Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fraunhofer
SCAI

Abschlussarbeit

im Studiengang Master Informatik

Entwicklung einer Steuerung für Löser linearer Gleichungssysteme

Kai Patrick Buschulte

Matrikelnummer: 9009274

kai.buschulte@smail.inf.h-brs.de

Erstprüfer:	Prof. Dr. Manfred Kaul
Zweitprüfer:	Prof. Dr. Simone Bürsner
Eingereicht am:	24. Oktober 2012

Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt bzw. nicht veröffentlicht.

Kai Patrick Buschulte

Vogelsangerstr. 49, 50823 Köln

Inhaltsverzeichnis

Akronyme

AMG	Algebraic MultiGrid.
Antlr	ANother Tool for Language Recognition.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
BLAS	Basic Linear Algebra Subprograms.
BNF	Backus Naur Form.
CG	Conjugate Gradient.
COO	Coordinate.
COTS	Commercial off-the-shelf.
CPU	Central Processing Unit.
CSR	Compressed Sparse Row.
CSS	Cascading Style Sheets.
CUDA	Compute Unified Device Architecture.
DIA	Diagonal.
DOM	Document Object Model.
DSEL	Domain Specific Embedded Language.
DSL	Domain Specific Language.
DTD	Document Type Definition.
EBNF	Extended Backus Naur Form.
ELL	ELLPack.

FAST	Family-Oriented Abstractions, Specifications and Translation.
FODA	Feature-Oriented Domain Analysis.
GAMG	Geometric-Algebraic MultiGrid.
GMRES	Generalized Minimal RESidual.
GPL	General Purpose Language.
GPU	Graphics Processing Unit.
GUI	Graphical User Interface.
HTML	Hypertext Markup Language.
ID	Identifikator.
ILU	Incomplete Lower-Upper.
JDS	Jagged Diagonal Storage.
JSON	JavaScript Object Notation.
LAMA	Library for Accelerated Mathematical Applications.
LU	Lower-Upper.
MPI	Message Passing Interface.
OpenFOAM	Open Field Operation And Manipulation.
OpenMP	Open Multi-Processing.
PEG	Parsing Expression Grammar.
PETSc	Portable, Extensible Toolkit for Scientific Computation.
PGAS	Partitioned Global Address Space.
rhs	Right-Hand-Side.

RSS	Really Simple Syndication.
SAX	Simple API for XML.
SCAI	Institute for Algorithms and Scientific Computing.
SOR	Successive Over-Relaxation.
SQL	Structured Query Language.
SVG	Scalable Vector Graphics.
UML	Unified Modelling Language.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.
XML	Extended Markup Language.
XSD	XML Schema Definition.

1. Einleitung

Computersimulationen oder Berechnungen von mathematischen Modellen benötigen als Basis für ihre Berechnungen Routinen zum Lösen linearer Gleichungssysteme. Die in der Simulationssoftware verwendeten Modelle werden häufig in großen dünnbesetzten Matrizen gespeichert. Diese Matrizen werden verwendet, um den Berechnungsoverhead zu verringern, der durch Null-Elemente in der Matrix entsteht. Zur Speicherung solcher Matrizen können unterschiedliche Formate verwendet werden, die eigene Implementierungen für die durch die Löser verwendeten mathematischen Operationen verlangen. Die Simulationssoftware Open Field Operation And Manipulation (OpenFOAM), die sich besonders auf Strömungsprobleme spezialisiert hat, verwendet eigene, fertig implementierte Löser für lineare Gleichungssysteme mit dünn- und dichtbesetzten Matrizen [siehe OpenFOAM, 2012a]. Die am Fraunhofer Institute for Algorithms and Scientific Computing (SCAI) entwickelte Software-Bibliothek mit dem Namen Library for Accelerated Mathematical Applications (LAMA) hat das Ziel, solche Standard-Löser durch LAMA-Löser zu ersetzen bzw. eine optimierte Alternative zu bieten [LibAMA, 2012]. Optimiert werden die LAMA-Löser durch Parallelisierung auf dem Hauptprozessor, die Auslagerung der Berechnungen auf Grafikkarten und die Verteilung der Berechnungen über mehrere Rechenknoten hinweg.

Die in der LAMA-Bibliothek angebotenen Löser haben verschiedene Konfigurationsmöglichkeiten, um ihr Verhalten während der Lösungsphase zu beeinflussen. Iterative Löser können beispielsweise Kriterien zugewiesen bekommen, die vorgeben, wann das Ergebnis der vorhergegangenen Iteration genau genug ist, um die Lösungsphase damit abzuschließen. Aktuell werden die Löser direkt in die Anwendersoftware eingebunden und dort entsprechend konfigu-

riert. Ändert der Nutzer die Konfiguration, da sie aus seiner Sicht nicht optimal ist, so muss die Anwendersoftware neu kompiliert werden. Dieser Schritt kostet viel Zeit und erfordert auf Seiten des Nutzers tiefere Kenntnisse über das Application Programming Interface (API) der LAMA-Bibliothek und der C++-Programmierung.

Das Ziel dieser Arbeit ist es, diesen Prozess für den Anwender zu vereinfachen. Dazu soll eine domänenspezifische Sprache in Form einer Konfigurationssprache entwickelt werden. Die Konfigurationsbeschreibung soll zur Laufzeit der Anwendersoftware durch den zu entwickelnden Interpreter eingelesen und anhand der enthaltenen Definitionen Löserinstanzen erzeugt und konfiguriert werden. Da zur Hauptanwendergruppe primär Ingenieure zählen, soll die Sprache auf diese ausgerichtet werden, indem bestehende Notationen für solche Konfigurationssprachen berücksichtigt werden, um die Benutzerfreundlichkeit hoch und den Einarbeitungsaufwand möglichst gering zu halten.

Neben den Haltekriterien für iterative Lösungsverfahren ist die Konfigurierbarkeit von Löserverknüpfungen ein sehr wichtiger Bestandteil dieser Arbeit. Das in der LAMA-Bibliothek implementierte Algebraic MultiGrid (AMG)-Verfahren verfügt aktuell nur wenige Konfigurationsmöglichkeiten, welche in dieser Arbeit geschaffen werden sollen.

Zunächst wird in der Arbeit auf die Bestandteile und Struktur des bestehenden LAMA-Projektes eingegangen und im Anschluss werden Grundlagen zur Entwicklung einer domänenspezifischen Sprache vorgestellt. Darauf folgend wird eine Anforderungserhebung durchgeführt und anhand der gesammelten Informationen Kriterien zur Auswahl einer Implementierungsmöglichkeit aufgestellt. Die vorausgewählten Implementierungsmöglichkeiten werden verglichen und bewertet. Anschließend wird zusammen mit der getroffenen Auswahl ein Konzept für die Umsetzung entwickelt, indem auch die konkrete Syntax der Sprache festgelegt wird. Der Ablauf der Implementierung und die Auswertung der Funktionsfähigkeit und Performanz werden in den zwei Folgekapiteln ausgeführt.

2. Grundlagen

In diesem Kapitel werden die Themen vorgestellt, die als Grundlage der Arbeit dienen und für das Verständnis der Folgekapitel nötig sind. Im ersten Teil des Kapitels wird die LAMA-Bibliothek vorgestellt. Dabei wird auf die wichtigsten Komponenten, deren Schnittstellen und die Funktionsweise der Bibliothek eingegangen. Darauf folgend werden die Idee, Einsatzzwecke und die grundlegenden Unterscheidungen von Domain Specific Languages (DSLs) veranschaulicht. Um einen Überblick über die aktuellen Möglichkeiten der Löserkonfiguration zu bieten, werden gängige Anwendungen für dieses Problemfeld vorgestellt.

2.1. Library for Accelerated Mathematical Applications

LAMA ist eine am Fraunhofer SCAI entwickelte Software-Bibliothek, welche Schnittstellen für mathematische Operationen aus der linearen Algebra bereitstellt und darauf aufbauend verschiedene Löser für lineare Gleichungssysteme anbietet. Die Anwendungsgebiete von LAMA sind z.B. Simulationsanwendungen, die die optimierten Routinen der Bibliothek nutzen. Derzeit wird die Bibliothek an die Simulationssoftware OpenFOAM gebunden. Da solche Modelle teilweise in sehr großen dünnbesetzten Matrizen gespeichert werden, unterstützt LAMA gängige Speicherformate für Matrizen mit vielen Null-Elementen. Die Bibliothek lässt sich in drei Schichten aufteilen, was in Abbildung 2.1 gezeigt wird. Auf der ersten Schicht liegen die Kernel, welche die Grundoperationen auf den konkreten Daten implementieren oder aufrufen. Darüber liegen die Daten-Container, die die Kernel so einsetzen, dass sie auch für verteilte Berechnungen eingesetzt werden können. Hierauf baut die Expressionschicht auf, mit der durch C++-Expressions versucht wird, eine einfache Möglichkeit zu bieten

mathematische Operationen zu beschreiben, indem Operatorüberladungen und C++-Templates verwendet werden. Diese Ausdrücke kommen ebenfalls bei der Entwicklung der nächsthöheren Schicht, dem Löser-Framework, zum Einsatz. Diese Schichten werden in den folgenden Abschnitten genauer erläutert.

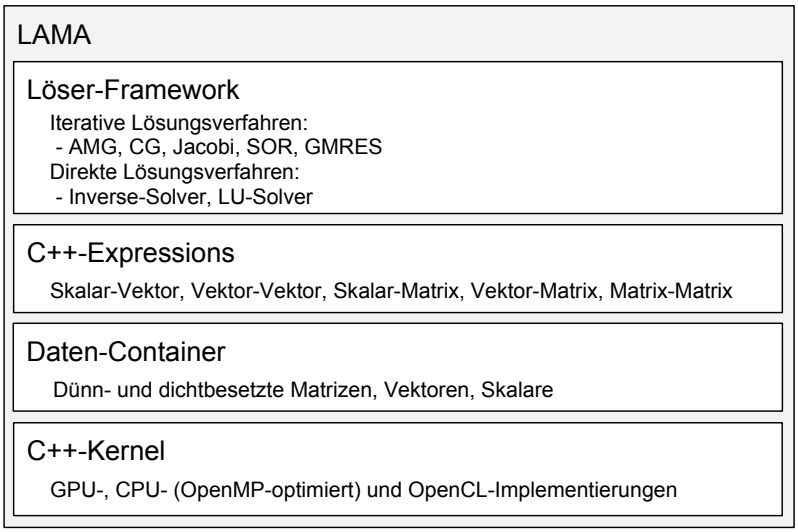

Abbildung 2.1.: Ein Überblick über die LAMA-Bibliothek und dessen grobe Aufteilung.

2.1.1. Kernel (Grundoperationen)

Auf der untersten Schicht befinden sich die Kernel, welche die nötigen Rechenschritte für eine gegebene Grundoperation der linearen Algebra bereitstellen, wie z.B. die Matrix-Vektor-Multiplikation oder ein Skalarprodukt. Diese Grundoperationen werden teilweise durch externe Bibliotheken, wie der Basic Linear Algebra Subprograms (BLAS)-Bibliothek abgedeckt [Netlib, 2012]. Fehlende Operationen, z.B. für dünnbesetzte Matrizen, wurden durch die Entwickler des LAMA-Projektes selbst implementiert und optimiert.

Kernel wurden für unterschiedliche Berechnungsorte implementiert, sodass Berechnungen neben der Central Processing Unit (CPU) auch auf einer Graphics Processing Unit (GPU) ausgeführt werden können. Berechnungsorte werden

in LAMA durch *Context*-Klassen beschrieben. Diese Klassen leiten von der abstrakten Klasse Context ab und implementieren alle nötigen Vor- und Nachbereitungsschritte, wozu Speicherallokation im Grafikspeicher, Kopieren auf und von dem Grafikspeicher gehören, um die Berechnungen an diesem Ort ausführen zu können. Aktuell wird für Grafikkarten eine Implementierungen mit der Compute Unified Device Architecture (CUDA)-Architektur von NVIDIA bereitgestellt [CUDA, 2012]. Die auf der CPU implementierten Routinen, wurden mit Open Multi-Processing (OpenMP) threadparallel für gemeinsamen Speicher optimiert [OpenMP, 2012].

2.1.2. Daten-Container (Matrix, Vektor und Skalar)

Aufbauend auf die Kernel wurden die Daten-Container implementiert, die alle relevanten Daten für die Berechnungen so verpacken, dass sie für verteiltes Rechnen eingesetzt werden können. Für die Speicherung von dünnbesetzten Matrizen unterstützt LAMA unterschiedliche Speicherformate, wozu aktuell Compressed Sparse Row (CSR), ELLPack (ELL), Jagged Diagonal Storage (JDS), Coordinate (COO) und Diagonal (DIA) zählen. Diese Formate sorgen dafür den Speicher- und Berechnungsoverhead, der durch solche Null-Elemente entsteht, möglichst gering zu halten. Welches der Speicherformate für eine gegebene Matrix geeignet ist, ist stark von der Matrixstruktur abhängig, da ggf. durch bessere Speicherzugriffe doch zusätzliche Null-Elemente mitgespeichert werden müssen oder die Zugriffsoperationen des Formates zu aufwändig sind. Für eine detaillierte Beschreibung dieser Speicherformate wird auf die Literatur verwiesen [Saad, 2003, S. 378ff]. Ein konkretes Beispiel zur Speicherung einer dünnbesetzten Matrix im CSR-Format befindet sich im Anhang A.1 auf Seite 114.

Die Klassenhierarchie wird im Diagramm in Abbildung 2.2 dargestellt. Die abstrakten Klassen *Matrix* und *Vector* sorgen für eine einheitliche Schnittstelle der konkreten Matrix- und Vektor-Implementierungen. Hierzu gehören der *DenseVector*, die *DenseMatrix* – also mit Speicherung von Null-Werten – und dünnbesetzte Matrix-Implementierungen, die von der Klasse *SparseMatrix* ab-

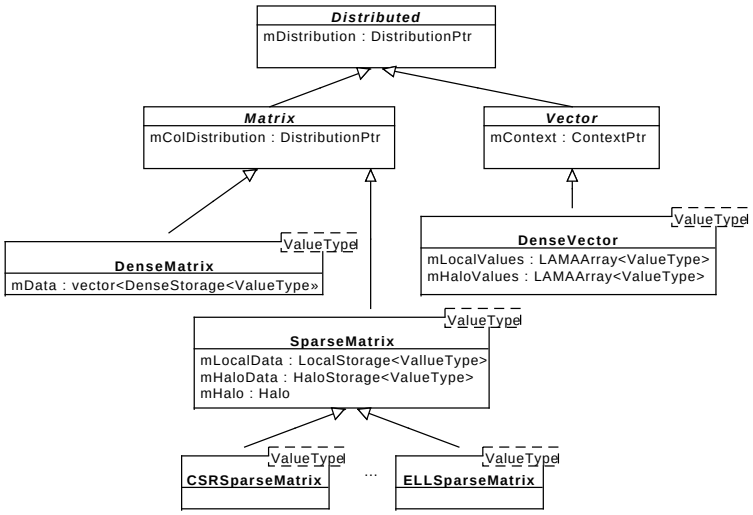

Abbildung 2.2.: Hierarchie der Daten-Container-Klassen. Wesentliche Unterscheidungen der Matrixklassen sind dicht- und dünnbesetzte Matrizen.

leiten. Die Klassen der dünnbesetzten Klassen beziehen sich immer auf eins der zuvor erwähnten Speicherformate. Die dünnbesetzten Matrizen halten ihre Daten in zwei separaten Speicherbereichen, deren Fließkommazahlgenauigkeit wird durch den Templateparameter *ValueType* vorgegeben. Jeder Speicherbereich (engl. Storage) entspricht der Speicherung in einem vorgegebenen Format und wird in einer getrennten Matrixstorage-Klasse implementiert.

