Domänenspezifische Sprachen zur Umsetzung numerischer Lösungsverfahren für gewöhnliche Differentialgleichungssysteme

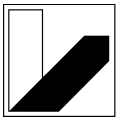
Benedikt Gleißner

Bayreuth Reports on Parallel and Distributed Systems

No. 12, March 2019

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de



UNIVERSITÄT
BAYREUTH

Fakultät für Mathematik, Physik und Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik II

Masterarbeit

Domänenspezifische Sprachen zur Umsetzung numerischer Lösungsverfahren für gewöhnliche Differentialgleichungssysteme

Benedikt Gleißner

Mat.-Nr.: 1305759

26. März 2019

Betreuung:
Dr. Matthias Korch

Zusammenfassung

Domänenspezifische Sprachen (domain-specific languages, DSLs) werden dazu eingesetzt, Lösungen für ein bestimmtes Problemfeld umzusetzen. Sie sind im Gegensatz zu General Purpose Languages (GPLs), wie Java oder C, speziell für Probleme innerhalb einer Domäne optimiert, wodurch Programme in der Regel einfacher und schneller implementiert werden können. Mögliche Fehlerquellen im Code können durch Validierungsmechanismen und die Struktur der DSL selbst frühzeitig erkannt und behoben werden. Durch automatische Codeerzeugung und Autotuning können aus einem mit einer DSL geschriebenem Programm mehrere Codevarianten erzeugt werden, die auf verschiedene Systeme portierbar sind und gleichzeitig auf dem jeweiligen System eine hohe Performanz erreichen. Im Rahmen dieser Arbeit wurden mehrere DSLs zur Implementierung numerischer Lösungsverfahren für gewöhnliche Differentialgleichungssysteme (ordinary differential equations, ODEs) entwickelt. Die verbreiteten Lösungsverfahren für ODEs lassen sich durch Datenflussgraphen beschreiben, in dem die Berechnungen des Verfahrens durch Basisoperationen als Knoten abgedeckt sind. Als ersten Schritt hin zu einem fertigen Programm wird die Struktur des Datenflussgraphen in einer DSL spezifiziert. In einer weiteren DSL können dann Verfahrenskoeffizienten, wie die Stufenzahl oder ein Butcher-Tableau, deklariert werden. Mit diesen beiden Sprachen ist das Verfahren bereits vollständig beschrieben. Zwei weitere DSLs befassen sich mit einer möglichst effizienten Implementierung auf Grafikprozessoren (graphics processing units, GPUs). Lösungsverfahren von ODEs sind durch die Eigenschaften von GPUs, wie kleinen Caches oder weitem SIMD, oft durch den Speicher limitiert. Daher gilt es für eine möglichst hohe Performanz die Anzahl an Speicherzugriffen zu minimieren, was durch die Fusion mehrerer Basisoperationen zu einem Kernel erreicht werden kann. Eine DSL befasst sich daher mit der Abbildung von Knoten des Datenflussgraphen auf Kernel. Durch Transformationen auf dem Graphen als vorbereitenden Schritt kann die Anzahl an Speicherzugriffen noch stärker reduziert werden. Diese Transformationen können in einer weiteren DSL deklariert werden.

Abstract

Domain-specific languages (DSLs) are used to implement solutions for a specific problem area. In contrast to general purpose languages (GPLs), such as Java or C, they are specifically optimized for problems within a domain, which usually makes it easier and faster to implement programs. Possible sources of errors in the code can be detected and corrected in an early stage by validation mechanisms and the structure of the DSL itself. Through automatic code generation and autotuning, several code variants can be generated from a program written with a DSL. These code variants can be ported to different systems and simultaneously achieve high performance on the respective system. In this work several DSLs for the implementation of numerical solution methods for ordinary differential equations (ODEs) have been developed. All common ODE solvers can be described by data flow graphs, in which all calculations of the method are covered by basic operations as nodes. As a first step towards a finished program, the structure of the data flow graph is specified in a DSL. In a second DSL, method coefficients such as the number of steps or a butcher tableau can then be declared. With these two languages the procedure is already completely described. Two further DSLs are concerned with an efficient implementation on graphics processing units (GPUs). ODE solvers are often limited by the properties of GPUs, such as small caches or large SIMD. Therefore it is important to minimize the number of memory accesses in order to achieve the highest possible performance. This can be achieved by fusing several basic operations into one kernel. A DSL therefore deals with the mapping of nodes of the data flow graph to kernels. By transformations on the graph as a preparatory step, the number of memory accesses can be reduced even more. These so-called enabling transformations can be declared in an additional DSL.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit sowohl expliziten als auch impliziten Lösungsverfahren für Anfangswertprobleme von gewöhnlichen Differentialgleichungssystemen (ordinary differential equations, ODEs). Diese sind wie folgt definiert:

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0, \quad t \in [t_0, t_e]$$

$f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ ist dabei die Funktion der rechten Seite (right-hand-side function, RHS), welche das ODE definieren. $t \in \mathbb{R}$ ist die unabhängige Variable (oft die Zeit), $y : \mathbb{R} \to \mathbb{R}^n$ die Lösungsfunktion, die im Intervall $[t_0, t_e]$ berechnet werden soll, und y_0 der initiale Wert zum Zeitpunkt t_0.

Anfangswertprobleme von ODEs haben oft keine analytische Lösung und müssen daher numerisch gelöst werden. Typischerweise wird dabei ein Zeitschrittverfahren angewendet, das bei t_0 beginnt und das Intervall bis hin zu t_e durchschreitet. Dabei wird für jeden Zeitschritt k eine Approximation $y_{k+1} \approx y(t_{k+1})$ berechnet. Implizite Verfahren verwenden zur Berechnung einer Approximation y_{k+1} das Ergebnis y_{k+1} selbst, während explizite Verfahren nur Approximationen aus vorherigen Zeitschritten nutzen. Weiter lassen sich Lösungsverfahren in Ein- und Mehrschrittverfahren unterteilen, wobei Einschrittverfahren nur das Resultat des letzten Zeitschritts und Mehrschrittverfahren die Approximationen mehrerer vorhergehender Zeitschritte verwenden. Als Repräsentanten der Verfahrensklassen werden in dieser Arbeit explizite Runge-Kutta-Verfahren, Peer-Verfahren, Adams-Bashforth-Verfahren und implizite Adams-Moulton-Verfahren betrachtet.

Lösungsverfahren für ODEs lassen sich durch Datenflussgraphen beschreiben, in dem durch Basisoperationen als Knoten alle Berechnungen des Verfahrens abgedeckt sind. Die eingehenden Kanten eines Knotens beschreiben die Abhängigkeiten von Ergebnissen anderer Operationen, die für die Berechnung des aktuellen Knotens benötigt werden. Die ausgehenden Knoten beschreiben die Weiterverarbeitung des Ergebnisses des aktuellen Knotens durch andere Operationen.

Aufgrund der oft hohen Rechenintensität numerischer Lösungsverfahren von ODEs und den fortdauernden Weiterentwicklungen von Parallelrechensystemen mit enormer und immer steigender Rechenleistung in den letzten Jahrzehnten wurden mehrere Methoden zur Parallelisierung dieser Verfahren vorgeschlagen. Diese Methoden werden gewöhnlich durch das Ausnutzen der Parallelität über das Verfahren, des Systems oder der Zeit klassifiziert[11]. Grafikprozessoren (graphics processing units, GPUs) sind hoch parallele Prozessoren mit vielen Kernen, die für einen hohen Datendurchsatz ausgelegt sind. Daher sind GPUs gut geeignet zur Beschleunigung von Berechnungen numerischer Lösungsverfahren. Dabei sind

jedoch bestimmte einschränkende Eigenschaften von GPUs zu beachten. So arbeiten GPUs gewöhnlich mit einer großen Zahl an Threads, haben kleine Caches und weiten SIMD. Wegen der hohen Zahl an Threads wird ein hoher Grad an Parallelität benötigt, der nicht durch den Ansatz der Parallelität über das Verfahren bereitgestellt werden kann. Stattdessen kann hier die Parallelität über das System genutzt werden.

In ihrem Artikel *Accelerating explicit ODE methods on GPUs by kernel fusion*[16] beschreiben Korch und Werner eine Technik der Kernel Fusion, mit der numerische Lösungsverfahren von ODEs auf GPUs erheblich beschleunigt werden können. Dabei wird die Tatsache ausgenutzt, dass die Auswertung der RHS Funktion bei vielen Anwendungen nur wenig Rechenaufwand benötigt und die ODE Verfahren daher oft durch den Speicher limitiert sind. Die Berechnung kann folglich beschleunigt werden, indem die Anzahl an Speicherzugriffen minimiert wird. Durch eine geschickte Verschmelzung mehrerer Basisoperationen zu einem Kernel finden weniger Zugriffe auf den Speicher statt, was sich deutlich auf die Laufzeit des Programms auswirkt. Durch eine vorherige Anwendung von sogenannten *Enabling Transformations* auf den Datenflussgraphen und anschließender Kernelfusion konnten Korch und Werner bestimmte Lösungsverfahren noch zusätzlich beschleunigen.

Domänenspezifische Sprachen (domain-specific languages, DSLs) werden dazu eingesetzt, Lösungen für ein bestimmtes Problemfeld umzusetzen. Sie sind im Gegensatz zu General Purpose Languages (GPLs), wie Java oder C, speziell für Probleme innerhalb einer Domäne optimiert. Dadurch haben DSLs folgende Vorteile im Vergleich zu GPLs[20]:

- *Produktivitätssteigerung:*
  Die für eine Domäne speziell entwickelte Sprache ist auf die Funktionen innerhalb der Domäne zugeschnitten und somit leicht lesbar. Die DSL beschränkt sich auf wesentliche Funktionen zur Lösung von domänenspezifischen Problemen und automatisiert andere Aspekte, wie beispielsweise das Speichermanagement. Die limitierte Ausdrucksmächtigkeit von DSLs beseitigt viele Fehlerquellen und durch speziell angepasste Validatoren können Fehler schnell erkannt und behoben werden.

- *Kommunikation mit Domänen Experten:*
  Für Domänen Experten ist ein DSL Programm leicht lesbar. Dadurch bekommen diese ein besseres Verständnis über die Funktionsweise des Systems und können so leichter mit Programmierern kommunizieren.

- *Alternatives Berechnungsmodell:*
  Die meisten GPLs verwenden ein imperatives Modell, bei dem der Programmierer genau spezifiziert, was in welcher Reihenfolge ausgeführt werden soll. Mit Hilfe von DSLs lassen sich alternative Berechnungsmodelle, wie eine Modellierung eines Datenflussgraphen oder die Benutzung deklarativer Programmierung, umsetzten.

Die Verwendung von DSLs birgt jedoch auch einige Probleme[20]:

- *Kosten für die Erstellung:*
  Die Entwicklung und Wartung von DSLs verursacht Kosten. Der Einsatz von DSLs ist daher nur sinnvoll, wenn der Nutzen die Kosten übertrifft. Für die Entwicklung und Wartung von DSLs sind spezielle Kenntnisse erforderlich, die eventuell erst erlernt werden müssen.

- *Sprachen Kakophonie:*
  Neue Sprachen müssen immer erst erlernt werden, was mit einem gewissen Aufwand verbunden ist. Dieser hält sich jedoch in der Regel im Vergleich zum Erlernen einer GPL in Grenzen, da eine GPL viel umfangreicher ist.

- *DSL wird zur GPL:*
  Wird die DSL ständig um neue Features erweitert um auf eine immer größer werdende Domäne anwendbar zu sein, so besteht die Gefahr, dass aus der DSL eine komplexe GPL wird. Dies kann vermieden werden, indem man mehrere DSLs verwendet anstatt zu versuchen, alle Probleme mit einer einzigen DSL zu lösen.

Bei HPC Systemen ist in letzter Zeit eine zunehmende Heterogenität zu beobachten. Um die Performanz eines Programms für ein bestimmtes System zu optimieren, sind jedoch Anpassungen des Codes entsprechend der Hardware des Systems vorzunehmen. Dadurch ergibt sich das Problem, dass ein Programm, das auf einem System eine gute Performanz erreicht, oft nicht auf andere HPC Systeme ohne weiteres portierbar ist. Das Portieren eines Programms auf ein anderes System bei gleichzeitiger Beihaltung der Performanz ist in der Regel mit viel manuellem Aufwand verbunden. Durch den Einsatz von DSLs kann die Struktur von Programmen auf einer hohen Abstraktionsebene spezifiziert werden, ohne sich dabei auf eine bestimmte Hardwarearchitektur festzulegen. Durch Anwendung von automatischer Codeerzeugung und Autotuning können dann aus dem DSL Programm mehrere Codevarianten generiert werden, die für verschiedene Systeme optimiert

sind. Somit ist das Programm leicht portierbar und gleichzeitig auf die Hardware der Systeme hin optimiert. Für den Bereich des wissenschaftlichen Rechnens wurden mit Hinblick auf diese Problematik bereits in mehreren Projekten DSLs eingeführt.

Im Rahmen dieser Arbeit wurden vier DSLs entwickelt, die die Implementierung von numerischen Lösungsverfahren für gewöhnliche Differenzialgleichungssysteme erleichtern sollen:

- *ODEMethod:*
  Beschreibung eines Lösungsverfahrens als Datenflussgraph. Basisoperationen des Verfahrens werden als Knoten dargestellt und Informationsfluss als gerichtete Kanten.

- *ODECoeffs:*
  Deklarierung von Verfahrenskoeffizienten, wie die Stufenzahl oder ein Butcher-Tableau.

- *ODETransformation:*
  Anwendung von *Enabling Transformations* auf den Datenflussgraphen als vorbereitenden Schritt zur Kernelfusion.

- *ODEKernelMapping:*
  Fusion einzelner Basisoperation zu Kernels zur Optimierung der Lokalität.

2 Related Work

Im Bereich des wissenschaftlichen Rechnens haben sich bereits mehrere Projekte mit der Entwicklung von domänenspezifischen Sprachen beschäftigt. Das mit DSLs spezifizierte Programm soll durch Anwendung von automatischer Codeerzeugung und Autotuning auf möglichst viele Systeme portierbar sein und gleichzeitig auf dem jeweiligen System eine möglichst hohe Performanz erreichen. Das Delite Compiler Framework [19] bietet dazu mehrere DSLs für die Bereiche Machine Learning, Graphanalyse und wissenschaftliches Rechnen an. Lift [13] kombiniert eine high-level Sprache mit einem System von sogenannten *rewrite rules*, mit denen Algorithmus- und Hardwarespezifische Optimierungen angewendet werden. Der Pochoir Stencil Compiler [7] bietet eine DSL speziell für Stencil Berechnungen auf Multicore Prozessoren. Das Projekt ExaStencils [1] behandelt ebenfalls die Domäne Stencils und bietet mit ExaSlang eine DSL auf vier Abstraktionsebenen.

Auch für den Bereich der gewöhnlichen Differentialgleichungen wurden schon einige DSLs entwickelt. Die domänenspezifische Sprache Melodee[10] wird verwendet, um ODE Systeme zu beschreiben. CellML[2] ist primär für die Beschreibung mathematischer Modelle im Bereich der Zellbiologie entwickelt, kann jedoch auch auf anderen Gebiete angewendet werden. Die mit CellML spezifizierten Programme können mit Flint[3] simuliert werden, unter anderem können damit auch ODEs numerisch gelöst werden. Flint unterstützt dazu Euler-Verfahren und das klassische Runge-Kutta-Verfahren. Die Implementierung eigener ODE Lösungsverfahren ist mit Flint nicht möglich. OpenModelica[6] ist eine Umgebung zur Modellierung, Simulation und Analyse von komplexen dynamischen Systemen. Mittels der Modellierungssprache Modelica werden Komponenten deklariert, die primär aus Gleichungen bestehen. OpenModelica stellt ähnlich wie Flint mehrere Lösungsverfahren für ODEs bereit, es können jedoch keine eigenen Löser implementiert werden. Die Lösungsverfahren von Flint und OpenModelica nutzen zudem keine Techniken zur Lokalitätsoptimierung.

In dieser Arbeit werden die Lösungsverfahren von ODEs als Datenflussgraphen dargestellt. Die Verwendung gerichteter Graphen zur Illustrierung von Informationsflüssen oder Abhängigkeiten innerhalb eines ODE Verfahrens ist bereits aus anderen Arbeiten, wie denen von Iserles[15] und Petcu[17], bekannt. Der Datenflussgraph, wie er in dieser Arbeit verwendet wird, ist stark an die Arbeit von Korch und Werner[16] angelehnt. Als Knoten verwenden Korch und Werner lediglich RHS Funktionsauswertungen, Map-Operationen und Reduktionen, während in dieser Arbeit zusätzlich Spezialknoten eingeführt werden, die beispielsweise zur Schrittweitenkontrolle genutzt werden.

Die Technik der Kernelfusion zur Optimierung von Lokalität auf GPUs wurde bereits auf mehreren Gebieten angewendet. Wang et al.[23] haben damit eine Technik zur Senkung des Stromverbrauchs von GPUs entwickelt. Wahib und Maruyama stellen die Anwendung von Kernelfusionen auf Stencil Berechnungen vor[21] und automatisieren diese[22]. Korch und Werner[16] wenden in ihrer Arbeit die Kernelfusionstechnik auf ODE Lösungsverfahren an. Dabei führen sie auf den Datenflussgraph, der ein ODE Verfahren beschreibt, Transformationen aus, um eine noch effizientere Kernelfusion zu ermöglichen. Dazu wurde eine C++-API verwendet, welche in Kapitel 7 mit den im Rahmen dieser Arbeit entwickelten DSLs verglichen wird.

