkeitsgrade für einen `RLDG`. Die noch unimplementierte `DPVectorProcessor`-Klasse berechnet Richtungsvektoren (siehe Anhang B) für einen `RVDG`. Abbildung 39 repräsentiert eine Übersicht der RDG-Datenstrukturen.

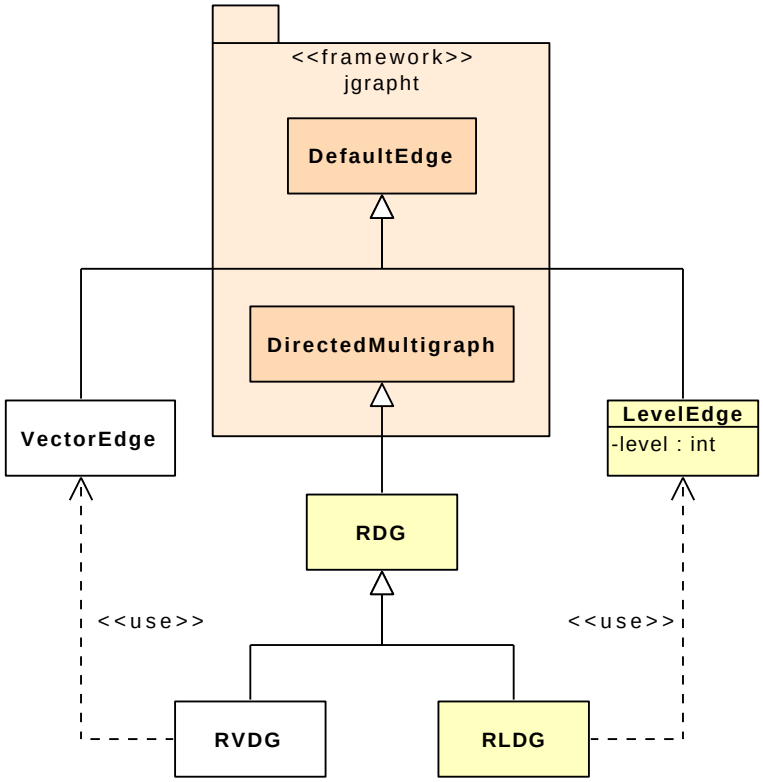


Abbildung 39: Klassendiagramm: Datenstruktur der Graphen

Schließlich wird der vollständige RLDG im `SURE Context` hinterlegt und die Datenabhängigkeitsanalyse für das nächste `DefiniteSure` beginnt.
Weitere Klassendiagramme für die Datenabhängigkeitsanalyse sind Abbildung 42 bzw. Abbildung 41 zu entnehmen.

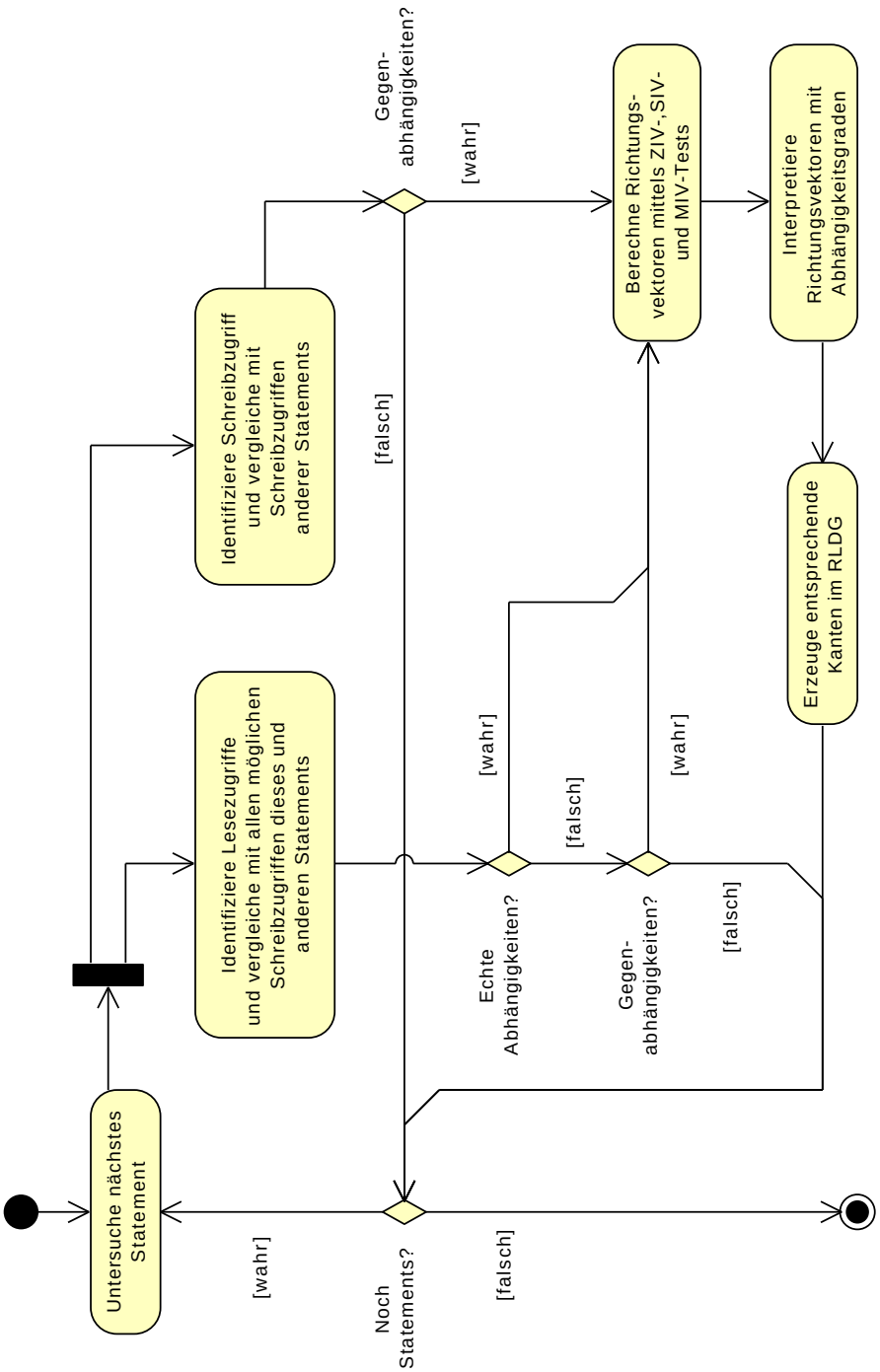


Abbildung 40: Aktivitätsdiagramm: `DPLevelProcessor.enterElement()`

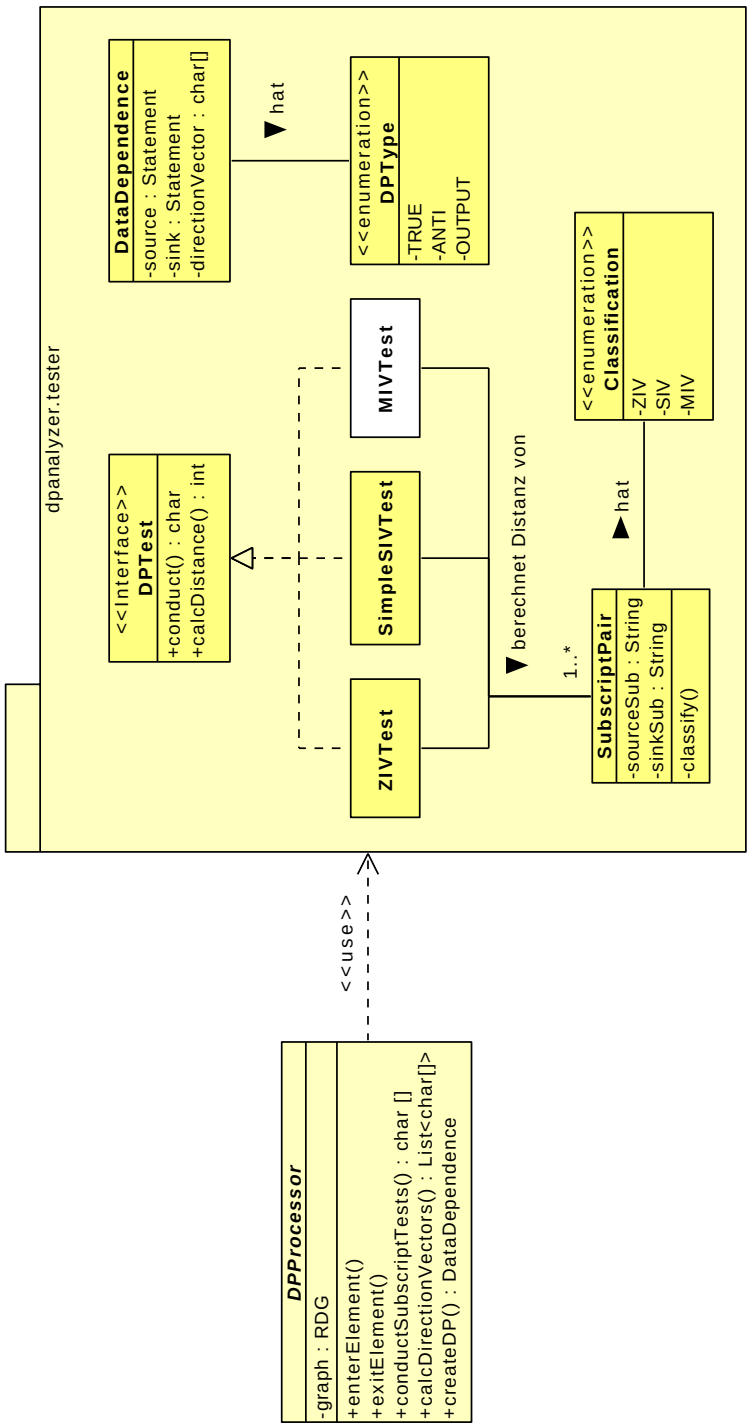


Abbildung 41: Klassendiagramm: Datenabhängigkeitstests

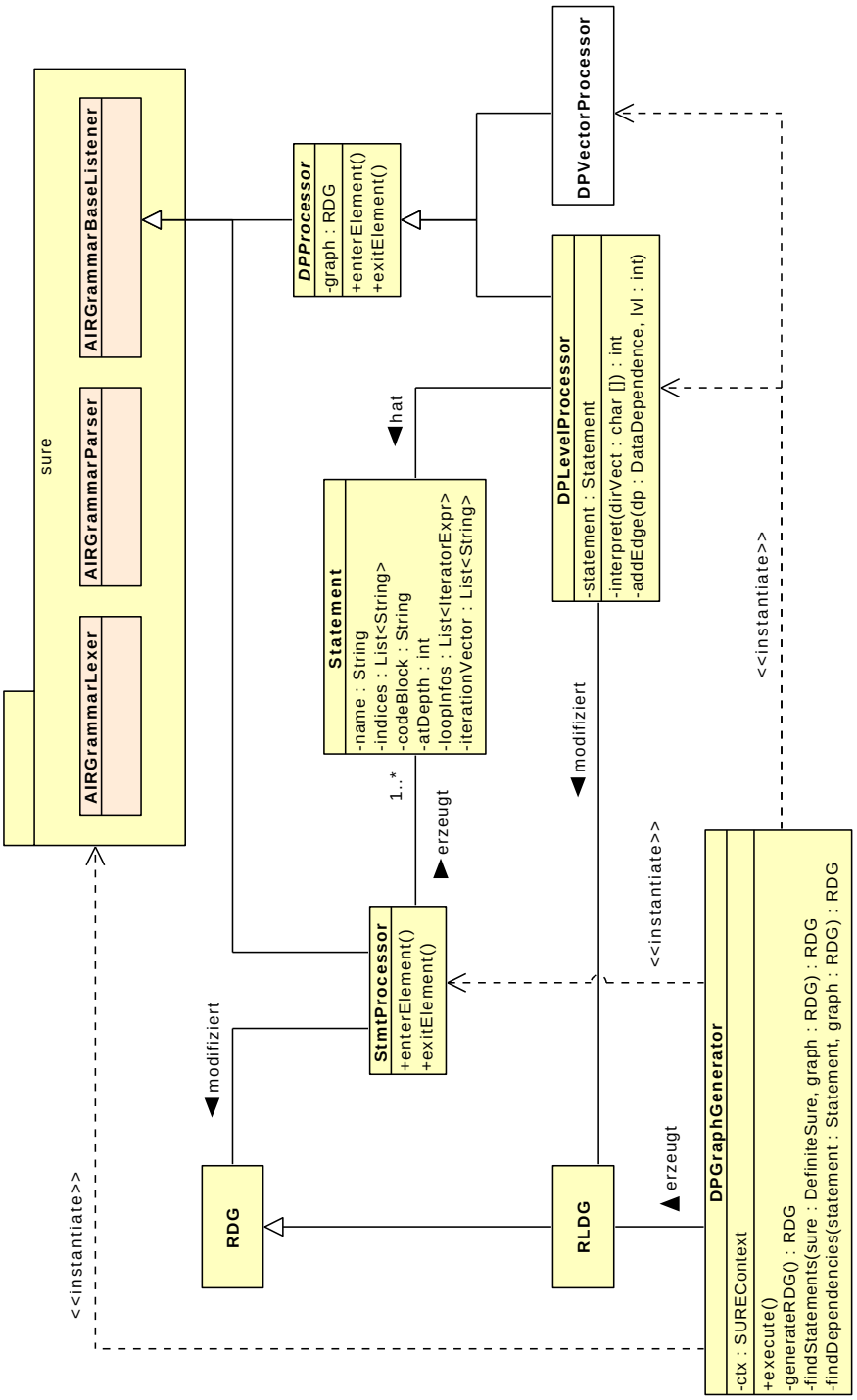


Abbildung 42: Klassendiagramm: Datenabhängigkeitsanalyse

5.6.3. Ablaufplanung

Eingabe: Ein `RLDG`

Aufgabe: Erzeugen eines Ablaufplans

Ausgabe: Ein `Schedule`

Die `execute()`-Methode ermittelt den anzuwendenden SURE-Algorithmus und initiiert die Ablaufplanung (siehe Abbildung 43). In diesem Fall wird der Algorithmus von Allen und Kennedy erkannt und eine `AKAlgorithm`-Instanz generiert. Dessen `run()`-Methode wird nun auf allen RDG-Eingaben (hier: RLDG-Eingaben) ausgeführt.

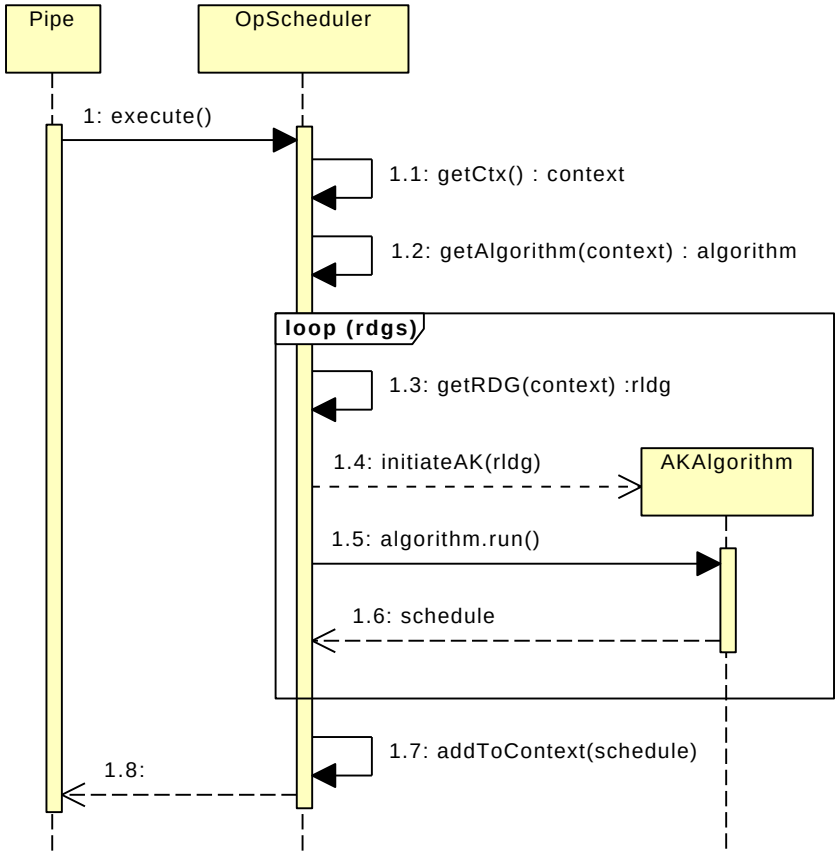


Abbildung 43: Sequenzdiagramm: `OpScheduler.execute()`

Die Klasse `AKAlgorithm` implementiert die Ausführung des Algorithmus von Allen und Kennedy. Die anderen (beiden) SURE-Algorithmen sind entsprechend in `WL-Algorithm` und `DVAlgorithm` zu implementieren. Durch Vorgabe der definierten

Schnittstelle `SUREAlgorithm` wird in jedem Fall ein `Schedule` erzeugt. Abbildung 44 zeigt die Realisierung des in Kapitel 4.4 definierten Ablaufplans:

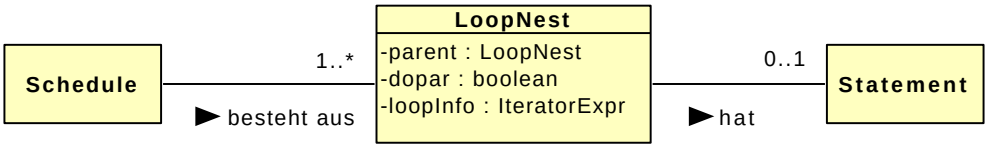


Abbildung 44: Klassendiagramm: `Schedule`-Datenstruktur

Ein `Schedule` setzt sich aus mindestens einem Schleifennest (`LoopNest`) zusammen. Das `loopInfo`-Attribut speichert die Kontrollstrukturen (Indexvariable, Abbruchbedingung) der repräsentierten `for`-Schleife. Durch das `parent`-Attribut wird stets eine Referenz auf das beinhaltende Eltern-`LoopNest` gespeichert.
Ein `LoopNest` kann ebenfalls ein `Statement` enthalten. Hiermit wird Bedingung 1 aus Kapitel 4.4 erfüllt. `LoopNests` werden in einer Prioritätsliste gehalten, damit die korrekte Anordnung sichergestellt ist (vergleiche Bedingung 2). Durch das Attribut `dopar` wird festgelegt, ob die Schleife parallel ausgeführt werden kann (vergleiche Bedingung 3).