Die Klassenhierarchie der Speicherbereiche ist in Abbildung 2.3 zu finden. Bei Einsatz einer *CSRSparseMatrix*, wird beispielsweise vorgeben, dass beide Speicherbereiche vom Typ *CSRStorage* sind und entsprechend im CSR-Format gespeichert werden. Es kann jedoch auch eine Instanz der Klasse *SparseMatrix* mit zwei unterschiedlichen Speicherbereichen eingesetzt werden.

Die Verwendung zweier separater Speicherbereiche hängt damit zusammen, dass es hiermit ermöglicht wird, die Daten-Container auf mehrere Rechenknoten zu verteilen. Die verwendeten Matrizen und Vektoren bekommen dafür bei ihrer Generierung eine Verteilung zugeordnet, sodass jeder Rechenknoten genau weiß, welcher Teil des Daten-Containers der eigene lokale Teil ist, welcher Teil

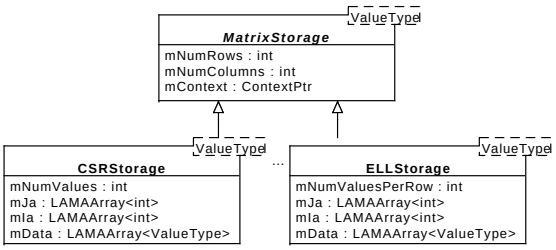

Abbildung 2.3.: Aufbau des templatisierten Matrixspeichers mit den Beispielspeicherbereichen ELL und CSR.

anderen Prozessen zugeordnet ist und welcher Teil bei bestimmten Operationen zwischen den Knoten ausgetauscht werden muss. Die *Distributed* Klasse ist die einzige Klasse, die die Vektor- und die Matrix-Klasse gemeinsam haben. Die Klasse hat eine Member-Variable, die einen Zeiger auf Verteilungsbeschreibung hält. Diese gilt für einen Vektor elementweise und für eine Matrix zeilenweise. Die Basisklasse *Matrix* beinhaltet deshalb noch eine Spaltenverteilung.

Für die Kommunikation zwischen den Prozessen können Hochgeschwindigkeitsübertragungstechnologien wie z.B. *InfiniBand* und *Myrinet* eingesetzt werden [InfiniBand, 2012; Myricom, 2012]. Wie die Daten-Container auf die Knoten verteilt werden, wird durch verschiedene verfügbare Verteilungsarten vorgegeben. In Abbildung 2.4 werden zur Veranschaulichung zwei Beispiele für Verteilungen eines Gleichungssystems gezeigt. Die Blockverteilung auf zwei Prozessoren wird auf der linken Seite de Abbildung gezeigt. Auf der rechten Seite befindet sich die zyklische Verteilung, diese kann Elementweise, wie hier im Beispiel, geschehen oder ebenfalls durch eine Blockgröße grobgranularer verteilt werden. Diese Art der Verteilung ist üblicherweise auch als Block-Zyklische-Verteilung bekannt.

Die zwei Speicherbereiche teilen sich in einen *lokalen* und einen s.g. *halo*-Speicherbereich auf. Der lokale Matrixspeicher eines Prozesses wird ohne nötige Kommunikation zu anderen Prozessoren verwendet. Der halo-Bereich steht ebenfalls jedem Prozess zur Verfügung, muss jedoch abhängig von der mathematischen Operation nach oder vor der Berechnung mit anderen Prozessen

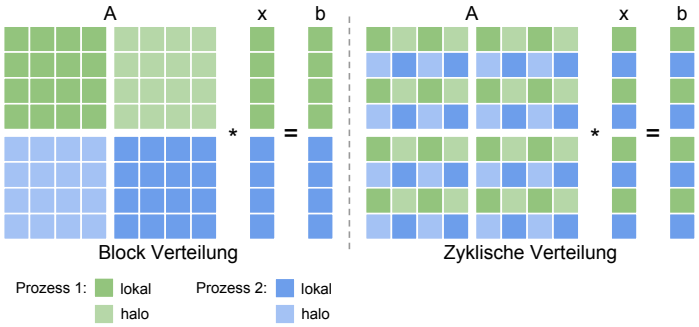

Abbildung 2.4.: Aufteilung eines linearen Gleichungssystems auf zwei Prozessoren bei einer einfachen Block-Verteilung oder einer zyklische Verteilung. Die Matrix erhält in diesem Beispiel die gleiche Verteilung für Zeilen und Spalten.

durch Synchronisierung abgeglichen werden. Wie diese Kommunikation abläuft, wird durch einen Kommunikationsplan in der Klasse *Halo* implementiert. Jede SparseMatrix erhält einen solchen Plan, um mit den anderen Prozessen kommunizieren zu können. Die restlichen Teile der Matrix oder des Vektors sind für den jeweiligen Prozessor irrelevant und werden durch Andere abgedeckt. Ein Kernel deckt mit seinen Berechnungen immer nur die Matrixzeilen ab, die dem Prozess anhand der gewählten Verteilung zugeordnet wurden.

2.1.3. C++-Expressions (Komplexe Ausdrücke)

Damit die Verwendung von Datenstrukturen und Kerneln komfortabler ist, sind die Funktionsaufrufe auch durch C++-Expressions möglich. Diese befinden sich in einer Schicht über den Kerneln und Datenstrukturen. Berechnungen können z.B. durch die Schreibweise in Listing 1 implementiert werden. C++-Expressions wurden eingesetzt, um die Berechnungsaufrufe nicht über direkte Kernel-Funktionsaufrufe zu starten, sondern um diese durch einfache mathematische Aufrufe ausführen zu können. Expressions bilden somit zusammen mit den Datenstrukturen eine benutzerfreundliche Schnittstelle.

```
...
DenseVector<float> a(100, 1.2f);
DenseVector<float> b(100, 0.5f);
a = a + b;
b += a;
...
```

Listing 1: Beispiel für C++-Expressions in der LAMA-Bibliothek.

2.1.4. Löser linearer Gleichungssysteme

Die einfache Schreibweise der Expressions kommt auch bei der Entwicklung des Löser-Frameworks zum Einsatz, welches Lösungsverfahren für lineare Gleichungssysteme bereitstellt. Die implementierten Löser und bilden eine weitere Abstraktionsschicht, die sich über den Expressions einordnet. Alle Löser werden für das direkte oder iterative Lösen eines linearen Gleichungssystem der Form $Ax = b$ eingesetzt. A ist dabei eine Matrix, b die rechte Seite (engl. Right-Hand-Side (rhs)) und x der Lösungsvektor. Derzeit stellt die Bibliothek die folgenden direkten und iterativen Lösungsverfahren für lineare Gleichungssysteme bereit:

- *direkt*
  - Dreieckszerlegung (auch Lower-Upper (LU)-Zerlegung)
  - Inverse Solver (hierbei wird die Matrix durch die Dreieckszerlegung invertiert und die Lösung durch Multiplizieren der Inversen mit dem Rechte-Seite-Vektor direkt berechnet)
- *iterativ*
  - Jacobi Method
  - Conjugate Gradient (CG) Method
  - Successive Over-Relaxation (SOR)
  - AMG Method
  - Generalized Minimal RESidual (GMRES)

Der Unterschied zwischen direkten und iterativen Lösern ist, dass direkte Löser unabhängig von einer Startlösung in einem einzigen Schritt lösen. Iterative Verfahren wie z.B. die Jacobi-Methode berechnen die Lösung schrittweise, wo-

bei ausgehend von einer Startlösung die Genauigkeit iterativ verbessert wird. Als Beispiel werden die mathematischen Details dieses Verfahrens im Anhang in Abschnitt A.2 auf Seite 115 erklärt. Für weitere Informationen zu den restlichen Löser linearer Gleichungssysteme wird auf die Literatur verwiesen [Kanzow, 2004; Saad, 2003; Barrett u. a., 1994; Griebel u. a., 1998; Förster und Kraus, 2011]. In diesem Abschnitt wird der Teil des Löser-Frameworks erläutert, der für die Umsetzung dieser Arbeit nötig ist. Dazu gehören primär die Schnittstellen des Löser-Frameworks.

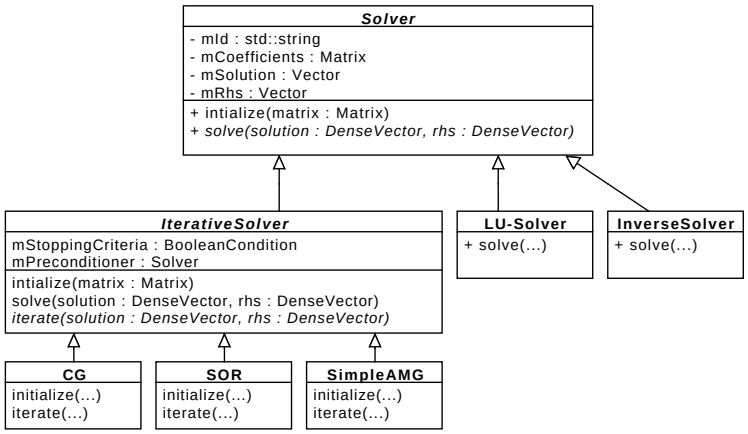

Abbildung 2.5.: Vereinfachtes Klassendiagramm des Löser-Frameworks.

Einen groben Überblick über die Basis des Löser-Frameworks gibt Abbildung 2.5. Die Basisklasse *Solver* gibt vor, dass alle ableitenden Klassen jeweils eine Initialisierungs und eine Lösungsmethode implementieren. Bei der Initialisierung werden alle für die Lösungsphase nötigen Vorbereitungsschritte durchgeführt, die nicht in jeder Iteration wiederholt werden müssen und matrixgebunden sind. Der Lösungsmethode werden dann die zwei noch benötigten Vektoren *solution* und *rhs* übergeben. Die Parameter der Lösungsmethode wurden so gewählt, da es damit möglich ist, unterschiedliche rechte Seiten bei gleichbleibender Matrix für die Berechnungen zu verwenden, ohne dass der Löser erneut mit der Matrix initialisiert werden muss.

Die Klassen *LU-Solver* und *InverseSolver* implementieren direkte Lösungsverfahren und leiten direkt von der Basisklasse *Solver* ab. Sie beinhalten die konkreten Implementierungen der *initialize*- und *solve*-Methoden, welche abhängig von den jeweiligen mathematischen Verfahren sind. Des Weiteren leitet die Klasse *IterativeSolver* von der Basisklasse *Solver* ab. Diese gibt vor, dass alle ableitenden Löser eine *iterate*-Methode implementieren und ist für den Umgang mit den Haltekriterien zuständig, welche nach jeder Iteration abgefragt werden, um festzustellen, ob die errechnete Lösung beispielsweise hinreichend genau ist.

Das Haltekriterium, das dies bestimmt, muss durch den Nutzer des Lösers vorgegeben werden. In dem Framework stehen derzeit die in Abbildung 2.6 aufgeführten Kriterien zur Verfügung.

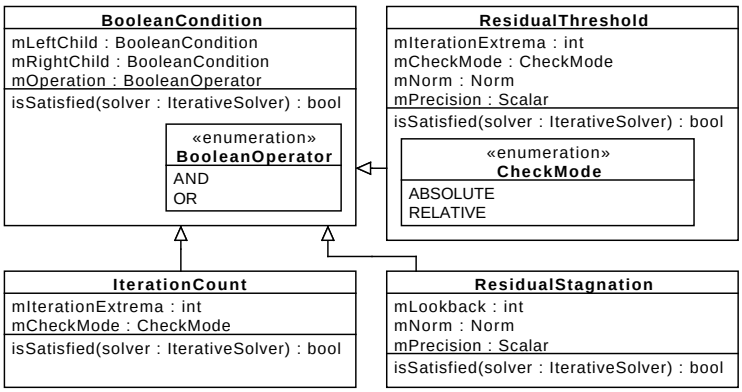

Abbildung 2.6.: Verfügbare Haltekriterien, welche durch logische Verknüpfungen untereinander verbunden werden können.

Die Kriterien können über logische Verknüpfungen miteinander kombiniert werden und anschließend dem iterativen Lösungsverfahren zugewiesen werden. Das Kriterium *IterationCount* begrenzt die maximale Anzahl von Iterationen. *ResidualStagnation* prüft anhand des aktuellen und der vorherigen *Residuen* iterationsweise, ob sich die Lösung noch ausreichend stark verbessert. Das Residuum beschreibt dabei die Abweichung von der optimalen (unbekannten) Lösung und

wird durch $residuum = (b - Ax)$, wobei x hier der Lösungsverktor der aktuellen Iteration ist. Die Angabe einer *Norm* ist dazu nötig, um die Größe des Residuen-Vektors in Form eines Skalars ermitteln zu können. Hierfür stehen die drei in den Klassen L1Norm (L^1 Norm), L2Norm (L^2 Norm) und MaxNorm (Maximumsnorm) implementierten Normen zu Verfügung. Die dritte Kriterienklasse ist *ResidualThreshold*. Mit ihr wird das Residuum nicht bezogen auf die vorherige Iteration verglichen, sondern durch eine konkrete Präzisionsvorgabe für die aktuelle Iteration überprüft.

Die Haltekriterien müssen entsprechend vor dem Aufruf der Lösungsmethode über eine separate Zuweisungsmethode gesetzt werden. Die Lösungsmethode ist öffentlich (`public`) und ruft solange die geschützte (`protected`) Iterationsmethode auf, bis das Haltekriterium erfüllt ist. Alle iterativen Löser, die von der Iterative-Solver-Klasse ableiten, implementieren die Initiailisierungs- und Iterierungsmethoden.

Das Framework ermöglicht es, die Verfahren untereinander zu verknüpfen. Beispielsweise ist es möglich, einen CG mit einem anderen Löser zu präkonditionieren. Die *Präkonditionierung* sorgt für eine bessere Ausgangssituation des CG-Verfahrens und wirkt sich damit auf die Gesamtlaufzeit aus (mehr dazu siehe [Kanzow, 2004, S. 250f] und [Barrett u. a., 1994, S. 35ff]).

Bei Mehrgitterverfahren spielt die Verknüpfbarkeit zu anderen Lösern eine große Rolle. Das AMG-Verfahren wird ähnlich, wie die zuvor vorgestellten iterativen Löser implementiert, jedoch werden für die LAMA-Implementierung gesonderte Setup-Klassen verwendet, die sich um die umfangreiche Initialisierungsphase (Setup) des Lösers kümmern. Die in Abbildung 2.7 gezeigte abstrakte Klasse *AMGSetup* ist die Schnittstelle, von der die konkreten Setup-Implementierungen ableiten. Diese werden aufgrund einer externen Closed-Source-Implementierung dynamisch an das Projekt gelinkt.

Ein AMG-Setup hat folgende Aufgaben:

- Erzeugung der Matrixhierarchie anhand der Eingabematrix und unterschiedlicher Vergröberungsstrategien.