3 ODE Verfahren

3.1 Explizite Runge-Kutta-Verfahren

Runge-Kutta-Verfahren[14] gehören zu den Einschrittverfahren. Für die Berechnungen eines Zeitschritts werden daher nur Ergebnisse des letzten Zeitschritts verwendet. Ein Schritt k zum Zeitpunkt t_k besteht aus s Stufen, jede Stufe i wiederum aus einer Auswertung der RHS Funktion an dem Punkt $(t_k+h_kc_i, Y_i)$ mit Ergebnis F_i. Y_i resultiert dabei aus der Linearkombination aus y_k und den Funktionsergebnissen F_1 bis F_{i-1}. Die Berechnungen der Stufe i lassen sich wie folgt beschreiben:

$$Y_i = y_k + h_k \sum_{j=1}^{i-1} a_{ij} F_j$$

$$F_i = f(t_k + h_k c_i, Y_i)$$

Nach der Berechnung aller Stufen eines Zeitschritts kann die neue Approximation y_{k+1} aus der Linearkombination y_k und den Ergebnissen der RHS Funktionen F_1 bis F_s gebildet werden:

$$y_{k+1} = y_k + h_k \sum_{j=1}^{s} b_j F_j$$

Die Schrittweite h_k kann zwischen den Zeitschritten variieren. Zur Steuerung der Schrittweite wird in einigen Runge-Kutta-Verfahren zusätzlich eine eingebettete Approximation $\widehat{y}_{k+1}$ berechnet:

$$\widehat{y}_{k+1} = y_k + h_k \sum_{j=1}^{s} \widehat{b}_j F_j$$

Wird ein eingebettetes Verfahren verwendet, so lässt sich aus y_{k+1} und $\widehat{y}_{k+1}$ mit einem Vektor S von Skalierungsfaktoren der Fehlervektor E bestimmen. Anschließend kann aus E der lokale Fehler ϵ ermittelt werden:

$$E_j = \frac{y_{k+1,j} - \widehat{y}_{k+1,j}}{S_j}, j = 1, ..., n$$

$$\epsilon = ||E||_\infty$$

Ist dieser Fehler ϵ größer als ein vom Benutzer vorgegebener Grenzwert, so wird der gesamte Zeitschritt verworfen und mit einer kleineren Schrittweite neu berechnet. Ansonsten wird mit einer größeren Schrittweite h_{k+1} der nächste Zeitschritt berechnet.

Die Koeffizienten Matrix $\mathrm{A} = (a_{ij} \in \mathbb{R}^{s \times s})$, sowie die Vektoren $b \in \mathbb{R}^s$, $\widehat{b} \in \mathbb{R}^s$ und $c \in \mathbb{R}^s$ werden oft in Form eines Butcher-Tableaus dargestellt:

$$\begin{array}{c|c} \mathrm{c} & \mathrm{A} \\ \hline & \begin{matrix} b \\ \widehat{b} \end{matrix} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & ... & a_{1s} \\ c_2 & a_{21} & a_{22} & ... & a_{2s} \\ ... & ... & ... & ... & ... \\ c_s & a_{s1} & a_{s2} & ... & a_{ss} \\ \hline & b_1 & b_2 & ... & b_s \\ & \widehat{b}_1 & \widehat{b}_2 & ... & \widehat{b}_s \end{array}$$

Abbildung 1 zeigt das Butcher-Tableau für das 4-stufige Runge-Kutta Verfahren, kurz RK4, das auch als klassisches Runge-Kutta Verfahren bezeichnet wird. Als Repräsentant der eingebetteten Verfahren ist ebenfalls ein Runge-Kutta-Fehlberg Verfahren mit 4 Stufen angegeben.

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

(a) Das **klassische Runge-Kutta Verfahren**.

$$\begin{array}{c|cccc} 0 & & & & \\ 1/4 & 1/4 & & & \\ 27/40 & -189/800 & 729/800 & & \\ 1 & 214/891 & 1/33 & 650/891 & \\ \hline & 214/891 & 1/33 & 650/891 & 0 \\ & 533/2106 & 0 & 800/1053 & -1/78 \end{array}$$

(b) Ein **Runge-Kutta-Fehlberg Verfahren** mit 4 Stufen.

Abbildung 1: Beispiele für Butcher-Tableaus.

3.2 Peer-Verfahren

Peer-Verfahren[24] lassen sich weder als reine Einschrittverfahren noch als reine Mehrschrittverfahren klassifizieren, da für die Berechnung eines Zeitschritts nicht nur die Ergebnisse der Stufen des aktuellen Zeitschritts verwendet werden, sondern

auch Ergebnisse der Stufen des vorherigen Zeitschritts. Sie gehören zu der Klasse der Generalisierten Linearen Modelle. Jeder Zeitschritt k besteht aus s Stufen, mit typischerweise $2 \leq s \leq 8$. Jede Stufe i berechnet eine Lösung $y_{k+1,i}$ zum Zeitpunkt $t_{k+1,i} = t_{k,i} + h_k$. Die Berechnung der neuen Stufen benötigt die Lösungen der Stufen des vorherigen Zeitschritts $y_{k,i}$, die Lösungen der RHS Funktionen $F_{k,i}$ und die variable Schrittweite h_k. $y_{k+1,i}$ wird aus einer Linearkombination aus den Vektoren $y_{k,j}$ und $F_{k,j}$ aller Stufen $j = 1, ..., s$ gebildet. $y_{k+1,i}$ und darauf folgend $F_{k+1,i}$ lassen sich also mit den Verfahrenskoeffizienten $A = (a_{ij}) \in \mathbb{R}^{s \times s}$ und $B = (b_{ij}) \in \mathbb{R}^{s \times s}$ wie folgt berechnen:

$$y_{k+1,i} = \sum_{j=1}^{s} (b_{ij} y_{k,j} + h_k a_{ij} F_{k,j})$$

$$F_{k+1,i} = f(t_{k+1,i}, y_{k+1,i})$$

Zur Bestimmung der variablen Schrittweite wird ähnlich zu den eingebetteten Runge-Kutta Verfahren anschließend ein Fehler Vektor E berechnet. Im Gegensatz zu den Runge-Kutta-Verfahren wird hier keine eingebettete Lösung benutzt. Der Fehlervektor E wird wie folgt berechnet:

$$E_j = \frac{|\sum_{l=1}^{s} e_l F_{k+1,l,j}|}{S_j}$$

$$S_j = atoli + rtoli|y_{k,1,j}|$$

e sind dabei verfahrensspezifische Koeffizienten und *atoli* sowie *rtoli* benutzerdefinierte Toleranzwerte. Analog zu den eingebetteten Runge-Kutta-Verfahren wird schließlich der lokale Fehler ϵ wie folgt gebildet:

$$\epsilon = ||E||_\infty$$

Ist ϵ größer als ein vom Benutzer vorgegebener Grenzwert, so wird der gesamte Zeitschritt verworfen und mit einer kleineren Schrittweite neu berechnet. Ansonsten wird der nächste Zeitschritt mit einer größeren Schrittweite h_{k+1} berechnet.

3.3 Adams-Bashforth-Verfahren

Explizite Adams-Bashforth-Verfahren [14] nutzen Ergebnisse mehrerer früherer Zeitschritte und sind somit ein bekannter Vertreter der Mehrschrittverfahren. Im Gegensatz zu Einschrittverfahren, in denen die Lösung nur aus der Differentialglei-

chung und dem Initialwert gebildet wird, ist bei Mehrschrittverfahren eine Startprozedur notwendig um die benötigten Eingabewerte für die ersten Zeitschritte zu ermitteln. Es gibt mehrere Möglichkeiten diese fehlenden initialen Werte zu finden. Zum Beispiel ist es möglich als Startprozedur ein Einschrittverfahren wie das Runge-Kutta-Verfahren zu verwenden.

In einem Adams-Bashforth-Verfahren mit s Schritten wird eine Lösung eines Zeitschritts y_{k+1} aus einer Linearkombination aus y_k und den Ergebnissen der RHS Funktionen der s letzten Zeitschritte verwendet:

$$y_{k+1} = y_k + h \sum_{j=0}^{s} b_j f(t_{k-j}, y_{k-j})$$

h ist dabei die Schrittweite. Der Vektor $b \in \mathbb{R}^{(s+1)}$ lässt sich abhängig von s wie folgt berechnen:

$$b_j = \frac{(-1)^j}{j!(s-j)!} \int_0^1 \prod_{i=0, i \neq j}^{s} (u+i) du, j = 0, ..., s$$

Abbildung 2 zeigt die Formeln für die Berechnung eines Zeitschritts mit $s = 1, 2, 3, 4$.

$$\begin{aligned}
s = 1: &\quad y_{k+1} = y_k + h f_k \\
s = 2: &\quad y_{k+1} = y_k + h(3/2 f_k - 1/2 f_{k-1}) \\
s = 3: &\quad y_{k+1} = y_k + h(23/12 f_k - 16/12 f_{k-1} + 5/12 f_{k-2}) \\
s = 4: &\quad y_{k+1} = y_k + h(55/24 f_k - 59/24 f_{k-1} + 37/24 f_{k-2} - 9/24 f_{k-3})
\end{aligned}$$

Abbildung 2: Adams-Bashforth mit verschiedener Schrittzahl[14].

3.4 Implizite Adams-Moulton-Verfahren

Adams-Moulton-Verfahren[12] sind den Adams-Bashforth-Verfahren sehr ähnlich. Der Unterschied besteht lediglich darin, dass für die Berechnung des Ergebnisses y_{k+1} eines Zeitschrittes der zu berechnende Wert selbst benutzt wird, y_{k+1} steht also auf beiden Seiten der Gleichung. Das Adams-Moulton-Verfahren zählt somit zu den impliziten Verfahren. y_{k+1} wird wie folgt berechnet:

$$y_{k+1} = y_k + h \sum_{j=-1}^{s-1} b_j f(t_{k-j}, y_{k-j}), 0 \leq s \leq k$$

Auch die Berechnung der Verfahrenskoeffizienten b_j ähnelt stark der Berechnung der Koeffizienten des Adams-Bashforth-Verfahrens:

$$b_j = \frac{(-1)^{j+1}}{(j+1)!(s-j-1)!} \int_0^1 \prod_{i=-1, i \neq j}^{s-1} (u+i) du, j = -1, 0, ..., s-1$$

4 Das Xtext Framework

Xtext ist ein Eclipse Framework zur Entwicklung von Programmiersprachen und DSLs[9]. Das Framework bietet Lösungen für viele Aspekte der Erstellung einer neuen Sprache und ermöglicht somit eine schnelle Entwicklung von DSLs. Insbesondere übernimmt Xtext selbstständig die lexikalische und syntaktische Analyse. Bei der lexikalischen Analyse wird ein Zeichenstrom aus dem Quellprogramm eingelesen und in einzelne Token zerlegt. Aus dem resultierendem Token Stream wird mit Hilfe einer Symboltabelle in der syntaktischen Analyse ein abstrakter Syntaxbaum (AST) erzeugt. Ausgehend von diesem können anschließend eine semantische Analyse und eine Codegenerierung folgen. Mehr dazu in Kapitel 4.2 und 4.3.

Zusätzlich bietet Xtext eine komplette Integration von DSLs in die Eclipse IDE mit folgenden Features:

- *Syntax Highlighting:*
  Verschiedene Elemente einer Sprache, wie zum Beispiel Kommentare und Keywords, werden farbig und durch visuelle Formatierung hervorgehoben. Das Highlighting gibt dem Programmierer direkt Feedback über den geschriebenen Code. So können beispielsweise sofort Rechtschreibfehler erkannt werden, falls ein Keyword falsch geschrieben wurde und somit nicht optisch hervorgehoben wird. Zudem hilft die visuelle Aufbereitung dem Programmierer die Struktur eines Programms schneller zu erfassen.

- *Parsen im Hintergrund:*
  Während des Schreibens eines Programms mit der DSL wird zeitgleich im Hintergrund der Code geparst und ein AST aufgebaut. Lexikalische, syntaktische und semantische Fehler werden somit sofort während des Programmierens erkannt und angezeigt. Ohne dieses Feature müsste der Programmierer

in regelmäßigen Abständen das Programm abspeichern, kompilieren und gegebenenfalls Fehler aus dem Output des Compilers im Code identifizieren und lokalisieren.

- *Fehlermarkierungen:*
  Wird im Code ein Fehler durch den Parser im Hintergrund entdeckt, so wird dieser Fehler nicht auf einer Konsole ausgegeben. Stattdessen werden die Programmelemente, die den Fehler verursachen, im Code direkt rot hervorgehoben. Zusätzlich können aussagekräftige Fehlermeldungen deklariert werden, die angezeigt werden, wenn der Cursor über die fehlerhafte Stelle im Code bewegt wird. Der Programmierer kann somit fehlerhafte Codestellen schnell lokalisieren und beheben.

- *Quickfixes:*
  Wird ein Fehler im Code erkannt, so kann dieser nicht nur optisch hervorgehoben werden. In einigen Fällen ist es möglich dem Programmierer direkt mögliche Lösungen zur Behebung des Problems anzubieten, die nach einer Bestätigung des Benutzers automatisch umgesetzt werden. So kann beispielsweise für den Fall, dass ein Name für zwei Variablen doppelt verwendet wurde, angeboten werden, diesen Namenskonflikt durch die Umbenennung einer Variable aufzulösen.

- *Content Assist:*
  Während des Programmierens werden dem Benutzer Vorschläge gemacht, wie ein angefangenes Statement oder ein Ausdruck vervollständigt werden kann. Für ein Objekt einer Klasse kann so zum Beispiel eine Auswahl aller wählbaren Member oder Methoden angezeigt werden. Der Benutzer kann somit eine DSL leichter verwenden, auch wenn dieser mit einigen Sprachkonstrukten kaum vertraut ist.

- *Automatischer Build:*
  Sobald eine Datei mit Programmcode abgespeichert wird, erfolgt automatisch eine Kompilierung des Programms. Der Benutzer muss also nicht manuell per Konsolenbefehl den Build starten. Es ist möglich diesen automatischen Aufruf beim Abspeichern um eigene Aktionen zu erweitern.

4.1 Die Xtext Grammatiksprache

Die Grammatiksprache des Xtext Frameworks[4] dient dazu, der zu erstellenden DSL eine konkrete Syntax zu geben. Diese beschreibt, wie der Anwender ein Programm entwickelt. Generell kann eine konkrete Syntax textuell, graphisch, tabellarisch oder jede Kombination daraus sein. Das Xtext Framework benutzt einen textuellen Ansatz, der Programmierer gibt also das Programm in Textform an. Aus der konkreten Syntax wird eine abstrakte Syntax erzeugt. Diese ist eine Datenstruktur, in der die Kerninformationen eines Programms abgespeichert werden. Details über die Notation, wie das Layout oder Kommentare werden nicht in der abstrakten Syntax gespeichert.

Xtext nutzt ANTLR (ANother Tool for Language Recognition) als Parser Generator. ANTLR generiert aus einer Grammatik einen Parser, der aus einer Eingabe in textueller Form einen abstrakten Syntaxbaum generiert. Es werden LL(k) und LL(*) Grammatiken unterstützt, typische Grammatiken benötigen jedoch lediglich einen begrenzten Lookahead. Zu beachten ist dabei, dass ein LL(k) Parser gewissen Beschränkungen unterliegt. So unterstützt ANTLR beispielsweise keine Grammatiken mit Linksrekursion. Diese können jedoch durch Left-Factoring beseitigt werden. Folgendes Beispiel zeigt eine Produktionsregel mit Linksrekursion und die Lösung durch Left-Factoring:

```
Exp ::= ID                                  Exp ::= Exp' (". "ID)*
      | STRING                  →
      | Exp ("."ID)+                        Exp' ::= ID
                                                   | STRING
```

Grammatiken können zudem mehrdeutig sein. Dies ist der Fall, wenn eine Sequenz aus Terminalsymbolen aus mehreren Parsbäumen erzeugt werden kann. Mehrdeutigkeit kann durch eine Aufteilung der Produktionsregeln in verschiedene Ebenen behoben werden. Folgendes Beispiel erlaubt, dass der Ausdruck $2*3+4$ je nach Reihenfolge der Rechenschritte entweder zu dem Ergebnis 10 oder 14 führen könnte. Nach der Umformulierung werden Multiplikationen immer vor Additionen ausgeführt:

```
                                            Exp ::= Exp "+ "Mult
Exp ::= NUM                                       | Mult
      | Exp "+ "Exp           →
      | Exp "* "Exp                         Mult ::= Mult "* "NUM
                                                   | NUM
```

Xtext Grammatiken werden in einer EBNF-ähnlichen Notation beschrieben. Durch Produktionsregeln wird gleichzeitig die konkrete Syntax und das Mapping zu der abstrakten Syntax spezifiziert. Folgendes Beispiel demonstriert die Verwendung von Regeln in Xtext:

```
1 Declaration:
2     name=ID ':' type=Type ('=' expression=Expression)?
3 ;
```

Quelltext 1: Die Produktionsregel für eine Variablendeklaration.

Die Regel *Declaration* erhält zunächst eine *ID* als Namen für die zu deklarierende Variable. *ID* ist dabei eine von mehreren eingebauten Terminalregeln. Anschließend folgt stets ein Doppelpunkt. Nach diesem wird der Typ der Variable spezifiziert. *Type* ist dabei eine weitere Produktionsregel. Schließlich kann die Variable optional mit einem Ausdruck initialisiert werden. Angenommen INT ist ein gültiges Terminal für *Type* und Zahlen sind gültige Terminale für *Expression*, so lässt sich mit der Eingabe "i : INT = 3"eine Variable i mit dem Wert 3 initialisieren.

Intern wird aus der Xtext Grammatik ein Ecore Modell[18] erzeugt. Das Ecore Meta-Model lässt sich vereinfacht durch die Klassen in Abbildung 3 zusammenfassen. Jede *EClass* besitzt einen Namen *name*, Attribute und Referenzen. Jedes *EAttribute* gehört zu genau einer Klasse, hat einen Namen und einen Datentyp. Jede *EReference* gehört ebenfalls zu genau einer Klasse, hat einen Namen und hat als Referenztyp wiederum eine *EClass*. Die Referenz kann eine Containment Beziehung sein.