Der in Kapitel 3.4 vorgestellte Algorithmus von Allen und Kennedy (dargestellt in Abbildung 11) beinhaltet bereits die Code-Transformation. Es mussten Anpassungen vorgenommen werden, sodass an Stelle des parallelisierten SUREs ein Ablaufplan berechnet wird. Abbildung 45 stellt die angepasste Arbeitsweise dar mit:

k	die Indexvariable der aktuellen Schleifentiefe
N	maximale Schleifentiefe des Statements im aktuell betrachteten (Sub-)Graphen

Eine Übersicht der Klassenstruktur ist Abbildung 46 zu entnehmen. Die gesamte Graphenfunktionalität wird durch Bibliothek `JGraphT` bereitgestellt und von allen SURE-Algorithmen in unterschiedlichem Ausmaß genutzt. Darunter fällt z.B. die Berechnung der SCCs oder das topologische Sortieren.

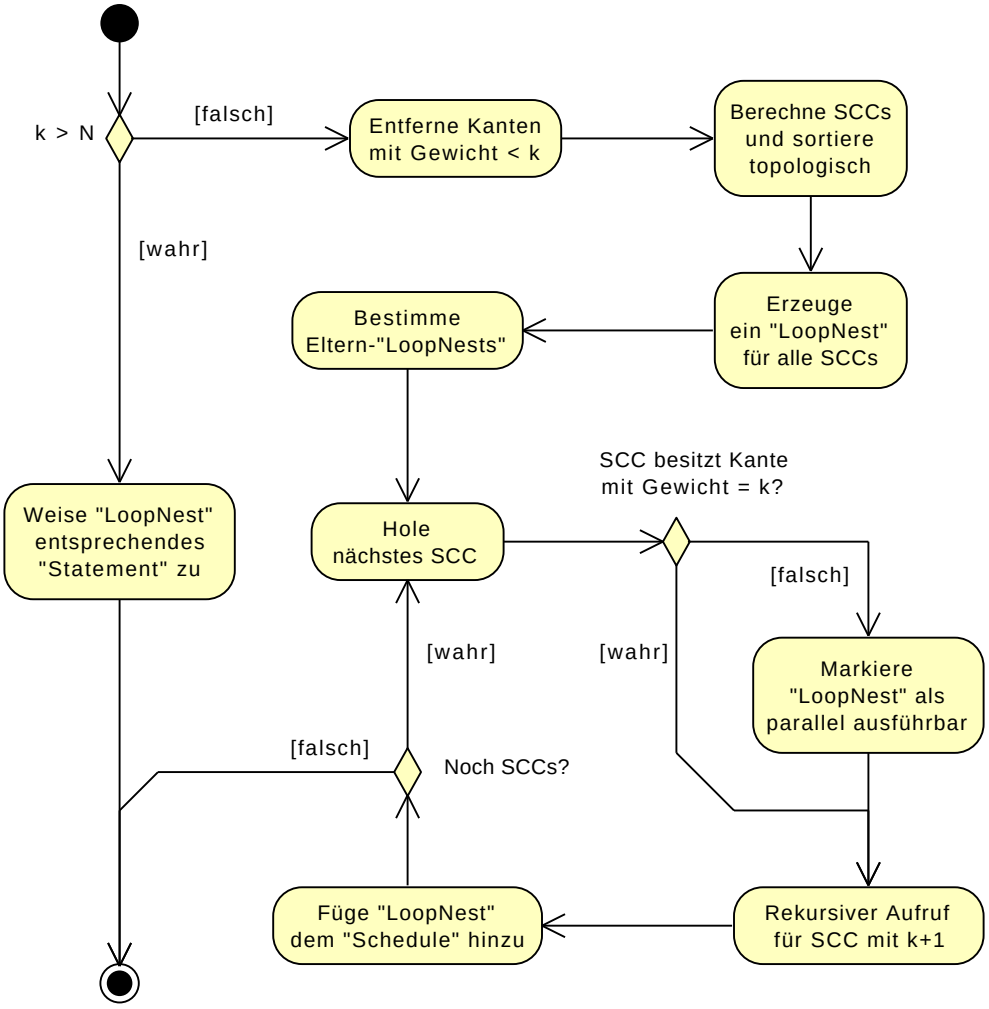


Abbildung 45: Sequenzdiagramm: `AKAlgorithm.run()`

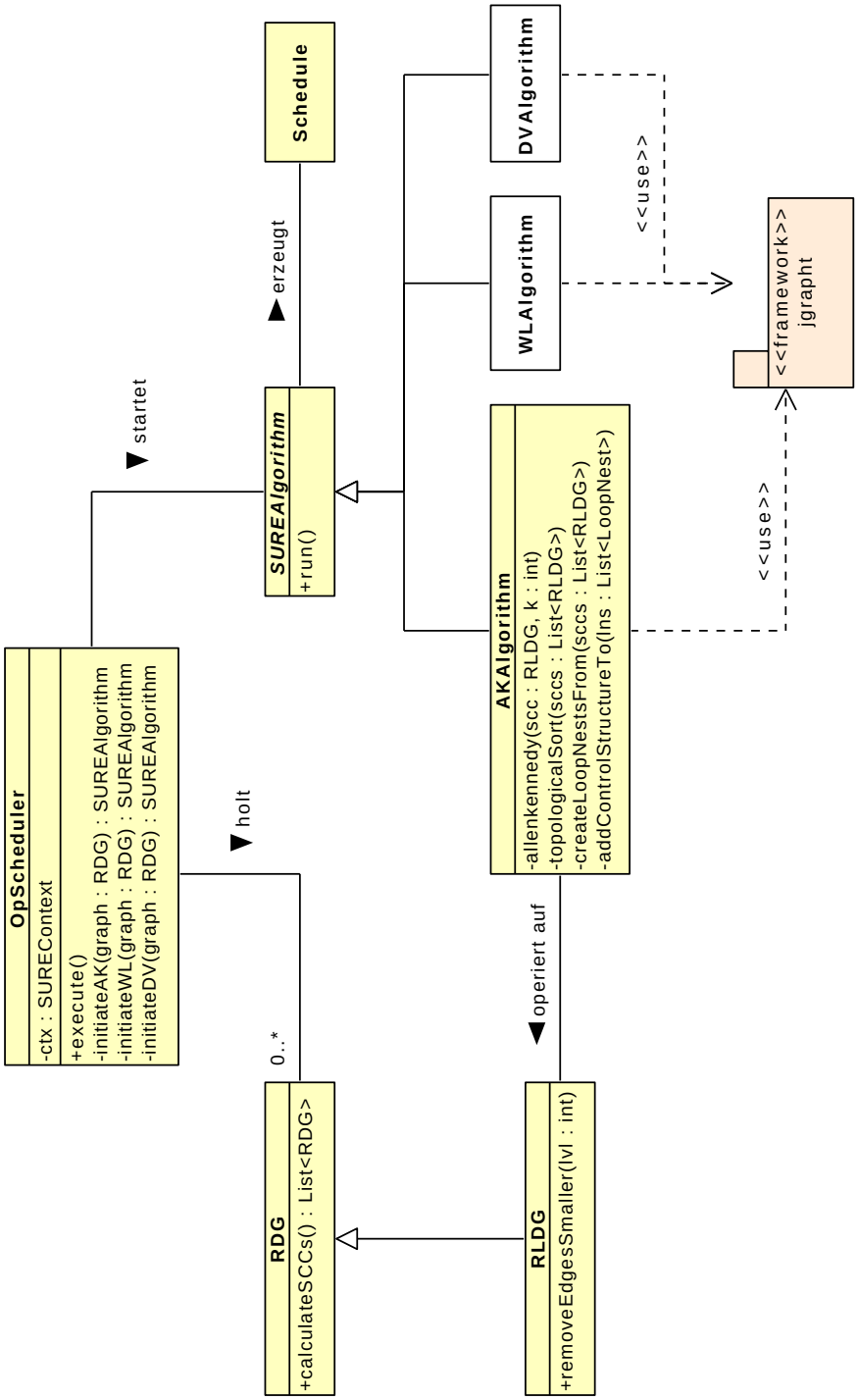


Abbildung 46: Klassendiagramm: Erzeugen eines Ablaufplans

5.6.4. AIR Transformation

Eingabe: Ein `Schedule` und das markierte `AIRprogram`

Aufgabe: Erzeugen eines parallel ausführbaren AIR-Programms

Ausgabe: Ein parallel ausführbares `AIRprogram`

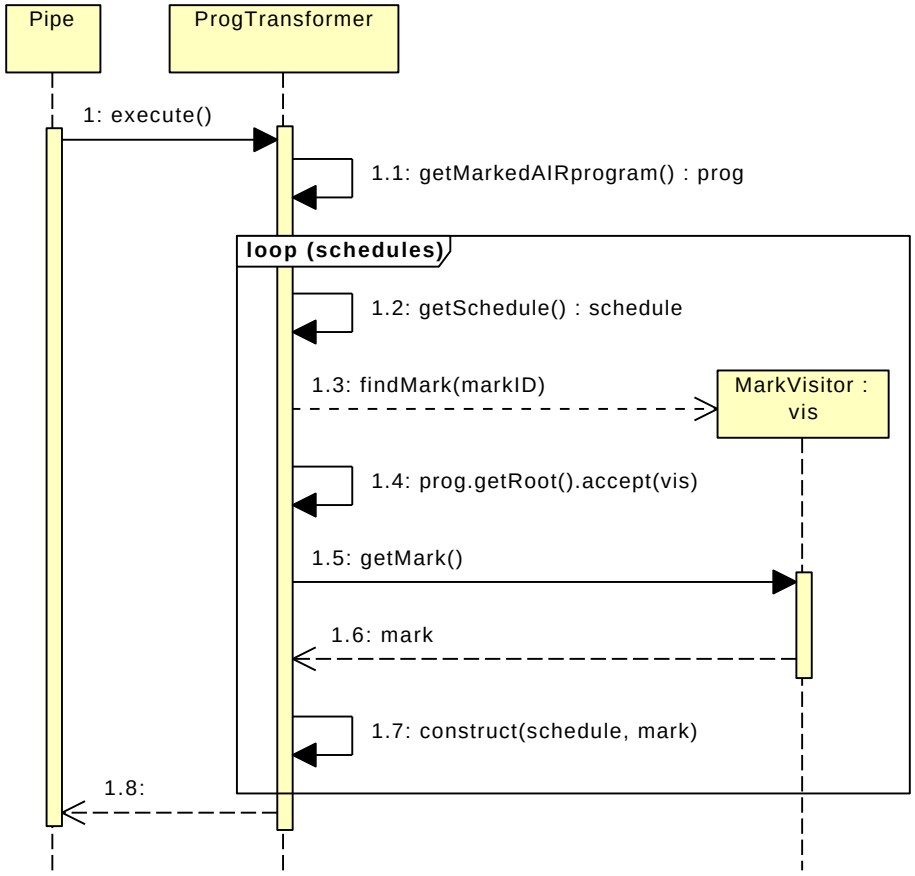


Abbildung 47: Sequenzdiagramm: `ProgTransformer.execute()`

Als Teil der `execute()`-Methode (siehe Abbildung 47) wird

1. der nächste `Schedule` aus dem `SUREContext` geholt,
2. die entsprechende Markierung im markierten `AIRprogram` gesucht,
3. aus dem `Schedule` eine parallelisierte Version des SUREs konstruiert und
4. die Markierung im `AIRprogram` durch das parallele SURE ersetzt

Für das Auffinden der Markierung wird die Besucher-Klasse `MarkVisitor` verwendet. Zurückgeliefert wird eine `MarkExpr`, die nun mittels `construct()`-Methode durch ein parallel ausführbares SURE-Konstrukt ersetzt wird. Die `construct()`-Methode, dargestellt in Abbildung49, definiert die Struktur des parallel ausführbaren SUREs. Zuerst werden die äußersten Schleifen (im Diagramm: *roots*) ermittelt. Diese entsprechen den `LoopNests` ohne Eltern. Anschließend wird ein rekursiver Bauprozess eingeleitet. Die "Übersetzung" eines `LoopNests` in ein Schleifenkonstrukt ist in Abbildung 48 veranschaulicht.

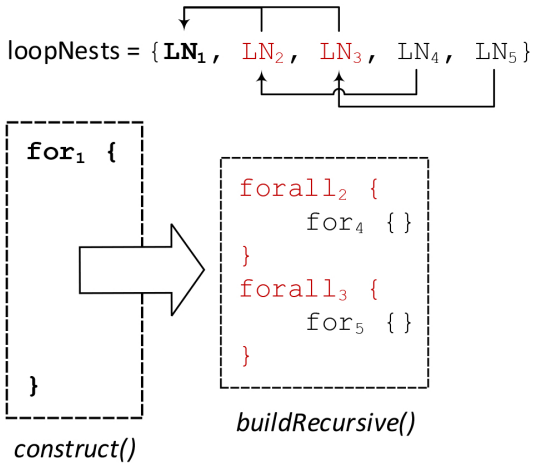


Abbildung 48: Rekursiver Bauprozess eines parallelen SUREs

Hierbei stellen die (schwarzen) Pfeile die Elternbeziehungen und die rot eingefärbten `LoopNests` eine *dopar*-Markierung dar. Der Bauprozess wird immer in Abhängigkeit einer vorhandenen *dopar*-Markierung eingeleitet:

vorhanden: `DOPARSkeletonBuilder` erbaut für das `LoopNest` eine parallel ausführbare Schleife (ein `DOPARSkeleton`)

abwesend: `ForStmtBuilder` erbaut für das `LoopNest` eine sequentielle Schleife (ein `ForStmt`)

Schließlich wird das erbaute parallele SURE an Stelle der `MarkExpr` in das `AIR-program` eingesetzt. Der bereits angedeutete rekursive Bauprozess ist in Abbildung 50 dargestellt. Dabei wird in jedem Aufruf ein entsprechendes Schleifenkonstrukt erzeugt. Entweder ein `ForStmt` oder ein `DOPARSkeleton`. Solange das aktuell betrachtete `LoopNest` weitere `LoopNests` als Kinder hat, wird dieser Prozess rekursiv wiederholt. Die neu erbauten Schleifenkonstrukte werden jeweils dem Eltern-Schleifenrumpf hinzugefügt.

Die verschiedenen SURE-Algorithmen transformieren `LoopNests` unterschiedlich:

- Der Algorithmus von Wolf und Lam bzw. Darte und Vivien manipulieren die Schleifenkontrollstrukturen (Start- und Endindizes). Verwendet werden vor allem *max()*- und *min()*-Funktionen. Zusätzlich werden die Operationen im Schleifenrumpf hinsichtlich der modifizierten Schleifenkontrollstrukturen angepasst.
- Der Algorithmus von Allen und Kennedy behält die ursprüngliche Schleifenstruktur bei

Aus diesem Grund wurde die Erzeugung eines Schleifenkonstrukts weiter aufgespalten. Die abstrakten Klassen `LoopConditionBuilder` und `EquationBuilder` erbauen die unterschiedlichen Schleifenkontrollstrukturen bzw. Statements in Abhängigkeit des SURE-Algorithmus. Deren Klassendiagramme werden durch Abbildungen 51 und 52 repräsentiert. Alle weiteren Klassen sind in Abbildung 53 dargestellt.