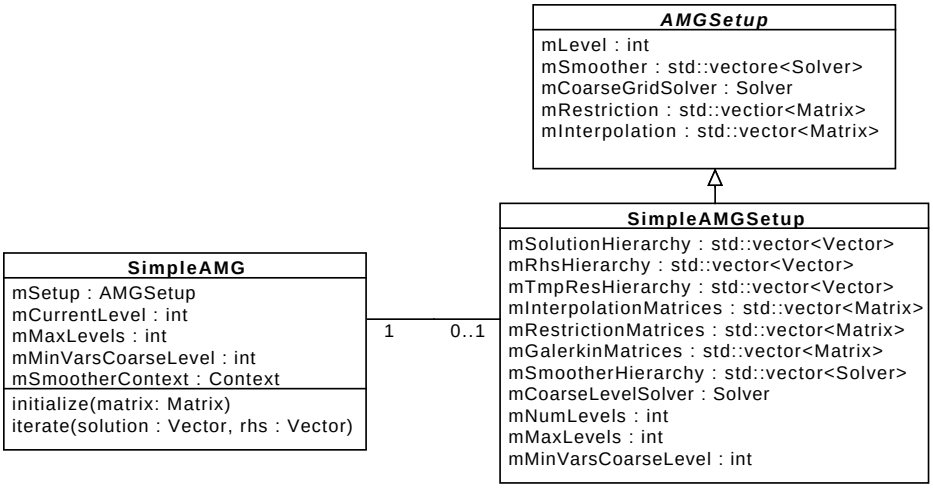

Abbildung 2.7.: Klassenstruktur des AMG-Mehrgitter-Verfahren und die Anbindung des Setups.

- Erzeugung und Konfiguration der Löserinstanzen, die als Grobgitterlöser oder Glätter auf den Matrixebenen (Gittern) eingesetzt werden.
- Initialisierung der erzeugten Löser.

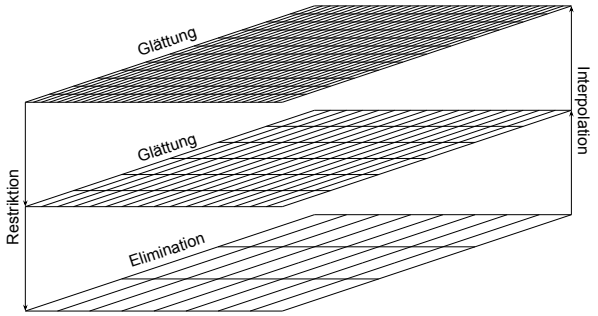

Abbildung 2.8.: Verlauf eines V-Zyklus eines Mehrgitterverfahrens, wie dem AMG.

Die Lösungsphase besteht aus mehreren Zyklus-Durchläufen. Ein Zyklus entspricht dem Durchlauf der gesamten Matrixhierarchie. Diese Zyklen können unterschiedliche Verläufe haben. Der einfachste Verlauf ist ein V-Zyklus, welcher in Abbildung 2.8 dargestellt wird. Hierbei wird ausgehend vom feinsten

Gitter jeweils eine Kombination von Glättung und eine Grobgitterkorrektur durchgeführt. Dieses Vorgehen wird rekursiv auf das jeweilige gröbere Gitter übertragen, bis das gröbste Gitter erreicht wird. Was das gröbste Gitter ist, wird bei der Erzeugung durch zuvor festgelegte Kriterien vorgegeben, wie z.B. die Mindestanzahl Punkte pro Zeile einer Matix. Auf dem gröbsten Gitter wird dann üblicherweise ein direkter Löser verwendet, der das Gleichungssystem exakt löst. Ein V-Zyklus führt zwischen Vor- und Nachglättung jeweils nur einem Grobgitterkorrekturschritt durch. Ein W-Zyklus führt jeweils zwei Korrekturschritte durch, was rekursiv zu einem W-förmigen Verlauf führt. Ein Zyklus-Durchlauf entspricht einer Iteration des AMG-Verfahrens. Die Anzahl der Iteration muss wie bei anderen iterativen Verfahren durch mindestens ein Haltekriterium begrenzt werden.

Für weiterführende Informationen zu dem sehr umfangreichen Verfahren, wird auf die Literatur verwiesen [siehe Griebel u. a., 1998; Förster und Kraus, 2011]. Nachdem nun das LAMA-Projekt grundlegend mit seinen Daten-Containern, Optimierungsansätzen und implementierten Lösungsverfahren vorgestellt wurde, wird in den folgenden Abschnitten auf die Entwicklung von domänenspezifischen Sprachen eingegangen.

2.2. Domänenspezifische Sprachen

Eine Domain Specific Language (DSL) ist eine Sprache, die anders als bei General Purpose Languages (GPLs) wie Java und C++ auf ein bestimmtes Problemfeld – die Domäne – ausgerichtet ist. Der wesentliche Grund für die Verwendung von DSLs, ist die *Abstraktion* des Bestandsprogrammcodes in einer eigenen Sprache, welche sich auf die *Features* (Softwaremerkmale) eines begrenzten Wissen- oder Interessensgebiet (die Domäne) fokussiert. Die Features ergeben sich aus den Anforderungen und sind nach Riebisch ein für den Kunden wertvoller Aspekt [Riebisch, 2003, S. 69]. DSLs erlauben die präzise Beschreibung der Anwendungslogik, wobei die semantische Distanz zwischen Problem und Programm reduziert wird [Bell u. a., 1994, S. 2f].

Ein weiteres Ziel beim Einsatz von DSLs ist die Wiederverwendbarkeit von Code, höhere Wartbarkeit des Codes und die Steigerung der Entwicklungseffizienz [Stahl u. a., 2007, S. 13ff]. Die Menschenlesbarkeit spielt eine wichtige Rolle bei der DSL-Entwicklung ([Ghosh, 2010, S. 11] und [Fowler und Parsons, 2011, S. 27]). Die Sprache soll so entwickelt werden, dass die Nutzer komplexe Strukturen möglichst einfach verstehen, bearbeiten oder entwickeln können.

Bekannte Beispiele für DSLs sind:

Hypertext Markup Language (HTML)
: als DSL, dessen Domäne Web-Anwendungen sind.

Cascading Style Sheets (CSS)
: als deklarative Sprache für Stilvorlagen von anderen strukturierten Sprachen, wie HTML und Extended Markup Language (XML).

Structured Query Language (SQL)
: als Datenbanksprache für Abfragen und das Bearbeiten relationalen Datenbanken.

Very High Speed Integrated Circuit Hardware Description Language (VHDL)
: als Hardwarebeschreibungssprache für die Entwicklung komplexer Schaltkreise bzw. Digital-Systeme.

2.2.1. DSL-getriebene Software-Entwicklung

Die DSL-Entwicklung ist gegenüber der herkömmlichen Software-Entwicklung sehr unterschiedlich. Für eine DSL muss zunächst sehr genau definiert werden, welche Komponenten der Ursprungssoftware zu der Domäne gehören und was der Zusammenhang zwischen ihnen ist [Ghosh, 2010, S. 4]. Da es bei der Domänen-Entwicklung hauptsächlich um die Wiederverwendbarkeit von Domänenwissen geht, wird bereits bei der Ermittlung der Anforderungen darauf geachtet, dass diese nicht zu statisch definiert werden. Stattdessen sollen die Anforderungen konfigurierbar sein, sodass für die Umsetzung ähnlicher Systeme nur geringe Anpassungen nötig sind.

Die DSL-Entwicklung verläuft, wie in Abbildung 2.9 gezeigt, parallel zu der herkömmlichen Entwicklung eines Softwareprodukts. Nach Mernik teilt sich die DSL-Entwicklung in drei Phasen auf [Mernik u. a., 2005, S. 323ff]. Die erste Phase ist die *Analysephase* (auch Domänen-Modellierung) dient der Domänenerfassung und Definition von wiederverwendbaren, konfigurierbaren domänenbezogenen Anforderungen. In der *Entwurfsphase* wird eine allgemeine Architektur für die Zieldomäne entwickelt und in der *Implementierungsphase* wird diese in wiederverwendbare Komponenten umgesetzt.

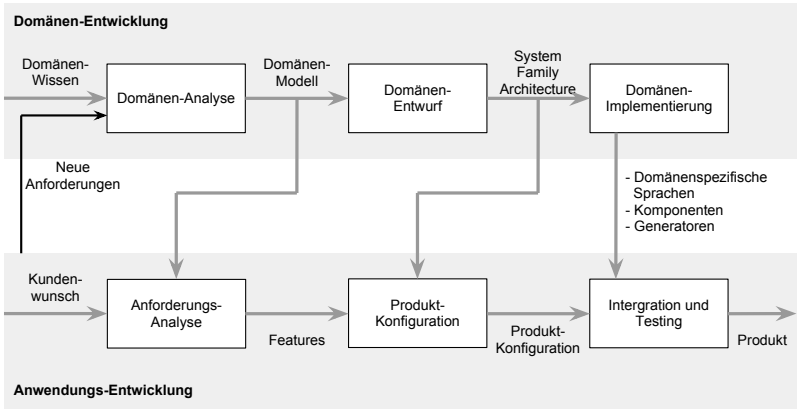

Abbildung 2.9.: Einsatz der DSL-Entwicklung in dem Anwendungsentwicklungsprozess [nach Czarnecki und Eisenecker, 2000, S. 21].

Wie genau DSLs eingesetzt werden und wie die einzelnen Phasen im Detail aussehen, wird in den folgenden Abschnitten beschrieben.

2.2.2. Domänenanalyse

In dieser Domänenanalyse wird analysiert, welche Teile der Software zu der Zieldomäne gehören. Dies kann entweder formell oder informell geschehen. Die informelle Analyse hat zwar einen geringeren Aufwand, jedoch werden oft wichtige Aspekte ungenau definiert, was in der späteren Entwicklungsphase zu Problemen führen kann [Martin-Vide, 2010, S. 424]. Formelle Analyse-Methoden sind beispielsweise Family-Oriented Abstractions, Specifications and Translation (FAST) entwickelt von Weiss und Lai und Feature-Oriented Domain Analy-

sis (FODA) entwickelt von Kang u. a. [Weiss und Lai, 1999; Kang u. a., 1990]. Alle formellen Analysemethoden sammeln Informationen und fassen diese in einem Domänenmodell zusammen, welches dann die Basis für das Domänen-Design bildet.

Neben der Sammlung und Formalisierung in der üblichen Anforderungsanalyse, wird bei der Domänenanalyse versucht weiteres Wissen über die Domäne zu gewinnen und dieses zu kategorisieren. Kategorisiert wird die Domäne nach Variabilitäten und Gemeinsamkeiten. *Variabilitäten* beschreiben genau, welche Informationen benötigt werden, um eine Systeminstanz zu spezifizieren und somit die oben erwähnte Konfigurierbarkeit abdecken. *Gemeinsamkeiten* einer Domäne sind dagegen für eine Familie von Domänen einsetzbar und somit wiederverwendbar. Formalisiert werden die Informationen in dem oben angesprochenen Domänenmodell, welches nach Mernik u. a. und Czarnecki und Eisenecker üblicherweise auch folgende Punkte abdeckt (siehe [Mernik u. a., 2005, S. 324]; [Czarnecki und Eisenecker, 2000, S. 23f]):

Domänen-Definition gibt die Abgrenzung der Domäne zum Rest der Software vor und beschreibt damit, was zur Domäne gehört. Die dazu nötigen Informationen werden den gesammelten Anforderungen entnommen.

Domänen-Terminologie umfasst die Terminologie der Sprache einschließlich des verwendeten Vokabulars. Hinzu kommt die *Ontologie*, durch diese werden Regeln für den Aufbau von Sprachkonstrukten und somit die formalisierte Struktur der Domäne (Metamodell) beschrieben. Die Terminologie aus den gesammelten Variabilitäten abgeleitet.

Featuremodell wird zur Variabilitätsanalyse eingesetzt, um sich ändernde Features in der Domäne zu erfassen. Die Analyse ist frei von Realisierungsaspekten, was die Analyse übersichtlicher hält [Vogel u. a., 2005, S. 300].

Die durch FODA beschriebenen Featuremodelle sind eine gängige Methode, um optionale und erforderliche Features in einer Domäne anhand der gesammelten Anforderungen zu ermitteln. Im Detail besteht ein solches Featuremodell aus [siehe Czarnecki und Eisenecker, 2000, S. 38ff]

Featurediagramme welche eine hierarchische Zerlegung der Features und ihrer

Charakter (erforderlich, alternativ oder optional) darstellen.

Semantik-Definition der Features.

Feature-Gestaltungsregeln die festlegen, welche Feature-Verknüpfungen erlaubt oder verboten sind.

2.2.3. Domänendesign

Die Aufgabe, die die Entwicklungsphase Domänendesign hat, ist die Spezifizierung der Softwarestruktur, die es in der Domänenimplementierung umzusetzen gilt. In der Designphase können die Entwurfsmuster *Spracherfindung* und *Sprachnutzung* verwendet werden. Bei der Spracherfindung wird eine Sprache vollständig neu entworfen. Dagegen wird bei der Sprachnutzung auf bestehende DSLs oder GPLs aufgebaut, um die neue Sprache zu entwickeln. Dabei gibt drei Varianten, eine Bestandssprache zu nutzen. Die erste Möglichkeit ist es, die existierende Sprache durch neue Features zu *erweitern* [Spinellis, 2001, S. 95]. Die zweite Möglichkeit ist es, existierende Features der Sprache weiter einzuschränken und somit die existierende Sprache nur teilweise zu nutzen, was auch als *Piggypack*-Methode bekannt ist [Spinellis, 2001, S. 93f]. Die dritte Möglichkeit ist die *Spezialisierung* der Bestandssprache, wobei diese durch das Entfernen bestehender Features eingeschränkt wird [Spinellis, 2001, S. 95f].

Mit diesen Mustern werden Beziehungen zwischen bestehender Sprache und DSL gefunden. Im Anschluss an die Wahl des Musters wird das Design formell oder informell spezifiziert. Formell geschieht die Syntaxbeschreibung durch reguläre Ausdrücke oder Grammatiken. Attribut-Grammatiken haben dabei den Vorteil, dass sie ebenfalls den semantischen Teil der Sprachbeschreibung abdecken. Dieser Prozess kann automatisiert werden.

2.2.4. Domänenimplementierung

Für die DSL-Entwicklung ist es wichtig, dass der richtige Implementierungsansatz verwendet wird. Diese Entscheidung wirkt sich stark auf den Umfang der

weiteren Entwicklung aus. Die Folgenden Realisierungsmöglichkeiten wurden dem Artikel von Mernik u. a. entnommen [Mernik u. a., 2005, S. 329ff].

Bei dem DSL-*Interpreter-Ansatz* werden DSL-Konstrukte zur Laufzeit erkannt und interpretiert. Dazu wird der Sprachteil zunächst gefetcht (abgerufen), dekodiert und anschließend ausgeführt. Da *Interpreter* zur Laufzeit eingesetzt werden, kann dies je nach Implementierung die Ausführungszeit stark beeinflussen.

DSL-Konstrukte werden bei dem DSL-*Compiler-Ansatz* bzw. Anwendungsgenerator zu Konstrukten oder Bibliotheksaufrufen in die Basissprache übersetzt. Ein *Compiler* interpretiert die DSL-Eingabe, überprüft diese statisch und übersetzt sie in ausführbare Maschinencode.

Bei der Verwendung eines *Präprozessors* werden die Sprachkonstrukte in die Basissprache übersetzt. Hierfür werden beispielsweise Makros in C oder C++ zur Sprachgenerierung definiert oder eine einfache lexikalische Analyse durchgeführt. Bei der *lexikalische Interpretation* werden wird auf Zeichenbasis gearbeitet, ohne eine komplizierte baumartige Syntax-Interpretation.