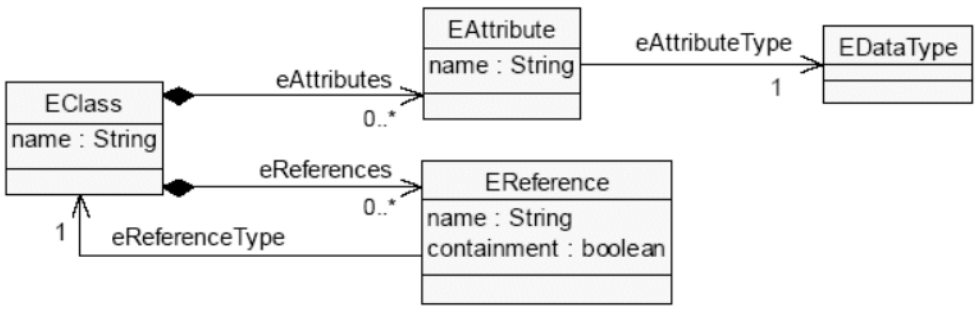


Abbildung 3: Das Ecore Meta-Model in stark vereinfachter Form.

Xtext generiert aus der Regel *Declaration* eine Klasse mit dem Attribut *name*. Zudem werden zwei Containment Referenzen *type* und *expression* zu den ebenfalls erzeugten Klassen *Type* und *Expression* gebildet (siehe Abbildung 4).

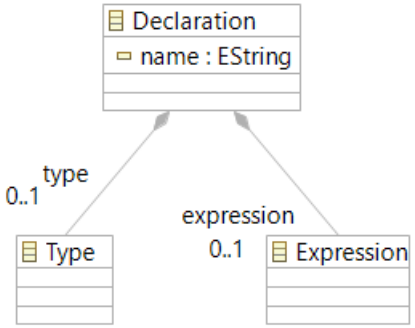


Abbildung 4: Die Regel *Declaration* erzeugt eine Klasse mit Attribut *name* und Referenzen zu *Type* und *Expression*.

Tatsächlich generiert Xtext beim Kompilieren der Grammatik ein Interface *Declaration* und eine Klasse *DeclarationImpl*. *Declaration* enthält get- und set-Methoden für die Attribute und Referenzen. Die Klasse implementiert das Interface und besitzt zusätzlich EMF-spezifische Methoden.

```
1 public interface Declaration extends EObject {
2     String getName();
3     void setName(String value);
4     Type getType();
5     void setType(Type value);
6     Expression getExpression();
7     void setExpression(Expression value);
8 }
```

Quelltext 2: Das automatisch generierte Interface Declaration.

Eine Besonderheit der Xtext Sprache gegenüber klassischen Grammatiksprachen ist die spezielle Behandlung von Referenzen auf benannte Elemente im Code. Folgendes Beispiel zeigt die Verwendung von bereits deklarierten Variablen in Ausdrücken:

```
TerminalExpression returns Expression:
    {StringConstant} value=STRING |
    {IntConstant} value=INT |
    {BoolConstant} value=('true' | 'false') |
    {Null} 'null' |
    {DeclarationRef} symbol=[Declaration] (arrays+=Array)* |
    '(' Expression ')'
;
```

Quelltext 3: Die Produktionsregel für eine *TerminalExpression*.

Die Option *DeclarationRef* wird verwendet um in einem Ausdruck eine bereits deklarierte Variable zu verwenden. In einer klassischen Grammatiksprache würde man diese Referenz durch eine Regel der Form "symbol=ID"lösen und in einem späteren Schritt versuchen die in einem Code angegebene ID auf einen Variablennamen zurückzuführen. Durch die eckigen Klammern kann in Xtext mit einer Regel der Form "symbol=[Declaration]"auf eine Variable referenziert werden. Xtext versucht dann automatisch während der Programmierung diese Referenz aufzulösen. Schlägt die Auflösung fehl, so wird die Referenz an entsprechender Stelle im Code mit einer Fehlermarkierung hervorgehoben.

Weiter lässt sich für jede Regel der Rückgabetyp angeben. Im obigen Beispiel ist eine *TerminalExpression* ein Spezialfall einer *Expression*. Wird eine *TerminalExpression* erkannt, so wird wegen dem Zusatz "returns Expression"ein Objekt der Klasse *Expression* erzeugt. Für die einzelnen Optionen einer *TerminalExpression* können durch die Angabe von Namen in geschweiften Klammern Unterklassen, die von der Klasse *Expression* erben, erzeugt werden. Wird also beispielsweise ein Integer erkannt, so wird ein Objekt der Klasse *IntConstant* erzeugt und als Objekt der Klasse *Expression* zurückgegeben. Objekte der Klasse *Expression* können später bei Bedarf mit "instanceof"auf die Unterklasse untersucht werden.

4.2 Validierung

Nicht alle Programme, die konform zu einer Xtext Grammatik sind, sind auch automatisch gültig. So lassen sich viele Restriktionen nicht rein durch die Struktur der Sprache umsetzen. Um diese Restriktionen zu implementieren bedarf es eines zusätzlichen Schritts zwischen der Erzeugung des abstrakten Syntaxbaums und der Codegenerierung. Eclipse erzeugt dazu beim Anlegen eines neuen Xtext Projektes automatisch eine Validator Klasse. Diese Klasse lässt sich durch eigene Methoden erweitern. Zu beachten ist hierbei, dass diese Klasse nicht in Java geschrieben ist,

sondern in der Java-ähnlichen Sprache Xtend[8]. Diese hat im Vergleich zu Java eine kompaktere Syntax und zusätzliche Features, wie Typinferenz und Lambda Ausdrücke. Besonders nützlich für die Codegeneration sind zudem Multi-line Template Expressions, mit denen sich Strings leicht erzeugen lassen. Um den Validator zu erweitern wird eine neue Methode mit dem Schlüsselwort "@Check"definiert. Folgende Methode zeigt die Verwendung von Validatoren:

```
@Check def void nodeHasName(NodeFunction f){
    for (a : f.arguments)
        if(a instanceof Label) return;

    error("Missing argument 'label'",
        null,
        MISSING_ARGUMENT)
}
```

Quelltext 4: Die Methode nodeHasName.

Diese Methode überprüft für jedes Objekt der Klasse *NodeFunction*, ob es im Code als Argument einen Namen zugewiesen bekommen hat. Dazu wird über alle Argumente des Objektes iteriert. Ist eines der Argumente vom Typ *Label*, so wird die Methode verlassen. Ansonsten wird mit Hilfe der Funktion *error* an der entsprechenden Stelle im Programmcode eine Fehlermarkierung mit entsprechender Meldung platziert:

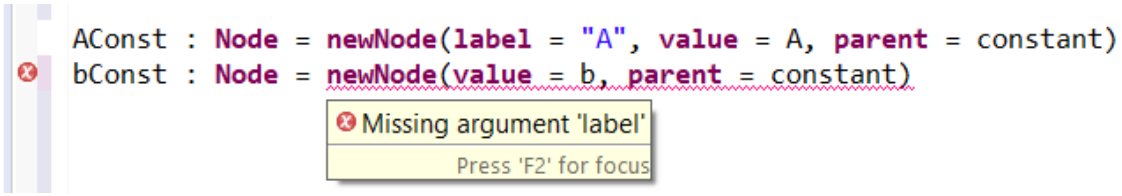


Abbildung 5: Wird dem Aufruf von *newNode* kein *label* übergeben, so wird die Stelle im Programmcode als Fehler markiert.

Wie häufig der Programmcode auf diese Restriktionen überprüft wird, kann im Validator für jede Methode spezifiziert werden. Durch eine Angabe von "@Check (CheckType.FAST) def void ..."wird die Restriktion nach jeder Änderung im Code geprüft. Dies ist die Standardeinstellung, falls im Methodenkopf der *CheckType*

nicht spezifiziert ist. In den meisten Fällen ist diese Einstellung sinnvoll, da der Anwender ein direktes Feedback beim Programmieren erhält. Ist die Überprüfung einer Restriktion rechenaufwendiger, da beispielsweise über alle Elemente des Modells iteriert werden muss, so kann die Standardeinstellung *FAST* jedoch zu merkbaren Reaktionsverzögerungen der IDE führen. In diesen Fällen empfiehlt sich die Einstellung *NORMAL*, wodurch die Überprüfung nur beim Abspeichern durchgeführt wird, oder *EXPENSIVE*, wodurch die Überprüfung explizit durch den Anwender aufgerufen werden muss.

Die im Validator spezifizierten Restriktionen lassen sich in die zwei Hauptkategorien Wohlgeformtheit und Typsystem unterteilen. Die oben beschriebene Regel *nodeHasName* gehört zu der Wohlgeformtheit des Programms. Das Typsystem befasst sich damit, dass alle Ausdrücke die Parameter des richtigen Typs erhalten und die Ergebnisse von Unterausdrücken wiederum kompatibel zu den Oberausdrücken sind. So sollen beispielsweise Ausdrücke der Form ""Test"== 3 "oder "(2 == 3) + 1 "verboten sein, da ein String nicht mit einem Integer vergleichbar ist und das Ergebnis eines Vergleichs (also ein Boolean) nicht mit einem Wert addiert werden kann. Dazu wird der Validator um folgende Restriktion erweitert:

```
@Check def void checkConformance(Expression exp) {
    val actualType = exp.typeFor
    val expectedType = exp.expectedType
    if (expectedType === null || actualType === null)
        return; // nothing to check
    if (!actualType.isConformant(expectedType)) {
        error("Incompatible types. Expected '" + expectedType.name
            + "' but was '" + actualType.name + "'",
            null, INCOMPATIBLE_TYPES)
    }
}
```

Quelltext 5: Die Methode checkConformance.

Für jede *Expression* wird also mit den Hilfsfunktionen *typeFor* und *expectedType* der tatsächliche und der erwartete resultierende Typ des Ausdrucks bestimmt. Sind diese beiden Typen nicht kompatibel so wird eine Fehlermeldung erzeugt. Der tatsächliche Typ wird durch "instanceof "Abfragen (als Switch Block) ermittelt:

```
def TYPE typeFor(Expression e) {
    switch (e) {
        DeclarationRef:
```

```
            e.symbol.type.SType.typeFor
        Null:
            TYPE.NULL_TYPE
        StringConstant:
            TYPE.STRING_TYPE
        IntConstant:
            TYPE.INT_TYPE
        BoolConstant:
            TYPE.BOOL_TYPE
        Or:
            TYPE.BOOL_TYPE
        Minus:
            TYPE.INT_TYPE
        // ...
    }
}
```

Quelltext 6: Die Hilfsmethode typeFor (Ausschnitt).

Offensichtlich muss für *Null*, *StringConstant*, *IntConstant* und *BoolConstant* der entsprechende Grundtyp zurück geliefert werden. Das Ergebnis eines Gleichheitsvergleichs und einer Disjunktion ist immer vom Typ Boolean, das Ergebnis einer Subtraktion immer vom Typ Integer (Mit der Annahme, dass nur der Datentyp Integer für Zahlen existiert). Ist der Ausdruck eine Referenz auf eine Variable, so wird der Typ der Variablen zurückgegeben.

expectedType ähnelt stark der Methode *typeFor* mit dem Unterschied, dass hier im Switch Block das Element untersucht wird, in dem die *Expression* enthalten ist:

```
def TYPE expectedType(Expression e) {
    val c = e.eContainer
    switch (c) {
        Declaration:
            c.type.SType.typeFor
        If:
            TYPE.BOOL_TYPE
        Or:
            TYPE.BOOL_TYPE
        Minus:
            TYPE.INT_TYPE
        Label:
```

```
13              TYPE.STRING_TYPE
14          // ...
15      }
16  }
```

Quelltext 7: Die Hilfsmethode expectedType (Ausschnitt).

If-Abfragen, sowie While- und For-Schleifen erwarten als Ergebnis des Bedingungsausdrucks immer den Typ Boolean. Wird eine Variable initialisiert, so sollte der Ausdruck immer zum Typ der Variablen passen. Ein Label erwartet immer einen String und die Minus-Operation einen Integer.

Wurden also die beiden Werte *actualType* und *expectedType* ermittelt, so kann mit der Methode *isConformant* überprüft werden, ob die beiden Typen kompatibel sind. Zwei Typen sind dann kompatibel, wenn sie identisch oder sehr ähnlich sind, wie zum Beispiel Integer und Double.

4.3 Generator

Nachdem ein mit der DSL geschriebenes Programm zu einem Modell geparsed und erfolgreich validiert wurde, kann dieses weiterverarbeitet werden. Das Modell kann direkt durch einen Interpreter ausgeführt oder durch einen Generator in Code übersetzt werden. Die Ausgabesprache ist dabei beliebig, es kann also beispielsweise Java, C oder auch eine XML Datei generiert werden. Zur Generierung wird ein Modell-zu-Text Transformationswerkzeug benötigt, welches als Eingabe eine Menge an Objekten erhält und daraus eine Sequenz von Textzeilen erzeugt. Eclipse liefert bei der Erstellung eines neuen Xtext Projektes automatisch eine Generator Klasse mit. Wie der Validator ist auch diese in Xtend geschrieben und sieht wie folgt aus:

```
1  class MyDslGenerator extends AbstractGenerator {
2      override void doGenerate(Resource resource, IFileSystemAccess2
           fsa, IGeneratorContext context) { }
3  }
```

Quelltext 8: Der von Eclipse erstellte Generator.

Die Klasse enthält zunächst nur die Methode *doGenerate*. Diese wird immer automatisch aufgerufen, wenn das DSL Programm verändert und abgespeichert wird, nachdem das Programm erfolgreich geparsed und validiert wurde. Wie im Codeausschnitt zu sehen ist, hat der Methodenkörper keinen Inhalt, die automatisch mitgelieferte Datei hat also zunächst keine Funktion. Es liegt also am Entwickler

der DSL, diese Datei mit Methoden auszustatten, die aus dem Modell eine Ausgabe erzeugen. Um eine Ausgabedatei zu generieren kann *doGenerate* wie folgt erweitert werden:

```
override void doGenerate(Resource resource, IFileSystemAccess2
    fsa, IGeneratorContext context) {
    for(f : resource.allContents.toIterable.filter(typeof(File)))
        fsa.generateFile("Output.java", f.compile)
}
```

Quelltext 9: Die erweiterte Methode doGenerate.

Über den Parameter *resource* kann auf alle Elemente des erzeugten Modells zugegriffen werden. Mit der Annahme, dass ein Objekt der Klasse File immer das Wurzelobjekt des Modells ist und nur einmal existiert, wird die For-Schleife genau einmal durchlaufen. Mit *generateFile* wird dann die Datei "Output.java "erzeugt. Als zweiten Eingabeparameter erwartet diese Methode einen String, der in die Datei geschrieben wird. Dazu wird der Generator um eine Methode "compile "mit Parameter vom Typ File erweitert, welche einen String zurückgibt. Als einfaches Beispiel könnte diese Methode wie folgt aussehen:

```
def compile(File file){
    '''
    package Generated;

    public class Output {
        «FOR d : file.elements»
        «IF d instanceof Declaration»«d.compile»«ENDIF»
        «ENDFOR»
    }
    '''
}
```

Quelltext 10: Die Methode compile.

Der Methodenkörper besteht aus einer einzigen Multi-line Template Expression. Dieses enthält statische Sequenzen, wie die Angabe des Paketes, und einen dynamisch erzeugten Teil, in dem aus den Elementen von *file* wiederum durch einen Funktionsaufruf Text kompiliert wird. So lassen sich mit Xtend ausgehend von dem Modell auch komplexere Programmstrukturen relativ einfach generieren.

5 Umgesetzte DSLs

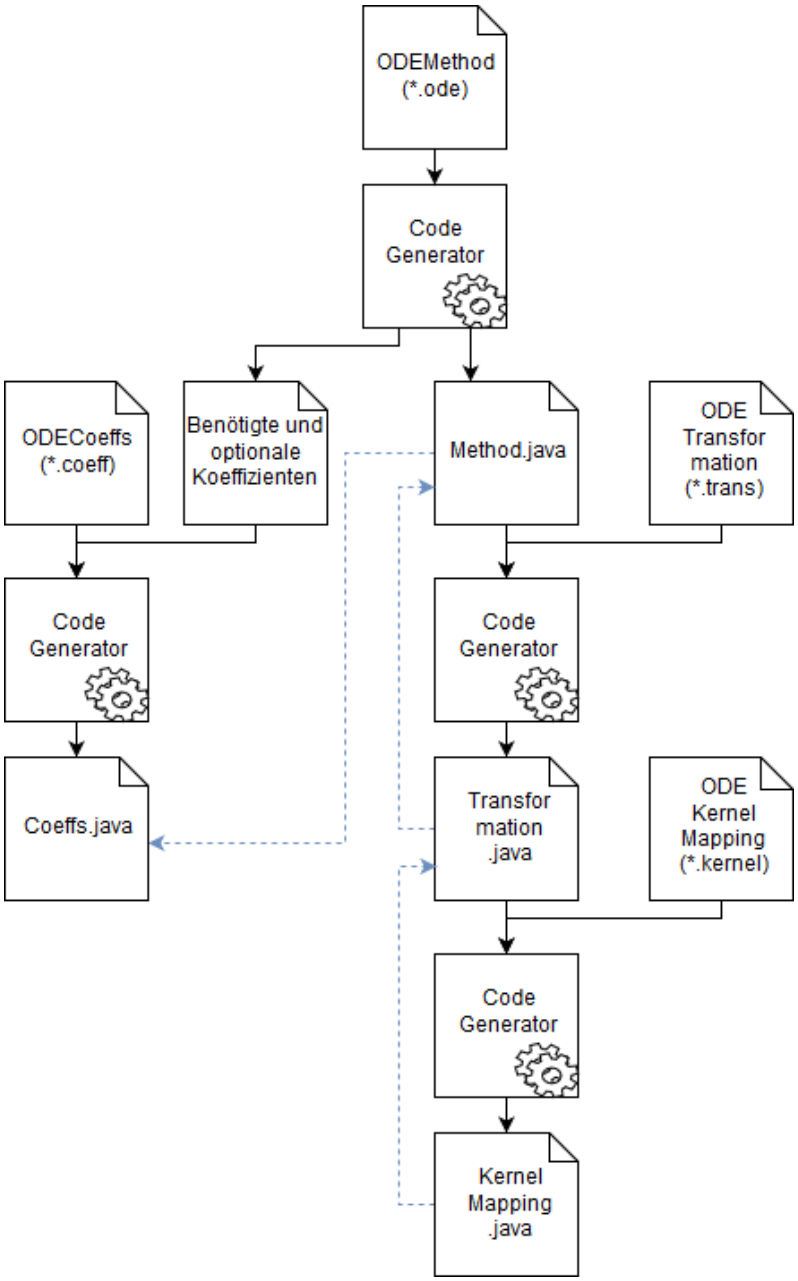


Abbildung 6: Die Pipeline zur Erzeugung eines ODE Solvers mit den DSLs. Gestrichelte Kanten zeigen die Abhängigkeiten zwischen den Java Klassen.