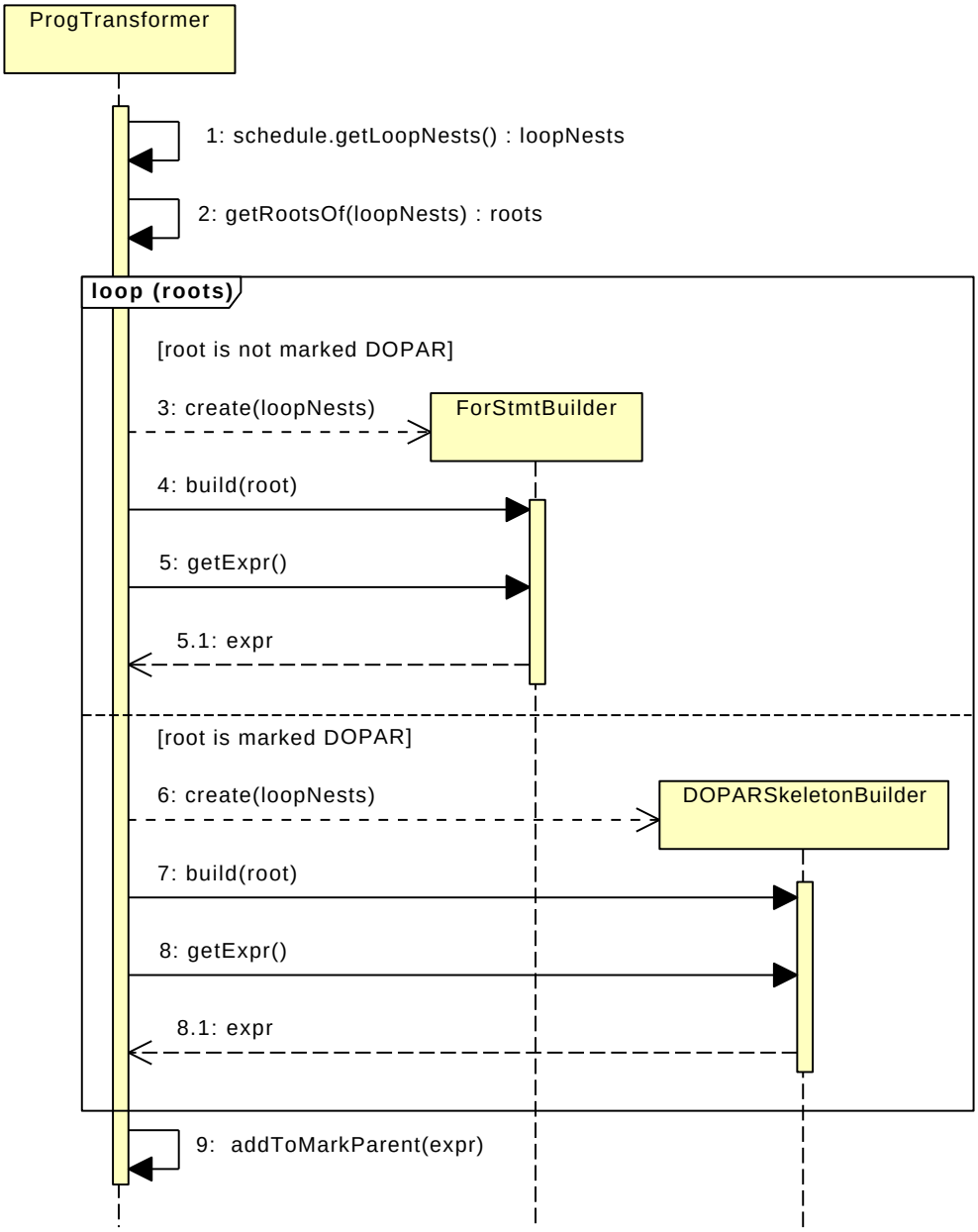


Abbildung 49: Sequenzdiagramm: `ProgTransformer.construct()`

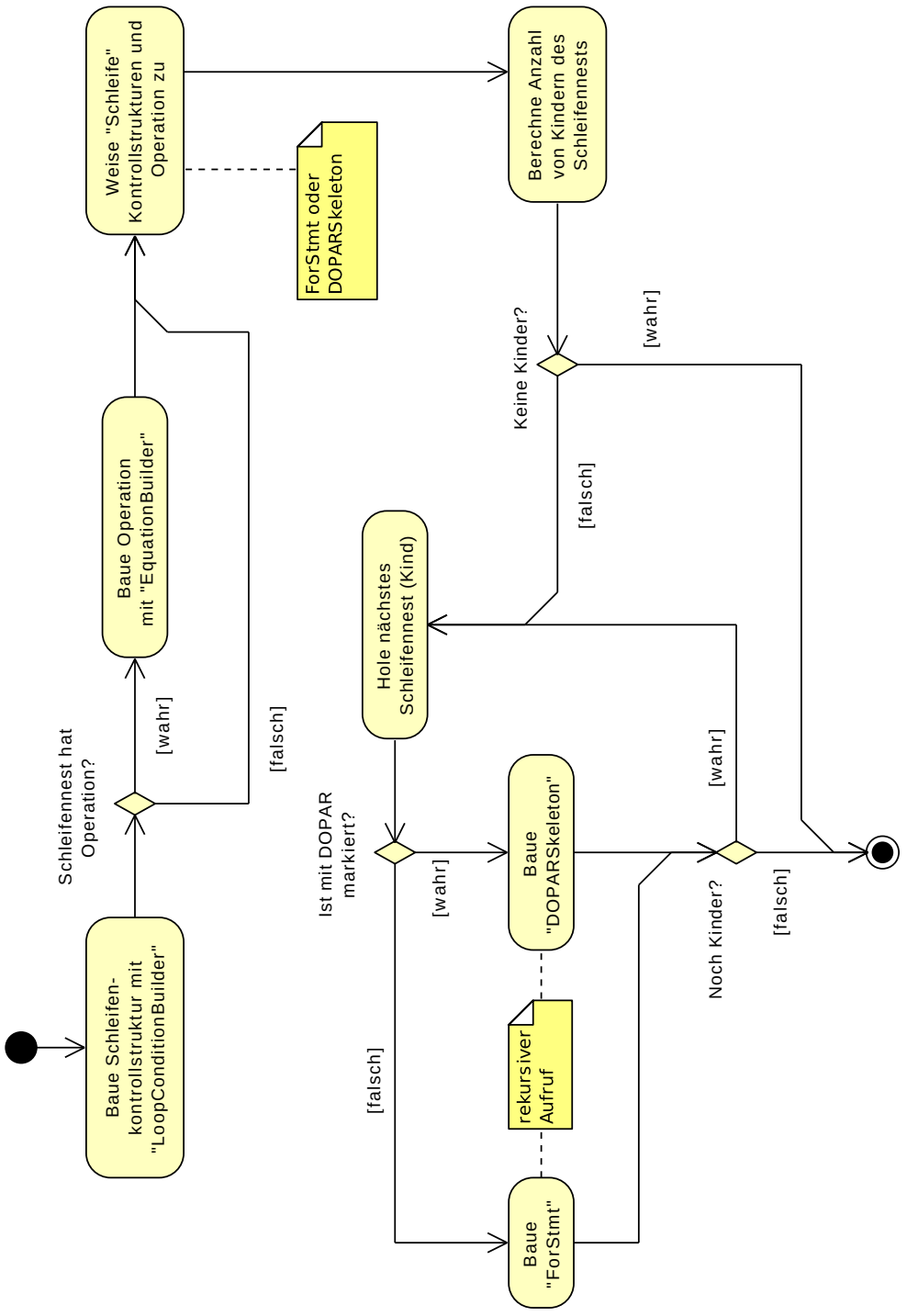


Abbildung 50: Aktivitätsdiagramm: Erbauen von Schleifen

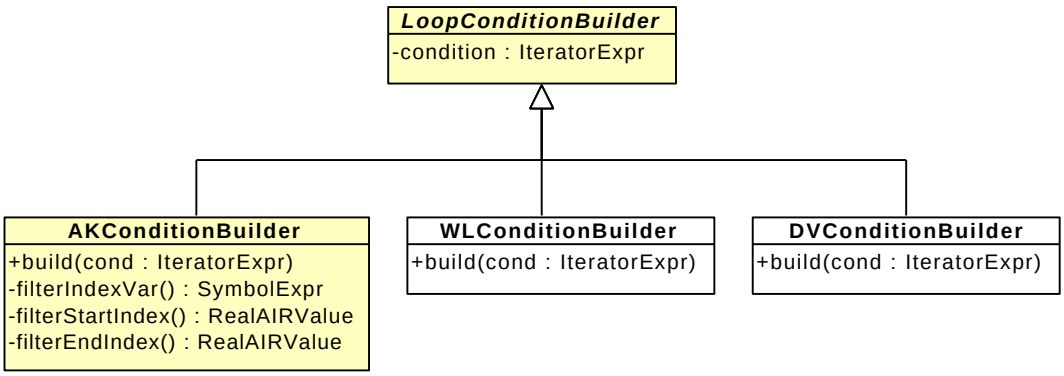


Abbildung 51: Klassendiagramm: `LoopConditionBuilder`

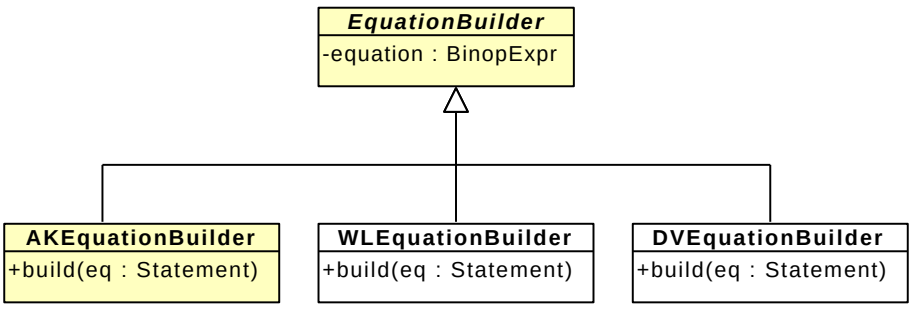


Abbildung 52: Klassendiagramm: `EquationBuilder`

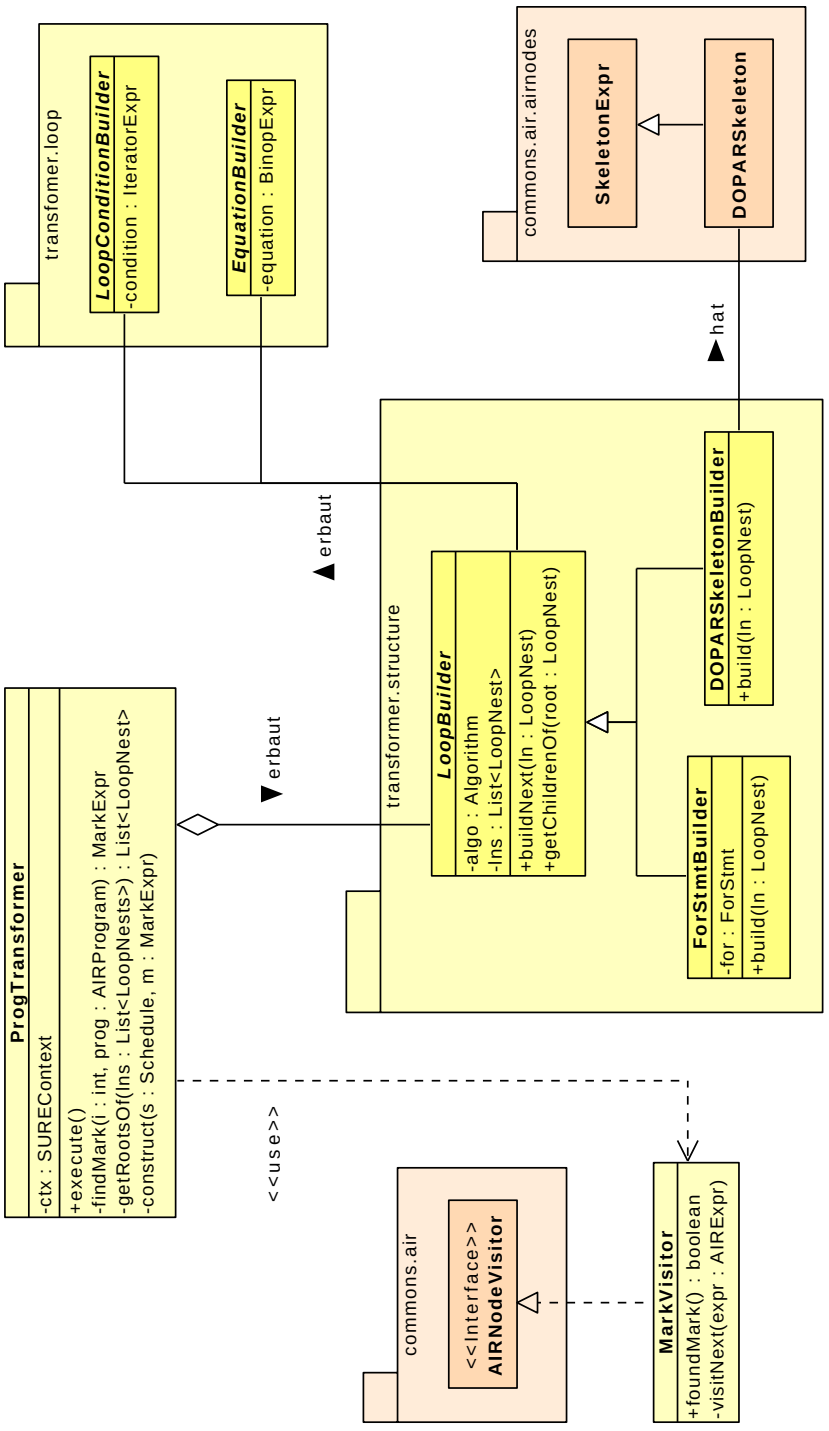


Abbildung 53: Klassendiagramm: AIR transformieren

6. Evaluierung

Die Evaluierung gliedert sich in zwei Hauptteile. Im ersten Teil wird die Qualität des implementierten PAMs beurteilt. Dabei wird sichergestellt, dass die essenziellen Aufgaben des `SURE`-Moduls korrekt ausgeführt werden. Es wurden eine Vielzahl von Testfällen erzeugt, welche die verschiedenen Aspekte der Mustererkennung und der Parallelisierungsanalyse auf die Probe stellen. Der zweite Teil befasst sich mit der Leistungsermittlung des parallelisierten Codes. Außerdem wurde in einigen Tests das Auftreten von Fehlern erzwungen, wodurch eine "saubere" Behandlung dieser getestet werden soll.

Bedingungen

Die korrekte Ausführung der aktuelllen Implementierung von `SURE` setzt folgende Bedingungen voraus:

1. Für den Iterator muss gelten:

   $$\texttt{for (}\mathit{Index}\ \texttt{in}\ \mathit{Startindex}:\mathit{Endindex}\texttt{)}$$
   $$\text{mit } \mathit{Startindex} \leq \mathit{Endindex}$$

   Außerdem werden ausschließlich numerische Werte für Start- und Endindizes vorausgesetzt. Numerische Variablen werden nicht erkannt.

2. Indizierung nur mittels einfacher arithmetischer Ausdrücke der Form:

   $$a * \mathit{Index}_i + c \quad : \quad a \in \{-1, 0, 1\},\ c \in \mathbb{Z}$$

   Dies schließt ebenfalls den R-Matrixbefehl `a[,,i]` aus, welcher einen Zugriff auf mehrere Zeilen bzw. Spalten einer Matrix darstellt.

3. Indizierung nur mittels Schleifenindexvariablen. D.h. eine Indizierung durch numerische Variablen, die außer- oder innerhalb der `for`-Schleife definiert wurden, sind nicht unterstützt:

```
1 V <- 42
2 for (i in 1:n) {
3   a[i] <- a[V]
4 }
```

4. Nur trennbare Matrixindizes (eng. *subscripts*). In [JS03] wird erklärt, dass ein Paar von Matrixindizes trennbar ist, falls alle beinhaltenden Indizes als exklusiv für dieses Paar gelten. Beispielsweise gilt für die Paare von Matrixindizes in einem Statement $m[i,\ 2j,\ j-1]$:

   Das erste Subscript ist trennbar, denn es enthält ausschließlich Index i. Das zweite und dritte Subscript sind jedoch gekoppelt, da sie sich die Indexvariable j teilen.

5. Kein "Aliasing"[7] für Matrizen. Ein Beispiel ist im nächsten Kapitel, präziser in Testfall 8 gegeben.

Die erste Bedingung bezieht sich ausschließlich auf die Implementierung des `DOPAR`-Skeletons. Eine Unterstützung für numerische Variablen ist implementierbar. Bedingung 2 ist auf PAM-Seite in der unvollständig implementierten Datenabhängigkeitsanalyse begründet. Eine Implementierung mächtigerer SIV- und MIV-Tests stellt die Lösung dieser Einschränkung dar. Eine Unterstützung für den R-Matrixbefehl `a[,,i]` ist nicht vorgesehen, da dieser Parallelisierungspotential entzieht.
Die Bedingungen 3 und 5 sind in ihrem Grad ähnlich schwer zu beseitigen. Momentan ist es unmöglich für `SURE` festzustellen, welchen Wert eine (Matrix-)Variable besitzt, die zur Laufzeit erzeugt und verändert wird.
Laut [JS03] existieren für trennbare Matrixindizes Methoden zur exakten Berechnung von Datenabhängigkeiten durch Abhängigkeitstests. Für gekoppelte[8] Matrixindizes gestaltet sich eine exakte Berechnung schwierig (vergleiche Bedingung 4).

6.1. Qualität von `SURE`

Aus der Fülle an ungefähr 45 durchgeführten Tests wurden die wichtigsten in Abhängigkeit derer Überdeckung ausgewählt. Im Folgenden werden die jeweiligen Testfälle kurz beschrieben und das resultierende Verhalten der Mustererkennung und der Parallelisierungsanalyse erläutert. Verwendet wird dabei die Implementierung des Algorithmus von Allen und Kennedy (im Folgenden: AK-Algorithmus) aus Kapitel 5. Veranschaulicht wird der Prozess anhand der berechneten Teilergebnisse, dem RLDG und des parallelisierten R-Programms in Pseudo-Code.
Bei den Testfällen handelt es sich ausschließlich um R-Programme mit verschachtelten `for`-Schleifen, da Programme ohne `for`-Schleifen ohnehin von der Mustererkennung ignoriert werden.