C++ bietet des Weiteren auch die Möglichkeit der templatebasierten Sprachgenerierung, die in der Meta-Programmierung verwendet wird [Abrahams und Gurtovoy, 2004]. Makros sind im Gegensatz zu C++-Template meist unabhängig von der Syntax der Basissprache und deswegen wird die syntaktische Korrektheit der Makros nicht garantiert. Die Fehler werden jedoch durch den Compiler oder Interpreter der Basissprache gemeldet. Templates erweitern die Basissprache zur Übersetzungszeit, was diese allerdings verlängern kann. Weitere Möglichkeiten für die Spracherweiterung in C++ ist die Überladung von Operatoren, bei der die eigentliche Semantik eines Operators überschrieben wird.

Ein weiterer Ansatz für die Umsetzung einer DSL ist die Verwendung von fertigen Commercial off-the-shelf (COTS) Werkzeugen und/oder Notationen. Hier werden beispielsweise fertige XML-Parser verwendet, die anhand von Domänen-Regeln arbeiten. Diese Regeln (die Grammatik) werden durch XML

Schema Definition (XSD) oder Document Type Definition (DTD) vorgegeben.

Da die Entwicklung eines eigenen Parsers für einen Interpreter oder Compiler sehr aufwändig sein kann und einiges an Wissen verlangt, können Parsergeneratoren eingesetzt werden. Parsergeneratoren arbeiten mit Grammatiken, welche beispielsweise in Backus Naur Form (BNF) definiert sind. Sie erzeugen aus der gegebenen Grammatik den benötigten Parser, welcher dann den in der DSL entwickelten Code-Ausschnitt interpretieren, auf eine GPL abbilden und ausführen kann. Sowohl ein Generator als auch ein Interpreter benötigen einen Parser, um die Abbildung von abstrakter zu konkreter Syntax und umgekehrt durchführen zu können. Ein Beispiel für einen solchen Parsergenerator ANother Tool for Language Recognition (Antlr), der beispielsweise für die Parsergenerierung in Xtext eingesetzt wird [ANTLR, 2012a; Xtext, 2012].

Um diese Ansätze zuordnen zu können, wurden diese in der gängigen Literatur zwei Gruppen von domänenspezifischen Sprachen unterteilt [Stahl u. a., 2007; Ghosh, 2010; Fowler und Parsons, 2011; Czarnecki und Eisenecker, 2000]. Dazu gehören zu einem *externe DSLs*, welche mit einer unabhängigen Implementierung eine eigene Syntax bereitstellen und somit eine hohe Flexibilität in der Spracherzeugung bieten. Externe DSLs erfordern jedoch die Entwicklung eigener Sprachwerzeuge, wie Compiler, Interpreter, Lexer und Parser.

Die zweiter Gruppe bilden *interne bzw. eingebettete DSLs*, welche auf einer existierenden GPL basieren und diese um Sprachkonstrukte erweitern, die den Code leserlicher machen und somit auch leichter wiederverwendbar. Bei dieser Form der DSL-Implementierung kommen Hauptsächlich die zuvor vorgestellten präprozessorbezogenen Ansätze, bei der die DSL-Beschreibung die Basissprache erweitert.

2.3. Gängige Konfigurationssprachen

In diesem Abschnitt werden gängige Konfigurationssprachen bzw. Konfigurationsmöglichkeiten für Löser und deren Funktionsumfang vorgestellt. Der Aufbau

dieser Sprachen wird für die Entwicklung der DSL dieser Arbeit berücksichtigt. Dabei werden Vorteile und Nachteile – in Form von Einschränkungen – der bestehenden Konfigurationssprachen herausgefiltert und für die zu entwickelnde Sprache berücksichtigt. Vorgestellt werden die Konfigurationsmöglichkeiten der Flüssigkeitssimulationssoftware OpenFOAM und Portable, Extensible Toolkit for Scientific Computation (PETSc) [OpenFOAM, 2012a; PETSc, 2012].

2.3.1. OpenFOAM

OpenFOAM benutzt s.g. Wörterbücher (engl. *Dictionaries*) zur Konfiguration der eigenen Löser. Diese Wörterbücher werden verschachtelt definiert und beinhalten zur Konfiguration eine Menge von Schlüssel-Wert-Kombinationen. Listing 2 zeigt eine Beispielkonfiguration zweier Löser, welches der OpenFOAM-Dokumentation entnommen wurde [siehe OpenFOAM, 2012b].

`U` entspricht hier einem Löser, der einer Geschwindigkeitsberechnung zugeordnet wurde, und `p` entspricht einem Löser, der einer Druckberechnung zugewiesen wurde. Diese beiden Löser-Wörterbücher beinhalten die Konfiguration. Mit dem Schlüsselwort `solver` wird der Lösertyp vorgegeben. Als Präkonditionierer wird hier ein von OpenFOAM entwickelter Geometric-Algebraic MultiGrid (GAMG) gewählt, welcher sowohl als algebraisches Mehrgitterverfahren – wie es in der LAMA-Bibliotheken umgesetzt wurde – als auch als geometrische Agglomeration eingesetzt werden kann. Die zweite Variante eignet sich mehr für die Präkonditionierung in diesem Anwendungsbereich [mehr dazu siehe Musy u. a., 2009; OpenFOAM, 2012b].

Die zwei Parameter `tolerance` und `relTol` geben die Genauigkeit der Lösung für iterative Verfahren an. Diese Parameter entsprechen dem Haltekritierum *ResidiualThreshold* des LAMA-Löser-Frameworks, können hier jedoch nicht untereinander verknüpft werden. Der für die Druckgleichung verwendete Präkonditionierer ist ein Incomplete Lower-Upper (ILU)-Verfahren, dass ebenfalls besonders für die Präkonditionierung geeignet ist [mehr dazu siehe Saad, 2003, S. 301ff].

```
solvers
{
  p
  {
    solver          PCG;
    preconditioner  GAMG;
    tolerance       1e-06;
    relTol          0;
  }

  U
  {
    solver          PCG;
    preconditioner  DILU;
    tolerance       1e-05;
    relTol          0;
  }
}
```

Listing 2: Dictionary-basierte Konfiguration der Löser in OpenFOAM.

2.3.2. PETSc

PETSc ist ebenso wie LAMA eine Bibliothek, die Datenstrukturen und Routinen zur Berechnung aus dem Bereich der Numerik anbietet. Optimiert wurden diese Routinen ebenfalls auf Grafikkarten, durch Threadparallelität und verteilt durch Message Passing Interface (MPI).

Um PETSc einzusetzen, muss deshalb zunächst eine Anwendung entwickelt werden, die die Daten für die PETSc-Daten-Container bereitstellt und die entsprechenden Löser aufruft. PETSc bietet drei verschiedene Wege, um die implementierten Löser zu verknüpfen und zu konfigurieren. Der Erste ist die Definition der Löserkonfiguration innerhalb der oben genannten Anwendung, welche somit als Standardwerte vorgegeben werden und bei Änderungen nur über ein erneutes Übersetzen der Anwendung wirksam werden. Zur Laufzeit können Konfigurationen entweder über eine Umgebungsvariable `PETSC_OPTIONS` oder über die Anwendungsaufrufparameter festgelegt werden. Ein Beispielaufruf mit

der Konfiguration über die Parameter wird in Listing 3 gezeigt.

```
./sys_petsc matrix_file
  -ksp_monitor_true_residual  # Logging Lösungsphase
  -ksp_view                # Logging Löserinitialisierung
  -ksp_type gmres          # Äußerer Löser
  -ksp_max_it 2            # Abbruchkriterium Iterationszahl
  -ksp_rtol 1e-7           # Relatives Abbruchkriterium Residuum
  -ksp_atol 1e-16          # Absolutes Abbruchkriterium Residuum
  -pc_type jacobi          # Innere Löser (Präkonditionierer)
  -sub_pc_type ilu         # Innere innere Löser
  -sub_pc_factor_mat_ordering_type rcm  # ILU-spezifischer Param.
  -sub_pc_factor_levels 0  # ILU spezifischer Parameter
```

Listing 3: Beispielaufruf einer Anwendung, die die PETSc-Bibliothek für ihre Berechnungen verwendet.

`sys_petsc` entspricht in diesem Beispiel der bibliotheksnutzenden Anwendung. Diese bekommt eine Matrixdatei aus der die Instanz einer PETSc-Matrix erzeugt und gefüllt wird. Diese wird dann für die folgenden Berechnungen eingesetzt. Die Konfiguration aus Listing 3 wird in der Anwendung eingelesen und auf die Löser angewandt. Die Konfigurationsparameter der PETSc-Bibliothek ähneln, denen in der LAMA-Bibliothek erheblich, jedoch sind die Möglichkeiten für die Löserverknüpfung und die Haltekriteriendefinition eingeschränkt. So können maximal drei Löser durch Präkonditionierung aneinander gekettet werden. Als Abbruchkriterien können nur die drei Parameter maximale Iterationsanzahl, relatives Residuum und absolutes Residuum definiert werden. Wie diese untereinander priorisiert und verknüpft werden, kann nicht beeinflusst werden. Für weitere Informationen zu der PETSc-Bibliothek wird auf die Literatur verwiesen [Balay u. a., 2004].

Im ersten Grundlagenkapitel wurde die LAMA-Bibliothek und dessen Aufteilung in drei Schichten vorgestellt. Die oberste Schicht bildet das Löserframework, welches ein wichtiger Teil dieser Arbeit ist. Die Konfiguration der Löser und deren direkte Abhängigkeiten, wie Logger und Haltekriterien sollen durch ein Steuerungsframework konzeptioniert und implementiert werden. Dazu soll

eine domänenspezifische Sprache entwickelt werden, die für Nutzer leicht zu verstehen und zu schreiben ist. Deshalb wurde in Abschnitt 2.2 ein Überblick über die Vorgehensweise der Domänen-Entwicklung gegeben. Aufbauend auf dieses Wissen wird nach dem beschriebenen Entwicklungsprozesses in den Folgekapiteln vorgegangen.

3. Anforderungs- und Domänenanalyse

In diesem Kapitel werden zunächst alle, für die weitere Entwicklung nötigen, Anforderungen gesammelt und festgehalten. Diese Anforderungen fließen dann in die darauf folgende Domänenanalyse ein. Die Anforderungen ergeben sich aus Befragungen einiger Projektmitglieder und der Analyse des Codes der LAMA-Bibliothek, die zusammenfassend in dem Grundlagenkapitel vorgestellt wurden.

3.1. Anforderungen

In diesem Abschnitt werden die ermittelten Anforderungen aufgelistet. Aufgeteilt werden die Anforderungen in allgemeine Anforderungen, die sich entweder auf die zu entwickelnde Sprache beziehen oder auf den Übersetzungsprozess. Die Anforderungen, die sich auf das Löserframework und dessen Konfigurierbarkeit beziehen, werden getrennt aufgelistet. Zusätzlich gibt es Anforderungen, die das Löserframework nicht nur indirekt betreffen, diese werden in einer dritten und vierten Auflistung beschrieben. Die dritte Auflistung bezieht sich auf alle Anforderungen, die sich auf Matrix-, Vektor-, Verteilungs-, Berechnungsort-Klassen beziehen. In der letzten Auflistung befinden sich, die Anforderungen die sich auf Strategieimplementierungen beziehen.

Allgemeine Anforderungen:

1. Konfigurationen können zur *Laufzeit* der Nutzeranwendung geändert werden, ohne dass diese neu übersetzt werden muss.
2. Erhöhen der Benutzfreundlichkeit durch eine *zielgruppenorientierte Sprache*, die für Ingenieure gut verständlich und einfach umzusetzen ist. Bei

der Syntaxdefinition der zu entwickelnden DSL sollen auch gängige Konfigurationssprachen berücksichtigt werden.

3. Das Steuerungsframework soll für neue Löser möglichst einfach *erweiterbar* sein, ohne dass der Entwickler eines neuen Lösers sich mit den Details der DSL-Entwicklung befassen muss. Parameteränderungen oder -erweiterungen sollen ebenfalls für die DSL leicht durch eine *Schnittstellenbeschreibung* definierbar sein. Die Schnittstellenbeschreibung schafft somit die Abbildung von Sprachkonstrukten auf Funktionsaufrufe, die den eingelesenen Konfigurationsparameter in der erzeugten Instanz setzt.
4. Für jede Konfiguriationsangabe sollen *Standardwerte* vorhanden sein, auf die ggf. zurückgegriffen werden kann, was die Angabe des Parameters optional macht. Diese Standardwerte werden in den folgenden Anforderungslisten zusammen mit den manuell setzbaren Konfigurationsparametern definiert.
5. Bei der Entwicklung der LAMA-Bibliothek wurde besonders darauf geachtet, dass die Bibliothek auf möglichst vielen Plattformen installiert werden kann. Bei der Umsetzung dieser Arbeit gilt es somit auch, bei der Anbindung neuer Bibliotheken *Plattformabhängigkeiten* zu vermeiden. Damit sind Einschränkungen für die Übersetzungs- und Installationsmöglichkeit des LAMA-Projektes bezüglich Betriebssystem- und/oder Architekturwahl gemeint.
6. *Abhängigkeiten* zu weiteren Bibliotheken oder Laufzeitumgebungen sollten bei der Entwicklung möglichst vermieden werden, um den Übersetzungsprozess möglichst einfach und schlank zu halten.
7. Die Entwicklung der Konfigurationssprache soll sich möglichst gut in den aktuellen Entwicklungsprozess integrieren.

Konfigurierbare Parameter aller Löser, die aktuell nach außen durch die LAMA-Bibliothek bereitgestellt werden:

8. Die vorhanden Logger (FileLogger und CommonLogger) sollen für eine Gruppe von Lösern zugeordnet werden können. Alle Löser, die durch das Framework bereitgestellt werden, können einen Logger über einen Logger-Identifikator (ID) zugeordnet bekommen. Standardwert: Kein Logging.

9. Löser sollen untereinander *verknüpft* werden können, z.B. durch Präkonditionierung von iterativen Lösern oder z.B. über Glättervorgaben oder Grobgitterlöser eines Mehrgitterverfahrens, wie dem AMG. Standardwert für den Präkonditionierer: Kein Löser. Standardwert für die Glätter: DefaultJacobi. Standardwert für den Grobgitter-Löser: InverseSolver.
10. Haltekriterien sollen durch den Nutzer definiert und einem iterativen Löser zugeordnet werden. Haltekriterien sind IterationCount, ResidualStagnation, ResidualThreshold. Wie bereits im Grundlagenkapitel Abschnitt 2.1.4 zum Löser-Framework erwähnt, können Haltekriterien logisch verknüpft werden (||,&&). Die logisch verknüpften Haltekriterien sollen durch eine eindeutige ID innerhalb einer Konfiguration wiederverwendbar sein. Standardwert: IterationCount mit einer Iteration und ohne einer logischen Verknüpfung zu einem weiteren Kriterium.
11. Die Konfigurierbarkeit des AMG soll geschaffen werden. Dazu gehört: Die Glätter sollen für alle Gitter definiert werden können. Ein Grobgitterlöser soll je AMG-Instanz definiert werden können. Kriterien für das AMG-Setup, wie die Anzahl der Level, die maximal durchlaufen werden, bis der Grobgitterlöser aufgerufen wird, oder die Mindestanzahl von Punkten einer Matrix, sollen gesetzt werden können.
12. Der Omega-Wert soll in Form eines Skalars gesetzt werden können. Dieser ist für Löser wie die spezialisierten Jacobi-Verfahren, das Standard-Jacobi-Verfahren und das SOR-Verfahren relevant. Er gibt vor, wie viel von der Lösung der vorherigen Iteration in die aktuelle Iteration einfließt. Standardwert: 1.0.
13. Das GMRES-Verfahren benötigt eine Angabe für die Dimension des Krylovraums [siehe Saad und Schultz, 1986, S. 1f]. Standardwert: 10.