5.1 Beschreibung von Verfahren - Die ODEMethod DSL

Als ersten Schritt hin zu einer automatischen Codeerzeugung eines ODE Verfahrens gilt es die Struktur des zu implementierenden Verfahrens zu beschreiben. Die ODEMethod DSL mit Dateiendung ".ode "wird dazu verwendet die Bestandteile eines Zeitschritts zu deklarieren. Wie alle mathematischen Verfahren lassen sich ODE Verfahren als ein Datenflussgraph aus Gleichungen beschreiben. Daher ist es sinnvoll, dass auch die grundlegende Datenstruktur hinter der DSL ein Graph mit gerichteten Kanten ist. Die Knoten des Graphen sind dabei die grundlegenden Rechenschritte eines Zeitschritts. Folgende Operationen sind typischerweise in einem ODE Verfahren vorhanden[16]:

- *Linearkombinationen (LC):*
  Eine Linearkombination von Vektoren ist eine Summe von Vektoren, von denen jeder mit einem Skalar multipliziert wird. Eine Linearkombination lässt sich daher in die Skalierung der einzelnen Vektoren und der anschließenden Addition unterteilen.

- *RHS Funktionsauswertungen:*
  Über die RHS Funktion können keine generellen Annahmen getroffen werden, da die Gleichungen eines ODE Systems beliebig gekoppelt werden können. Bei einem Aufruf $f(t, y)$ kann der gesamte Argumentvektor y benötigt werden.

- *Reduktionen:*
  Bei einer Reduktion wird aus einem Vektor ein einzelner Skalar berechnet.

Hinzu kommen noch spezielle Knoten, die beispielsweise für ein eingebettetes Verfahren den Fehlervektor E berechnen. Knoten lassen sich auch zur besseren Übersicht hierarchisch anordnen.

Eine gerichtete Kante von einem Knoten A zu einem Knoten B beschreibt den Datenfluss zwischen beiden Operationen. Knoten A erzeugt ein Resultat, das B wiederum für Berechnungen benötigt. Häufig kommt es vor, dass ein Knoten die Ergebnisse eines Knotens aus dem vorherigen Zeitschritt benötigt. Diese Abhängigkeiten über Zeitschritte hinweg lassen sich durch Annotation der Zeitschrittdifferenz an den Kanten ausdrücken. Eine Abhängigkeit innerhalb eines Zeitschritts wird durch Zeitschrittdifferenz 0 annotiert, die Abhängigkeit von einem Ergebnis des letzten Zeitschritts durch 1.

Diese Elemente reichen aus um einen Zeitschritt eines ODE Verfahrens vollständig und korrekt abzubilden. Abbildung 8 zeigt das vierstufige eingebettete Runge-Kutta-Verfahren in Form eines Datenflussgraphen. Linearkombinationen sind gelb, Auswertungen der RHS Funktion rot und Reduktionen grün hervorgehoben. Die grundlegenden Operationen des Runge-Kutta-Verfahrens sind hier wiederzufinden: Die Berechnung der Werte Y_2 bis Y_4 ($Y_1 = y_k$ muss nicht berechnet werden), die Auswertungen der RHS Funktionen F_1 bis F_4, die Berechnung der neuen Approximation y_{k+1} sowie die eingebettete Lösung $\widehat{y}_{k+1}$, die Berechnung des Fehlervektors E und die anschließende Reduktion sind alle durch Knoten im Graphen realisiert. Dazu kommen noch Verfahrenskoeffizienten, die im Elternknoten *Constants* enthalten sind. Kanten innerhalb eines Zeitschrittes sind als durchgehende Linien dargestellt, Abhängigkeiten über Zeitschritt hinweg durch gestrichelte Linien mit annotierter Zeitschrittdifferenz. Da Runge-Kutta ein Einschrittverfahren ist, ist die maximale Differenz 1.

Abbildung 7 zeigt das implizite Adams-Moulton-Verfahren. Die Knoten y, F und ein Skalierungsknoten bilden darin einen Zyklus innerhalb eines Zeitschritts.

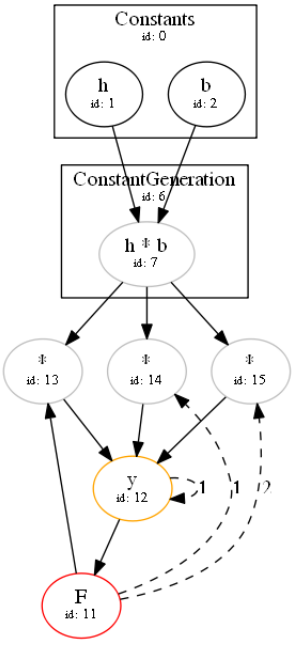


Abbildung 7: Der Datenflussgraph für das Adams-Moulton-Verfahren.

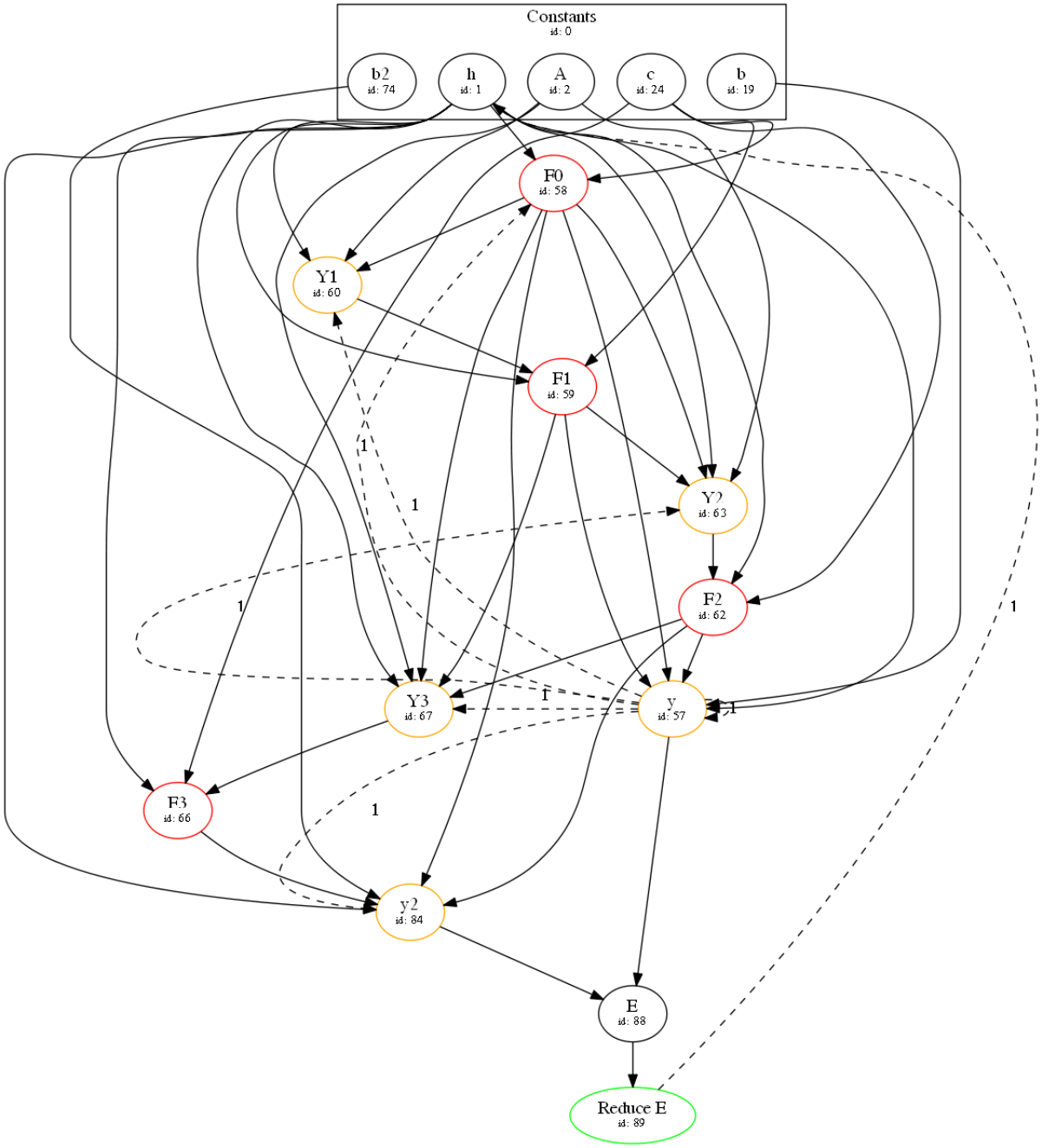


Abbildung 8: Der Datenflussgraph für das vierstufige eingebettete Runge-Kutta-Verfahren.

Die wichtigsten Bestandteile der ODEMethod DSL sind also die Möglichkeiten Knoten und Kanten zu definieren. Knoten lassen sich durch eine Variablendeklaration hinzufügen. Die für eine Variablendeklaration nötigen Produktionsregeln lauten wie folgt:

```
1 Declaration: name=ID ':' type=Type ('=' expression=Expression)? ;
2
3 Type: sType=SimpleType (arrays+=Array)* ;
4
5 SimpleType: 'ID' | 'String' | 'N' | 'R' | 'Bool' | 'Node' ;
```

Quelltext 11: Variablendeklarationsregeln der ODEMethod DSL.

Zur Initialisierung eines Knotens wird als Expression eine Option der Regel *NodeFuntction* gewählt:

```
1 NodeFunction:
2     {Node} 'newNode' '(' arguments+=Argument (','
          arguments+=Argument)* ')' |
3     {Accumulation} 'newAccumulation' '(' arguments+=Argument (','
          arguments+=Argument)* ')' |
4     {RHS} 'newRHS' '(' arguments+=Argument (','
          arguments+=Argument)* ')' |
5     {Reduction} 'newReduction' '(' arguments+=Argument (','
          arguments+=Argument)* ')' |
6     {Basic} 'newBasic' '(' arguments+=Argument (','
          arguments+=Argument)* ')'
7 ;
```

Quelltext 12: Produktionsregeln zur Initialisierung von Knoten.

Der Knoten kann also den Typen *Accumulation*, *RHS*, *Reduction*, *Basic* oder, falls der Knoten nicht näher spezifiziert werden soll, *Node* haben. Knoten vom Typ *Basic* beschreiben die grundlegenden Rechenoperationen Multiplikation und Addition. Linearkombinationen lassen sich nicht direkt als Typ angeben, sondern werden aus Knoten vom Typ *Accumulation* und *Basic* gebildet.

Bei der Initialisierung können folgende Argumente übergeben werden:

label	Der Name des Knotens
type	*(Nur Basic)* Die grundlegende Rechenart ('Addition' oder 'Multiplication')
value	*(Optional)* Der Wert des Knotens, falls dieser bereits bekannt ist (z.B. Eingabekoeffizienten)
time	*(Optional)* Die zum Zeitpunkt des Zeitschritts relative Zeit
result	*(Optional)* Der Ergebnistyp ('ErrorVector', 'StateVector' oder 'SlopeVector')
parent	*(Optional)* Ein Elternknoten als Referenz

Folgendes Beispiel zeigt die Deklaration von Knoten in der DSL:

```
initH : R = 1
constant : Node = newNode(label = "Constants")
h : Node = newNode(label = "h", value = initH, parent = constant)
```

Quelltext 13: Beispiel einer Knotendeklaration.

Zunächst wird ein Knoten *constant* mit Namen *Constants* deklariert. Der Knoten *h* erhält als Wert 1 und wird dem Elternknoten *constant* untergeordnet.

Kanten lassen sich durch die Verwendung der Regel *DEPENDENCYAssign* erzeugen:

```
ASSIGNStatement returns Statement:
    {Assign} source=[Declaration] (arrays+=Array)*
       (right=EQUALAssign | right=DEPENDENCYAssign)
;

DEPENDENCYAssign:
    '--' (time=Expression)? '->' destination=[Declaration]
       (arrays+=Array)*
;
```

Quelltext 14: Produktionsregeln zur Erzeugung von Kanten.

Mit dem Ausdruck "A $--$ 1 $->$ B"erhält man also eine Kante von A nach B mit Zeitschrittdifferenz 1.

Damit die DSL für alle Verfahren einer Verfahrensklasse gilt ist die Verfahrensbeschreibung allgemein zu halten. Beispielsweise können Runge-Kutta-Verfahren beschrieben werden, ohne eine konkrete Anzahl an Stufen s in der ODEMethod DSL anzugeben. Somit ist das in der DSL spezifizierte Programm auf alle Runge-Kutta-Verfahren anwendbar. Will man mit einer anderen Stufenzahl arbeiten muss

das Verfahren somit nicht neu spezifiziert oder umgeschrieben werden. Es muss lediglich der Variablenwert s geändert werden. Die Stufenzahl und andere Koeffizienten eines speziellen Verfahrens können in die ODECoeffs DSL ausgelagert werden. Die ODEMethod DSL greift auf die in der ODECoeffs DSL deklarierten Variablen zu um den Datenflussgraphen aufbauen zu können und benötigt daher eine Auflistung von benötigten und optionalen Variablen, die mit der ODECoeffs DSL zu deklarieren sind. Dazu wird in der ODEMethod DSL eine Input Struktur deklariert. Diese folgt folgenden Produktionsregeln:

```
Input:
    {Input} 'input' '{' required=Required (optionals+=Optional)*
        '}' ;

Required:
    {Required} 'required' '{' (variables+=Declaration)* '}' ;

Optional:
    {Optional} 'optional' sname=ID '{' (variables+=Declaration)*
        '}' ;
```

Quelltext 15: Produktionsregeln zur Erzeugung eines Input Blockes.

Eine Input Struktur enthält stets einen *Required* Block gefolgt von beliebig vielen optionalen Blöcken. Im *Required* Block werden alle Variablen angegeben, die zur Generierung des Verfahrens mittels der DSL notwendig sind. In optionalen Blöcken, die einen Namen erhalten, werden Variablen deklariert, die für zusätzliche optionale Features benötigt werden. Eine gültige Input Struktur für das Runge-Kutta-Verfahren sieht wie folgt aus:

```
input {
    required {
        s : N  # Stufenzahl
        p : N  # Ordnung
        A : R[s][s]
        b : R[s]
        c : R[s]
    }

    optional embedded {
        pe : N # Ordnung des eingebetteten Verfahrens
        b2 : R[s]
```

```
13      }
14
15      optional embeddedDiff{
16          b2Diff : R[s]
17      }
18  }
```

Quelltext 16: Ein Input Block für das Runge-Kutta-Verfahren.

Die DSL benötigt also die Stufenzahl s, die Ordnung des Verfahrens p, sowie die Verfahrenskoeffizienten A, b und c um einen Datenflussgraphen für das Runge-Kutta-Verfahren aufbauen zu können. Ein Runge-Kutta-Verfahren kann eingebettet sein, was durch den optionalen Block *embedded* umgesetzt wird. Will der Entwickler ein eingebettetes Verfahren mit Zeitschrittweitensteuerung implementieren, so muss er zusätzlich in der ODECoeffs DSL die Variablen für die Ordnung des eingebetteten Verfahrens *pe* und die zusätzlichen Verfahrenskoeffizienten $b2$ angeben.

Auf die in einem Input Block deklarierten Variablen kann zur Konstruktion des Datenflussgraphen problemlos zugegriffen werden, obwohl diese in der ODEMethod DSL nicht initialisiert wurden. Soll ein Verfahren um ein Feature, wie zum Beispiel eine Zeitschrittweitensteuerung, erweitert werden, müssen dem Graph gewisse Knoten hinzugefügt werden. Diese sollen jedoch nur präsent sein, wenn das Feature auch tatsächlich implementiert werden soll. Zur Abfrage dieser Kondition besitzt die DSL das Sprachkonstrukt *optionalInput*. Gefolgt von einem Namen eines optionalen Blockes gibt dieses *true* zurück, falls das Feature implementiert werden soll, und *false* falls nicht. Das Sprachkonstrukt kann in folgender Form verwendet werden, um optionale Knoten und Kanten im Graphen hinzuzufügen:

```
1  if ( optionalInput embedded ) {
2      # Fuege optionale Knoten und Kanten hinzu
3  }
```

Quelltext 17: Ein Beispiel für die Verwendung des Sprachkonstruktes *optionalInput*.

Die Bedingung des If-Statements gibt genau dann *true* zurück, wenn alle Variablen, die im optionalen Block mit Namen *embedded* angegeben sind, mit der ODECoeffs DSL implementiert wurden. Fehlt mindestens eine Variable, so gibt das Sprachkonstrukt *false* zurück. Dadurch kann der Programmierer mittels der ODECoeffs DSL implizit die Struktur des Graphen beeinflussen.

Möchte man die Koeffizienten eines Verfahrens nicht explizit angeben, so kann

man diese mittels eines Aufrufs eines externen Programms berechnen und einlesen. Der Aufruf wird aus der folgenden Produktionsregel gebildet:

```
1 OtherFunction:
2     {External} 'callExternal' '(' file = Expression (',' 
          params+=Expression)* ')'
3 ;
```

Quelltext 18: Die Produktionsregel für *callExternal*.