[7] Hier: Die Existenz mehrerer Referenzen für eine identische Matrix
[8] Zwei unterschiedliche Paare von Matrixindizes teilen eine gemeinsame Indexvariable

Ferner ist anzumerken, dass es sich in diesem Unterkapitel um größtenteils pragmatische Beispiele handelt. Es werden evtl. keine sinnvollen Berechnungen dargestellt. Es wird davon ausgegangen, dass die verwendeten Matrizen und Variablen ausreichend initialisiert sind.

1. Testfall

1.1 Mustererkennung

Einfaches mehrdimensionales Beispiel mit einem Statement S_1. Es existieren Lese- und Schreibzugriffe auf eine Matrix a:

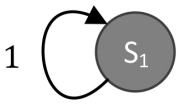
```
1 for (i in 2:n) {
2   for (j in 2:n) {
3       S1: a[i-1, j] <- a[i-1, j] + a[i, j];
4   }
5 }
```

Listing 6: R-Programm für Testfall 1

Das Schleifennest wird korrekt als SURE identifiziert.

1.2 Parallelisierungsanalyse

Die Datenabhängigkeitsanalyse ermittelt eine echte Abhängigkeit von Statement S_1 auf sich selbst. Der AK-Algorithmus erkennt die Datenabhängigkeit in Schleifentiefe 1. Folglich ist nur die `for`-Schleife der Tiefe 2 parallelisierbar:




```
1 for (i in 2:n) {
2   forall (j in 2:n) {
3     S1
4   }
5 }
```

Abbildung 54: Teilergebnisse von Testfall 1

2. Testfall

2.1 Mustererkennung

Komplexeres mehrdimensionales Beispiel mit zwei Statements. Ein Statement verwendet die *max()*-Methode. Es existieren Lese- und Schreibzugriffe auf beide Matrizen a und b. Zu beachten sind auch die unterschiedlichen Endindizes der Schleifen:

```
1 for (i in 2:m) {
2   for (j in 2:n) {
3     for (k in 2:o) {
4       S1 : a[i, j, k] <- max(b[2, j-1, k], a[i, j, k-1]);
5       S2 : b[i, j, k] <- b[i, j, k-1] + a[i-1, j, k];
6     }
7   }
8 }
```

Listing 7: R-Programm für Testfall 2

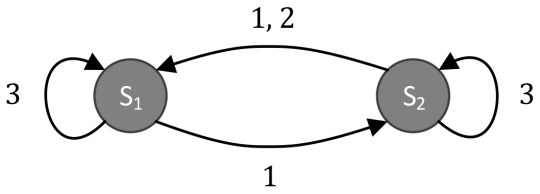
Das Schleifennest wird ebenfalls korrekt als SURE identifiziert. Sowohl die unterschiedlichen Endindizes als auch die Verwendung von Hilfsfunktionen (hier: *max()*) haben keinen Einfluss auf die Mustererkennung.

2.2 Parallelisierungsanalyse

Bemerkenswert in diesem Fall sind die durch Lesezugrif `b[2, j-1, k]` aus S_1 und Schreibzugriff `b[i, j, k]` aus S_2 erzeugten Datenabhängigkeiten:

- Echte Abhängigkeit ((Abhängigkeits-)Grad 1),
- Gegenabhängigkeit (Grad 1) und
- Echte Abhängigkeit (Grad 2)

Dies resultiert aus dem Vergleich zwischen Matrixindizes `i` und `2`. Der Wert von Matrixindex `i` kann durch die Schleifenindizierung kleiner, gleich oder größer als Matrixindex `2` sein. Folglich die drei Abhängigkeiten. Der *AK*-Algorithmus kann das Schleifennest nicht komplett aufspalten. Grund dafür ist der "Abhängigkeitszyklus" in Schleifentiefe 1. Erst ab Tiefe 2 werden die Statements in separate Schleifennester aufgespalten (*loop distribution*):




```
1  for (i in 2:m) {
2    forall (j in 2:n) {
3      for (k in 2:o) {
4        S2
5      }
6    }
7    forall (j in 2:n) {
8      for (k in 2:o) {
9        S1
10     }
11   }
12 }
```

Abbildung 55: Teilergebnisse von Testfall 2

3. Testfall

3.1 Mustererkennung

Ein Beispiel mit einem "eingeschobenen" (trivialen) Statement ohne Matrixzugriffe. Das zweite Statement S_2 besitzt Lese- und Schreibzugriffe auf Matrix a:

```
1 for (i in 1:m) {
2   S1: b <- i;
3   for (j in 2:n) {
4     S2: a[j] <- a[j] + (2 * a[j-1]);
5   }
6 }
```

Listing 8: R-Programm für Testfall 3

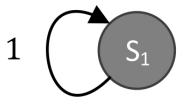
In diesem Fall wird nur das innere Schleifennest als SURE identifiziert, da die Matrixzellen von a nicht mittels Indexvariable i indiziert werden.
Ferner ist zu bemerken, dass dieses äußere Schleifennest korrekterweise nicht als SURE identifiziert wird, falls die Matrixzellen von a mittels Indexvariable i indiziert werden. Das resultierende SURE würde inkompatible Statements[9] beinhalten (vergleiche auch 4. Testfall).

3.3 Parallelisierungsanalyse

Wegen der echten Abhängigkeit innerhalb S_2 mit Grad 1 kann der AK-Algorithmus

[9] Weder Lese- noch Schreibzugriffe auf Matrixzellen

die Schleife nicht parallelisieren. Das ursprüngliche (unveränderte) Programm wird zurückgegeben:




```
1 for (i in 1:m) {
2   S1
3   for (j in 2:n) {
4     S2
5   }
6 }
```

Abbildung 56: Teilergebnisse von Testfall 3

4. Testfall

4.1 Mustererkennung

Ein beliebiges Beispiel mit inkompatiblen Statements im Schleifenrumpf:

```
1 for (i in 2:m) {
2   for (j in 2:n) {
3     for (k in 2:n) {
4       S1: a[i, j, k] <- b[i, j-1, k] - a[i, j, k-1];
5       S2: c <- randomFunc(c);
6       S3: b[i, j, k] <- c + b[i, j, k] + a[i-1, j, k];
7     }
8   }
9 }
```

Listing 9: R-Programm für Testfall 4

Die Mustererkennung identifiziert keinerlei kompatible SURE. Durch die Anwesenheit des Statements S_2 wird das angedeutete SURE beinträchtigt. Es ist nicht klar aber unwahrscheinlich, dass die entstehende Abhängigkeitsbeziehung durch einen SURE-Algorithmus verarbeitet werden kann.

4.2 Parallelisierungsanalyse

Es wurde kein SURE für diesen Testfall erkannt. Folglich kann auch keine Parallelisierungsanalyse durchgeführt werden, weshalb der Prozess an dieser Stelle abgebrochen wird (siehe auch nächsten Abschnitt “Fehlerbehandlung“).

5. Testfall

5.1 Mustererkennung

Ein Beispiel mit mehrfachen äußeren als auch inneren Schleifen. Ebenfalls zu beachten ist, dass manche Lese- und Schreibzugriffe auf Matrixzellen a und b durch numerische Konstanten indiziert werden:

```
1  for (i in 2:m) {
2    for (j in 3:n) {
3      S1: a[i, j] <- 2 * a[i-2, j];
4    }
5    for (k in 2:n) {
6      S2: b[i, k] <- b[1, k-1] + a[i, k];
7    }
8  }
9  for (i in 2:m) {
10   S3: a[i, 1] <- a[i, 2] * pi;
11 }
```

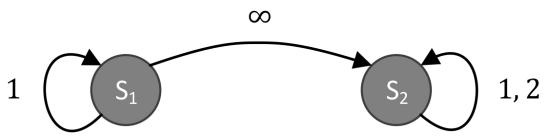
Listing 10: R-Programm für Testfall 5

Für dieses R-Programm werden zwei separate SURE erkannt, die durch die beiden äußersten Schleifen (mit Indexvariable i) eingeleitet werden. Die Konstante `pi` in Statement S_3 wird nicht zur Indizierung verwendet. Sie hat demnach keinen Einfluss auf die Mustererkennung.

5.2 Parallelisierungsanalyse

Für das erste SURE erkennt die Datenabhängigkeitsanalyse eine echte Datenabhängigkeit mit Grad ∞. Diese entsteht durch den Schreibzugriff `a[i, j]` aus S_1 und Lesezugriff `a[i, k]` aus S_2. Da weder die j- noch die k-Schleife zwischen den beiden Statements eine Abhängigkeit tragen kann (siehe *loop carried dependence* aus Kapitel 3.3) folgt daraus eine schleifenunabhängige Datenabhängigkeit (*loop independent dependence*):

Die Analyse und Parallelisierung des zweiten SUREs ist trivial. Es existieren keine Abhängigkeiten, also parallelisiert der AK-Algorithmus alle `for`-Schleifen im aktuellen SURE. In diesem Fall nur die i-Schleife:




```
1  for (i in 2:m) {
2    forall (j in 3:n) {
3      S1
4    }
5  }
6  for (i in 2:m) {
7    for (k in 2:n) {
8      S2
9    }
10 }
```

Abbildung 57: Teilergebnisse des ersten SURE von Testfall 5


```
1  forall (i in 2:m) {
2    S3
3  }
```

Abbildung 58: Teilergebnisse des zweiten SURE von Testfall 5

In der Transformationsphase des SURE-Verfahrens werden die beiden parallelisierten SURE zusammengeführt:

```
1  for (i in 2:m) {
2    forall (j in 3:n) {
3      S1
4    }
5  }
6  for (i in 2:m) {
7    for (k in 2:n) {
8      S2
9    }
10 }
11 forall (i in 2:m) {
12   S3
13 }
```

Listing 11: Finaler Pseudo-Code für Testfall 8

6. Testfall

6.1 Mustererkennung

Dieser Testfall stellt eine Abwandlung des vorhergehenden Beispiels dar. Es soll die

Problematik von separaten Schleifennestern mit gleicher Indexvariable verdeutlicht werden. Der Schreibzugriff beider Statements bezieht sich auf die selbe Matrix a:

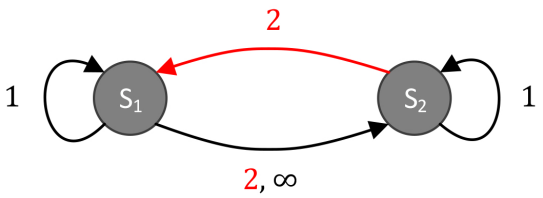
```
1 for (i in 2:n-1) {
2   for (j in 2:n) {
3     S1: a[i, j] <- a[i+1, j-1];
4   }
5   for (j in 5:n) {
6     S2: a[i, j-1] <- a[1, j-1] + a[i, j];
7   }
8 }
```

Listing 12: R-Programm für Testfall 6

Das komplette Schleifennest wird (wie auch Testfall 5) als SURE identifiziert.

6.2 Parallelisierungsanalyse

Im derzeitigen Zustand der Implementierung ist es nicht möglich zwischen zwei `for`-Schleifen mit gleicher Indexvariable zu unterscheiden. Aus diesem Grund werden die Ausgabeabhängigkeiten (Grad 2) erzeugt, was zu einem Abhängigkeitszyklus führt (vergleiche Testfall 2). Daraufhin schlägt die Ablaufplanung mit dem AK-Algorithmus fehl. Eine Aufspaltung der Schleifen wird nur ermöglicht, falls kein Abhängigkeitszyklus zwischen zwei Statements existiert:




```
1 for (i in 2:n-1) {
2   for (j in ?) {
3     S1
4     S2
5   }
6 }
```

Abbildung 59: Fehlerhafte Teilergebnisse von Testfall 6

In den folgenden beiden (Unter-)Testfällen wird dieses Verhalten noch einmal genauer untersucht. Speziell wird dabei auf die Rolle des Datenabhängigkeitstyps eingegangen. Das erste R-Programm (siehe Listing 13) enthält zwei Gegenabhängigkeiten, während das zweite R-Programm (siehe Listing 14) zwei echte Abhängigkeiten enthält:

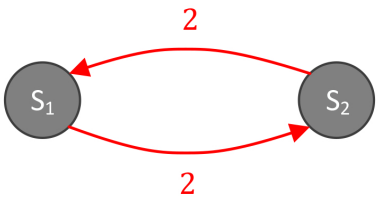
```
1 for (i in 1:m) {
2   for (j in 2:n+1) {
3     S1: a[i, j-1] <- b[i, j+1];
4     S2: b[i, j] <- a[i, j];
5   }
6 }
```

Listing 13: R-Programm des ersten Untertestfalls

```
1 for (i in 1:m) {
2   for (j in 2:n+1) {
3     S1: a[i, j] <- b[i, j-1];
4     S2: b[i, j] <- a[i, j-1];
5   }
6 }
```

Listing 14: R-Programm des zweiten Untertestfalls

Die Parallelisierungsanalyse für beide R-Programme verläuft identisch. Der Abhängigkeitszyklus in Schleifentiefe 2 verhindert eine Aufspaltung von Statement S_1 und S_2. Es ist wichtig zu erwähnen, dass dieses Verhalten nur bei Zyklen mit maximalem[10] Grad eintritt. Ein Testfall mit alternierenden Datenabhängigkeiten konnte nicht erzeugt werden, was eine Existenz jedoch nicht ausschließt:




```
1 for (i in 1:m) {
2   for (j in ?) {
3     S1
4     S2
5   }
6 }
```

Abbildung 60: Fehlerhafte Teilergebnisse von Testfall 13 und 14

[10] D.h. für die innerste Schleife

7. Testfall

7.1 Mustererkennung

In diesem Testfall indiziert der Schreibzugriff von Statement S_2 mittels Indexvariable der äußeren `for`-Schleife. Lediglich die Lesezugriffe auf Matrix c verwenden Indexvariable j. Außerdem ist Matrix a, genau wie Matrix b 1-dimensional:

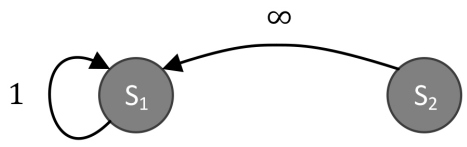
```
1 for (i in 1:m) {
2   S1: b[i] <- b[i] / a[i];
3   for (j in 2:n) {
4     S2: a[i] <- c[j] + c[j-1];
5   }
6 }
```

Listing 15: R-Programm für Testfall 7

Wird ebenfalls korrekt als SURE erkannt. In diesem Testfall handelt es sich bei S_1 nicht um ein triviales Statement (vergleiche auch mit 3. Testfall), sondern um eine Matrixmodifikation mit Lese- und Schreibzugriffen.

7.2 Parallelisierungsanalyse

Der RLDG dieses Testfalls weist eine Ausgabeabhängigkeit von S_2 auf sich selbst aus, zumal Matrix a (Schreibzugriff) nicht genügend Matrixindizes für alle umschließenden `for`-Schleifen besitzt (vergleiche auch Beispiel aus [AK87]):




```
1 for (i in 1:m) {
2   forall (j in 2:n) {
3     S2
4   }
5 }
6 forall (i in 1:m) {
7   S1
8 }
```

Abbildung 61: Teilergebnisse von Testfall 7

8. Testfall

8.1 Mustererkennung

Dieser einfache Testfall verdeutlicht noch einmal die Problematik von “Aliasing“ im generellen Sinne:

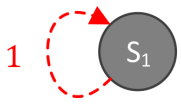
```
1 b <- a;
2 ...
3 for (i in 2:n) {
4   S1 : a[i] <- b[i-1];
5 }
```

Listing 16: Programm für Testfall 8

Die Schleife wird zurecht als SURE identifiziert, jedoch wird dieses Programm fälschlicherweise parallelisiert.

8.2 Parallelisierungsanalyse

In R wird durch den Befehl `b <- a` ein Kopie von `a` erzeugt. Demnach wären die Teilergebnisse in Abbildung 62 korrekt.
Im Folgenden wird das Problem in Bezug auf Sprachen, die lediglich Referenzen erzeugen (z.B. Java), betrachtet. Dem `SURE`-Modul ist die Referenz von Matrix b auf Matrix a zur Laufzeit unbekannt. Die Datenabhängigkeitsanalyse würde zwei unterschiedliche Matrizen identifiziert und könnte somit die echte Datenabhängigkeit nicht feststellen. Die `for`-Schleife wird parallelisiert und liefert bei einer Ausführung potentiell falsche Ergebnisse:




```
1 b <- a;
2 ...
3 forall (i in 2:n) {
4   S1
5 }
```

Abbildung 62: Fehlerhafte Teilergebnisse von Testfall 8

Wie bereits erwähnt ist “Aliasing“ ausschließlich ein Problem bezüglich Referenzierung und hat somit keine Auswirkungen auf die Parallelisierung von R-Programmen.

Zusammenfassend lässt sich sagen, was in [DRV01] bereits bewiesen wurde: Der AK-Algorithmus ist optimal für eine Abhängigkeitsapproximation durch Abhängigkeitsgrade, kann jedoch nicht zwischen Programmen mit identischem RLDG unterscheiden.
Falls Abhängigkeitszyklen mit maximalem Grad existieren, so schlägt die Parallelisierung mit dem AK-Algorithmus fehl.

Fehlerbehandlung

Dieser Teil der Evaluierung umfasst ungefähr 25 Tests in Bezug auf die Fehlerbehandlung des `SURE`-Moduls:

Fehlerhafte Benutzereingaben , wie z.B. Syntaxfehler oder unzureichend initialisierte Matrizen, werden hauptsächlich von R-ALCHEMY abgefangen. Entsprechende Fehlermeldungen können der R-Konsole entnommen werden. Dabei gilt es zu beachten, dass alle konfigurierten ALCHEMY-Transmutatoren (also auch `SURE`) trotz dieser auftretenden Fehler ausgeführt werden.