Anforderungen zur Konfigurierbarkeit Matrix-, Vektor-, Berechnungsort-, Verteilungs-Klassen:

14. Die Verteilung einer Matrix soll vorgegeben werden können. Zur Verfügung stehen die Block-Verteilung, die zyklische Verteilung, die generelle Block-Verteilung, die generelle Verteilung und keine Verteilung (rein lokal). Standardwert: keine Verteilung.

15. Der Berechnungsort soll für eine Matrix vorgegeben werden können. Zur Verfügung stehen die CPU und ggf. die GPU mit CUDA-Implementierungen. Standardwert: CPU also nur durch OpenMP optimierter Code.
16. Der Typ des Speicherbereichs für den Halo-Matrixanteil und den lokalen Matrixanteil sollen vorgegeben können. Zur Auswahl sollen alle durch die Bibliothek zur Verfügung gestellten Speicherberichtypen (CSR, ELL, DIA, JDS …) stehen. Standardwert: CSR für den lokalen Anteil und den Halo-Anteil.

Durch die Möglichkeit der Einbindung von *Strategien* sollen dem Anwender Entscheidungen abgenommen werden, die er durch z.B. fehlendes Wissen über die gegebene Architektur, Löser oder weiterer beliebiger Kriterien nicht selbst treffen kann. Strategien sollen für Matrix- und Löserwerte setzbar sein. Dabei sollen folgende Parameter durch eine Strategie konfiguriert werden können:

17. Eine Strategie für Berechnungsorte kann dem Nutzer beispielsweise anhand der gegebenen Matrixgröße die Entscheidung abnehmen, ob sich die Auslagerung der Berechnung auf die Grafikkarte lohnt.
18. Strategien für Matrix- bzw. Speicherbereichtypen können ebenfalls abhängig von z.B. der Matrixdichte und Struktur oder dem Berechnungsort entscheiden, welcher Speicherbereichtyp sinnvoll ist.
19. Strategien für Verteilungen können Matrixbezogen sinnvoll sein, um Umverteilungen der Matrix anhand ihrer Größe durchzuführen.
20. Strategien für Haltekriterien sind sinnvoll, wenn der Nutzer nicht einschätzen kann, ab wann ein Kriterium streng genug ist, um ein geeignetes Ergebnis zu erzielen.

3.2. Analysephase in der DSL-Entwicklung

Im Folgenden wird anhand der gesammelten Anforderungen ein Domänen-Modell aufgebaut und die damit verbundenen Aufgaben schrittweise abgearbeitet. Um eine domänenspezifische Sprache zu entwickeln, muss zunächst der Umfang der Domäne genau spezifiziert werden.

3.2.1. Domänen-Definition

Die erste Domäne – das Löser-Framework – beinhaltet alle direkten oder iterativen Löser, deren Schnittstellen, Haltekriterien und Löser-Logger. Es geht jedoch nicht um dessen Funktionalität, sondern um den konfigurierbaren Teil der Software-Komponenten. Diese Abgrenzung ergibt sich aus den Anforderungen 8 bis 13.

Alle Vektor- und Matrix-Konfigurationen werden in einer eigenen Domäne untergebracht. Zu dieser Domäne gehören auch der Berechnungsort (Anforderung 15), die Speicherbereiche (Anforderung 16) und die Verteilungen (Anforderung 14), da diese nur durch einen Löser überschrieben werden können, jedoch ursprünglich den Daten-Containern zugeordnet sind. Diese Trennung in zwei Domänen wird auf Seite 34 genauer begründet. Dort wird diese auch genauer spezifiziert.

3.2.2. Featuremodell

In diesem Abschnitt werden die Features der Domänen, die sich aus den o.g. Anforderungen ergeben, vorgestellt. Um diese zu unterteilen und deren Abhängigkeiten zu klären, werden sie in geeigneten Diagrammen dargestellt. Hierfür werden die in den Grundlagen (Abschnitt 2.2.2, S. 16) vorgestellten Techniken angewandt. Dafür wird die FODA-Analysemethode eingesetzt, welche sich in der gängigen Literatur wiederfindet [siehe Czarnecki und Eisenecker, 2000; Stahl u. a., 2007; Riebisch, 2003] und eine gut zu verstehende grafische Darstellung durch Featurediagramme bietet.

Da es sich bei LAMA um eine Software-Bibliothek handelt und es kein direkt durch den Nutzer zu verwendendes Produkt ist, wird der Entwickler, der die Bibliothek nutzt, als Kunde bzw. Nutzer betrachtet. Die für ihn relevanten Features werden in den folgenden Diagrammen dargestellt. Featurediagramme werden üblicherweise als zusammenhängendes domänenbezogones Diagramm präsentiert. In diesem Abschnitt wird jedoch schrittweise vorgegangen und es

werden immer nur Ausschnitte des vollständigen Featurediagramms betrachtet. Die Zusammenhänge zwischen den Abbildungen werden erläutert.

Ein Featuremodell wird verwendet, um eine Hierarchie von Features zu bilden und somit deren Zusammenhänge zu klären. Features können dabei Subfeatures haben, welche ggf. in erweiterten Versionen von Featuremodellen Multiplizitäten besitzen können, wie sie aus der Unified Modelling Language (UML) bekannt sind [Riebisch, 2003, S. 67]. In einem Featurediagramm wird das Modell grafisch festgehalten. Jedes Feature kann als

erforderlich ausgefüllter Kringel,
optional leerer Kringel,
alternativ bzw. 1-aus-n nicht ausgefüllter Bogen zwischen den Assoziationen oder
OR bzw. n-aus-m ausgefüllter Bogen zwischen den Assoziationen

angegeben werden.

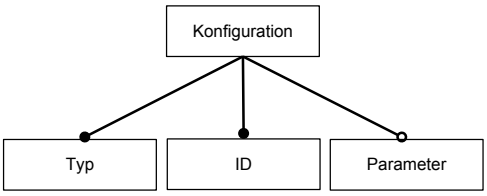

Abbildung 3.1.: Eine Konfiguration besteht aus drei grundlegenden Teilen.

Das Wurzelfeature ist eine Konfiguration, die in Abbildung 3.1 dargestellt wird. Eine Konfiguration besteht aus einem erforderlichen Typ, einer erforderlichen ID und optionalen Parametern. Mit Typen sind Bezeichner von Lösern, Loggern oder Haltekriterien gemeint. Für Löser sind dies beispielsweise CG oder SOR, für Logger FileLogger oder CommonLogger und für Haltekriterien IterationCount, ResidualStagnation, ResidualThreshold. Typen für abstrakte Klassen, wie z.B. die IterativeSolver-Klasse sind ausgeschlossen. Abbildung 3.2 zeigt das passende Featurediagramm zu einigen Typen. Der nicht ausgefüllte Bogen steht für die alternative Assoziation bzw. die oben genannte 1-aus-n Beziehung zwischen dem Feature *Typ* und dessen Subfeatures. Die Angabe des Identifikators

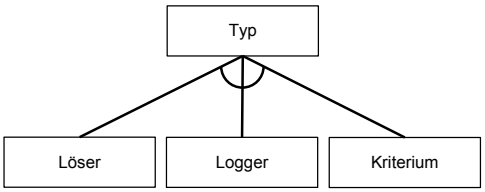

Abbildung 3.2.: Auswahl der Typen für Löser, Logger und Haltekriterium.

ist nicht optional, da jeder Löser eine eindeutige ID benötigt, um beispielsweise für das Logging der Löserinstanzen oder Fehlermeldungen ein Unterscheidungsmerkmal zu haben. Nach der Angabe des Typen und dessen ID werden die Parameterangaben festgelegt. Diese sind optional anzugeben, um Anforderung (4) zu erfüllen. Welche Parameter konkret eingelesen werden sollen, ist abhängig vom eingelesenen Typen, da jeder Typ (Logger, Löser oder Kriterium) unterschiedliche Konfigurationsmöglichkeiten hat. Was konfigurierbare

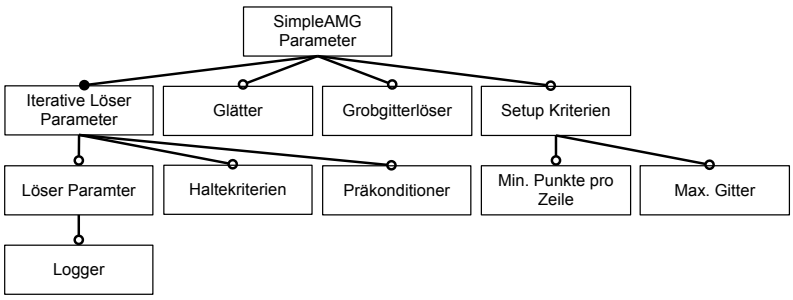

Abbildung 3.3.: Parameter des SimpleAMG als Beispiel.

Parameter sind, wird an dem Beispiel des SimleAMGs in Abbildung 3.3 gezeigt. Die Parameter setzen sich entsprechend der Klassenhierarchie des Löserframeworks zusammen. Im Fall des SimpleAMGs also auch aus den Parametern der IterativeSolver- und der Solver-Klasse. In der abstrakten Klasse Solver lässt sich nur der Logger des Lösers konfigurieren. Auf die Features der Logger wird in einem späteren Abschnitt eingegangen. Für die iterativen Löser können, ein anderes Verfahren, das als Präkonditionierer (Anforderung 9) dient oder Haltekriterien (Anforderung 10) definiert werden. Die Anforderung 11, in der die Konfigurierbarkeit der AMG-Implementierung gefordert wird, wird durch die

restlichen Subfeatures, die in dem Diagramm gezeigt werden, abgedeckt.

Das Featurediagramm in Abbildung 3.4 zeigt den Aufbau von Haltekriterien. Ein Haltekriterium kann sich – wie in den Grundlagen beschrieben – aus Iterationsbegrenzungen, Residuumsbegrenzungen oder Stagnationsangaben für das Residuum zusammensetzen. Kriterien können durch Ausdrücke verknüpft werden. Ausdrücke entstehen durch logische Verknüpfungsoperatoren und Kriterien als Operanden.

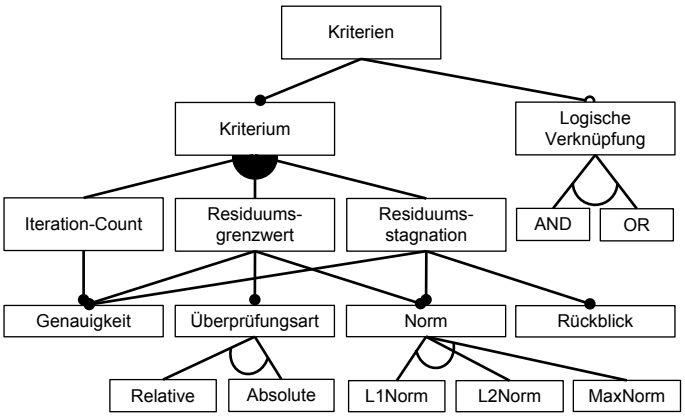

Abbildung 3.4.: Haltekriterien und deren logische Verknüpfung.

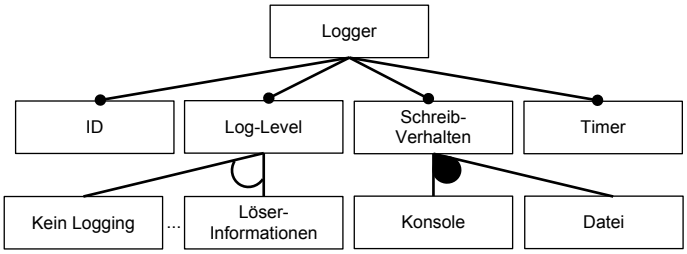

Abbildung 3.5.: Logger, die verschiedenen Lösern zugeordnet werden können.

In Abbildung 3.5 wird das Featurediagramm zum Logger dargestellt. Informationen, die der Logger liefert, sind beispielsweise Laufzeit einer Iteration und das der Iteration zugehörige Residuum. Ein Logger kann einem oder mehreren Lösern zugeordnet werden, um dem Nutzer über den aktuellen Berechnungsverlauf zu informieren. Jeder Logger benötigt dafür eine ID, die bei der Ausgabe

mit ausgegeben wird, um die Informationen voneinander trennen zu können. Neben der ID benötigt der Logger auch einen Parameter, der angibt, in welchem Detailgrad (Log-Level) die Löserinformationen ausgegeben werden sollen. Diese werden entsprechend durch die Ausgabeaufrufe in den Lösern priorisiert. Die Logger-Ausgabe kann wahlweise einfach auf der Konsole geschehen und/oder in eine Datei geschrieben werden. Damit ein Logger Zeitangaben ausgeben kann, werden Timer benötigt, die falls der oder die Löser etwas berechnet, aktiviert werden und die Zeitspanne der Berechnung erfassen.

Nachdem nun ein Überblick über die Features der Löserframework-Domäne gegeben wurde, werden in diesem Abschnitt die matrix- und vektorbezogenen Features vorgestellt, mit denen deren Domäne ebenfalls abgegrenzt wird. Zunächst wird genauer begründet, warum die Bildung einer weiteren Domäne sinnvoll ist.

Verteilungen, Berechnungsorte und Matrixtypen sollen, wie in den Anforderungen beschrieben, ebenfalls konfiguriert werden können. Sollten die Löser diese Konfiguration übernehmen, müssten sie die Daten-Container ggf. umverteilen, auf den oder vom Grafikspeicher kopieren oder möglicherweise neu erzeugen, um in einen neuen Speichertypen umgewandelt zu werden. Da Matrizen und Vektoren unabhängig von Lösern instanziiert und nicht zwangsweise zusammen mit ihnen verwendet werden, sollten sie unabhängig konfiguriert werden. Die Daten-Container sollten zudem erst dann ihre Daten erhalten, wenn sie bereits konfiguriert wurden, da sonst auch hier ein Überschreiben der alten Werte nötig ist und daraus ein erhöhter Rechen- und Datentransferaufwand entstehen kann.

Verteilungen oder Berechnungsorte sind direkt an Matrizen oder Vektoren gebunden und bilden somit gemeinsame Features dieser LAMA-Daten-Container. Ein Problem, das in diesem Zusammenhang mit der LAMA-Bibliothek besteht, ist, dass nicht klar ist welcher Daten-Container, welche Verteilung oder welchen Berechnungsort bekommt, falls diese durch Expressions miteinander verbunden werden. Expressions vereinfachen zwar die Definition von mathematischen Aus-

drücken, jedoch ist es nicht möglich, Verteilungen oder Berechnungsorte einer Expression direkt zuzuordnen. Es ist also nicht klar definiert, welcher Berechnungsort oder welche Verteilung verwendet werden soll, wenn diese unterschiedlich den Operanden einer Expression zugewiesen wurden. Wenn $a = b + a$ für die Vektoren a und b aufgerufen wurde und a eine andere Verteilung als b hat, so sollte in diesem Fall die Verteilung von b an die Verteilung von a angepasst werden. Bei einem Ausdruck wie $a = A * b$ sollten die Matrix A und die rechte Seite b vor der Berechnung wie a umverteilt werden, da a hier die Zielverteilung vorgibt, mit der weiter gerechnet werden soll. Allerdings bedeutet dies, dass bei diesem Ausdruck die beteiligte Matrix A vollständig umverteilt werden muss, was einen hohen Aufwand mit sich bringt. Der Nutzer sollte sich deshalb immer vor der Verwendung von Expressions Gedanken darüber machen, welche Verteilungen er haben möchte, und die beteiligten Vektoren und Matrizen zuvor entsprechend selbst umverteilen. Tut er dies nicht, müssen die in der Bibliothek implementierten Expressions-Rules die Entscheidung übernehmen.