Der Funktionsaufruf *callExternal* erwartet als ersten Parameter den Namen des externen Programms. Danach können beliebig viele weitere Parameter angegeben werden, welche wiederum dem externen Programm als Parameter übergeben werden. Ein Aufruf zum Einlesen von Koeffizienten des Adams-Bashforth-Verfahrens kann beispielsweise wie folgt erfolgen:

```
1 b : R[r] = callExternal("ABCoeffsGenB.exe", r)
```

Quelltext 19: Die Koeffizienten b des Adams-Bashforth-Verfahrens werden durch ein externes Programm berechnet.

b ist ein Vektor der Größe r, der mit den Ergebnissen des externen Programms *ABCoeffsGenB.exe* gefüllt werden soll. Als Übergabeparameter erhält das Programm die Variable r, also die Größe von b. Damit das Einlesen der Werte funktioniert, muss das externe Programm genau r Werte ausgeben. Für jeden Wert muss eine eigene Zeile ausgegeben werden. Mit dieser Methode können auch zweidimensionale Arrays eingelesen werden.

Zur Steuerung von DSL spezifischen Features können im Programm folgende Flags gesetzt werden:

@VALIDATION_OFF	Das Programm wird nicht validiert. Die semantische Wohlgeformtheit wird nicht überprüft.
@COMPILATION_OFF	Die durch den Generator erzeugte Java Klasse wird nicht automatisch übersetzt.
@VISUALIZATION_OFF	Der durch die DSL erzeugte Graph wird nicht visualisiert.

Der Validator der DSL führt folgende Checks durch:

checkConformance (Expression exp)	Der erwartete Typ und tatsächliche Typ eines Ausdrucks müssen kompatibel sein.
checkOptionalInputRef (OptionalInputRef r)	Ein Aufruf *optionalInput* darf sich nur auf einen existierenden optionalen Block beziehen.
checkEqualityConformance (Expression exp)	Die Typen der Ausdrücke links und rechts eines '==' Vergleichs müssen kompatibel sein.
checkComponents (File f)	Das Programm muss immer die Variablen *class* und *name* besitzen.
checkArrayMin (Declaration dec)	Wird ein Array initialisiert, so müssen die Dimensionen des Ausdrucks mit den deklarierten Dimensionen übereinstimmen.
checkNoDuplicateDeclarations (File f)	Namen für Variablen dürfen nicht doppelt vergeben werden.
checkNodeHasName (NodeFunction f)	Jeder Knoten erwartet als Argument einen Namen.
checkBasicNodeHasType (NodeFunction f)	Jeder Knoten vom Typ *Basic* erwartet die Angabe der grundlegenden Rechenart.
checkNonBasicNodeHasNoType (NodeFunction f)	Knoten, die nicht vom Typ *Basic* sind, dürfen keine Angabe einer grundlegenden Rechenart erhalten.

Der Generator erzeugt aus dem DSL Programm eine Java Klasse *Method*, die in der Methode *buildGraph* den Aufbau des resultierenden Datenflussgraphen beschreibt. Dazu werden die Hilfsklassen *Node* und *Dependency* verwendet:

```
1 public class Node {
2     public String name;
3     public int id;
4     public ArrayList<Dependency> ingoing;
5     public ArrayList<Dependency> outgoing;
6
```

```
7     public ArrayList<Node> children;
8     public Node parent;
9
10    public Kernel kernel;
11
12    public double relativeTime;
13    public ResultType resultType;
14    public double value;
15    public double[] value1dim;
16    public double[][] value2dim;
17
18    public boolean hasValue = false;
19    // ...
20 }
```

Quelltext 20: Die Hilfsklasse Node.

Neben einem Namen und einer ID hat jedes Objekt der Klasse Node eine Liste an eingehenden und ausgehenden Kanten. Zur hierarchischen Anordnung von Knoten wird eine Liste von Kindern und eine Referenz zum Elternknoten verwaltet. Für ein späteres Kernelmapping ist die Referenz zu einem Kernel vorgesehen. Weiterhin gespeichert wird die relative Zeit, der Ergebnistyp und der Wert, falls in der DSL angegeben.

```
1 public class Dependency {
2     public Node other;
3     public int timeDiff = 0;
4     // ...
5 }
```

Quelltext 21: Die Hilfsklasse Dependency.

Eine Dependency wird immer einem Knoten zugeordnet. Sie enthält eine Referenz zu dem Knoten, zwischen dem die Kante verläuft, und die Zeitschrittdifferenz. Wird also eine Kante von Knoten A nach Knoten B deklariert, so wird der Liste *outgoing* von A eine Dependency mit Referenz zu B und der Liste *ingoing* von B eine Dependency mit Referenz zu A hinzugefügt.

Bei der Codegenerierung werden zusätzlich Dateien erstellt, die später von der ODECoeffs DSL eingelesen werden. Die Datei "«Klassenname»Inputs.txt"enthält den Namen und Typ aller für das Verfahren benötigten Variablen. Die Namen aller optionalen Blöcke werden in der Datei "«Klassenname»OptionalInputs.txt"abgespeichert. Für jeden optionalen Block wird eine Datei "«Optionalname»«Klassen-

name»OptionalInputs.txt"erzeugt, welche wiederum alle für den optionalen Block benötigten Variablen mit Typ enthält.

5.2 Deklarierung von Koeffizienten - Die ODECoeffs DSL

Nachdem das Verfahren mit der ODEMethods DSL deklariert wurde, gilt es nun die im Input Block gelisteten Variablen zu implementieren. Diese werden mit der ODECoeffs DSL angegeben. Variablen können wie in der ODEMethod DSL deklariert werden. Folgender Programmausschnitt zeigt die DSL Features, die bei der Deklaration eines Arrays verwendet werden können:

```
1 s  : N = 4
2
3 A : R[s][s] =
4     [
5         [ ],
6         [ 1/4 ],
7         [ -189/800, 729/800 ],
8         [ 214/891, 1/33, 650/891 ]
9     ]
```

Quelltext 22: Deklaration eines zweidimensionalen Arrays.

Die Elemente des Arrays können mit Ausdrücken gefüllt werden. Wird ein Teilarray nicht mit genügend Elementen gefüllt, so wird dieser automatisch um angehängte Nullen ergänzt.

Die DSL überprüft automatisch, ob alle in der ODEMethod DSL als benötigt deklarierte Variablen implementiert wurden. Dazu wird die von der ODEMethod DSL erzeugte Datei "«Klassenname»Inputs.txt"eingelesen. Für jeden Eintrag dieser Datei wird überprüft, ob eine Variable mit entsprechendem Namen und Typen deklariert wurde. Ist dies nicht der Fall, so erzeugt der Validator eine Fehlermeldung. Ebenso werden die Dateien aller optionalen Blöcke eingelesen. Für jeden optionalen Block wird überprüft, ob die benötigten Variablen mit entsprechenden Namen und Typen deklariert sind. In der Ausgabedatei "Coeffs.java ", die vom Generator der DSL angelegt wird, wird für jeden optionalen Block der ODEMethod DSL eine boolean Variable erzeugt. Diese erhält den Wert true, falls alle für einen optionalen Block benötigten Variablen vorhanden sind. Falls mindestens eine Variable fehlt oder einen falschen Typ hat wird die boolean Variable auf false gesetzt. Die von der ODEMethod generierte Java Klasse kann auf die boolean

Variablen zugreifen und somit beim Bau des Graphen die *optionalInput* Aufrufe auflösen.

Mit dem in Quelltext 16 angegebenen Input Block kann folgendes Programm mit der ODECoeffs DSL erzeugt werden um ein eingebettetes Runge-Kutta-Verfahren mit vier Stufen zu erzeugen:

```
class : ID = "RK"
name : String = "RKF 2(3) s=4"
id   : ID = "RKF23_4"

description : String = " A Runge-Kutta-Fehlberg method of order
   2(3) with 4 stages."

s  : N = 4  # number of stages
p  : N = 2  # order of the method
pe : N = 3  # order of the embedded solution

A : R[s][s] =
    [
        [ ],
        [ 1/4 ],
        [ -189/800, 729/800 ],
    [ 214/891, 1/33, 650/891 ]
]

b : R[s] = [ 214/891, 1/33, 650/891, 0 ]

b2 : R[s] = [ 533/2106, 0, 800/1053, -1/78]

c : R[s] = [ 0, 1/4, 27/40, 1 ]
```

Quelltext 23: Deklaration der Verfahrenskoeffizienten für RKF23_4.

Aus dem Programm wird folgender Java Code generiert:

```
public class Coeffs {
    boolean embedded = true;
    boolean embeddedDiff = false;

    double[] b2Diff;

    String className = "RK";
```

```
8     String name = "RKF 2(3) s=4";
9     String id = "RKF23_4";
10    String description = " A Runge-Kutta-Fehlberg method of order
          2(3) with 4 stages.";
11    int s = 4;
12    int p = 2;
13    int pe = 3;
14    double[][] A = {{0, 0, 0, 0}, {0.25, 0, 0, 0}, {-0.23625,
          0.91125, 0, 0}, {0.24017957351290684, 0.030303030303030304,
          0.7295173961840629, 0}};
15    double[] b = {0.24017957351290684, 0.030303030303030304,
          0.7295173961840629, 0.0};
16    double[] b2 = {0.25308641975308643, 0.0, 0.7597340930674265,
          -0.01282051282051282};
17    double[] c = {0.0, 0.25, 0.675, 1.0};
18 }
```

Quelltext 24: Die generierte Klasse Coeffs.java.

Da die beiden Variablen *pe* und *b*2, die für den optionalen Block *embedded* benötigt werden, deklariert wurden, erhält die Variable *embedded* den Wert true. $b2Diff$ wurde nicht angegeben, daher wird die Variable $embeddedDiff$ auf false gesetzt. Nicht angegebene Variablen optionaler Blöcke, wie $b2Diff$, müssen von der DSL ergänzt werden, da diese in der von der ODEMethod DSL generierten Klasse Method.java benutzt werden. Da der Zugriff auf Variablen optionaler Blöcke in der Regel innerhalb eines konditionalen *optionalInput* Programmteils erfolgt wird auf die Variable jedoch nicht zugegriffen, weshalb diese nicht initialisiert werden muss.

Zur Steuerung von DSL spezifischen Features können im Programm folgende Flags gesetzt werden:

@VALIDATION_OFF	Das Programm wird nicht validiert. Die semantische Wohlgeformtheit wird nicht überprüft.
@REQUIRED_INPUT _VALIDATION_OFF	Das Programm wird nicht darauf geprüft, ob alle in der ODEMethod DSL als benötigt deklarierten Variablen implementiert wurden.

Der Validator der DSL führt folgende Checks durch:

checkConformance (Expression exp)	Der erwartete und tatsächliche Typ eines Ausdrucks müssen kompatibel sein.
checkEqualityConformance (Expression exp)	Die Typen der Ausdrücke links und rechts eines '==' Vergleichs müssen kompatibel sein.
checkComponents (File f)	Das Programm muss immer die Variablen *class*, *name* und *id* besitzen.
checkArrayMin (Declaration dec)	Wird ein Array initialisiert, so müssen die Dimensionen des Ausdrucks mit den deklarierten Dimensionen übereinstimmen.
checkNoDuplicateDeclarations (File f)	Namen für Variablen dürfen nicht doppelt vergeben werden.
checkInputVariables (File f)	Das Programm muss alle in der ODE-Method DSL als benötigt deklarierten Variablen implementieren.

5.3 Graphtransformationen - Die ODETransformation DSL

Durch die ODEMethod und die ODECoeffs DSL wurde der Aufbau eines Datenflussgraphen beschrieben und die benötigten Verfahrenskoeffizienten deklariert. Somit ist das Verfahren komplett beschrieben und es ist möglich, mit einem Generator aus den Java Klassen einen lauffähigen Code zu erzeugen, der Differentialgleichungen löst. Will man das Programm auf einer CPU laufen lassen, so sind keine weiteren Angaben nötig. Will man jedoch mit den DSLs Code generieren, der effizient auf GPUs läuft, so ist ein zusätzliches Mapping der einzelnen Knoten auf Kernels notwendig. Ein simpler Ansatz dabei wäre, jeden Knoten in einem eigenen Kernel zu berechnen. Der Artikel *Accelerating explicit ODE methods on GPUs by kernel fusion* von Korch und Werner[16] zeigt, dass durch geschickte Fusion einiger Kernel die Performanz eines ODE Lösers signifikant verbessert werden kann. Ziel der Kernel Fusionen ist die Erhöhung der Lokalität von Speicherzugriffen. Besonders bei Anwendungen, bei denen die Auswertung der RHS Funktionen nicht sehr rechenintensiv ist, wird der Speicher oft zum Flaschenhals. Daher ist es keine Überraschung, dass durch Kernel Fusion ein Speedup von über 3,5 erreicht werden kann.

Korch und Werner haben des weiteren festgestellt, dass die Performanz zusätzlich verbessert werden kann, indem auf den Datenflussgraphen sogenannte *Enabling transformations* angewendet werden. Diese Transformationen ändern die Struktur des Graphen so, dass Kernel Fusionen durchgeführt werden können, welche die Anzahl an DRAM Zugriffen noch stärker reduzieren. Durch die Transformationen können Abhängigkeitsrestriktionen, auf die in Kapitel 5.4 näher eingegangen wird, gelöst werden. Folgende Transformationen sind möglich:

- *Cloning Transformation:*
  In einigen Fällen ist es sinnvoll einen Knoten zu klonen. Folgendes Beispiel beschreibt so einen Fall: Angenommen ein Kernel lädt bereits alle Parameter, die als Eingabe für einen Knoten benötigt werden. Dann minimiert das Hinzufügen dieses Knotens zu dem Kernel die Anzahl an DRAM Zugriffen. Zusätzlich wird angenommen, dass der Knoten jedoch eine ausgehende Kante besitzt, die das Hinzufügen des Knotens zu dem Kernel verbietet. Oft kann dann der Knoten geklont und die ausgehenden Kanten zwischen dem originalen und geklonten Knoten so aufgeteilt werden, dass einer der Knoten dann dem Kernel hinzugefügt werden kann. Folglich muss der Knoten zwar doppelt berechnet werden, insgesamt kann diese Transformation jedoch trotzdem die Performanz erhöhen.

- *Copy Transformation:*
  Die Copy Transformation ist ein spezieller Fall der Cloning Transformation, bei dem der Klonprozess nicht innerhalb eines Zeitschritts statt findet. Stattdessen wird ein Knoten eines vorherigen Zeitschritts in den aktuellen kopiert. Alle eingehenden Kanten des originalen Knotens werden dem Klon mit einer um 1 erhöhten Zeitschrittdifferenz hinzugefügt. Alle ausgehenden Kanten mit Zeitschrittdifferenz 0 werden dem originalen Knoten und alle sonstigen Kanten dem Klon zugeordnet.

- *Splitting Transformation:*
  Die Splitting Transformation kann auf Mapping Operationen, insbesondere Linearkombinationen angewendet werden. Die zugrunde liegende Eigenschaft, dass Mapping Operationen in eine Kombination von mehreren partiellen Operationen zerteilt werden können, wird dabei ausgenutzt um einen Knoten des Datenflussgraphen aufzuteilen. Die eingehenden Kanten werden dann zwischen den beiden Teilen der Mapping Operation aufgeteilt. Wie bei der Cloning Transformation können somit auch durch eine Splitting Transformation Abhängigkeitsrestriktionen aufgelöst werden und Knoten zu einem

Kernel hinzugefügt werden, die sonst nicht mit diesem fusionierbar gewesen wären.

Die ODETransformation DSL nutzt als Eingabe den von der ODEMethod DSL erzeugten Datenflussgraphen. Die *Enabling Transformations* können darin durch folgende Produktionsregeln deklariert werden:

```
SplitNodeFunction:
    {Split} 'split' '(' arguments+=Argument (',' arguments+=Argument)*')'
;

CloneNodeFunction:
    {Clone} 'clone' '(' arguments+=Argument (',' arguments+=Argument)*')'
;

CopyFunction:
    {Copy} 'copy' '(' arguments+=Argument (',' arguments+=Argument)*')'
;
```

Quelltext 25: Die Produktionsregeln für die Enabling Transformations.

Den Argumenten der Transformationen müssen Referenzen zu den im Graphen enthaltenen Knoten übergeben werden. Eine Referenz eines Knotens kann durch folgende Produktionsregel erzeugt werden:

```
GetNodeFunction:
    {Node} 'getNode' '(' expression=Expression ')'
;
```

Quelltext 26: Die Produktionsregel zur Erzeugung einer Referenz auf einen Knoten.

Als Ausdruck erwartet *getNode* entweder den Namen des Knotens als String oder die von der DSL automatisch erzeugte ID des Knotens. Letztere kann beispielsweise aus der Visualisierung des Graphen abgelesen werden.

Jede Transformation erwartet das Argument *node*, welches den Knoten spezifiziert, auf den die Transformation ausgeführt werden soll. Die Splitting und Cloning Transformationen erwarten das zusätzliche Argument *dependencies*, der eine Liste von Knoten übergeben werden kann. Bei der Cloning Transformation wird damit angegeben, welche ausgehenden Kanten dem geklonten Knoten übergeben werden.

Bei der Splitting Transformation wird mit dem Argument *dependencies* deklariert, welche eingehenden Kanten dem neu erzeugten Knoten übergeben werden. Die Copy Transformation benötigt kein Argument *dependecies*, da die Zuordnung der eingehenden und ausgehenden Kanten zu den Knoten eindeutig ist.

Abbildung 9 zeigt die Graphen vor und nach der Anwendung der Transformationen mit folgenden Statements:

```
1 cl : Node = clone(node = b, dependencies = {d, e})
```

Quelltext 27: Die in Abbildug 9 dargestellte Cloning Transformation.