Interne Fehler können durch unerwartete Parameterwerte auftreten. Präziser ausgedrückt, entstehen sie durch falsche Verwendung `SURE`-interer Schnittstellen. Die Mustererkennung als auch die Datenabhängigkeitsanalyse basiert auf einem textuellen Parser. Dieser ist eng an die XML-Repräsentation von AIR gekoppelt. Verändert sich diese ohne Anpassung des Parsers, so treten Lesbarkeitsfehler auf.

Transmutationsfehler entstehen, falls eine Parallelisierung mit der verwendeten `SURE`-Konfiguration nicht durchführbar ist. Hauptsächlich werden diese in der Ablaufplanungsphase bei der Durchführung eines SURE-Algorithmus ausgelöst. Die im vorhergehenden Abschnitt "Parallelisierungsanalyse" behandelten Testfälle 2 und 6 fallen in diese Kategorie. Tritt ein Transmutationsfehler auf, so wird dieser detailliert über die Konsole ausgegeben. Die Parallelisierung für das aktuell analysierte SURE wird abgebrochen und der ursprüngliche Code wiederhergestellt.

Warnungen sind Benutzermitteilungen, welche detaillierte Informationen über unkritische Fehler bereitstellen. Beispielsweise wird über die Konsole eine Warnung ausgegeben, falls keine SURE erkannt wurden (vergleiche Testfälle 3 und 4). Ein weiteres Beispiel wäre die Erkennung einer MIV-Klassifizierung (siehe Kapitel 3.3). Zum aktuellen Stand ist kein MIV-Test implementiert. In diesem Fall wird von einer oder mehreren Datenabhängigkeiten ausgegangen, obwohl diese nicht notwendigerweise existieren. Warnungen treten in allen Komponenten des `SURE`-PAMs auf und haben lediglich Auswirkung auf das Ergebnis, nicht aber die Durchführung.

Anderweitige Fehler, die z.B. durch Nichteinhaltung der in Abschnitt "Bedingungen" aufgeführten Konditionen entstehen, werden nicht behandelt.

6.2. Leistung

In diesem Unterkapitel wird die Leistung der parallelisierten Programme anhand von Messungen verglichen und beurteilt. Alle Messungen wurden auf einem gewöhnlichen Desktop-PC mit den Spezifikationen

CPU: AMD FX-8120 (8-Kerne) @ 3.1 GHz

RAM: 8 GB

OS: Ubuntu 12.04 (64-bit)

durchgeführt. Um Fehler bei der Berechnung der Messergebnisse zu relativieren, wurden alle Messungen 10x wiederholt und daraus das arithmetische Mittel berechnet. Für die Tests wurde jeweils die Laufzeit in Sekunden und die Beschleunigung gemessen. Die Beschleunigung (engl. *SpeedUp*) ergibt sich aus der Laufzeit eines sequentiellen Programms ($P_{sequentiell}$) im Vergleich zur Laufzeit des parallelisierten Gegenstücks ($P_{parallel}$):

$$\text{Beschleunigung} = \frac{\text{Ausführungszeit}(\text{P}_{\text{sequentiell}})}{\text{Ausführungszeit}(\text{P}_{\text{parallel}})}$$

Einschränkungen des `ArBB`-Backends

Damit eine parallele Ausführung durch das `ArBB`-Backend ermöglich werden kann, müssen folgende Einschränkungen eingehalten werden:

1. Maximale Matrizendimension von **2** darf nicht überschritten werden (fehlende ALCHEMY-Implementierung)
2. Es sind keine **äußeren** parallelisierten Schleifen, präziser `DOPAR`s, erlaubt (keine vorhandene[11] `ArBB`-Unterstützung)

[11] `ArBB`-Projekt wurde eingestellt

Dies bedeutet ausschließlich parallelisierte SUREs der Form:

```
1 for (i in 1:n) {
2   Schleife_1 { S_1 }
3   Schleife_2 { S_2 }
4   ...
5   Schleife_m { S_m }
6
7 }
```

mit $Schleife_i \in \{for,\ forall\}$ können erfolgreich ausgeführt und gemessen werden. Ferner erlaubt das `ArBB`-Backend für schreibende Matrixzugriffe nur eine elementare Indizierung der Form:

$$\texttt{Array[}Index_1\texttt{, }Index_2\texttt{, ...,}Index_n\texttt{]}$$

Messungen

Auf Grund der strengen Einschränkungen des `ArBB`-Backends wurde für die Messungen ein anderes Backend [Ana13][Boa97] gewählt:

doMC ist ein paralleles R-Backend, welches die Schleifenfunktion `foreach` in sequentieller als auch paralleler Form verwendet. Für die Parallelisierung wird *R multicore* [Urb] verwendet.

C/OpenMP ist eine Programmierschnittstelle für C, C++ und Fortran. Unter anderem sind spezielle Compiler-Direktive zur verteilten Abarbeitung von `for`-Schleifen definiert.

Die folgenden Messungen wurden mit Matrixelementen zwischen 1K bis 500M[12] getestet. Die sequentielle bzw. parallele Laufzeit und die berechnete Beschleunigung sind jeweils in Diagrammen veranschaulicht:

1. Messbeispiel

Das erste Beispiel stammt aus [Kum02] und stellt eine einfache Multiplikation einer Matrix a mit einem Vektor x dar:

[12] In Abhängigkeit des *Break-Even-Points* dem zur Verfügung stehenden Arbeitsspeicher

```
1 for (i in 1:n) {
2   for (j in 1:n) {
3     S1: y[i] <- y[i] + a[i,j] * x[j]
4   }
5 }
```

Listing 17: Sequentielles R-Programm für Beispiel 1

Die Messungen wurde mit 1K, 50K, 100K und 500K Matrixelementen mittels *doMC* durchgeführt:

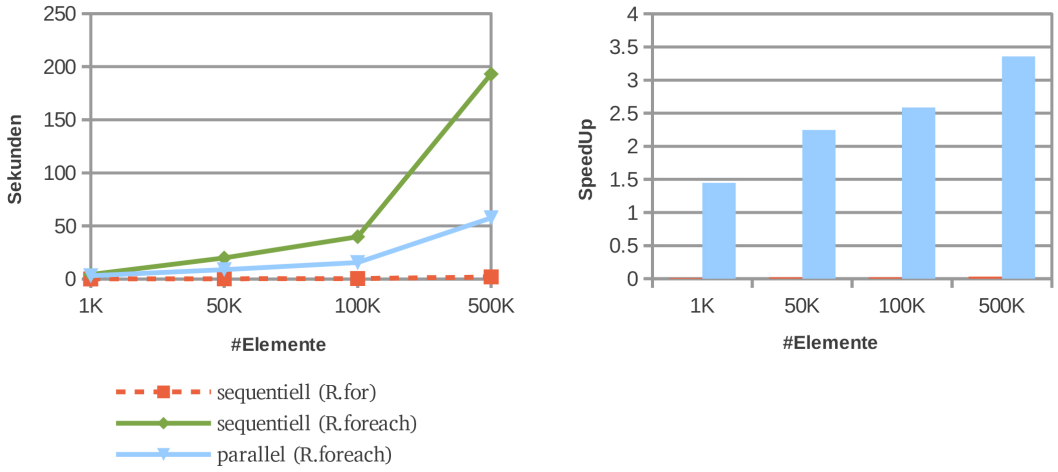


Abbildung 63: Laufzeit und SpeedUp von Beispiel 1 mit *doMC*

Den Leistungsdaten aus Abbildung 63 ist zu entnehmen, dass die sequentielle Ausführungszeit der `foreach`-Schleife weitaus länger ist, als die der parallelen `foreach`-Schleife. Für 500K Matrixelemente benötigt das sequentielle Programm 193,19 Sekunden, das parallele Programm hingegen nur 57,544 Sekunden. Eine Beschleunigung von 1,4 zeichnet sich schon für bereits 1K Matrixelemente ab. Die maximale Beschleunigung liegt bei 3,4.
Vergleicht man diese Werte jedoch mit den Messergebnissen einer regulären sequentiellen `for`-Schleife, ist ein hohes Leistungsgefälle erkennbar. Dies liegt an der Implementierung von `foreach`. Für jede Iteration einer sequentiellen (bzw. parallelen) `foreach`-Schleife werden:

- mehrere lokale Variablen erzeugt,
- Werte zwischengespeichert und
- Fehler behandelt

Die Berechnung und Koordination all dieser Zusatzdaten (eng. *Overhead*) erzeugt den dargestellten Leistungsabfall. Die Operationen im Schleifenrumpf sind zudem nicht arbeitsintensiv genug, weshalb sich dieser Mehraufwand nicht relativieren lässt.
Aus diesem Grund wurde das aktuelle Messbeispiel mit *C/OpenMP* wiederholt. Um darstellbare Ergebnisse liefern zu können, wurde die Eingabegröße auf 100K, 500K, 1M, 10M und 100M angepasst:

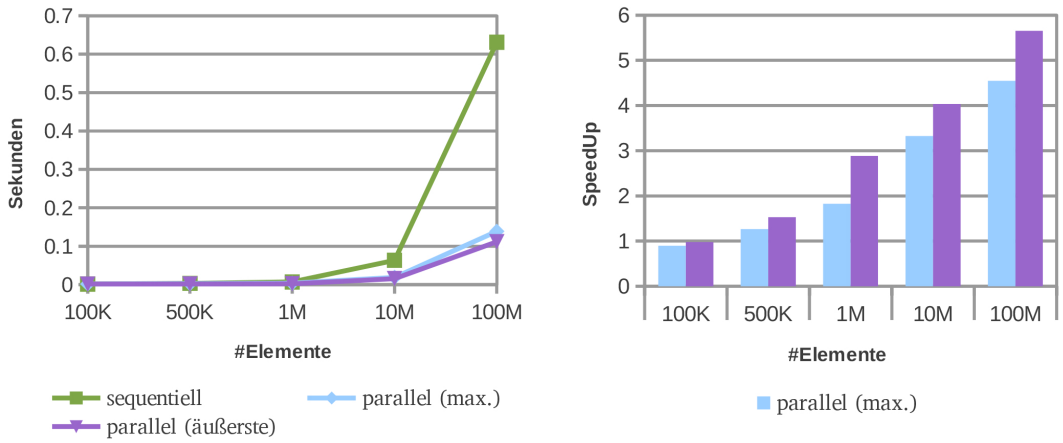


Abbildung 64: Laufzeit und SpeedUp von Beispiel 1 mit *C/OpenMP*

Im Vergleich zum vorhergehenden Messbeispiel zeichnet sich entsprechend Abbildung 64 ein signifikanter Leistungzuwachs ab. Es wurden zwei Fälle betrachtet:

1. Der parallelisierte Programm-Code in unverändertem Zustand mit maximaler Parallelität[13] und

2. Eine angepasste Version des parallelisierten Programm-Codes, aus der alle parallel ausführbaren `for`-Schleifen (`forall`-Schleifen) bis auf die **äußerste** entfernt wurden

In Fall 1 wurde der *Break-Even-Point* (BEP) bei ungefähr 300K Elementen erreicht. Ferner zeigen die Leistungsdaten einen weiteren Leistungszuwachs von durchschnittlich 27% für den 2. Fall. Der BEP wurde bereits bei ungefähr 100K Elementen überschritten. Diese Erkenntnis wird in den folgenden Messungen weiterverfolgt.
Die von `SURE` berechneten Teilergebnisse sind in Abbildung 65 einsehbar.

[13] In Anbetracht des Algorithmus von Allen und Kennedy


```
1 forall (i in 1:n) {
2   forall (j in 1:n) {
3     S1
4   }
5 }
```

Abbildung 65: Teilergebnisse von Beispiel 1

In Anhang D werden die jeweiligen Code-Beispiele für *doMC* und *C/OpenMP* samt kurzer Erläuterung gegeben. Alle weiteren Messungen wurden ausschließlich mit *C/OpenMP* durchgeführt.

2. Messbeispiel

Das zweite Beispiel ist ebenfalls [Kum02] entnommen worden und stellt eine Multiplikation zweier Matrizen a und b dar:

```
1 for (i in 1:n) {
2   for (j in 1:n) {
3     for (k in 1:n) {
4       S1 : c[i,j] <- c[i,j] + a[i,k] * b[k,j]
5     }
6   }
7 }
```

Listing 18: Sequentielles R-Programm für Beispiel 2

Die Messungen wurden mit 10K, 100K, 500K und 1M Matrixelementen durchgeführt.

Ähnlich zu Messbeispiel 1 fallen auch in diesem Beispiel die Leistungsdaten sehr positiv aus. Da es sich hierbei um drei geschachtelte Schleifen handelt wurde ein weiter Fall betrachtet:

3. Eine angepasste Version des parallelisierten Programm-Codes, aus welcher lediglich die **innerste** `forall`-Schleife entfernt wurde

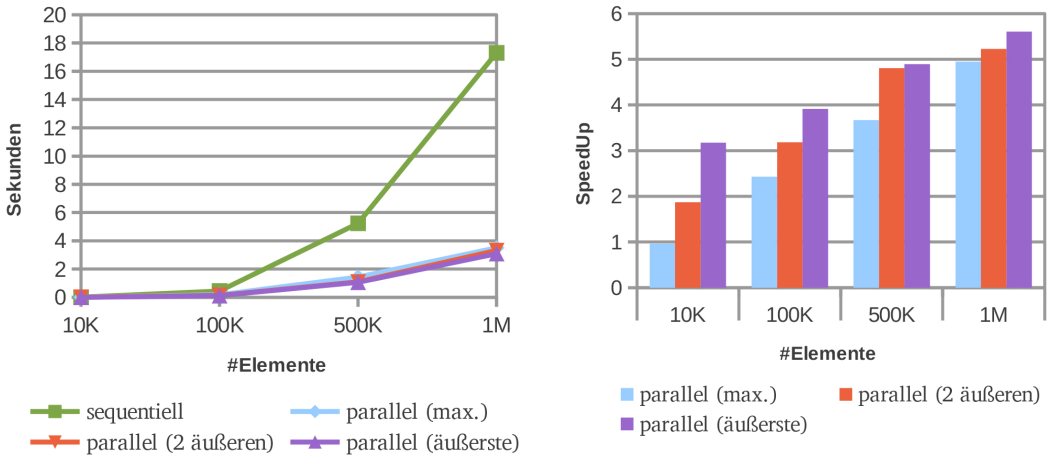


Abbildung 66: Laufzeit und SpeedUp von Beispiel 2

Wurde nur die äußerste `forall`-Schleife parallel ausgeführt lag die Beschleunigung zwischen 3,2 und 5,6. Der BEP wurde ebenfalls ab einer geringeren Eingabegröße erreicht:

	parallel (max.)	*parallel (2 äußeren)*	*parallel (äußerste)*
BEP	**~10K**	**~5K**	**~3K**

Wie die Leistungsdaten aus Abbildung 66 belegen, fiel die Beschleunigung umso höher aus, je mehr `forall`-Schleifen (von innen nach außen) entfernt wurden. Mögliche Begründungen[14] dafür wären:

- Ein erhöhter Kommunikationsaufwand zwischen den Ausführungssträngen (eng. *Threads*)
- Häufigere Wartezeiten, die durch das "Aufwecken" schlafender Threads auftreten
- Implementierungsbedingter Overhead
- Kombination aus allen bisher genannten Begründungen

Die berechneten Teilergebnisse sind in Abbildung 67 dargestellt.

[14] Ausschließlich auf Basis der berechneten Leistungsdaten


```
forall (k in 1:n) {
  forall (j in 1:n) {
    forall (i in 1:n) {
      S1
    }
  }
}
```

Abbildung 67: Teilergebnisse von Beispiel 2

3. Messbeipiel

Dieses künstliche Beispiel stammt aus [AK87]. Es ist anzumerken, dass `SURE` aufgrund der echten Datenabhängigkeit zwischen Statement S_4 und S_1 (MIV) zwei nicht existierende Abhängigkeiten berechnet. Für dieses Beispiel wurde deshalb der parallelisierte Code aus [AK87] verwendet (siehe Abbildung 69):

```
for (i in 1:n) {
  S1: x[i] <- y[i] + 10
  for (j in 1:n) {
    S2: b[j] <- a[j,n]
    for (k in 1:n) {
      S3: a[j+1,k] <- b[j] + c[j,k]
    }
  S4: y[i+j] <- a[j+1,n]
  }
}
```

Listing 19: Sequentielles R-Programm für Beispiel 3

Die Messungen wurde mit 100K, 1M, 5M, 8M und 10M Matrixelementen durchgeführt.