Für die Löser gilt dieses Problem ebenfalls, da diese für die internen Berechnungen ebenfalls Expressions verwenden. Jeder Löser benötigt für seine Berechnungen eine Matrix und zwei Vektoren, deren Verteilungen aufeinander abgestimmt werden müssen. Stellt ein Löser fest, dass Unterschiede in den Verteilungen oder den Berechnungsorten bestehen, überschreibt der Löser diese Werte immer mit denen der Matrix, die in der Initialisierungsphase angegeben wurden.

Das Featurediagramm in Abbildung 3.6 zeigt die Features, die mit den Hauptfeatures Vektor und Matrix zusammenhängen. Beide Daten-Container haben, wie bereits erwähnt, die Gemeinsamkeiten Berechnungsort und Verteilung. Beide Daten-Container haben jedoch mit ihren Speichern ihre eigenen Features und müssen für die Implementierung getrennt betrachtet werden.

LAMA-Matrizen und Vektoren bekommen ihre Daten durch den LAMA-Matrix-Creator oder durch die Nutzersoftware. Um eine Matrix zu erzeugen, benötigt der Konstruktor lediglich die Angabe der Matrixgröße. Diese ist besonders wichtig, um die matrixbezogene Verteilung erzeugen zu können. Die Verteilungs- und die Berechnungsortvorgabe sind wiederum die Voraussetzungen, um die

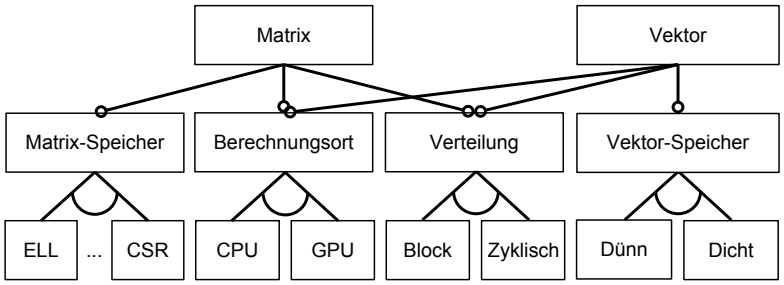

Abbildung 3.6.: Die Matrix-Vektor-Domäne und dessen Features.

jeweiligen Speicherbereiche (Storages) zu erstellen.

Aktuell wird für den Vektor nur ein dichtbesetzer Speicher zur Verfügung gestellt. Die Funktionalität des dünnbesetzten Vektors soll aber in einer der kommenden LAMA-Versionen integriert werden. Deshalb wird die Unterscheidung der Vektorspeicherbereiche hier auch schon berücksichtigt.

Da es nun zwei Domänen gibt, werden die in den Anforderungen geforderten Strategien abhängig von ihrer Funktionalität der jeweiligen Domäne zugeordnet. Die verfügbaren Strategien werden in Abbildung 3.7 zusammengefasst. Hier können keine konkreten Angaben zu deren Parametervorgaben gemacht werden, da dies nicht Teil dieser Arbeit ist und hierfür weitere umfangreiche Untersuchungen nötig sind.

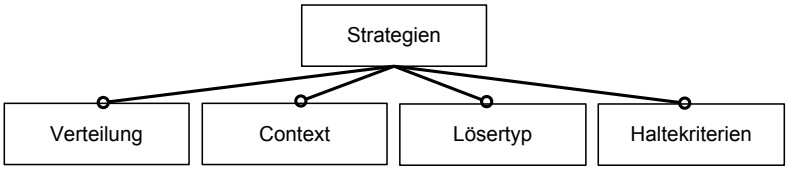

Abbildung 3.7.: Konfigurationswerte, für die Strategien herangezogen werden können.

Die Haltekriterien sind direkt an die iterativen Löser gebunden und werden somit der Löserdomäne zugeordnet. Die Matrix-Vektor-Domäne bekommt die restlichen Strategietypen Berechnungsort, Verteilung und Speicherbereich zu-

geordnet.

Alle noch fehlenden Featurediagramme befinden sich im Anhang A.3 ab Seite 116. Hierzu gehören im Wesentlichen nur die Feature-Diagramme, die zu den Konfigurationen der einzelnen LAMA-Löser passen.

In diesem Kapitel wurden die Anforderungen zu dieser Arbeit gesammelt, indem Mitarbeiter befragt wurden und der Bestandscode, die bestehende Dokumentation und Artikel untersucht wurden. Mit der gewählten FODA-Analysemethode wurden die Grenzen und Zusammenhänge der zwei ermittelten Domänen beschrieben.

4. Auswahl eines Implementierungsansatzes für DSLs

In diesem Kapitel werden anhand der gesammelten Anforderungen die Kriterien erstellt, mit Hilfe derer eine geeignete Umsetzungsmöglichkeit für die Konfigurationssprache gefunden werden soll. Mit den Kriterien und den dazugehörigen Gewichtungen werden die einzelnen Umsetzungsmöglichkeiten bewertet und eine Auswahl getroffen.

4.1. Kriterien für die Auswahl einer Umsetzungsmethode

In der folgenden Auflistung werden die aus den Anforderungen ermittelten Kriterien für die Wahl einer geeigneten Realisierungsmöglichkeit festgelegt:

1. Für die *Erweiterbarkeit* der Sprache ist es wichtig, dass die Sprache einen modularen Aufbau hat und somit leicht neue Sprachelemente hinzugefügt und wiederverwendet werden können. Dieses Kriterium ist auch wichtig, um zu bewerten, wie gut sich eine Umsetzungsmöglichkeit in den aktuellen Entwicklungsprozess integrieren lässt (aus Anforderung 3).
2. Da die Sprache auf die Konfiguration von Lösern ausgerichtet ist und Sprachkonstrukte und Formatierungen aus anderen gängigen Konfigurationssprachen für die Lesbarkeit durch Ingenieure mit in die Zielsprache einfließen sollen, ist eine hohe *Anpassbarkeit* von großer Bedeutung (aus Anforderung 2).
3. Die Sprache sollte *Ausdrücke* einfach und verständlich einlesen können, um beispielsweise logische Verknüpfungen zwischen den Haltekriterien zu unterstützen (aus Anforderung 10).

4. Bei der Entwicklung der LAMA-Bibliothek wurde besonders darauf geachtet, dass die Bibliothek auf möglichst vielen Plattform installiert werden kann und somit bei der Anbindung neuer Bibliotheken *Plattformabhängigkeiten* vermieden werden sollten (aus Anforderung 5).
5. Des Weiteren ist es für das Projekt wichtig, dass die Anzahl der neu hinzukommenden *Abhängigkeiten* zu anderen Bibliotheken möglichst gering gehalten wird (aus Anforderung 6).
6. Werden bei der Umsetzungsmöglichkeit zusätzliche *Werkzeuge* z.B. zur Editorgenerierung mitgeliefert, welche wie bei xText automatische Vervollständigung der Sprachkonstrukte bieten, soll dies auch in die Auswahl mit einbezogen werden [Xtext, 2012].
7. Wichtig für die Auswahl sind auch verfügbare *Dokumentationen*, Tutorials, Community-Foren und Literatur, die bei der Einarbeitung und Entwicklung weiterhelfen. Hier werden auch Aktualität, Qualität und Umfang mit berücksichtigt.
8. Bei der Entwicklung sollen neue Änderungen und Ergänzungen kontinuierlich getestet werden. Die *Testbarkeit* der Implementierungsmöglichkeit der DSL wird deshalb ebenfalls mit in die Entscheidung einfließen. Neben der Testbarkeit ist für die Entwicklung auch *Debug-Fähigkeit und Fehlerbehandlung* für die Ermittlung von Programmierfehlern und falscher Sprachkonstrukte gewünscht.

Tabelle 4.1 fasst die Kriterien zusammen. Jedes Kriterium erhält dafür eine Maximalpunktzahl von 2 Punkten, um die späteren Ergebnisse miteinander vergleichen zu können.

Jede Umsetzungsmöglichkeit kann somit maximal 16 Punkte erreichen. Für jedes einzelne Kriterium werden 0 Punkte vergeben, wenn das Kriterium nicht erfüllt wurde. Wenn das Kriterium nur teilweise erfüllt wurde, wird ein Punkt vergeben und 2 Punkte, wenn das Kriterium vollständig erfüllt wurde. Unterschiedliche Gewichtungen der Kriterien wurden hier nicht berücksichtigt, da die Kriterien keine großen Unterschiede in ihrer Wichtigkeit für das LAMA-Projekt haben.

Kriterium	Punkte
1. Erweiterbarkeit	2
2. Anpassbarkeit	2
3. Ausdrücke	2
4. Plattformunabhängig	2
5. Abhängigkeit zu anderen Bibliotheken	2
6. Dokumentation	2
7. Testbarkeit + Debugging + Fehlerbehandlung	2
8. Zusätzliche Werkzeuge	2
Summe	**16**

Tabelle 4.1.: Gewichtung der Auswahlkriterien.

4.2. Umsetzungsmöglichkeiten für eine DSL

In diesem Abschnitt sollen nun einige Umsetzungsmöglichkeiten für DSLs vorgestellt und anhand der gegebenen Kriterien bewertet werden, um am Ende des Abschnittes eine Auswahl für den weiteren Entwicklungsprozess zu finden.

Durch die Anforderungen wird gefordert, dass Konfigurationen zur Laufzeit der Nutzersoftware eingelesen werden können, um Änderungen an der Konfiguration vornehmen zu können. Dies begrenzt die Auswahlmöglichkeiten bereits. Da solche Anforderungen bereits in der Vorauswahl berücksichtigt werden, tauchen diese nicht in den Kriterien auf.

Für die Vorauswahl werden unterschiedliche Ansätze verglichen, welche den in den Grundlagen Abschnitt 2.2.4 auf Seite 18 vorgestellten Realisierungsmöglichkeiten zugeordnet werden. Da *Parsergeneratoren* den Implementierungsaufwand durch Beschreibung des Parsers durch Grammatikbeschreibungen deutlich reduzieren, wird hierfür der bekannte objektorientierte Parsergenerator Antlr untersucht, der auch C++-Parser generieren kann. Boost.Spirit wird als *Domain Specific Embedded Language (DSEL)* implementiert, wozu Techniken der Template-Meta-Programmierung und der Expression-Templates eingesetzt wurden. Die Grammatik wird dabei direkt im C++-Code der Zielanwendung eingebettet. Neben diesem Ansatz wird auch ein *COTS*-Ansatz hinzugezogen, bei dem der Implementierungsaufwand durch fertige Werkzeuge gering gehalten

werden kann. Hierfür werden fertige XML-Parser vorgestellt, welche zusammen mit XSD oder DTD eigene Sprachen einlesen können.

4.2.1. Boost.Spirit

Boost.Spirit ist eine C++-Bibliothek, die Komponenten zur Parsergenerierung durch den C++-Compiler bereitstellt. Dies geschieht durch den Einsatz von Templates und Operatorüberladungen, die in der Template-Meta-Programmierung eingesetzt werden. Boost.Spirit baut auf einer fertigen Boost-Meta-Programmbibliothek auf. Spirit setzt diese so ein, dass dem Anwender die Möglichkeit geboten wird, die Grammatikbeschreibung, der zu entwickelnden DSL direkt in seinen Code zu integrieren.

Boost.Spirit teilt sich in drei wesentliche Bestandteile auf:

Spirit.Qi erlaubt dem Nutzer der Bibliothek Parser durch Grammatikbeschreibungen zu erstellen, welche rekursiv arbeiten. Qi übernimmt hiermit die Aufgabe des Interpreters.

Spirit.Lex ist der Teil der Bibliothek, der auf Zeichenbasis arbeitet. Lex erlaubt die Definition von lexikalischen Scannern (kurz Lexern). Lex arbeitet nicht rekursiv und erlaubt auf Zeichenbasis das Zurückspringen in den DSL-Konstrukten. Wie die Konstrukte zerlegt werden, um sie zu interpretieren, wird durch reguläre Ausdrücke beschrieben.

Spirit.Karma ist für den generativen Teil der Boost.Spirit-Bibliothek zuständig. Karma arbeitet ebenfalls – wie Qi – mit einer im Code eingebetteten Grammatikbeschreibung und definiert damit, wie das Ausgabeformat passend zu einer Datenstruktur auszusehen hat. Karma erzeugt deshalb domänenspezifische Sprachkonstrukte anhand einer gegebenen Datenstruktur.

Für die Erzeugung eines Parsers verwendet Boost.Spirit Grammatikbeschreibungen, welche durch die oben bereits erwähnten Techniken, wie Operatorüberladung und Funktionsobjekte in der Gastgebersprache (C++) definiert werden können. Die Beschreibung in C++ ähnelt dabei der Notation einer Ex-

tended Backus Naur Form (EBNF)-Grammatikbeschreibung [mehr dazu siehe ISO/IEC JTC 1, 1996]. Genauer gesagt hält Boost.Spirit sich mehr an eine Ableitung der EBNF mit dem Namen Parsing Expression Grammar (PEG), da diese sich leichter auf rekursiv absteigende Parser abbilden lässt. Mit PEGs können kontextfreie Grammatiken in linearer Zeit geparst werden [Mizushima u. a., 2010; Ford, 2004]. Es wurde PEG verwendet, weil bei EBNF mehrere Lösungen für eine Eingabe zulässt [de Guzman und Kaiser, 2011, S. 25].

Qi verwendet s.g. synthetische Attribute, was bedeutet, dass es zu jedem Datentypen, wie z.B. einem `int` einen passenden Attributtypen, beschrieben durch `int_`, gibt. Soll eine Sequenz von Attributen in Qi einlesen werden, so werden die Attribute, wie bei `std::string` durch `>>` verbunden.

Für das Verschachteln von Grammatikbeschreibungen, werden Regeln (implementiert durch `qi::rules`) eingesetzt, um Nicht-Terminale zu definieren, denen diese Grammtikstücke zugeordnet werden können. So kann eine konkrete Grammatik aus einer Zusammenstellung von vielen Regeln bestehen, denen zusätzlich Multiplizitäten zugewiesen werden können. Regeln können deshalb als Subgrammatiken betrachtet werden, die eigene Eingabewerte haben können und ggf. einen Rückgabewert liefern. Ein kleines Beispiel für eine Grammatik wird in Listing 4 gezeigt.

```
1  template <typename Iterator>
2  struct WortListe : qi::grammar<Iterator, string(), ascii::space>
3  {
4      WortListe() : WortListe::base_type(start)
5      {
6          start = *wort           [ push_back( _val, _1 ) ];
7          wort = +char_("a-zA-Z") [ _val = _1 ];
8      }
9      // Regel Deklarationen
10     rule<Iterator, std::vector<string>(), ascii::space> start;
11     rule<Iterator, string(), ascii::space> wort;
12 };
```

Listing 4: Beispielgrammatik mit Verwendung von Regeln.

Das `struct` mit dem Namen *WortListe* repräsentiert die Grammatik und bildet

die Basis für die Parserbeschreibung. `grammar` ist eine in Qi definierte Schnittstelle, von der die eigene Grammatik ableiten muss. Der Templateparameter `Iterator` wird von Qi benötigt, um den Basiseingabetypen während des Parsens durchlaufen zu können. Üblicherweise wird ein Stringiterator verwendet.