Eine Referenz auf den durch die Cloning Transformation erzeugten Knoten *b_2* wird von dem *clone* Aufruf zurückgegeben. Alle eingehenden Kanten von *b* werden auch *b_2* hinzugefügt. Als ausgehende Kanten erhält *b_2* die angegebenen Abhängigkeiten zu *d* und *e*, welche vom Knoten *b* entfernt werden.

```
1 cp : Node = copy(node = c)
```

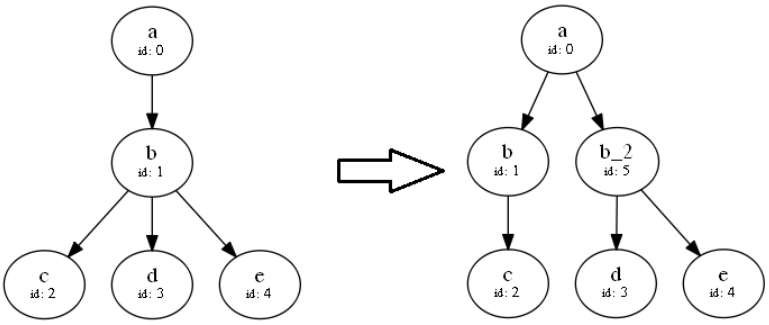
Quelltext 28: Die in Abbildug 9 dargestellte Copy Transformation.

Alle in Knoten *c* eingehenden Kanten sind Abhängigkeiten, die auch von *c_2* benötigt werden, daher werden diese übernommen. Die Zeitschrittdifferenz der Kanten muss dabei um 1 erhöht werden. Die ausgehenden Kanten von *c* werden aufgeteilt. Alle ausgehenden Abhängigkeiten mit Zeitschrittdifferenz 0 bleiben *c* erhalten, alle ausgehenden Kanten mit Zeitschrittdifferenz größer 0 werden dem Knoten *c_2* zugeordnet. Da *c_2* einen Zeitschritt später berechnet wird, muss die Zeitschrittdifferenz aller ausgehenden Knoten um 1 verringert werden.

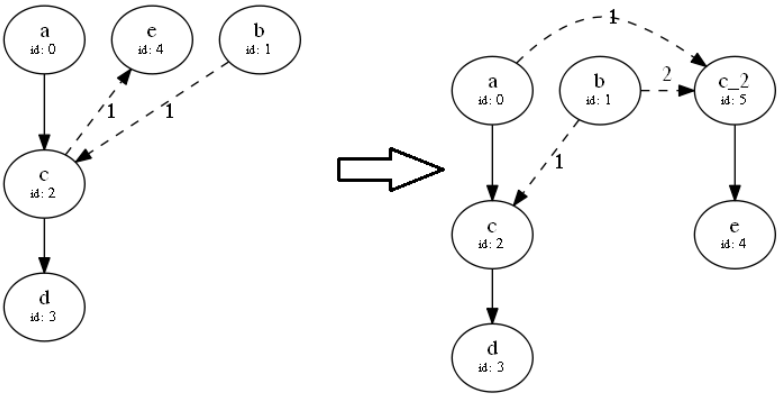
```
1 sp : Node = split(node = d, dependencies = {a, b})
```

Quelltext 29: Die in Abbildug 9 dargestellte Splitting Transformation.

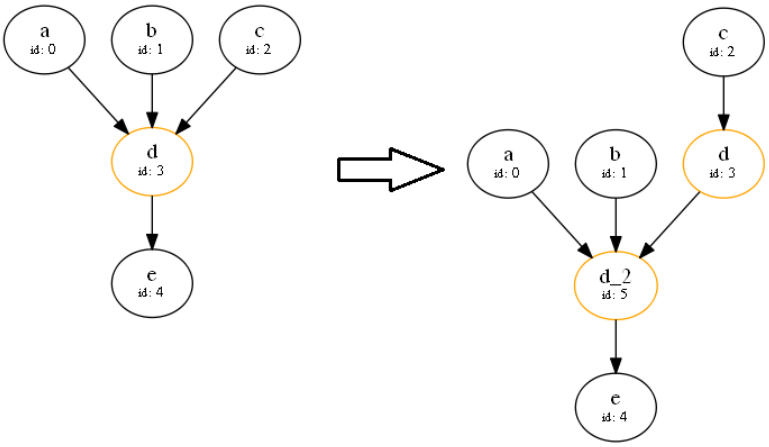
Die Akkumulation *d* wird geteilt. *d* erhält dabei eine ausgehende Kante zum neu erzeugten Knoten *d_2*. Die mit dem Argument *dependencies* deklarierten eingehenden Abhängigkeiten werden von *d* entfernt und dem neuen Knoten zugeordnet. Alle ausgehenden Kanten von *d* werden entfernt und *d_2* hinzugefügt.



(a) Die **Cloning Transformation**.



(b) Die **Copy Transformation**.



(c) Die **Splitting Transformation**.

Abbildung 9: Visualisierung der *Enabling Transformations*.

Zur Steuerung von DSL spezifischen Features können im Programm folgende Flags gesetzt werden:

@VALIDATION_OFF	Das Programm wird nicht validiert. Die semantische Wohlgeformtheit wird nicht überprüft.
@TRANSFORMATION _VALIDATION_OFF	Die Existenz eingehender und ausgehender Kanten sowie Knoten und dessen Typ wird nicht überprüft.
@COMPILATION_OFF	Die durch den Generator erzeugte Java Klasse wird nicht automatisch übersetzt.
@VISUALIZATION_OFF	Der durch die DSL erzeugte Graph wird nicht visualisiert.

Der Validator der DSL führt folgende Checks durch:

checkConformance (Expression exp)	Der erwartete Typ und tatsächliche Typ eines Ausdrucks müssen kompatibel sein.
checkComponents (File f)	Das Programm muss immer die Variablen *class* und *name* besitzen.
checkNoDuplicateDeclarations (File f)	Namen für Variablen dürfen nicht doppelt vergeben werden.
checkNodeExists (Node n)	Prüft, ob ein durch *getNode* referenzierter Knoten existiert.
checkSplitNodeIsAccumulation (Split s)	Der Knoten, auf dem eine Splitting Transformation ausgeführt werden soll, muss eine Akkumulation sein.
checkForNecessarySplitArguments (Split s)	Eine Splitting Transformation erwartet die Argumente *node* und *dependencies*.
checkSplitDependenciesExist (Split s)	Prüft, ob die mit *dependencies* angegebenen eingehenden Abhängigkeiten eines zu teilenden Knotens existieren.
checkForNecessaryCloneArguments (Clone c)	Eine Cloning Transformation erwartet die Argumente *node* und *dependencies*.
checkCloneDependenciesExist (Clone c)	Prüft, ob die mit *dependencies* angegebenen ausgehenden Abhängigkeiten eines zu klonenden Knotens existieren.

Der Generator erzeugt aus dem DSL eine Java Klasse *Transformation*, die ein Objekt von der von ODEMethod DSL generierten Klasse *Method* initialisiert. Mit der in *Transformation* enthaltenen Methode *applyTransformations* werden die in der DSL angegebenen Transformationen mit Aufrufen der Methoden *applySplit*, *applyClone* und *applyCopy* auf den Datenflussgraph angewendet.

5.4 Mapping der Graphelemente auf Kernel - Die ODEKernelMapping DSL

Nachdem der Datenflussgraph durch Anwendung von Transformationen vorbereitet wurde, gilt es nun die einzelnen Knoten auf Kernels abzubilden. Als Anfangszustand ist dabei anzunehmen, dass jede Operation in einem eigenen Kernel ausgeführt wird. Diese Kernels können dann unter Berücksichtigung bestimmter Regeln fusioniert werden. Folgende Restriktionen nennen Korch und Werner in ihrer Arbeit[16] :

- Jede Basisoperation muss auf genau einen Kernel gemappt werden
- Es dürfen keine zyklischen Abhängigkeiten zwischen Kernel entstehen
- Basisoperationen können aufgrund ihres Typen und ihrer Abhängigkeiten nicht miteinander fusionierbar sein

Zwei Basisoperationen unterliegen, je nach Abhängigkeit zueinander, folgenden Regeln[16]:

- *Keine Abhängigkeit:*
  Existiert keine Abhängigkeit zwischen zwei Basisoperationen, so lassen sich beide in einen Kernel fusionieren.
- *Eine direkte Abhängigkeit:*
  Angenommen, es existiert eine direkte Abhängigkeit von Knoten a zu Knoten b. a und b können in den gleichen Kernel fusioniert werden, wenn a keine Reduktion ist, weil das Ergebnis einer Reduktion immer erst verfügbar ist, wenn alle Workitems ihre Arbeit erledigt haben, und b keine RHS Funktionsauswertung ist, da eine RHS Funktionsauswertung auf den Eingabevektor willkürlich zugreift. Abbildung 10 zeigt die Fusionierbarkeit von Knoten bei einer direkten Abhängigkeit.

- *Eine indirekte Abhängigkeit:*
  Existiert eine indirekte Abhängigkeit zwischen zwei Knoten, so können diese Knoten nur fusioniert werden, falls die indirekte Abhängigkeit nur über Linearkombinationen getragen wird. Wird die Abhängigkeit über eine RHS Funktionsauswertung oder eine Reduktion getragen, so sind die Knoten nicht fusionierbar.

a → b	b		
a	RHS	Map	Reduce
RHS	-	Ja	Ja
Map	Nein	Ja	Ja
Reduce	Nein	Nein	-

Abbildung 10: Fusionierbarkeit der Knoten a und b im Falle einer direkten Abhängigkeit [16].

In der DSL lassen sich Kernels mit folgender Produktionsregel deklarieren:

```
KernelFunction:
    {Kernel} 'newKernel' '(' arguments+=Argument (',' arguments+=Argument)* ')'
;
```

Quelltext 30: Die Produktionsregeln zur Deklaration eines Kernels.

Das Statement erwartet die beiden Argumente *label* zur Angabe des Namens und *nodes*, mit dem eine Liste von Knoten deklariert wird, die dem Kernel zugeordnet werden soll. Eine Referenz zu einem im Graphen existierenden Knoten kann analog zur ODETransformation DSL mit dem Keyword *getNode* erzeugt werden.

Bei der Angabe der Knotenliste eines Kernels überprüft die DSL automatisch, ob die Knoten fusionierbar sind. Dazu werden die Knoten paarweise auf einen Verstoß der oben genannten Restriktionen geprüft. Wird eine Regel verletzt, so wird das zugehörige *newKernel* Statement mit einer Fehlermarkierung hervorgehoben. Die Fehlermeldung spezifiziert möglichst genau den Regelverstoß. Damit werden Probleme beim Kernelmapping frühestmöglich automatisch erkannt und können vom Benutzer leicht zurückverfolgt und behoben werden.

Werden in der DSL keine Kernels explizit angegeben, so wird ein Kernelmapping generiert, bei dem jede Basisoperation einem Kernel zugeordnet wird. Linearkombinationen werden automatisch erkannt und die zugehörigen Akkumulations- und Skalierungsknoten in einen Kernel fusioniert. Mit dem Datenflussgraphen aus Abbildung 8 als Eingabe wird ohne expliziter Deklaration von Kernels das in Abbildung 11 visualisierte Kernelmapping erzeugt.

Die Abhängigkeiten der Kernel werden in Abbildung 12 visualisiert. Während zwischen Knoten durchaus zyklische Abhängigkeiten existieren dürfen, was typischerweise bei impliziten Verfahren der Fall ist, darf eine zyklische Abhängigkeit innerhalb eines Zeitschritts zwischen Kernels nicht auftreten. Um eine zyklische Abhängigkeit von Kernels bei einem impliziten Verfahren zu vermeiden fusioniert die DSL daher automatisch alle Knoten, die einen Zyklus innerhalb eines Zeitschritts bilden, zu einem Kernel.

Die DSL stellt als zusätzliches Feature das Flag *@AUTO_MAX_FUSION* bereit. Wird dieses gesetzt, so versucht die DSL automatisch ein optimales Kernelmapping zu erzeugen. Dabei werden in einem iterativen Verfahren Knoten zu Kernels fusioniert. Pro Iterationsschritt werden für einen Kernel K alle Kernel mit direkter Abhängigkeit zu K als Kandidaten einer Kernelfusion betrachtet. Für jeden Kandidaten wird dann geprüft, ob alle enthaltenen Knoten mit den Knoten von K fusionierbar sind. Ist dies der Fall, so werden die beiden Kernel fusioniert. Dieser Prozess wird so lange wiederholt, bis in einer Iteration keine Änderung mehr auftritt. Nimmt man wieder das in Abbildung 8 abgebildete Runge-Kutta-Verfahren als Eingabe und setzt das *@AUTO_MAX_FUSION* Flag, so werden die Operationen in vier Kernels fusioniert. Abbildung 13 zeigt das resultierende Kernelmapping.

Die in Korch und Werner vorgestellten Implementierungsvarianten von Peer Verfahren lassen sich mit Hilfe der DSLs ohne großen Aufwand umsetzen 14. Die mittels der *ODEMethod* DSL festgelegten Struktur des Verfahrens muss für die Alternativen dabei nicht abgeändert werden. Zur Umsetzung von Peer-A müssen keine Transformationen am Graphen getätigt werden. Auch in der *ODEKernelMapping* DSL müssen keine Fusionen explizit angegeben werden und das *@AUTO_MAX_FUSION* Flag wird nicht gesetzt. Die Peer-B Variante kann analog zu Peer-A erzeugt werden mit dem einzigen Unterschied, dass das Flag *@AUTO_MAX_FUSION* gesetzt wird. Für Peer-C ist das Kopieren der beiden RHS Knoten des letzten Zeitschritts mittels des *copy* Statements in der *ODETransformation* DSL notwendig. Die Funktion zur automatischen maximalen Kernelfusion erzeugt das Kernelmapping der Variante Peer-C.

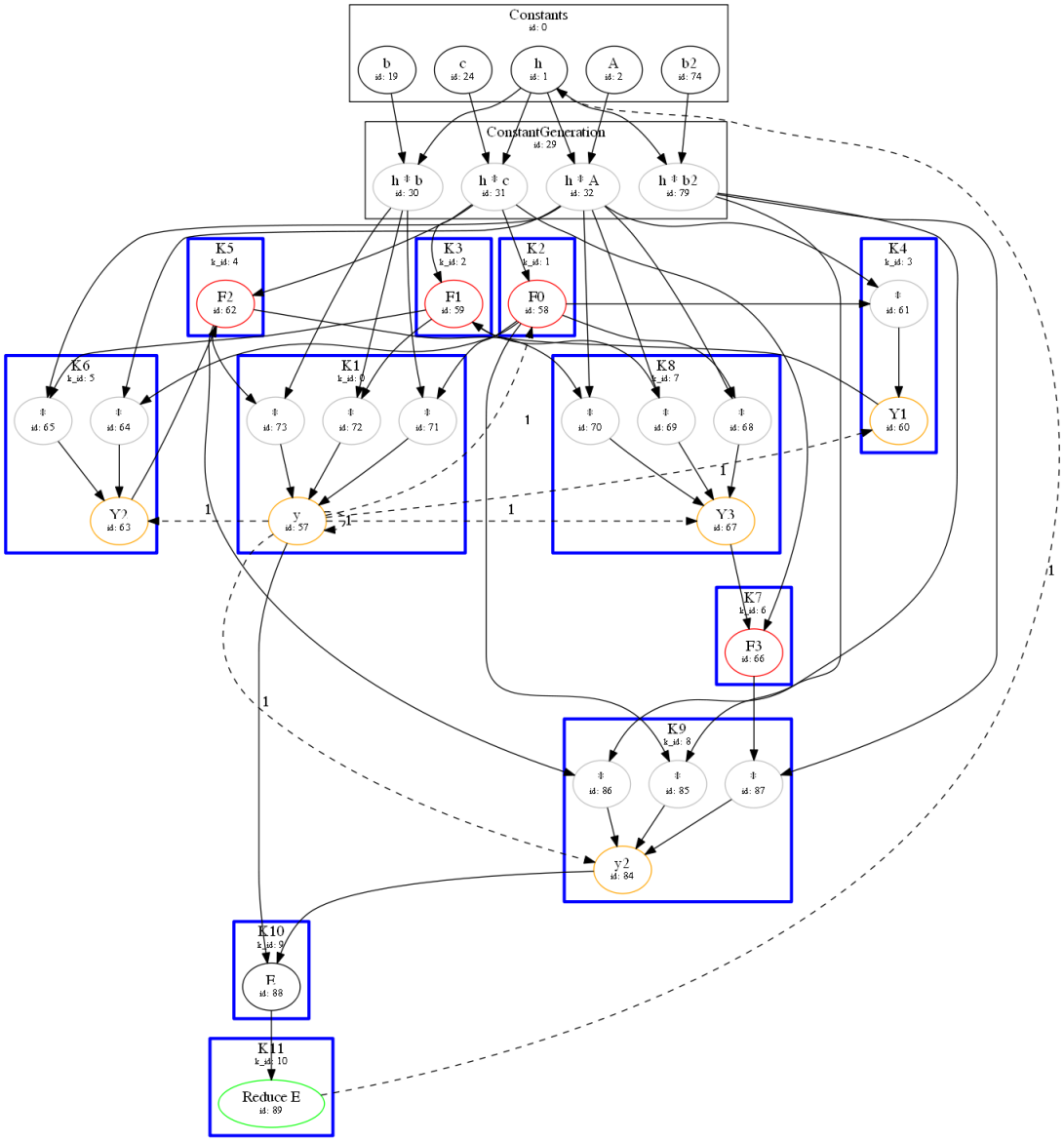


Abbildung 11: Das Standard Kernelmapping für das vierstufige eingebettete Runge-Kutta-Verfahren.