Der parallelisierte Code für Beispiel 3 weist mehrere innere `forall`-Schleifen auf. Trotz des hohen Grades an Parallelität ist die berechnete Beschleunigung (siehe Abbildung 68) mit maximal 2,1 für 10M Elemente zwar noch gut, aber nicht mehr so spektakulär wie in Messbeispiel 1 und 2. Der BEP wurde bei ungefähr 3M Elementen erreicht. Um eine potentielle Ursache des Leistungsabfalls zu überprüfen, wurden zusätzliche Leistungsdaten ermittelt:

4. Aus dem parallelisierten Programm-Code wurden **alle** `forall`-Schleifen entfernt.

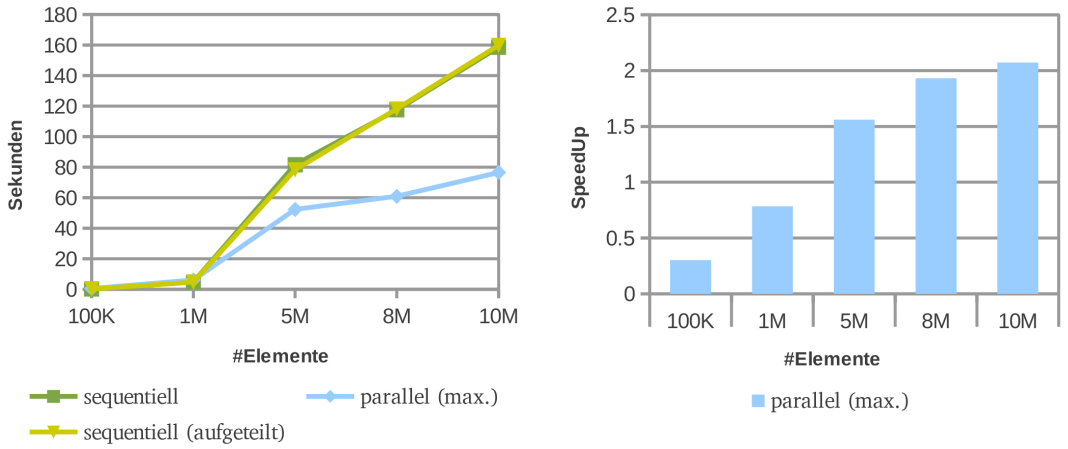
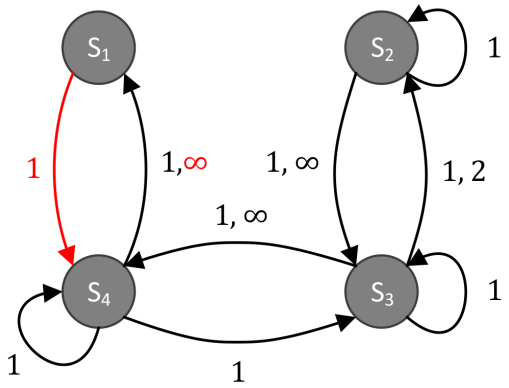


Abbildung 68: Laufzeit und SpeedUp von Beispiel 3

Somit wurde überprüft, ob die Schleifenneststruktur einen Einfluss auf die parallele Laufzeit hat. Das Messergebniss zeigt, dass Fall 4 in diesem Beispiel sogar bis zu 4% schneller ausgeführt wurde, als die unaufgeteilte sequentielle Version.




```
1  for (i in 1:n) {
2    for (j in 1:n) {
3      S2
4      forall (k in 1:(n/2)) {
5        S3
6      }
7    }
8    forall (j in 1:n) {
9      S4
10   }
11 }
12 forall (i in 1:n) {
13   S1
14 }
```

Abbildung 69: Teilergebnisse von Beispiel 3

4. Messbeispiel

Das 4. Messbeispiel bezieht sich auf ein Code-Beispiel aus [DV96a]:

```
1 for (i in 2:n) {
2   for (j in 2:n) {
3     S1: a[i,j] <- i
4     S2: b[i,j] <- b[i,j-1] + a[i,j] * c[i-1,j]
5     S3: c[i,j] <- 2 * b[i,j] + a[i,j]
6   }
7 }
```

Listing 20: Sequentielles R-Programm für Beispiel 4

Es wurden Eingaben der Größe 1M, 10M, 100M und 400M verwendet.

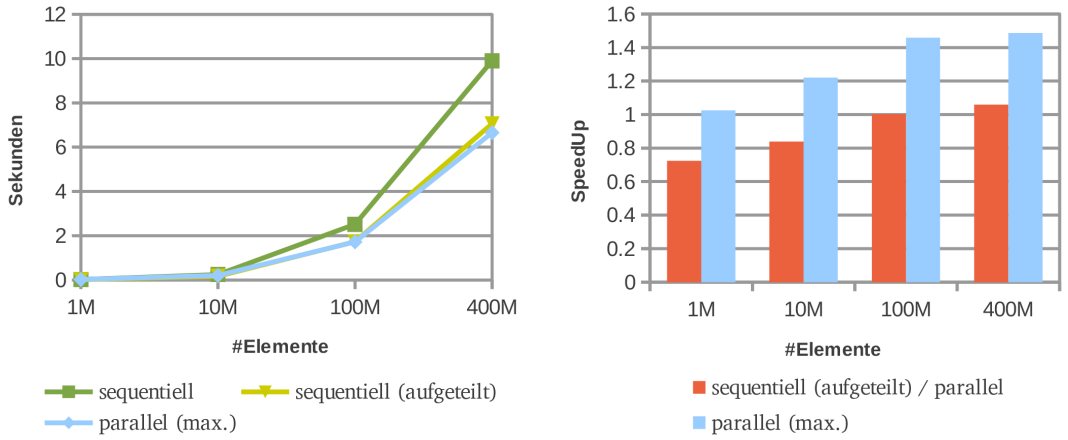
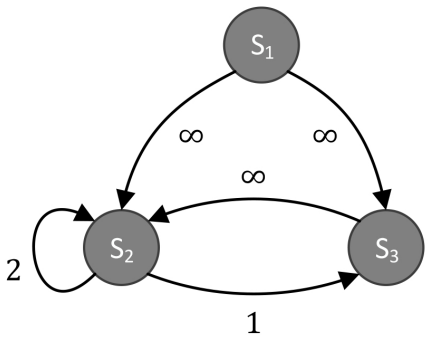


Abbildung 70: Laufzeit und SpeedUp von Beispiel 4

Das parallelisierte Programm besitzt zwar einen hohen Grad an Parallelität, jedoch weist das Statement im parallelen Schleifenennests (siehe Abbildung 71, Zeile 1-5) keine Berechnungen auf. In Abhängigkeit des Overheads wirken sich diese parallelen `forall`-Schleifen vielleicht sogar negativ auf die Gesamtlaufzeit aus.
Es wurde wie im vorhergehenden Messbeispiel Laufzeitdaten von einer aufgeteilten Version des sequentiellen R-Programms ermittelt. Dessen Laufzeit unterscheidet sich nur marginal von der erzeugten parallelen Version. Folglich liegt die Beschleunigung nur knapp über 1,0. Gegenüber dem regulären, sequentiellen Programm lässt sich eine Beschleunigung von ungefähr 1,5 für 400M Elemente berechnen.

Diese Daten belegen, dass die Aufteilung (*loop distribution*) der Schleifenstruktur die parallele Programmlaufzeit positiv beinflusst. In diesem Beispiel ist die reguläre, sequentielle Laufzeit um durchschnittlich 43% länger.




```
1  forall (i in 2:n) {
2    forall (k in 2:n) {
3      S1
4    }
5  }
6  for (i in 2:n) {
7    for (j in 2:n) {
8      S2
9    }
10   forall (j in 2:n) {
11     S3
12   }
13 }
```

Abbildung 71: Teilergebnisse von Beispiel 4

5. Messbeispiel

Gundlage für das 5. Messbeispiel bietet das Code-Beispiel für den AK-Algorithmus aus [DRV01]:

```
1 for (i in 2:n) {
2   for (j in 2:(n-1)) {
3     for (k 2:n) {
4       S1 : a[i,j,k] <- a[i-1,j+1,k] + a[i,j,k-1] - b[i,j-1,k]
5       S2 : b[i,j,k] <- b[i,j-1,k] + a[i-1,j,k]
6     }
7   }
8 }
```

Listing 21: Sequentielles R-Programm für Beispiel 5

Der BEP wurde erst bei etwa 30M Matrixelementen überschritten. Im Gegensatz zum letzten Messbeispiel fällt der Leistungszuwachs durch eine Schleifennestaufteilung mit durchschnittlich 10% etwas kleiner aus. Die maximale Beschleunigung liegt bei 500M Matrixelementen mit 1,3 gegenüber der regulären und 1,2

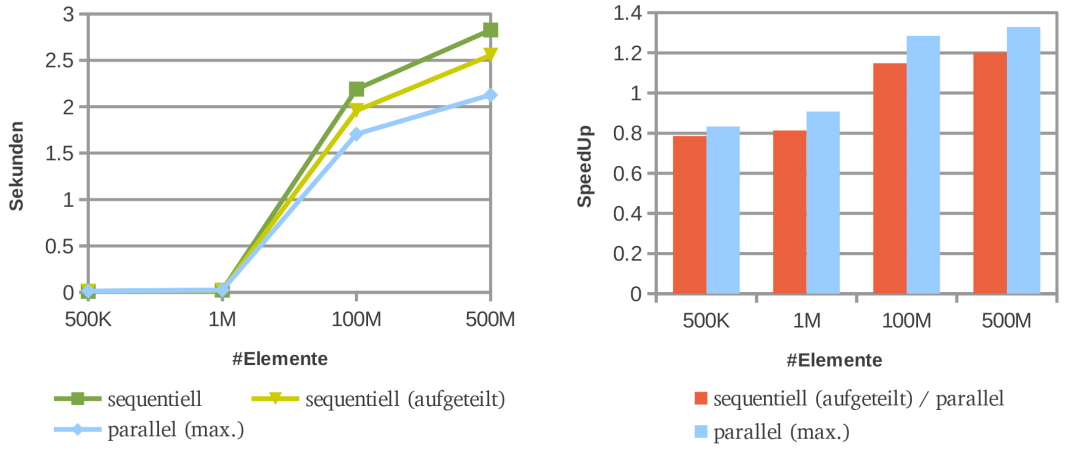
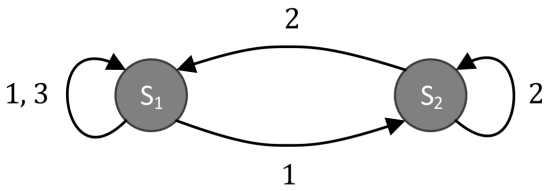


Abbildung 72: Laufzeit und SpeedUp von Beispiel 5

gegenüber der aufgeteilten, sequentiellen Laufzeit. Abbildung 73 ist zu entnehmen, dass hauptsächlich innere `for`-Schleifen parallel ausführbar sind.




```
1  for (i in 2:n) {
2    for (j in 2:(n-1)) {
3      forall (k in 2:n) {
4        S2
5      }
6    }
7    forall (j in 2:(n-1)) {
8      for (k in 2:n) {
9        S1
10     }
11   }
12 }
```

Abbildung 73: Teilergebnisse von Beispiel 5

6. Messbeispiel

Dass eine Ausführung mancher (durch das SURE-Verfahren) parallelisierten Programme erst für sehr große Eingaben einen Leistungszuwachs zeigen, belegt das Code-Beispiel aus [Dar]:

```
1  for (k in 1:n) {
2    for (j in 1:n) {
3      for (i in 1:n) {
4        S1: a[i,j,k] <- a[i,j-1,k] + b[i,j-1,N] + c[i,j-1]
5        S2: b[i,j,k] <- a[i,j,k-1] + b[i,j,k-1]
6      }
7      S3: c[i,j] <- b[i-1,j,N] + c[i-1,N]
8      S4: d[i,j] <- a[i-1,j,N] + c[i,j] + d[i,j-1]
9    }
10 }
```

Listing 22: Sequentielles R-Programm für Beispiel 6

Die Messungen wurden mit der maximal möglichen Anzahl an Matrixelementen 100K, 500K, 1M, 100M und 500M durchgeführt:

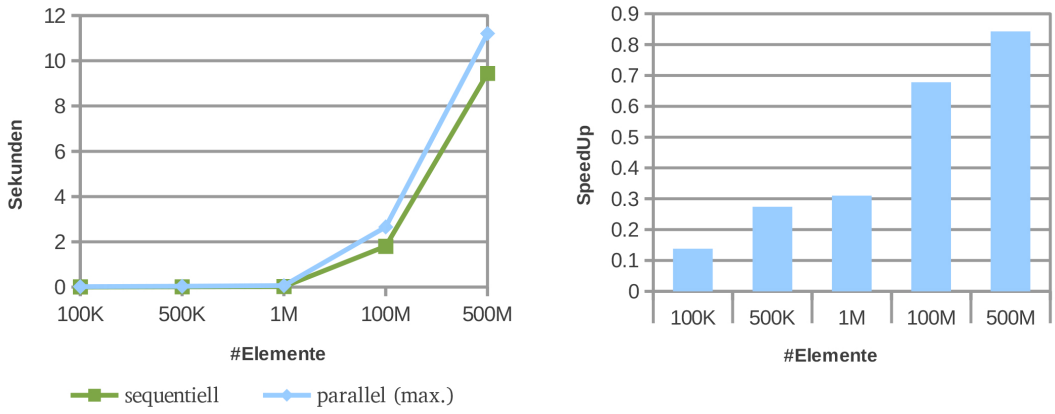
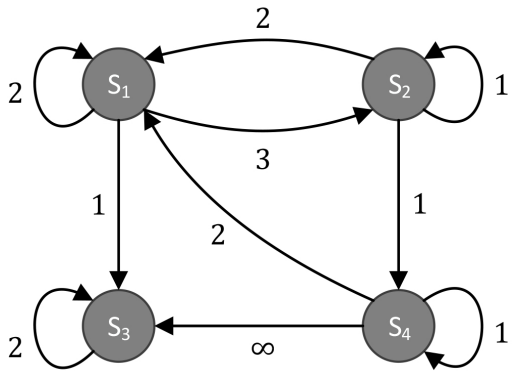


Abbildung 74: Laufzeit und SpeedUp von Beispiel 6

Es konnten 3 von 7 `for`-Schleifen der aufgespaltenen Version parallelisiert werden (siehe Abbildung 75). Die maximale Beschleunigung bei 500M Elementen ist 0,85, d.h. der BEP wurde unter Vorbehalt der Eingeabegrößen nie erreicht. Aufgrund der steigenden Beschleunigung (von 0,14 bis 0,85), lässt sich jedoch vermuten, dass der BEP jenseits von 500M Elementen noch erreicht wird.

Für die folgenden beiden Messbeispiele 7 und 8 wurden noch einmal die Erkenntnisse aus den ersten beiden Messbeispielen für größere Dimensionen getestet.




```
for (i in 1:n) {
  forall (j in 1:n) {
    S3
  }
  for (j in 1:n) {
    forall (k in 1:n) {
      S1
    }
    for (k in 1:n) {
      S2
    }
  }
}
forall (i in 1:n) {
  for (j in 1:n) {
    S4
  }
}
```

Abbildung 75: Teilergebnisse von Messung 6

7. Messbeispiel

Als Grundlage dient ein einfaches, arbiträres Code-Beispiel mit Matrizen a und b der Dimension 5. Ferner ist zu bemerken, dass Datenabhängigkeiten auf hoher Schleifenebene erzwungen wurden:

```
for (i in 0:(n-1)) {
  for (j in 1:n) {
    for (k in 0:(n-1)) {
      for (l in 1:n) {
        for (m in 1:n) {
          S1: a[i,j,k,l,m] <- a[i,j-1,k,l,m] + a[i,j,k+1,l,m]
                - b[i,j,k,l,m]
          S2: b[i,j,k,l,m] <- b[i+1,j,k,l-1,m-1]
                + a[i,j-1,k,l,m]
        }
      }
    }
  }
}
```

Listing 23: Sequentielles R-Programm für Beispiel 7

Die Messungen wurden mit 500K, 1M, 100M, 250M und 500M Matrixelementen durchgeführt.

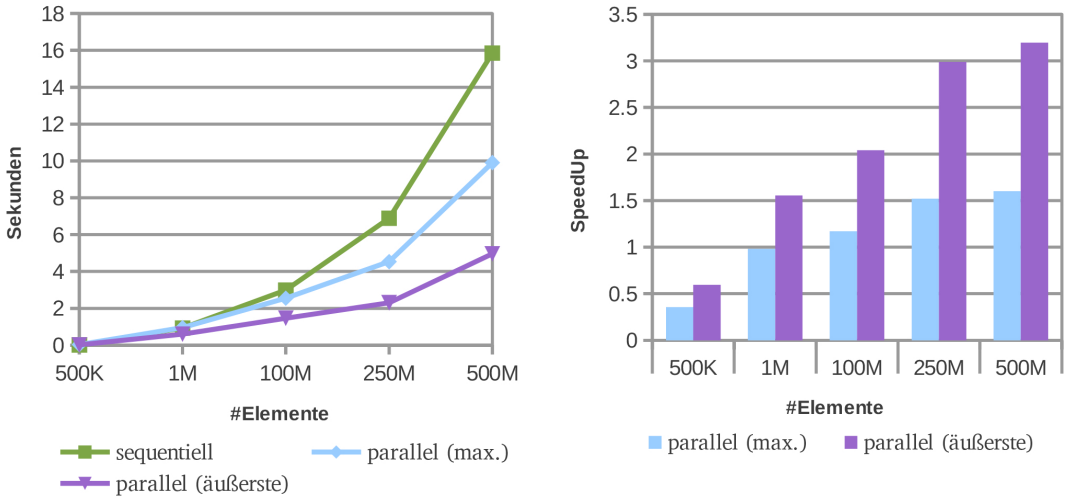


Abbildung 76: Laufzeit und SpeedUp von Beispiel 7

Die berechneten Teilergebnisse sind in Anhang E einsehbar. Die Leistungsdaten ergeben eine durchschnittliche[15] Beschleunigung von ungefähr 1,1 und maximal 1,6 bei 500M Elementen für das maximal parallelisierte Programm. Für die angepasste Version[16] (vergleiche Fall 2 aus Messbeispiel 1) ist die Beschleunigung um durchschnittlich 79% höher mit maximal 3,2 für 500M Elemente. Tendenziell steigt dieser Leistungszuwachs für Eingabegrößen jenseits 500M. Für den BEP gilt:

	parallel (max.)	*parallel (äußerste)*
BEP	**~5M**	**~750K**

8. Messbeispiel

Es wird eine leicht angepasste Version von Messbeispiel 7 verwendet. In diesem Fall wurden Datenabhängigkeiten auf tiefer Schleifenebene erzwungen. Die Komplexität der Berechnungen wurde nicht verändert:

[15] über alle Eingabegrößen
[16] Ausschließlich die Schleifen in Zeile 3 und 11 werden parallel ausgeführt (siehe Abbildung 85)

```
for (i in 0:(n-1)) {
  for (j in 1:n) {
    for (k in 0:(n-1)) {
      for (l in 1:n) {
        for (m in 1:n) {
          S1: a[i,j,k,l,m] <- a[i,j,k-1,l-1,m] + a[i,j,k,l,m+1]
                - b[i,j,k,l,m]
          S2: b[i,j,k,l,m] <- b[i,j,k,l-1,m-1] + b[i,j-1,k,l,m+1]
        }
      }
    }
  }
}
```

Listing 24: Sequentielles R-Programm für Beispiel 8

Die Messungen wurden ebenfalls mit 500K, 1M, 100M, 250M und 500M Matrixelementen durchgeführt um eine Vergleichbarkeit zu gewährleisten.