Innerhalb der Grammatikstruktur werden die zwei Regeln `start` und `wort` definiert. Die genaue Beschreibung, was eine Regel einliest und was für eine semantische Aktion darauf folgt, wird in dem Konstruktor von *WortListe* festgelegt. In der Regeldeklaration in Zeile 10 wird als zweites Template-Argument eine Funktionssignatur vorgegeben, welche durch die Regel implementiert wird. Die Regel `start` erzeugt entsprechend einen Standard-Vektor von Strings. Die Regel `start` liest Wörter ein, deren Form durch die Regel `wort` vorgegeben wird (Zeile 6 und 7). Der Sternchen-Operator wird in `qi` und `karma` nicht wie üblich als Zeiger-Dereferenzierung verwendet, sondern als Kleensche-Stern. Somit wird hier die Multiplizität 0…n für die Regel `wort` innerhalb von `start` vorgegeben. Dies geschieht über eine Boost.Spirit-interne Operatorüberladung. `start` wird in Zeile 4 als Startpunkt der Grammatik in der Membervariable `base_type` vorgegeben.

Zu jeder Regel kann eine semantische Aktion definiert werden, welche in den eckigen Klammern hinter der Regeldefinition beschrieben werden kann. Ohne semantische Aktionen können Parser nur auswerten, ob ein gegebener Ausdruck gültig ist oder nicht. Semantische Aktionen definieren, was mit dem eingelesenen Attribut passieren soll. Eine semantische Aktion kann beispielsweise ein konkreter C++ Funktions- oder Methodenaufruf sein, der den durch die Regel gelesenen Wert als Übergabeparameter erhält. In der Regeldefinition von `start` wird mit dem `push_back`-Aufruf dem Vektor ein neues String-Objekt hinzugefügt. `_val` ist ein Platzhalter und repräsentiert die Vektor-Instanz, die durch die Regel zurückgegeben wird. `_1` entspricht der eingelesenen Rückgabe von `wort`, also einem Stringobjekt.

Die Regel `wort` besteht aus mindestens einem Buchstaben. Die Multiplizität gibt das `+`-Zeichen vor und entspricht 1…n. Der Ausdruck `[_val = _1 ]` steht für die Zuweisung des eingelesenen Strings, der durch die Regel repräsentiert

bzw. an die Startregel zurückgegeben wird.

Nachdem nun eine grundlegende Einführung in die Funktionsweise der Bibliothek Boost.Spirit gegeben wurde, wird die Bibliothek nun anhand der Kriterien bewertet.

Der geforderte modulare Aufbau ist durch den Einsatz von Regeln in der PEG-Beschreibung gegeben. Des Weiteren lässt sich Boost.Spirit gut in den Entwicklungsprozess integrieren, da keine externen Aufrufe für die Parsergenerierung nötig sind, sondern diese zur Übersetzungszeit geschehen und die Grammatikbeschreibung in das vorhandene Softwarepaket eingebettet werden kann. Damit wird das Kriterium Erweiterbarkeit voll erfüllt.

Durch die bestehenden Lexer-Module wie `char_("a-zA-Z0-9")` und die Beschreibung in einem EBNF ähnlichen Stil der Grammatikbeschreibung, welcher direkt in C++ durch Operatorüberladungen durch die Bibliothek integriert wird, ist die Erzeugung von äußerst komplexen, aber auch auf den Anwender ausgerichteten Sprachen möglich. Damit werden die Kriterien Anpassbarkeit und Ausdrücke voll erfüllt.

Die Boost-Bibliothek wird bereits in LAMA verwendet, um beispielsweise Aufrufparameter oder verschiedene intelligente Zeiger mit automatischer Speicherbereinigung (engl. Garbage-Collection) zu nutzen. Die für die LAMA-Bibliothek geschriebenen Testes arbeiten ebenfalls mit einer Boost-Bibliothek. Um die Komponenten der Boost.Spirit-Bibliothek nutzen zu können, werden lediglich einige Header-Dateien aus der Standard-Boost-Bibliothek benötigt, ohne zusätzliche Bibliotheken linken zu müssen. Deshalb entstehen durch die Integration von Boost.Spirit keine Einschränkungen bezüglich der Plattformabhängigkeit für die LAMA-Bibliothek.

Boost.Spirit ist zwar Bestandteil der Standard-Boost-Bibliothek, jedoch ist es inkompatibel zu alten Boost.Spirit-Versionen, da sich grundlegende Konzepte geändert haben [de Guzman und Kaiser, 2011, S. 7ff]. Die Löser dürfen somit keine Abhängigkeiten zu dem Steuerungsframework haben, da auch weiterhin ältere Boost-Versionen unterstützt werden sollen. Das Steuerungsframework

könnte in diesem Fall aus dem Übersetzungsprozess der Bibliothek ausgenommen werden. Da die Option der optionalen Verwendung besteht und somit keine zwangsweise Einschränkung besteht, wird das Kriterium für Abhängigkeiten zu anderen Bibliotheken trotzdem erfüllt.

Zusätzliche Werkzeuge sind nicht bekannt.

Auf der Webseite von Boost.Spirit findet sich eine umfangreiche Dokumentation mit einfachen bis komplexeren Beispielen, einer tiefergehenden API-Referenz und fortgeschrittenen Themen [siehe de Guzman und Kaiser, 2011]. Neben dieser Möglichkeit gibt es auf der Seite aktuelle Blog-Einträge über Erfolgsmethoden (engl. Best-Practices) und Videos zu spiritbezogenen Themen. Für die Dokumentation gibt es deshalb 2 Punkte.

Debugging ist in Boost.Spirit sehr simpel anzuwenden, indem die zu untersuchende Regel der in Qi bereitgestellten Funktion übergeben wird. Während der Ausführung wird dann durch Boost.Spirit ausgegeben, in welcher Regel welcher Input zum Erfolg und welcher zum Misserfolg geführt hat. Jeder Regel sollte zuvor jedoch ein Name zugewiesen werden. Für die Fehlerbehandlung von Eingabefehlern gibt es ebenfalls vorgefertigte Funktionen, welche es ermöglichen, dem Entwickler die Fehlermeldung selbst an seine Anwendung anzupassen und damit dem Anwender bestmögliche Hilfestellung zu geben. Im Fehlerfall können die folgenden Ausgabeinformationen angegeben werden:

1. Bis wohin die Eingabe bereits gelesen wurde.
2. An welcher Stelle der Eingabe der Fehler aufgetreten ist.
3. Welcher Teil der Eingabe noch zu lesen wäre.
4. Welches Sprachkonstrukt eigentlich als Eingabe erwartet wird.

Somit werden für die Boost.Spirit-Bibliothek mit dem Kriterium Debugging und Fehlerbehandlung ebenfalls 2 Punkten vergeben.

Zusammengefasst werden die Punkte in einer zusammenfassenden Tabelle am Ende des Kapitels.

4.2.2. XML + XSD mit einem C++-Parser/Interpreter

XML ist eine weitverbreitete Methode hierarchisch aufgebaute Textdateien zu definieren und ist die Basis vieler bekannter Formate (z.B. Really Simple Syndication (RSS) oder Scalable Vector Graphics (SVG)). Zusammen mit DTD oder XSD kann XML domänenspezifisch definiert werden, da diese für Schema-Definitionenen verwendet werden können und damit auch das benötigte Meta-Modell erstellt wird. Bei XML entsprechen Nicht-Terminale den XML-Elementen – definiert durch die spitzen Klammern – und Terminale entsprechen den konkreten Daten [Mernik u. a., 2005, S. 330].

Für XML steht eine große Auswahl an freien Parsern zur Verfügung, welche für das Einlesen der Sprache verwendet werden können. Beispiele sind Xerces und Libxml2 [Xerces, 2012b; Libxml2, 2012]. Diese unterstützen die APIs Simple API for XML (SAX) (ereignisbasiert) und Document Object Model (DOM) (baumbasiert) und sind weitestgehend plattformunabhängig sowie frei verfügbar. Da es fertige C++-Bibliotheken gibt, ist die Anbindung an die LAMA-Bibliothek nicht weiter einschränkend.

Eine Beispielimplementierung wird für Xerxes in der Dokumentation gezeigt [siehe Xerces, 2012a]. Hierbei wird ein Parser für die ereignisbasierte SAX2 API verwendet. Es wird daher durch den Parser eine Methode aufgerufen, die durch den Nutzer implementiert wird, um den Inhalt des gelesenen Elementes zu behandeln.

Durch die gegebenen Parser, welche sich durch einfache und wenige Aufrufe integrieren lassen, wird das Kriterium Erweiterbarkeit zumindest auf der syntaktischen Ebene voll erfüllt. Die semantische Erweiterbarkeit ist dem Entwickler überlassen. Das Kriterium Erweiterbarkeit wird insgesamt voll erfüllt.

Durch den syntaktischen Aufbau von XML mit zwangsweisem Start- und End-Element, werden Dokumente teilweise sehr groß, verglichen zu anderen Sprachen, wie JavaScript Object Notation (JSON). Dies fällt besonders auf, wenn bei der XML-Beschreibung auf Attribute der Elemente verzichtet wird. Bei der Struktur von XML-Dokumenten wurde auf einen guten Kompromiss zwischen

Menschenlesbarkeit und jedoch nicht besonders gut auf den Nutzer ausrichten, um sie leicht und verständlich zu machen, und der Schreibaufwand bei der Erstellung eines Sprachdokuments ist durch die schließenden Tags unkomfortabel (Anpassbarkeit: 0 Punkte).

```
<Criterion connective="AND">
    <LeftChild>
        <IterationCount extrema = "10">
    </LeftChild>
    <RightChild>
        <ResidualThreshold
            norm = "L2Norm"
            precision = "0.001"
            checkmode = "absolute">
    </RightChild>
</Criterion>
```

Listing 5: Beschreibung von Haltekriterien in XML mit Attributwerten.

Eine logische Und-Verknüpfung der Haltekriterien `IterationCount( 10 )` und `ResidualThreshold(L2Norm, 0.001, Absolute)` könnte wie in Listing 5 mit Attributangaben in der Elementdeklaration beschrieben werden. XML ist deshalb eher für Schlüssel-Wert-Kombinationen geeignet, jedoch nicht für eine kompakte Beschreibung von Ausdrücken. Im Anhang A.4 auf Seite befindet sich ein Beispiel für die XML Schemadefinition, passend zum Haltekriterium ResidualThreshold. Das Kriterium für Ausdrücke (3) wird deshalb nicht erfüllt, da XML mit seiner festen Dokumentstruktur zu unflexibel ist.

Das Kriterium Werkzeuge wird durch den XML-Ansatz voll erfüllt, da es neben den vielen verfügbaren Parsern auch Tools wie Jaxe gibt [Jaxe, 2012]. Jaxe ist ein Editor-Generator der anhand des XML-Schemas einen Editor generiert. Eine umfangreiche Liste von XML-Editoren findet sich in dem EduTechWiki [EduTechWiki, 2012].

Auch das Kriterium Dokumentation wird durch den XML-Ansatz sehr gut erfüllt. Bücher und online verfügbare Literatur gibt es in großem Umfang.

Xerces bietet ebenfalls, die Möglichkeit, Fehler in der XML-Beschreibung durch

selbst definierbare Exception-Handler einfach behandeln zu können und eine gute Fehlerbeschreibung zu liefern [siehe Xerces, 2012a].

4.2.3. ANTLR

Der Parsergenerator Antlr erzeugt TreeParser durch Grammatikbeschreibungen in EBNF. Zuerst werden die durch den Lexer ermittelten Token aus der Eingabe an den Antlr-Parser weitergereicht. Dieser Parser generiert automatisch einen abstrakten Syntaxbaum (engl. Abstract Syntax Tree (AST)) anhand der Grammatikbeschreibung. Zur Laufzeit wird dann ein TreeParser erzeugt, der semantische Aktionen beinhaltet und somit die Schnittstelle zur Zielsprache bildet. Antlr wird in Java entwickelt, es werden aber auch C++ und weitere Sprachen zur Parsergenerierung unterstützt. Für C++ werden nur Header-Dateien erzeugt, sodass keine zusätzliche Bibliothek an LAMA gebunden werden muss.

Beschrieben werden die Sprachen in Grammatiken. Diese bestehen aus Terminalen und Nicht-Terminalen. Terminale, wie beispielsweise Integer, Fließkommazahlen, Zeichen oder Zeichenketten, können durch Nicht-Terminale gruppiert bzw. modularisiert werden, um eine Grammatik logisch zu strukturieren. Nicht-Terminale sind wie in Boost.Spirit durch Grammatikregeln definierbar. Jedes Terminal und Nicht-Terminal kann auch hier Multiplizitäten, wie den Kleenschen-Stern (0…n) und das Pluszeichen (1…n) zugeordnet bekommen. Die Sprachelemente können über logische Operatoren oder Sequenzen miteinander verknüpft werden. In Listing 6 wird ein Beispiel für eine Grammatik-Beschreibung gezeigt.

Terminale werden in einer Antlr-Grammatik-Beschreibung groß geschrieben, da sie den Lexer betreffen, Nicht-Terminale klein, da sie den Parser betreffen. Das Beispiel in Listing 6 zeigt eine zu dem Boost.Spirit-Beispiel (Listing 4, Seite 41) äquivalente Grammatikdefinition. Der Startpunkt ist das Nichtterminal bzw. die Regel `start`. Durch das Terminal `WORT` wird vorgegeben, dass ein Wort mindestens einen Groß- oder Kleinbuchstaben beinhalten muss.

```
grammar WortListe;

options {
  language = Cpp;
}

// Weitere sprachabhängige Optionen wie
// @namespace{ name }
// @include{ ... }

// Parser Rule (Nicht-Terminale)
start returns [ string result ]
    : WORT* { $result += $WORT.text; }
    ;

//\gls{glos:lexer}-Regeln (Terminale)
WORT : ( 'a'..'z' | 'A'..'Z' )+;
```

Listing 6: Beispielgrammatik definiert in Antlr.

Nach der Fertigstellung der oben beschriebenen Grammatik wird durch Antlr eine Parser- und eine Lexer-Klasse erzeugt. Diese könnten dann z.B. durch LAMA eingebunden werden und durch die fertigen Konstruktoren erzeugt werden. Aufgerufen werden Parser und Lexer durch einen Funktionsaufruf. Diese Funktion ist gleichnamig mit der Wurzelregel (im Beispiel `start`).

Ähnlich wie Boost.Spirit kann die Sprachbeschreibung in terminale oder nichtterminale Grammatikregeln eingeteilt werden. Dadurch, dass jedoch externe Tools für Erweiterungen in der Entwicklung nötig sind, wird das Kriterium *Erweiterbarkeit* nur teilweise erfüllt. Die Flexibilität bei der Erzeugung der Sprache ist ebenfalls hoch und somit ist eine auf den Nutzer ausgerichtete Syntax möglich (Anpassbarkeit: 2 Punkte). Die nötigen Ausdrücke, um z.B. Haltekriterien miteinander verknüpfen zu können, werden ebenfalls unterstützt (Ausdrücke: 2 Punkte).

Gebaut wird Antlr mit Maven, einem Build-System für Java-Projekte [Maven, 2012]. Der Antlr Parser-/Lexer-Generator setzt Java als Sprache voraus. Ja-

va schränkt die Plattformabhängigkeit nicht weiter ein, somit gibt es hierfür 2 Punkte. Jedoch sind weitere Abhängigkeiten zu Eclipse und Java für die Erweiterbarkeit der Domäne gegeben (Abhängigkeiten: 0 Punkte).