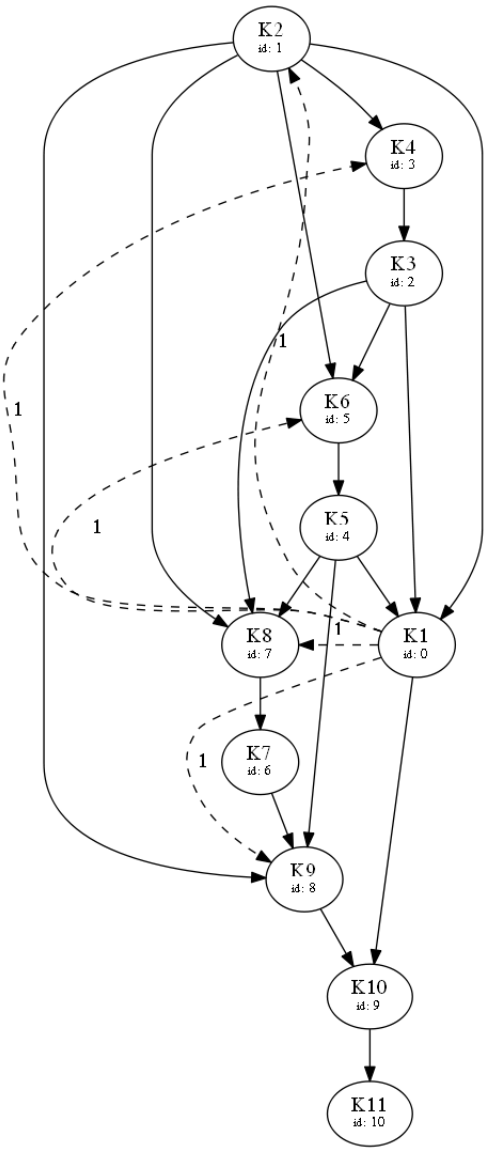


Abbildung 12: Die Abhängigkeiten der Kernels des in Abbildung 11 visualisierten Graphen.

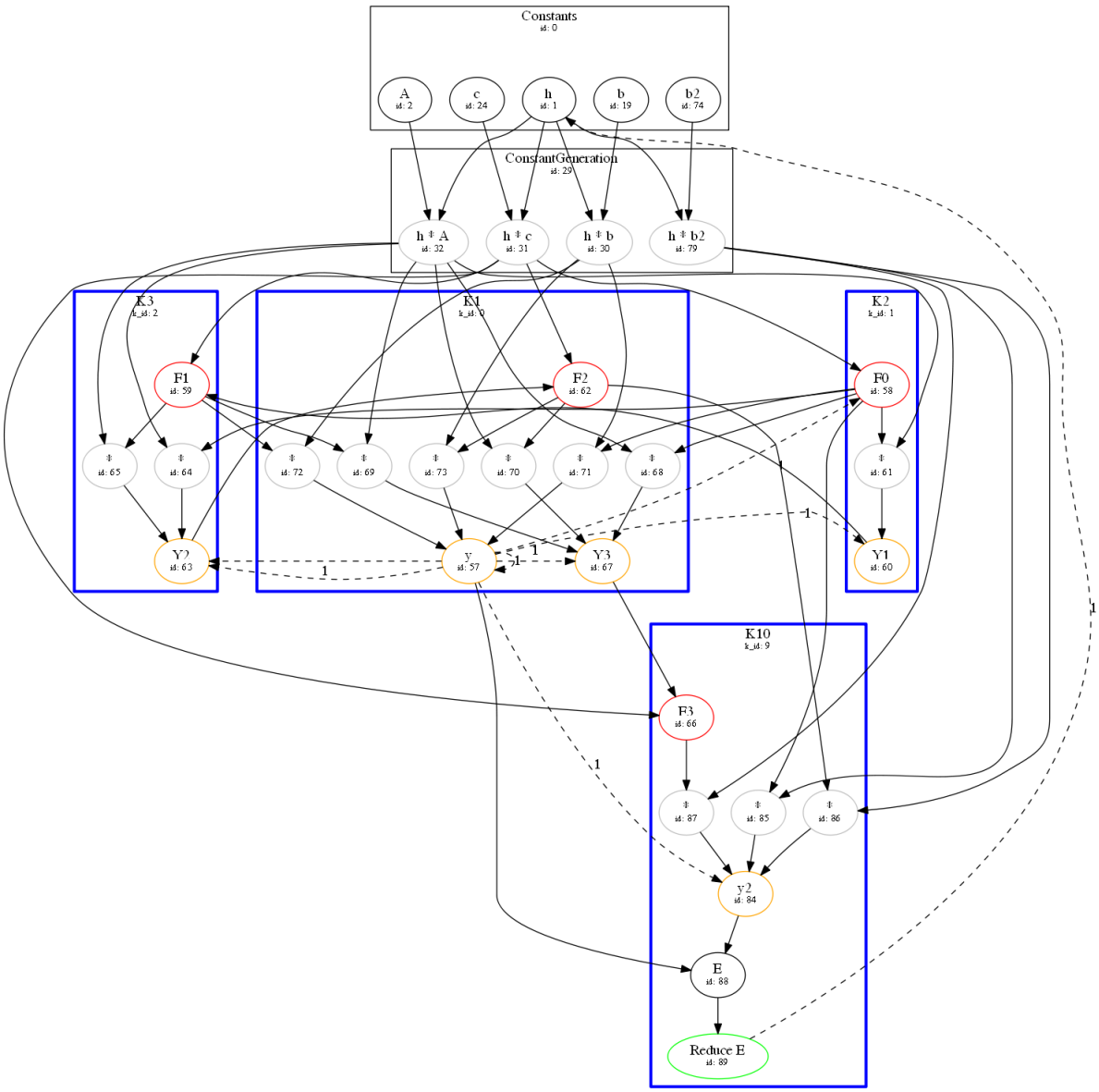
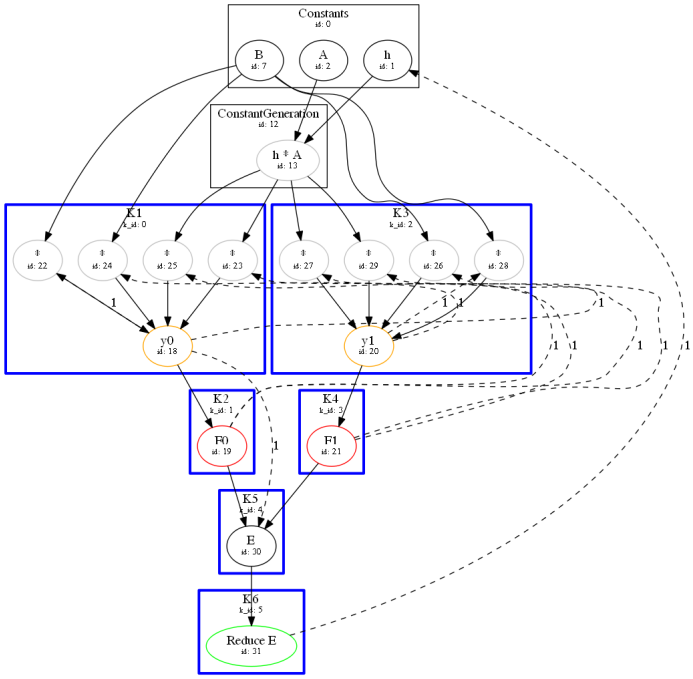
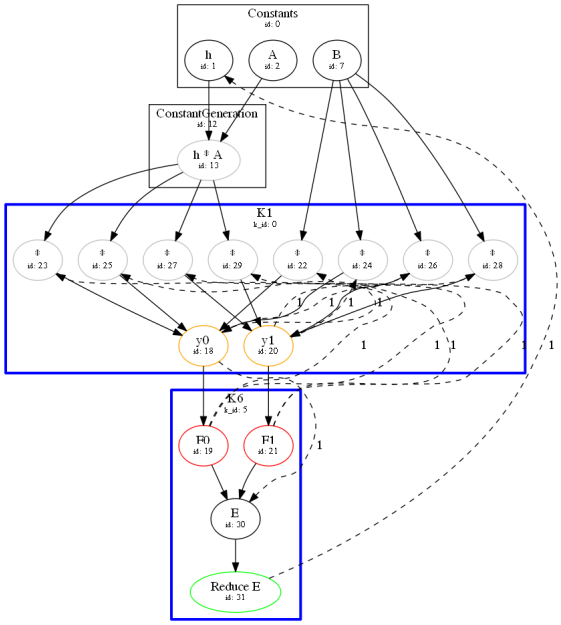


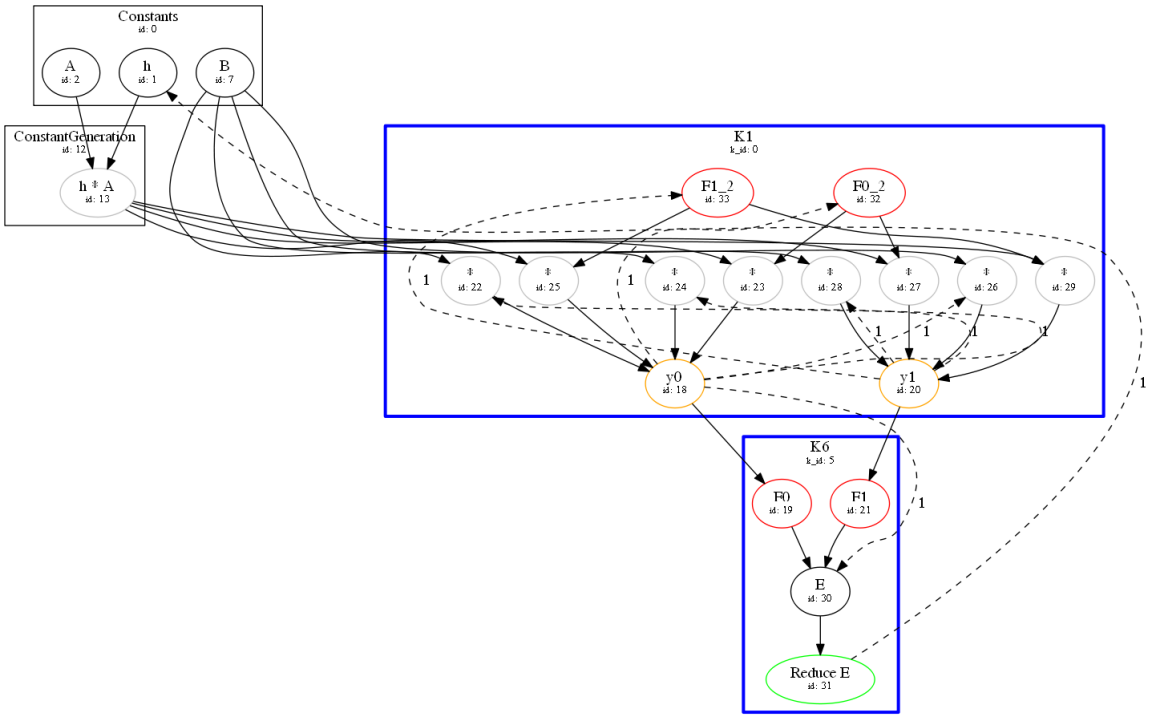
Abbildung 13: Automatische Kernelfusion des in Abbildung 11 visualisierten Graphen.



(a) **Peer-A**.



(b) **Peer-B**.



(c) **Peer-C**.

Abbildung 14: Die von Korch und Werner vorgestellten Implementierungsvarianten der Peer Verfahren.

Zur Steuerung von DSL spezifischen Features können im Programm folgende Flags gesetzt werden:

@VALIDATION_OFF	Das Programm wird nicht validiert. Die semantische Wohlgeformtheit wird nicht überprüft.
@KERNEL_VALIDATION _OFF	Die Kompatibilität von Knoten, die zu einem Kernel fusioniert werden sollen, wird nicht geprüft.
@LC_DETECTION_OFF	Die automatische Erkennung und Fusion von zu einer Linearkombination gehörenden Knoten wird deaktiviert.
@COMPILATION_OFF	Die durch den Generator erzeugte Java Klasse wird nicht automatisch übersetzt.
@VISUALIZATION_OFF	Der durch die DSL erzeugte Graph wird nicht visualisiert.
@AUTO_MAX_FUSION	Die DSL versucht automatisch ein möglichst optimales Kernelmapping zu generieren.

Der Validator der DSL führt folgende Checks durch:

checkConformance (Expression exp)	Der erwartete Typ und tatsächliche Typ eines Ausdrucks müssen kompatibel sein.
checkComponents (File f)	Das Programm muss immer die Variablen *class* und *name* besitzen.
checkNoDuplicateDeclarations (File f)	Namen für Variablen dürfen nicht doppelt vergeben werden.
checkNodeExists (Node n)	Prüft, ob ein durch *getNode* referenzierter Knoten existiert.
checkNodesCompatible (Kernel k)	Prüft, ob alle Knoten eines zu erzeugenden Kernels kompatibel sind.
checkNodeOnlyInOneKernel (File f)	Jeder Knoten darf explizit maximal einem Kernel zugeordnet werden.
checkForNecessaryKernelArguments (Kernel k)	Eine Deklarierung eines Kernels erwartet die Argumente *label* und *nodes*.

6 Zusätzliche Funktionalitäten

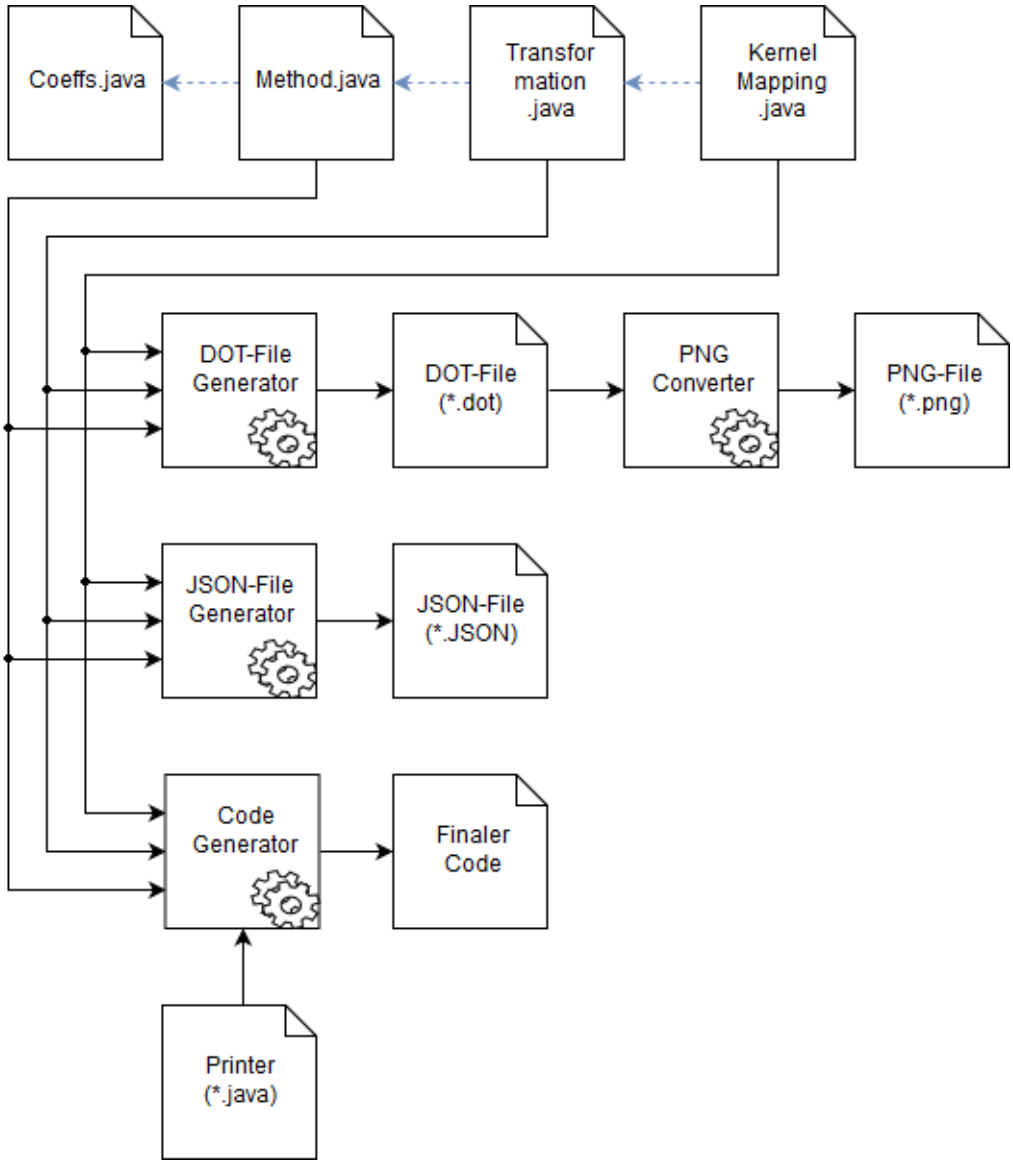


Abbildung 15: Die Zusätzlichen Funktionalitäten des DSL Frameworks.

6.1 Visualisierung

Der mit der ODEMethod spezifizierte Datenflussgraph kann recht schnell komplex werden, so besitzt das eingebettete Runge-Kutta-Verfahren mit vier Stufen beispielsweise 91 Knoten. Um den Überblick darüber nicht zu verlieren, ist es hilfreich den Graphen zu visualisieren. Im Rahmen dieser Arbeit wurde die Open Source Software Graphviz [5] zur Visualisierung von Graphen verwendet. Mit der Beschreibungssprache DOT lassen sich hierarchische Visualisierungen von gerichteten Graphen umsetzen. Dabei ist es möglich die Visualisierung von Knoten und Kanten selbst mit Sprachelementen zu Steuern. So lassen sich unter anderem die Form, Farbe und Beschriftung von Kanten deklarieren.