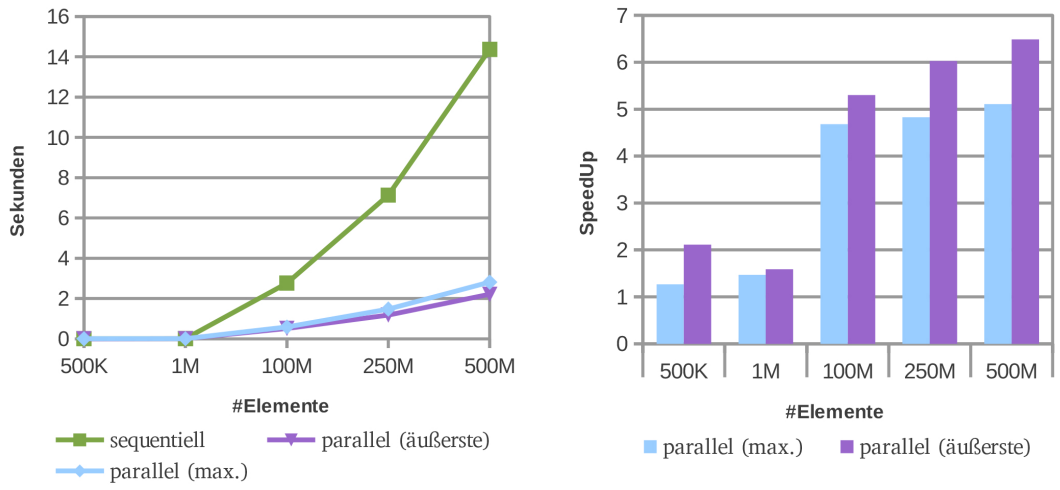


Abbildung 77: Laufzeit und SpeedUp von Beispiel 8

Es ergibt sich eine durchschnittliche Beschleunigung von ungefähr 3,4 und maximal 5,1 bei 500M Elementen für das maximal parallelisierte Programm. Der Leistungsunterschied zwischen den beiden parallelen Versionen wurde im Durchschnitt auf 26% reduziert.
Interessant ist ein Vergleich der Leistungsdaten mit denen des vorhergehenden Messbeispiels:

	Messbeispiel 7	Messbeispiel 8
parallel (max.)		
parallele Laufzeit (500M)	**9,9060**s	**2,8124**s
Beschleunigung (500M)	**1,6**	**5,1**
durchschnittl. Beschleunigung	**1,1**	**3,5**
BEP	**∼5M**	**∼300K**

	Messbeispiel 7	Messbeispiel 8
parallel (äußerste)		
parallele Laufzeit (500M)	**4,9624**s	**2,2148**s
Beschleunigung (500M)	**3,2**	**6.5**
durchschnittl. Beschleunigung	**2,1**	**4,3**
BEP	**∼750K**	**∼100K**

Die dargestellten Leistungsdaten belegen, dass der Leistungszuwachs eines parallelisierten Programms gegenüber des sequentiellen Gegenstücks stark davon abhängt in welcher Schleifentiefe sich eine parallele `for`-Schleife befindet.
Ferner wird durch eine Parallelisierung auf höherer Schleifeneben der BEP bereits für geringere Eingabegrößen erreicht.

Nicht parallelisierbare Beispiele

In einigen Testfällen aus Kapitel 6.1 wurde bereits auf die Schwächen des *AK*-Algorithmus hingewiesen. An dieser Stelle sollen diese noch einmal anhand von zwei reellen Beispielen verdeutlicht werden:

1. Longest-Common-Subsequence (LCS)

Die Berechnung der längsten gemeinsamen Teilfolge zweier oder mehrerer Zeichenfolgen stammt aus dem Bereich der dynamischen Programmierung. Das Code-Beispiel wurde der Arbeit [Pfa12] von D. Pfaff entnommen. Die sequentielle R-Implementierung ist in folgendem Listing 25 enthalten:

```
1 for (i in 2:m) {
2   for (j in 2:n) {
3     S1: d[i,j] <- max(max(d[i,j-1]+2, d[i-1,j]+3),
4           d[(i-1,j-1]+1)
5   }
6 }
```

Listing 25: R-Programm für "LCS"

Nach erfolgreicher Erkennung findet die Datenabhängigkeitsanalyse von SURE nur ein Statement S_1 mit drei echten Abbhängigkeiten. Zwei davon mit Abhängigkeitsgrad 1 und eine mit Abhängigkeitsgrad 2. Folglich wird der in Abbildung 78 dargestellte RLDG berechnet.

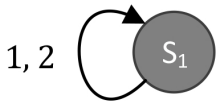


Abbildung 78: RLDG zu Listing 25

Der *AK*-Algorithmus realisiert die Abhängigkeiten mit Abhängigkeitsgraden 1 bzw. 2 und entscheidet, dass keine der beiden Schleifen parallelisierbar ist. Das ursprüngliche (sequentielle) Programm wird zurückgegeben.
Geht man davon aus, dass die Abhängigkeiten in Schleifentiefe 1 nicht existieren, dann würde der entsprechende EDG wie in Abbildung 79 aussehen.

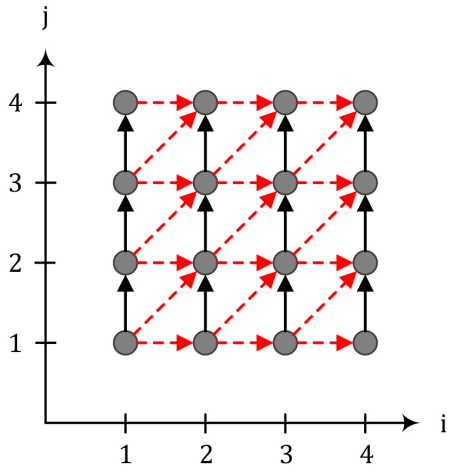


Abbildung 79: EDG zu Listing 25 mit entfernten (roten) Abhängigkeiten

Aus einem entsprechenden RLDG ohne Datenabhängigkeiten mit Abhängigkeitsgrad 1 kann der AK-Algorithmus ein parallelen Ablaufplan berechnen. Dies führt zu dem parallelisierten Pseudo-Code in Listing 26:

```
1 forall (i in 2:m) {
2   for (j in 2:n) {
3     S1: d[i,j] <- max(max(d[i,j-1]+2, d[i,j]+3),
4           d[i,j-1]+1)
5   }
6 }
```

Listing 26: Parallelisiertes R-Programm in Pseudo-Code für "LCS"

Auf Basis des parallelisierten Programms aus Listing 26 wurden nun Leistungsdaten gesammelt. Als Eingabegrößen dienten 1M, 50M, 100M und 400M Elemente:

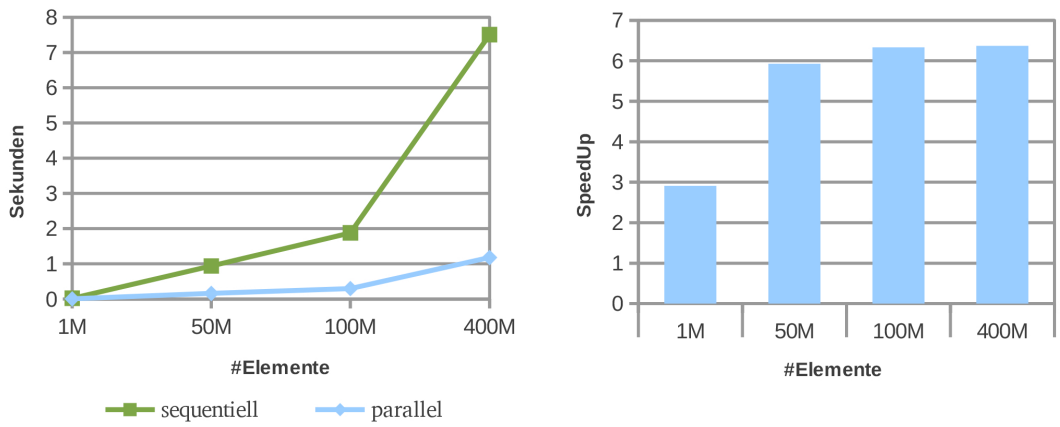


Abbildung 80: Laufzeit und SpeedUp des "LCS"-Beispiels

Klar ist, dass es sich hierbei nur um eine theoretische Leistungsermittlung handelt. Durch die Manipulation der Abhängigkeiten wird das zu berechnende Ergebnis vollkommen verfäscht. Es ist viel mehr Ziel dieses (und des nächsten) Versuchs einen Anhaltspunkt für zukünftige Arbeiten zu liefern.
Aus den Daten in Abbildung 80 lässt sich eine Beschleunigung von bis zu 6,4 für 400M Matrixelemente ablesen. Im Durchschnitt über alle Eingabegrößen liegt die Beschleunigung bei 5,4.

2. Gauss-Seidel-Iteration (GSI)

Das Gauss-Seidel-Verfahren stammt aus der numerischen Mathematik zur näherungsweisen Lösung linearer Gleichungssysteme. Das Beispiel wurde aus [DV96b] entnommen und wird in seiner sequentiellen Form durch Listing 27 repräsentiert:

```
1 for (i in 2:(m-1)) {
2   for (j in 2:(n-1)) {
3     S1: g[i,j] <- (f[i,j] + g[i-1,j] + g[i+1,j]
4           + g[i,j-1] + g[i,j+1]) / 4
5   }
6 }
```

Listing 27: R-Programm für "GSI"

Durch die Datenabhängigkeitsanalyse werden zwei echte und zwei Gegenabhängigkeiten mit jeweiligen Abhängkeitsgraden 1 und 2 erkannt. Der berechnete RLDG[17] ist Abbildung 81 zu entnehmen.

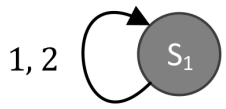


Abbildung 81: RLDG zu Listing 27

Der *AK*-Algorithmus agiert identisch zum vorhergehenden "LCS"-Beispiel. Es wird wieder das sequentielle Programm zurückgeliefert.
Erneut wird davon ausgegangen, dass alle Abhängigkeiten in Schleifentiefe 1 nicht existieren. Der entsprechende EDG ist in Abbildung 82 dargestellt.
Der *AK*-Algorithmus berechnet aus einem analogen RLDG den parallelisierten Pseudo-Code in Listing 28.

[17] Identisch zu RLDG des "LCS"-Beispiels, da mehrfache Abhängigkeiten mit gleichem Abhängigkeitsgrad ignoriert werden

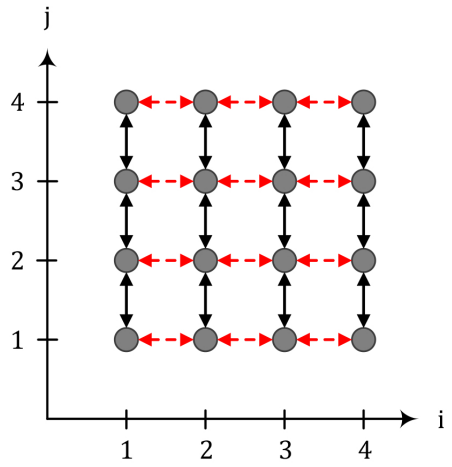


Abbildung 82: EDG zu Listing 27 mit entfernten Abhängigkeiten

```
1 forall (i in 2:(m-1)) {
2   for (j in 2:(n-1)) {
3     S1: g[i,j] <- (f[i,j] + g[i,j] + g[i,j]
4           + g[i,j-1] + g[i,j+1]) / 4
5   }
6 }
```

Listing 28: Parallelisiertes R-Programm in Pseudo-Code für "GSI"

Die Messungen wurden mit 1M, 50M, 100M und 400M Matrixelementen durchgeführt (siehe Abbildung 83.

Aus der theoretischen Leistungsermittlung (siehe Abbildung 83 geht eine Beschleunigung von bis zu 6.3 für 400M Matrixelemente hervor. Im Durchschnitt über alle Eingabegrößen liegt die Beschleunigung bei 5.1. Aufgrund der ähnlichen Programm- und Schleifenstruktur des "LCS"- und des "GSI"-Beispiels unterscheiden sich Laufzeit, als auch Beschleunigung nur marginal von einander.

Zusammenfassend lässt sich sagen, dass sich für steigende Eingabegrößen eine annähernd immer höhere Beschleunigung einstellt. Besonders deutlich zeigen dies die angepassten, parallelen Versionen aus Messbeispielen 2 und 8.
Die Beschleunigung eines parallelisierten Programms gegenüber eines sequentiellen ist umso höher, je höher die Schleifenebene ist, auf der sich eine parallele Schleife befindet. Ergibt sich aus einer Parallelisierung Programm-Code mit ausschließlich parallelen Schleifen auf tiefster Schleifenebene, wird der BEP erst für sehr große

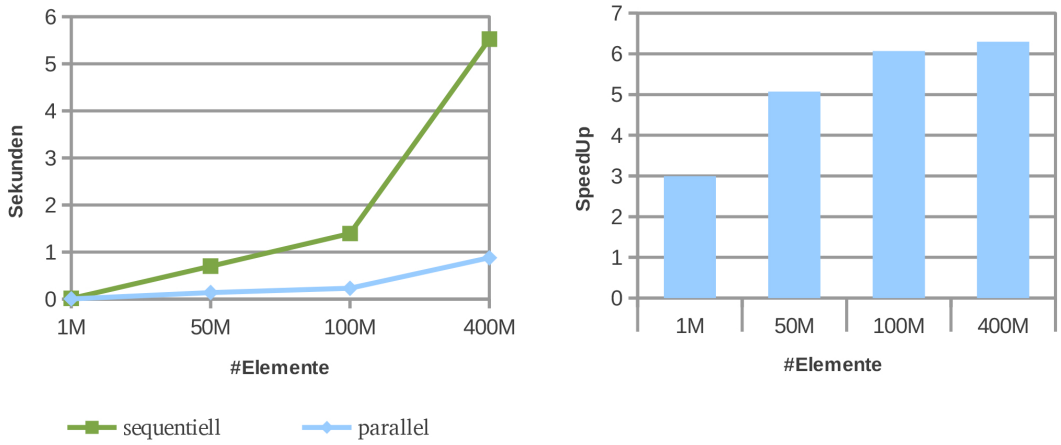


Abbildung 83: Laufzeit und SpeedUp des “GSI“-Beispiels

Eingaben erreicht. Ein Programm mit mehreren verschachtelten parallelen Schleifen erzielte in allen Tests eine niedrigere Beschleunigung als das selbe Programm, dessen parallele Schleifen auf die äußerste reduziert wurde.

7. Schlussfolgerung

Diese Arbeit beschreibt die theoretischen Grundlagen, den Entwurf und die Realisierung des SURE-Verfahrens und seine Integration in die R-Experimentierplattform ALCHEMY. Es wurde ein Transmutator zur Parallelisierung von verschachtelten Schleifen erzeugt.
Es wurde belegt, dass durch die Verwendung eines definierten Musters, SUREs in der XML-Repräsentation eines *AIR Programms* erfolgreich erkannt werden können. Ferner wurde gezeigt, dass die implementierte Datenabhängigkeitsanalyse nahezu alle Abhängigkeiten in gängigen Programmbeispielen erkennt. Der Algorithmus von Allen und Kennedy wurde implementiert und eine Basis für die Implementierung weiterer SURE-Algorithmen wurde geschaffen. Der durch das `SURE`-Modul parallelisierte AIR-Code lässt sich unter Einhaltung der analysierten Einschränkungen problemlos durch das `ArBB`-Backend ausführen.
Die Schwächen des SURE-Verfahrens unter Verwendung des Algorithmus von Allen und Kennedy wurden hinsichtlich Ansatz und Implementierung aufgezeigt. Die durchgeführten Messungen belegten unter Vorbehalt des gewählten Backends die folgenden Annahmen:

1. Die Beschleunigung eines parallelisierten Programms gegenüber eines sequentiellen ist umso höher, je höher die Schleifenebene ist, auf der sich eine parallele Schleife befindet

2. Ein Programm mit mehreren verschachtelten parallelen Schleifen erzielt eine niedrigere Beschleunigung, als das selbe Programm, dessen parallele Schleifen auf die äußerste reduziert wurde

Generell wurde gezeigt, dass durch eine Parallelisierung mit dem SURE-Verfahren gute Resultate erzeugt werden können, mit absehbarem Potential für Optimierung.