Abbildung 8 zeigt den mit DOT visualisierten Datenflussgraphen für das vierstufige Runge-Kutta-Verfahren, welches mit der ODEMethod DSL beschrieben wurde. Die im Graphen enthaltenen Knoten sind farblich nach der Basisoperation gekennzeichnet: Auswertungen der RHS Funktion sind rot, Akkumulationen gelb und Reduktionen grün markiert. Konstanten und spezielle Knoten, wie der Knoten E im Datenflussgraphen des Runge-Kutta-Verfahrens zur Berechnung des Fehlervektors, erhalten eine schwarze Umrandung. Jeder Knoten wird mit einer einzigartigen ID dargestellt, die in der ODETransformation und ODEKernelMapping DSL verwendet werden kann, um Knoten des Graphen zu referenzieren. Durchgezogene Kanten weisen auf den Datenfluss zwischen Knoten innerhalb eines Zeitschritts hin. Kanten zwischen Zeitschritten werden durch gestrichelte Linien dargestellt. Diese haben zusätzlich eine Annotation über die Zeitschrittdistanz der Abhängigkeit. Da das Runge-Kutta-Verfahren zu den Einschrittverfahren gehört, sind im dazugehörigen Graphen nur Kanten mit Zeitschrittdistanz 1 vorhanden. Hierarchien zwischen Knoten, wie die Zugehörigkeit von Konstanten zum Elternknoten mit Label *Constants*, können ebenfalls mit DOT visualisiert werden.

Ist die Visualisierung des Graphen nach der Deklarierung mit der ODEMethod DSL aktiviert, so werden automatisch drei Graphen mit unterschiedlichem Detailgrad gezeichnet. Der Graph mit höchster Genauigkeit zeigt alle Knoten, insbesondere alle Elemente der konstanten Vektoren und Matrizen. In der Visualisierung mit mittlerem Detailgrad werden Konstanten zu einem Knoten zusammengefasst. Die Visualisierung mit geringstem Detailgrad eliminiert alle Skalierungsknoten. Nach der Durchführung von Enabling Transformations mittels der ODETransformation DSL kann der Graph ebenfalls visualisiert werden. Nach dem Kernelmapping werden alle zu einem Kernel gehörigen Graphelemente in einem blau umrandeten Kasten dargestellt. Zusätzlich wird nach dem Kernelmapping eine Visualisierung der Abhängigkeiten zwischen den Kernels erstellt.

6.2 Finale Codegeneration

Mit den in dieser Arbeit vorgestellten DSLs werden Java Klassen erzeugt, die den Aufbau eines Datenflussgraphen, Transformationen darauf und das Mapping einzelner Knoten auf Kernels spezifizieren. Diese Klassen können nicht direkt verwendet werden, um ein numerisches Lösungsverfahren auf einem Rechensystem auszuführen. Dazu benötigt es eines finalen Codegenerators, der die Struktur des Datenflussgraphen durchschreitet und dabei ausführbaren Code erzeugt, mit dem Differentialgleichungssysteme gelöst werden können. Die Klasse *Generator* besitzt dafür die Methode *Generator* , die als Parameter einen *Printer*, eine Liste von Knoten, eine Liste von Kernels und den Namen der Klasse benötigt. Die Listen von Knoten und Kernels können dabei direkt der von den DSLs erzeugten Klassen entnommen werden. Die Methode ist wie folgt aufgebaut:

```
1  public static void generate(Printer printer, ArrayList<Kernel>
       kernelList, ArrayList<Node> nodeList, String className) throws
       Exception {
2      // ...
3
4      fileWriter = new FileWriter(className + "Kernel" +
           printer.getFileEndingName());
5      printWriter = new PrintWriter(fileWriter);
6
7      printer.initialize(className + "Kernel", kernelList, nodeList,
           printWriter);
8
9      printer.printFileHead();
10
11     for(Node n : nodeList) // Detect constants
12         if(n.value != 0 || n.value1dim != null || n.value2dim !=
               null)
13             setEvaluated(n);
14
15     for(Node n : nodeList) // Constant generation
16             if(n.kernel == null && inputEvaluated(n))
17                 evaluate(n);
18
19     while(evaluatedKernels != totalKernels) {
20         ArrayList<Kernel> candidates = getKernelCandidates();
21         if(candidates.size() == 0) throw new Exception("Some
```

```
            kernels could not be evaluated");

        Kernel next = candidates.get(0);
        evaluate(next);
    }

    if(evaluatedNodes != totalNodes) throw new Exception("Some
        nodes could not be evaluated");

    printer.printFileBottom();
    printWriter.close();
}
```

Quelltext 31: Die Methode generate zur finalen Codeerzeugung.

Der Code wird mit Hilfe von Methodenaufrufen des Printers generiert. Nach der Initialisierung des Printers wird zunächst der Kopf der Datei geschrieben. Danach werden Konstanten erkannt und alle Knoten zur Konstantengeneration evaluiert. Hinter dem Methodenaufruf *evaluate* steht eine Switch-Anweisung, die je nach Typ des Knotens wiederum spezielle Methoden des Printers aufruft, um entsprechenden Code zu erzeugen. Anschließend werden in einer Schleife die Kernel evaluiert. Um die Kernel in korrekter Reihenfolge auszuführen, wird in jedem Schleifendurchlauf für jeden Kernel geprüft, ob alle eingehenden Abhängigkeiten evaluiert worden sind. Erst dann kann dieser Kernel ebenfalls ausgewertet werden. Bei der Auswertung eines Kernels wird wiederum der gleiche Ansatz zur Findung einer richtigen Reihenfolge zur Evaluierung der im Kernel enthaltenen Knoten angewendet. Zuletzt wird das Ende der Datei durch entsprechenden Printer Aufruf ergänzt.

Das Printer Interface gibt vor, welche Methoden eine Klasse benötigt um zusammen mit der Generator Klasse finalen Code zu erzeugen:

```
public interface Printer {
    // The printer needs to be initialized first.
    public void initialize(String className, ArrayList<Kernel>
        kernelList, ArrayList<Node> nodeList, PrintWriter
        printWriter);

    public String getFileEndingName();

    public void printFileHead();
```

```
8       public void printFileBottom();
9
10      public void printMultiplicationNode(Node n);
11      public void printAdditionNode(Node n);
12      public void printOtherNode(Node n);
13      public void printRHSEvaluationNode(Node n);
14      public void printReductionNode(Node n);
15      public void printAccumulationNode(Node n);
16      public void printUndefinedNode(Node n);
17
18      public void printKernelHead(Kernel k);
19      public void printKernelBottom(Kernel k);
20  }
```

Quelltext 32: Das Printer Interface.

Zur Initialisierung des Printers werden benötigte Listen von Knoten und Kernels übergeben. Ebenfalls benötigt der Printer einen PrintWriter. Die Methode *getFileEndingName* gibt die Endung der zu erstellenden Datei zurück. Weiterhin muss eine Printer Klasse die Methoden zur Generation von Code aus den verschiedenen Knotentypen implementieren. Jeder Kernel und die Datei selbst haben in der zu erzeugenden Datei einen Kopf und ein Ende, die ebenfalls durch entsprechende Aufrufe an den Printer ergänzt werden.

Mit der Kombination aus Generator Klasse und Printer Interface lassen sich aus dem mit den DSLs erzeugten Datenflussgraphen der Code eines numerischen Lösungsverfahrens in einer beliebigen Programmiersprache generieren. Will der Benutzer Code in einer bestimmten Programmiersprache erzeugen, so muss er lediglich das Printer Interface in einer neuen Klasse so implementieren, dass Code in der gewählten Sprache erzeugt wird. Ebenfalls können für eine Programmiersprache mehrere Implementierungen des Printer Interfaces angefertigt werden, um verschiedene Codevarianten des gleichen Verfahrens in der gleichen Programmiersprache zu generieren. Die daraus resultierenden Varianten können beispielsweise für Autotuning Ansätze benutzt werden.

6.3 Modell Export

Zur Weiterverarbeitung oder Analyse können die von den DSLs erzeugten Datenflussgraphen exportiert werden. Mit Hilfe der Methoden der Klasse *Graph2File* werden die Elemente des Graphen mit allen Eigenschaften als JSON Datei abgespeichert. Diese enthält einen Array *nodes*, der Informationen zu allen Knoten

speichert, einen Array *edges*, der alle Kanten beinhaltet, sowie, falls vorhanden, einen Array *kernels*, der das Kernelmapping beschreibt.

7 Vergleich mit einer C++-API

In folgendem Kapitel werden die entwickelten DSLs mit einer C++-API verglichen, die im Rahmen der Arbeit von Korch und Werner[16] entwickelt wurde. Ähnlich wie in dieser Arbeit wird dabei zunächst ein Graph für ein Verfahren definiert. Anschließend werden die Basisoperationen auf Kernels abgebildet.

Die Verwendung der API setzt Vorkenntnisse mit Sprachkonzepten von C++ voraus, insbesondere von Zeigerarithmetik und Speicherverwaltung. Diese Aspekte entfallen bei der Entwicklung mit den DSLs. Dadurch ist das Programm für Domänenexperten, die wenig Erfahrung mit der Entwicklung mit C++ haben, zugänglicher. Zudem ist das Programm als DSL kompakter und weniger anfällig für Fehler, da die DSL viele irrelevante Aspekte automatisiert. Zur Deklarierung eines Datenflussgraphen inklusive Kernelmapping mit den DSLs müssen lediglich vier recht kurze Dateien angegeben werden. Die C++-Schnittstelle besteht im Gegensatz dazu aus einer Vielzahl von Klassen und Methoden, die wiederum in einer Vielzahl von Header- und Quelldateien mit Querverweisen abgelegt sind. Zur Erweiterung der Schnittstelle um ein neues Verfahren ist eine Kenntnis über Implementierungsdetails vieler Klassen und Methoden zwingend notwendig. Eine klare Trennung der verschiedenen Schritte zu einem Lösungsverfahren existiert im Gegensatz zu den DSL Programmen nicht. Das Resultat bei der Verwendung der Schnittstelle ist immer ein C++-Programm, während aus den DSL Programmen Code in einer beliebigen Programmiersprache generiert werden kann. Zudem lassen sich aus den DSL Programmen verschiedene Codevarianten erzeugen.

Die verschiedenen Basisoperationen sind in der API als Klassen implementiert und werden zur Erstellung eines Datenflussgraphen instanziiert und in einer Vector Struktur gespeichert. Informationsfluss zwischen den Operationen werden anders als bei der ODEMethod DSL nicht mit Kanten, sondern ebenfalls mit einem Vector beschrieben. Die Deklarierung von Kanten mit einem speziellem Sprachkonstrukt in der DSL entspricht daher eher der dahinter liegenden Struktur eines Datenflussgraphen.

Die API besitzt keine Methoden zur automatischen Umstrukturierung des Datenflussgraphen durch Enabling Transformations. Die Knoten und Abhängigkeiten müssen daher bei der Deklaration des Graphen geändert werden. Die Durchführung einer Enabling Transformation erfordert im Gegensatz dazu bei der Verwendung

von DSLs lediglich einen Funktionsaufruf. Die Spezifikation des Graphen in der ODEMethod DSL bleibt dabei unberührt.

Die C++-Schnittstelle stellt keine Mechanismen zur Überprüfung der Korrektheit einer Kernelfusion bereit. Der Entwickler muss somit selbst prüfen, welche Basisoperationen fusionierbar sind. Die DSL führt automatisch Tests im Hintergrund durch und zeigt eine aussagekräftige Fehlermeldung an, falls eine Restriktion verletzt wurde. Zudem bietet die DSL im Gegensatz zu der API eine automatische maximale Kernelfusion an.

8 Zusammenfassung

Xtext bietet nützliche Funktionalitäten, wie die Bereitstellung eines Validators und eines Generators, sowie die automatische Integration der entwickelten DSL in die Eclipse IDE. Somit ist es mit Hilfe des Frameworks möglich, schnell und einfach neue DSLs zu entwickeln.

Im Rahmen dieser Arbeit wurden mehrere DSLs entwickelt, die die Programmierung numerischer Lösungsverfahren für gewöhnliche Differentialgleichungen erleichtern. Durch die Verwendung von mehreren DSLs ist es möglich, gezielt Aspekte zu einem bestimmten Zeitpunkt der Entwicklung umzusetzen. Durch die Beschränktheit der DSLs auf die wesentlichen Aspekte zur Umsetzung eines Verfahrens muss sich der Anwender nicht um irrelevante Implementierungsdetails kümmern und kann somit schneller und mit weniger Programmcode sein Ziel erreichen. Die automatischen Validierungsmechanismen geben dabei schon zur Zeit der Entwicklung Feedback über die Korrektheit des Programmcodes. Eventuelle Fehler bei der Deklarierung des Datenflussgraphen, Durchführung von Enabling Transformations oder der Anwendung einer Kernelfusion werden somit frühzeitig erkannt und können schnell behoben werden. Durch die visuelle Aufbereitung der Datenflussgraphen können Verfahren auch ohne mathematische Formeln oder Code einfach kommuniziert werden. Ebenso simpel ist der Export der mit den DSLs erzeugten Graphen als JSON Format. Die Kombination aus Generator Klasse und Printer Interface sorgt dafür, dass mit relativ wenig Aufwand aus einem Graphen verschiedene Codevarianten eines Lösungsverfahrens in verschiedenen Programmiersprachen erzeugt werden kann. Die generierten Varianten können zukünftig beispielsweise für Autotuning Ansätze benutzt werden. Zusammen mit einer Hardwarespezifikation, beispielsweise wiederum in Form eines DSL Programms, kann Code generiert werden, der für ein spezielles System optimiert ist.

Literatur

[1] *Advanced Stencil-Code Engineering (ExaStencils).* www.exastencils.org/, Zugriff Januar 2019.

[2] *The CellML project .* www.cellml.org/, Zugriff Januar 2019.

[3] *Flint - a simulator for biological and physiological models .* www.flintproject.github.io/, Zugriff Januar 2019.

[4] *The Grammar Language.* www.eclipse.org/Xtext/documentation, Zugriff Januar 2019.

[5] *Graphviz - Graph Visualization Software.* www.graphviz.org, Zugriff Januar 2019.

[6] *OpenModelica.* www.openmodelica.org/, Zugriff Januar 2019.

[7] *The Pochoir Stencil Compiler.* www.people.csail.mit.edu/yuantang/pochoir.html, Zugriff Januar 2019.

[8] *Xtend Documentation.* www.eclipse.org/xtend/documentation, Zugriff Januar 2019.

[9] Bettini, Lorenzo: *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition.* 2nd. Packt Publishing, 2016. – ISBN 1786464969, 9781786464965

[10] Blake, Robert C. ; Gonsiorowski, Elsa ; O'Hara, Thomas ; Richards, David F.: Modular Expression Language for Ordinary Differential Equation Editing. In: *Proceedings of Seventh International Work-shop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, November 2017

[11] Burrage, Kevin: *Parallel and Sequential Methods for Ordinary Differential Equations.* New York, NY, USA : Clarendon Press, 1995. – ISBN 0-19-853432-9

[12] Butcher, J. C.: *Numerical Methods for Ordinary Differential Equations (2Nd Ed.).* John Wiley & Sons, Ltd, 2008. – ISBN 9780470753767

[13] Hagedorn, Bastian ; Stoltzfus, Larisa ; Steuwer, Michel ; Gorlatch, Sergei ; Dubach, Christophe: High Performance Stencil Code Generation with Lift. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. New York, NY, USA : ACM, 2018 (CGO 2018), S. 100–112. – ISBN 978-1-4503-5617-6

[14] Hairer, E. ; Nørsett, S. P. ; Wanner, G.: *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems*. Berlin, Heidelberg : Springer-Verlag, 1993. – ISBN 0-387-56670-8

[15] Iserles, Arieh ; Norsett, Syvert: On the Theory of Parallel Runge—Kutta Methods. In: *Ima Journal of Numerical Analysis* 10 (1990), 10, S. 463–488

[16] Korch, Matthias ; Werner, Tim: Accelerating explicit ODE methods on GPUs by kernel fusion. In: *Concurrency and Computation: Practice and Experience* 30 (2018), Nr. 18, S. e4470

[17] Petcu, Dana: Solving Initial Value Problems with a Multiprocessor Code. In: Malyshkin, Victor (Hrsg.): *Parallel Computing Technologies*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1999, S. 452–465. – ISBN 978-3-540-48387-8

[18] Steinberg, David ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. – ISBN 0321331885

[19] Sujeeth, Arvind K. ; Brown, Kevin J. ; Lee, HyoukJoong ; Rompf, Tiark ; Chafi, Hassan ; Odersky, Martin ; Olukotun, Kunle: Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. In: *ACM Trans. Embedded Comput. Syst.* 13 (2014), S. 134:1–134:25

[20] Voelter, Markus ; Benz, Sebastian ; Dietrich, Christian ; Engelmann, Birgit ; Helander, Mats ; Kats, Lennart C. L. ; Visser, Eelco ; Wachsmuth, Guido: *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. – ISBN 978-1-4812-1858-0

[21] Wahib, Mohamed ; Maruyama, Naoya: Scalable Kernel Fusion for Memory-bound GPU Applications. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Piscataway, NJ, USA : IEEE Press, 2014 (SC '14), S. 191–202. – ISBN 978-1-4799-5500-8

[22] Wahib, Mohamed ; Maruyama, Naoya: Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA : ACM, 2015 (HPDC '15), S. 259–270. – ISBN 978-1-4503-3550-8

[23] Wang, Guibin ; Lin, YiSong ; Yi, Wei: Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In: *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*. Washington, DC, USA : IEEE Computer Society, 2010 (GREENCOM-CPSCOM '10), S. 344–350. – ISBN 978-0-7695-4331-4

[24] Weiner, Rüdiger ; Biermann, Katja ; Schmitt, Bernhard A. ; Podhaisky, Helmut: Explicit two-step peer methods. In: *Computers & Mathematics with Applications* 55 (2008), Nr. 4, S. 609 – 619. – ISSN 0898-1221

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel eingesetzt habe. Die Arbeit wurde noch nicht zur Erlangung eines akademischen Grades eingereicht.

Ort, Datum	Unterschrift