Ausblick

Zukünftige Arbeiten in diesem Bereich könnten folgende Themen beinhalten:

- Die Datenabhängigkeitsanalyse ist zum aktuellen Stand nur rudimentär implementiert. Die Realisierung mächtigerer SIV- und vor allem MIV-Tests für eine genauere Bestimmung von Datenabhängigkeiten würde die Einschränkungen der Indiziierungsfunktion beheben. In Folge dessen könnten weitaus komplexere Szenarien untersucht werden.

- Das `SURE`-PAM unterstützt aktuell nur einen SURE-Algorithmus. Durch eine Implementierung weiterer SURE-Algorithmen, können bisher nicht parallelisierbare Programme analysiert und parallelisiert werden. Von besonderem

Interesse dürften die Algorithmen von Wolf und Lam bzw. Darte und Vivien sein, welche auf einer präziseren Abhängigkeitsapproximation operieren, dem RDG mit Richtungsvektoren.

- Das in ALCHEMY integrierte `ArBB`-Backend weist keine gute Unterstützung für das von `SURE` verwendete `DOPAR`-Skeleton auf. Ein ähnliches Backend auf Basis von *OpenMP*, welches in den durchgeführten Messbeispielen überzeugende Laufzeiten lieferte, würde ein gute Alternative darstellen.

Literatur

[AK87] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 1987.

[Ana13] Revolution Analytics. domc: Foreach parallel adaptor for the multicore package, 2013. http://cran.r-project.org/web/packages/doMC/index.html.

[BC90] Guy Blelloch and Siddhartha Chatterjee. VCODE: A Data-Parallel Intermediate Language. In *In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[Bel03] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[BMR98] Frank Buschmann, Regine Meunier, and Hans Rohnert. *Pattern-orientierte Software-Architektur: A pattern system*. Addison-Wesley, Bonn, 1998.

[Boa] OpenMP Architecture Review Board. OpenMP Spezifikation. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[Boa97] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming, 1997. http://http://openmp.org/wp/.

[Col] Murray Cole. Skeletal Parallelism. http://homepages.inf.ed.ac.uk/mic/Skeletons/.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel computation*. MIT Press, 1991.

[Dar] Alain Darte. Detection of parallel loops: from Allen-Kennedy?s algorithm (the simplest algorithm) to multi-dimensional affine schedules (Feautrier?s algorithm). http://perso.ens-lyon.fr/alain.darte/DEA04/loops.pdf?

[Dar98] Alain Darte. *Mathematical Tools for Loop Transformations: From Systems of Uniform Recurrence Equations to the Polytope Model*, pages 147–183. Springer Verlag, 1998.

[DRV01] Alain Darte, Yves Rober, and Frédéric Vivien. *Compiler Optimizations for Scalable Parallel Systems*, chapter Loop Parallelization Algorithms, pages 141–171. Springer Verlag, 2001.

[DV95] Alain Darte and Frédéric Vivien. A Comparison of Nested Loops Parallelization Algorithms. Technical Report RR95-11, ENS Lyon, 1995.

[DV96a] Alain Darte and Frédéric Vivien. On the Optimality of Allen and Kennedy's Algorithm for Parallel Extraction in Nested Loops. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, pages 379–388. Springer-Verlag, 1996.

[DV96b] Alain Darte and Frédéric Vivien. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, 1996.

[GR88] Alan Gibbons and Wojciech Rytter. Efficient parallel algorithms. 1988.

[JS03] Tim Jacobsen and Gregg Stubbendieck. Dependency Analysis of FOR-Loop Structures for Automatic Parallelization of C Code, 2003. http://www.cse.unt.edu/ sweany/CSCE5650/HANDOUTS/Jacobson.pdf.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., 2002.

[Kie13] Marc Aurel Kiefer. A Backend for the Translation of AIR Intermediate Code Into Intel ArBB for the R Parallelization Platform ALCHEMY. Master's thesis, Saarland University, 2013.

[KMM+05] Kazuhiko Kakehi, Kiminori Matsuzaki, Akimasa Morihata, Kento Emoto, and Zhenjiang Hu. Parallel Dynamic Programming using Data-Parallel Skeletons. In *In Proceedings of the 22nd JSSST Conference*, 2005.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *J. ACM*, 1967.

[Kri09] Wim P. Krijnen. Applied Statistics for Bioinformatics using R. *The R Journal*, 2009. http://cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf.

[Kum02] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[Kyr07] Konstantinos Kyriakopoulos. *Advancing Data Dependence Analysis in Practice*. Dissertation, University of Texas at San Antonio, 2007. http://www.cs.utsa.edu/uploads/theses/Kostas.pdf.

[Lam74] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, 1974.

[Mir11] Michael Mirold. ALCHEMY - An Experimentation Laboratory for the Automatic Parallelization of Programs Written in the R Language. Master's thesis, Saarland University, 2011.

[MLS07] Xiaosong Ma, Jiangtian Li, and Nagiza F. Samatova. Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing. In *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[NLR01] Michael Na Li and A.J. Rossini. RPVM: Cluster Statistical Computing in R. *R News*, 1(3), 2001. http://cran.r-project.org/doc/Rnews.

[NSs94] Esko Nuutila and Eljas Soisalon-soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 1994.

[Par12] Terence Parr. Code Generation Targets, 2012. http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, 2013.

[PAS08] Ignacio Peláez, Francisco Almeida, and Fernando Suárez. DPSKEL: A Skeleton Based Tool for Parallel Dynamic Programming. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, 2008.

[Pfa12] David Pfaff. Implementation of a Method for Automatic Parallelization of Dynamic Programming in R-programs using Data-parallel Programming Patterns. Bachelor's thesis, Saarland University, 2012.

[PM12] Frank Padberg and Mirold Michael. An Experimentaion Platform for the Automatic Parallelization of R Programs. *APSEC 2012*, 2012.

[RK99] Doug Rosenberg and Scott Kendall. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley, 1999.

[Tea] R Developement Core Team. The R Project for Statistical Computing. http://www.r-project.org/.

[Urb] Simon Urbanek. multicore - Parallel processing in R on machines with multiple cores or CPUs. http://www.rforge.net/multicore/index.html.

[Wes] Steve Weston. Foreach looping construct for R. http://cran.r-project.org/web/packages/foreach/foreach.pdf.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. *SIGPLAN Not.*, 1991.

[WLY85] B. W. Wah, Guo-jie Li, and Chee Fen Yu. Multiprocessing of Combinatorial Search Problems. *Computer*, 18(6):93–108, 1985.

[Yu02] Hao Yu. Rmpi: Parallel Statistical Computing in R. *R News*, 2(2), 2002. http://cran.r-project.org/doc/Rnews.

A. AIR Beispiel

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <AIR>
3    <Program>
4      <ForStmt>
5        <condition>
6          <IteratorExpr>
7            <iterator>
8              <SymbolExpr name="i" />
9            </iterator>
10           <collection>
11             <SymbolExpr name="n" />
12           </collection>
13         </IteratorExpr>
14       </condition>
15       <body>
16         <ExprList>
17           <BinopExpr op="&lt;-">
18             <lhs>
19               <SubscriptExpr>
20                 <collection>
21                   <SymbolExpr name="a" />
22                 </collection>
23                 <subscripts>
24                   <SymbolExpr name="i" />
25                   <ConstantExpr type="real">
26                     <RealValue data="1.0" />
27                   </ConstantExpr>
28                 </subscripts>
29               </SubscriptExpr>
30             </lhs>
31             <rhs>
32               <ConstantExpr type="real">
33                 <RealValue data="0.0" />
34               </ConstantExpr>
35             </rhs>
36           </BinopExpr>
37         </ExprList>
38       </body>
39     </ForStmt>
40   </Program>
41 </AIR>
```

Listing 29: Vollständige AIR von Listing 1 als XML-Code

B. Richtungsvektoren

Ein *Distance Set* (siehe Kapitel 3.3) kann durch einen n-dimensionalen Vektor, den "Richtungsvektor", repräsentiert werden. Seine Komponenten sind aus

$$\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+, -\})$$

Die i-te Komponente stellt dabei eine Approximation der i-ten Komponenten aller möglichen Distanzvektoren dar:

z	Falls die Abhängigkeit uniform in dieser Dimension mit eindeutigem Wert z ist
$z+$ (bzw. $z-$)	Falls alle i-ten Komponenten größer (bzw. kleiner) als z sind
$*$	Falls beliebig

Generell wird für den Fall $1+$ (bzw. $(-1)-$) die Notation $+$ (bzw. $-$) verwendet. Diese Repräsentation findet seine Verwendung in den PAs aus [WL91] und [DV96b]. Ein entsprechender RDG mit Richtungsvektoren ist in Abbildungen 84 und 9 dargestellt.

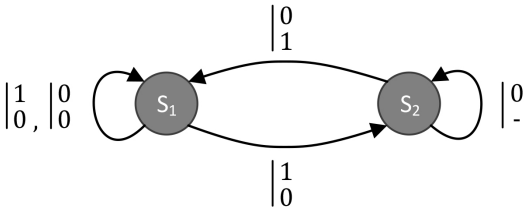
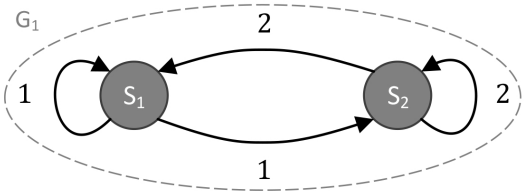


Abbildung 84: RDG von Listing 3 mit Richtungsvektoren

C. Durchführung des Algorithmus von Allen und Kennedy

Eingabe für die Durchführung des Algorithmus ist der RLDG aus Abbildung 10. Der Einfachheit halber werden Eigenkanten mit Abhängigkeitsgraden $c_i = \infty$ entfernt, da diese keine Aussagekraft besitzen. In den Abbildungen sind ausschließlich die reduzierten RLDGs $G(k)$ dargestellt. Ferner ist zum Vergleich in jedem Schritt der transformierte Pseudo-Code angegeben.

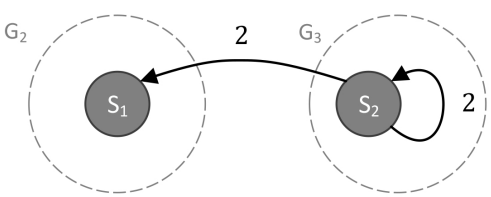
1. `Allen-Kennedy(`G, `1):`




```
for i=1 to n
    Allen-Kennedy(G1,2)
endfor
```

Für den initialen Aufruf wird lediglich eine SCC G_1 für $G(1)$ berechnet. Weil nur eine einzige SCC existiert ist folglich keine Sortierung oder *loop distribution* nötig. G_1 besitzt 2 Kanten mit $c_i = k = 1$, also muss die Schleife der Tiefe $k = 1$ sequentiell aus geführt werden (Markierung mit "for").

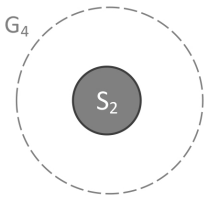
2. `Allen-Kennedy(`G_1, `2):`




```
for i=1 to n
    for j=1 to n
        Allen-Kennedy(G3,3)
    endfor
    forall j=1 to n
        Allen-Kennedy(G2,3)
    endforall
endfor
```

Für $G_1(2)$ lassen sich 2 SCCs G_2 und G_3 mit der topologischen Sortierung $\{G_3, G_2\}$ berechnen. Unter Berücksichtigung dieser wird nun eine *loop distribution* durchgeführt. G_3 besitzt eine Kante mit $c = 2$, also ist keine parallele Ausführung möglich. G_2 besitzt keine Kante mit $c = 2$, weshalb diese Schleife der Tiefe $k = 2$ parallel ausgeführt werden kann (Markierung mit "forall").

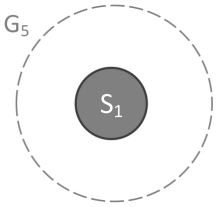
3. `Allen-Kennedy(`G_3`, 3)`:




```
for i=1 to n
    for j=1 to n
        forall k=1 to n
            S2
        endforall
    endfor
    forall j=1 to n
        Allen-Kennedy(G2,3)
    endforall
endfor
```

Für $G_3(3)$ wird SCC G_4 berechnet, welche keine Kanten mehr besitzt. D.h. alle Schleifen ab der aktuellen Tiefe $k = 3$ können mit "forall" markiert werden. Anstelle eines weiteren rekursiven Aufrufs wird gleich die Operation S_2 in den Schleifenrumpf geschrieben.

4. `Allen-Kennedy(`G_2`, 3)`:




```
for i=1 to n
    for j=1 to n
        forall k=1 to n
            S2
        endforall
    endfor
    forall j=1 to n
        forall k=1 to n
            S1
        endforall
    endforall
endfor
```

Die Arbeitsschritte für $G_2(3)$ werden analog zu Schritt 3 durchgeführt.

D. Beispiel-Code der Messungen

Im Folgenden die Programm-Code-Beispiele aus Messung 1 (siehe Kapitel 6.2). Aus Gründen der Einfachheit wurden etwaige Initialisierungen ausgespart:

Verwendung des *doMC*-Backends

Die Befehlsfolge für die Ausführung des sequentiellen R-Codes:

```
1 > R
2 > library(doMC)
3 > registerDoMC(8)
4 > {
5 + y <- foreach (j=1:n .combine='cbind') %do% {
6 +
7 +       foreach (i=1:n .combine='c') %do% {
8 +         y[i] + a[i, j] * x[j]
9 +       }
10 + }
11 + }
```

Damit `foreach` von *doMC* verwendet werden kann, ist es nötig die `registerDoMC()`-Methode aufzurufen. Sie erhält die Anzahl an verfügbaren Prozessorkernen als Parameter übergeben. Aus der Paketbeschreibung [Wes] erfährt man, dass das `foreach`-Konstrukt die Iteration über alle Elemente eines Vektors erlaubt. Speziell ist im Vergleich zu der R-eigenen `lapply`-Funktion, der definierbare Rückgabewert. Dieser wird mittels `.combine`-Funktion definiert:

`'c'` Konkateniert das berechnete Ergebnis zu einem Vektor

`'cbind'` Kombiniert Vektoren zu einer Matrix

Für eine sequentielle Berechnung wird der Operator %do% verwendet. Das parallele Gegestück mit Operator %dopar% entspricht:

```
1 > ...
2 > {
3 + y <- foreach (j=1:n .combine='cbind') %dopar% {
4 +
5 +       foreach (i=1:n .combine='c') %dopar% {
6 +         y[i] + a[i, j] * x[j]
7 +       }
8 + }
9 + }
```

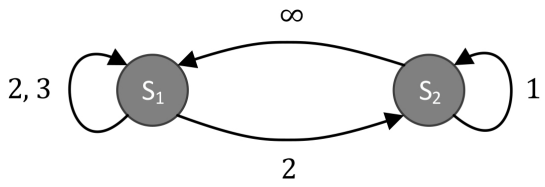
Verwendung der Programmierschnittstelle *C/OpenMP*

Paralleles C-Code-Beispiel:

```
1 #pragma omp parallel for
2 for (int i=0; i< N; i++) {
3
4   #pragma omp parallel for
5   for (int j=0; j<N; j++) {
6     y[i] = y[i] + a[i][j] * x[j];
7   }
8 }
```

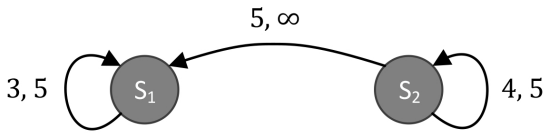
Laut der *OpenMP*-Spezifikation werden sämtliche Compiler-Direktive durch den `#pragma`-Mechanismus eingeleitet. Im Beispiel wird eine vordefinierte Abkürzung des `parallel`-Konstrukts zur parallelen Ausführung einer oder mehrerer Schleifen verwendet. Der Hauptausführungsstrang (eng. *Thread*) unterteilt das Programm in weitere Ausführungsstränge für eine parallele Abarbeitung des umschlossenen Programmteils. Eine ausführliche Beschreibung des Ausführungsmodells ist ebenfalls der Spezifikation [Boa] zu entnehmen.

E. Teilergebnisse von Messbeispiel 7 und 8




```
for (i in 0:(n-1)) {
  for (j in 1:n) {
    forall (k in 0:(n-1)) {
      forall (l in 1:n) {
        forall (m in 1:n) {
          S2
        }
      }
    }
    for (k in 0:(n-1)) {
      forall (l in 1:n) {
        forall (m in 1:n) {
          S1
        }
      }
    }
  }
}
```

Abbildung 85: Teilergebnisse von Messbeispiel 7




```
forall (i in 0:(n-1)) {
  forall (j in 1:n) {
    forall (k in 0:(n-1)) {
      for (l in 1:n) {
        for (m in 1:n) {
          S2
        }
      }
    }
  }
  forall (j in 1:n) {
    for (k in 0:(n-1)) {
      forall (l in 1:n) {
        for (m in 1:n) {
          S1
        }
      }
    }
  }
}
```

Abbildung 86: Teilergebnisse von Messbeispiel 8