Ein Antlr-Eclipse-Plugin, welches einen eigenen Editor für Grammatikbeschreibungen bietet, wird auf der Projektseite bereitgestellt. Allerdings hat das Plugin nur einen geringen Mehrwert gegenüber der Boost.Spirit-Grammatikbeschreibung, da Spirit die Auto-Vervollständigung von der verwendeten Entwicklungsumgebung nutzt, dadurch, dass die Beschreibung in C++ geschrieben wird. Einen Editor für die Spracheingabe selbst wird eigens durch Antlr leider nicht erzeugt. Allerdings wird ein Interpreter mitgeliefert, der eine Syntax-Hervorhebung bietet. Deshalb gibt es für das Kriterium *zusätzliche Werkzeuge* einen Punkt.

Eine umfangreiche Dokumentation stehen für Antlr zur Verfügung [siehe ANTLR, 2012b; Parr, 2007]. Zusätzlich zu diesen Quellen gibt es noch weitere Internetquellen, die neben Video-Tutorials auch andere Beispiele und Tutorials bieten. Für die Dokumentation gibt es deshalb 2 Punkte.

Antlr bietet Möglichkeiten der Fehlererkennung, um falsche Eingaben des Nutzers zu melden, worin beschrieben wird, was gelesen wurde und welcher Ausdruck eigentlich erwartet wird. Um verschiedene Szenarien testen zu können, kann der Interpreter, der mit dem Plugin mitgeliefert wird, verwendet werden. Dieser kann solche Szenarien durchlaufen und prüft sie auf ihre Gültigkeit anhand der gegebenen abstrakten Syntax.

4.2.4. Gegenüberstellung der Umsetzungsmöglichkeiten

Die vorgestellten Umsetzungmöglichkeiten werden in Tabelle 4.2 zusammengefasst. Boost.Spirit hat einen vergleichbaren Umfang wie Antlr. Nicht überzeugt hat der XML-Ansatz, da er nicht genügend auf den Nutzer ausgerichtet werden kann und keine Ausdrücke unterstützt. Leider hat Boost.Spirit nicht die Möglichkeit, einen Editor mitgenerieren zu lassen, der die Erstellung der Konfigurationsdateien erleichtern würde. Boost.Spirit überzeugt jedoch aufgrund

seiner äußerst guten Dokumentation. Antlr hat eine gut gelungene Integration in die weit verbreitete Entwicklungsumgebung Eclipse. Dagegen eignet Antlr sich weniger für das LAMA-Projekt, da es mit den Abhängigkeiten zu einer Java-Laufzeitumgebung für die Entwickler umständlicher ist, das Steuerungsframework zusammen mit dem Löser-Framework aktuell zu halten. Dadurch dass die Grammatikbeschreibung von Boost.Spirit direkt in das Projekt integriert werden kann, ist es leichter, die Erweiterbarkeit der LAMA-Bibliothek und der Steuerung zu gewähren.

Kriterium	Boost.Spirit	XML+XSD	Antlr
Modularer Aufbau	2	2	1
Anpassbarkeit	2	0	2
Ausdrücke	2	0	2
Plattformunabhängigkeit	2	2	2
Abhängigkeiten	2	2	0
Zusätzliche Werkzeuge	0	2	1
Dokumentation	2	2	2
Debugging & Fehlerbehandlung	2	2	2
Summe	**14**	**12**	**12**

Tabelle 4.2.: Vergleich bekannter DSL-Umsetzungsmöglichkeiten.

5. Konzept zur Implementierung des Steuerungsframeworks

Nachdem nun in dem vorherigen Kapitel eine geeignete Implementierungsmöglichkeit gefunden wurde, wird in diesem Kapitel das Konzept zur Implementierung entwickelt. Hier geht es hauptsächlich darum, die Frage zu klären, wie die in der Analyse ermittelten Features und deren Konfigurationsmöglichkeiten durch die DSL und das Steuerungsframework implementiert werden sollen. Dabei wird mit der Definition der konkreten Syntax begonnen. Anschließend werden ausgehend von der Benutzerschnittstelle, die die Features bereitstellt, alle dafür erforderlichen Datenstrukturen modelliert.

5.1. Aufbau der Sprache

In diesem Abschnitt wird entschieden, wie die konkrete Syntax in der zu entwickelnden Konfigurationssprache aussehen soll. Die syntaktische Struktur wird dabei an die in dem Grundlagenkapitel 2.3 vorgestellte Konfigurationssprache von OpenFOAM angelehnt, um die Anforderung 2 zu erfüllen.

```
TYP [identifikator] {
    schlüssel1 = wert1;
    schlüssel2 = wert2;
}
```

Diese Form der Sprachkonstukte hat einen geringen Formulierungsaufwand, da durch die Klammerung nach dem Typ (z.B. SOR) und die Reduzierung auf den Zuweisungsoperator `=` und das abschließende Semikolon des Schlüssel-Wert-

Paares keine weiteren festen sich für jeden Eintrag wiederholenden Syntax-Elemente vorhanden sind. Martin Fowler bezeichnet dies als *syntaktisches Rauschen*, was beispielsweise in einem XML-Dokument, durch die öffnenden und schließenden Tags, erhöht wird [siehe Fowler, 2012].

Diese oben vorgestellte Konfigurationsbeschreibung hat als optionale Angabe (angegeben durch die eckigen Klammern) den Identifikator, der für Löser und Logger wichtig ist. Wie dieser weiter eingesetzt werden kann und wie mehrere Konfigurationen in einer Datei definiert werden, wird im Folgenden beschrieben.

Neben den Ausdrücken, die durch Haltekriterien-Verknüpfung entstehen, ist auch die Verknüpfung unter den Lösern, Loggern und Haltekriterien selbst ein sehr wichtiger Aspekt, der bei der Struktur der Sprache berücksichtigt werden sollte.

Vorgestellt werden zwei Umsetzungsmöglichkeiten für die Löserverknüpfung. Die erste Möglichkeit ist die Verschachtelung der Löser-Konfigurationen, wodurch direkt eine Hierarchie von Lösern entsteht. Ein Beispiel für eine solche verschachtelte Konfigurationsbeschreibung wird in folgendem Listing gezeigt.

```
CG {
    id = beispielCG;
    preconditioner = SOR { omega = 1.5; };
}
```

Hier wird ein CG-Löser mit einem SOR-Löser präkonditioniert. Diese Möglichkeit beschreibt zugleich eine klare Ausführungsreihenfolge für die Initialisierungen und den Ablauf der Lösungsphase. Beim Parsen könnten die Instanzen zu den Lösern – ausgehend von den Blättern bis hin zu der Wurzel der Löser-Verknüpfung – erzeugt werden.

Die zweite Möglichkeit ist die getrennte Konfiguration. Die Konfiguration werden dabei nacheinander unabhängig voneinander beschrieben. Die Löserverknüpfung kann in diesem Fall über deren IDs geschehen. Die zu dem vorherigen Beispiel passende Konfiguration wird als nicht-verschachtelte Variante in

folgendem Listing gezeigt.

```
SOR myPrec {
    omega = 1.0
}

CG root {
    preconditioner = myPrec
}
```

Die fortlaufende Konfiguration und nicht verschachtelte Verknüpfung der Löser ist deutlich übersichtlicher und hat den Vorteil, dass Konfigurationen an mehreren Stellen in der Konfiguration für die Löserverknüpfung eingesetzt werden können. Somit kann eine Löserinstanz beispielsweise als Präkonditionierer für zwei verschiedene Verfahren eingesetzt werden. Das Problem, das mit dieser Art der Löserverknüpfung entsteht, ist, dass die Löserkonfigurationsreihenfolge nicht mehr klar definiert ist. So kann aktuell beispielsweise ein CG-Löser nicht erst instanziiert werden, wenn er mit einem nicht-instanziierten Löser präkonditioniert wird. Also muss der Nutzer auf die Reihenfolge achten oder die Konfigurationen müssen nach dem Parsen direkt in die richtige Reihenfolge gebracht werden.

Für Logger und Haltekriterien wurde die Wiederverwendbarkeit in den Anforderungen 8 und 10 gefordert. Für die Instanzen der Logger und Kriterien ist die Verwendung von IDs ebenfalls sinnvoll, da diese mit mehreren Lösern verknüpft werden könnten. Listing 7 zeigt ein Beispiel für die Konfiguration von Lösern, Loggern, Haltekriterien und ihre mögliche Verknüpfung.

Bei Loggerdefinitionen wurde auf die Angabe von Schlüssel-Wert-Kombinationen verzichtet, da, wenn ein Logger definiert wird, auch alle Angaben benötigt werden (siehe Featurediagramm 3.5, S. 32). Die Folge von Datenwerten ohne Schlüssel, wird ähnlich wie in gängigen Programmiersprachen, wie C und Java in runden Klammern angegeben. Dadurch dass in diesem Fall auf den Schlüssel und das Zuweisungszeichen verzichtet wird, ist der Schreibaufwand geringer, jedoch die Reihenfolge der Parameterangaben fest vorgeben.

```
CommonLogger myLogger ( loggerDesciption, solverInforation,
    toConsoleOnly, OpenMPTimer )

Criterion myCriteria = ( IterationCount( 10 )
    AND ResidualThreshold( L2Norm, 0.001, Absolute ) )

SOR myPre { logger = myLogger; }

CG root
{
    logger = myLogger;
    criteria = myCritria;
    preconditioner = myPre;
}
```

Listing 7: Beispielkonfiguration zweier Löser mit je nur einer Loggerinstanz.

Logische Verknüpfungen von Kriterien haben ebenfalls eine runde Klammerung, um Verknüpfungen verschachteln zu können. Die Verknüpfungsoperatoren sind AND und OR (siehe Featurediagramm 3.4, S. 32).

Nachdem nun die konkrete Syntax festgelegt wurde, werden in den Folgekapiteln hauptsächlich die semantischen Aspekte und die benötigten Datenstrukturen konzeptioniert. Hier wird beispielsweise geklärt, wie genau die Instanzverknüpfung umgesetzt werden kann.

5.2. Benutzer-Schnittstelle

Der Nutzer benötigt die Möglichkeit, die Sprachkonstrukte als String, Bytecode oder als Datei an die Bibliothek weiterzureichen. Die Bibliothek muss dafür eine geeignete Schnittstelle bieten. Diese Schnittstelle liefert einen Löser – die Wurzel der verknüpften Löser –, der die Aufrufe für die Initialisierung mit der Eingabematrix und den Lösungsaufruf mit Lösungsvektor und rechter Seite entgegennimmt. Dafür gibt es zwei mögliche Umsetzungen.

Die Erste ist die Entwicklung einer Konfiguratorklasse, die verschiedene Me-

thoden liefert, um die Konfiguration entgegenzunehmen, und Zeiger auf den erzeugten Wurzellöser liefert. Auf den zurückgelieferten Löser kann dann der Nutzer seine Aufrufe ausführen.

Die zweite Möglichkeit ist die Bereitstellung eines gesonderten Lösers, der die Konfiguration über Dieser Löser wird in der Klasse MetaSolver implementiert, leitet direkt von der abstrakten Solver-Klasse ab und bietet eigene initialize- und solve-Methoden. Der MetaSolver als Benutzschnittstelle hat den Vorteil, dass dem Nutzer die Fehlerbehandlung abgenommen und die Herausgabe eines fehleranfälligen nicht-konstanten Zeigers auf den Wurzellöser vermieden wird. Hierbei übernimmt der MetaSolver auch die Aufgabe der Speicherfreigabe für die erzeugten Instanzen über den eigenen Destruktor oder wenn eine neue Konfiguration entgegengenommen wird und die alte verworfen werden soll. Der MetaSolver-Ansatz wird deshalb für die weitere Konzeptionierung verwendet.

5.3. Aufgaben des MetaSolvers

In diesem Abschnitt soll ein Überblick gegeben werden, was hinter der nach außen gerichteten Schnittstelle genau passieren soll. Dazu gehören die internen Aufgaben des MetaSolvers, was nach dem Einlesen mit den vom Nutzer übergebenen Konfigurationen passiert, welche Erweiterungen nötig sind und welche Abhängigkeiten zwischen ihnen bestehen.

Da ein Konfigurationsstring, aus mehreren aufeinanderfolgenden Konfigurationen besteht, müssen diese schrittweise durch den MetaSolver abgearbeitet werden. Als erstes wird der Typ eingelesen, darauf folgend die ID und dann abhängig vom Typen die Konfigurationsparameter (siehe dazu Featurediagramm 3.1, S. 30). Dies kann bereits mit den in Listing 8 gezeigten Regeln umgesetzt werden. Beschrieben wird die Grammatik in der Notation von PEGs, die in Boost.Spirit zur Dokumentation verwendet wird. Eine Konfiguration kann eine Löser-, Logger- oder eine Kriterien-Konfiguration sein. Die Matrix-Vektor-bezogenen Konfigurationen werden am Ende dieses Kapitels betrachtet, da diese getrennt implementiert werden.

```
configuration_sequence ← *configuration
configuration ← solver_config / logger_config / criterion_config

solver_config ← solver_type id '{' parameter_list '}'
solver_type ← 'CG' / 'SOR' / ... / 'InverseSolver'

logger_config ← common_logger / file_logger

criterion_config ← 'Criterion' id '=' criteria

id ← [a-zA-Z_][a-zA-Z_0-9]*
```

Listing 8: Grammatikbeschreibung für Löser-, Logger- und Haltekriterien-Konfigurationen.

Die Interpretation der Konfigurationensequenz und die Initialisierung der Löser müssen voneinander getrennt werden, weil bei gleichbleibender Konfiguration unterschiedliche Matrizen und Vektoren für die Berechnungen verwendet werden können. Deshalb sollte die Konfigurationsinterpretation in einer gesonderten Funktion stattfinden, die vor der Initialisierung des MetaSolvers aufgerufen wird. Diese könnte mit dem Konstruktor oder gesondert aufgerufen werden, was eine MetaSolver-Instanz wiederverwendbar macht, indem mehrere Konfigurationen auf einen MetaSolver angewandt werden können. Dieser überschreibt dann intern die alte Konfiguration.

Da die Erweiterbarkeit des Löser-Frameworks durch die Implementierung des MetaSolvers nicht deutlich komplexer werden soll, ist es wichtig, dass die Abhängigkeiten zwischen Lösern und dem MetaSolver möglichst gering gehalten und keine Änderungen am MetaSolver nötig sind. Entsprechend sollte der MetaSolver bei neu hinzugefügten Lösern im Framework automatisch deren Konfigurationsmöglichkeit aufnehmen. Die Implementierung der Parameter-Interpretation sollte also nicht direkt in dem MetaSolver, sondern extern erfolgen.

Aufgrund der Inkompatibilitäten zwischen der aktuellen Boost.Spirit-Versionen 2.x zu den alten Boost.Spirit-Versionen 1.x, sollten die Löser keine Boost.Spirit-Abhängigkeiten besitzen, sodass die Löser selbstständig trotz veralteter Boost-

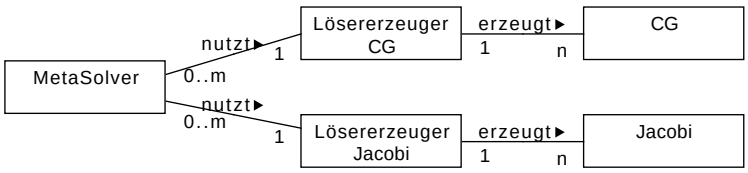

Abbildung 5.1.: Verbindung zwischen MetaSolver, Löser-Erzeuger und daraus entstandenen Löserinstanzen.

Version nutzbar bleiben. Dazu werden gesonderte *Erzeugerklassen* benötigt, die die Parameter interpretieren, Instanzen kreieren und diese konfigurieren. Abbildung 5.1 stellt diesen Zusammenhang noch einmal dar.

5.4. Instanzerzeuger

Der MetaSolver benötigt mit den Erzeugerklassen nur den Zugriff auf ihre Erzeugermethoden und kein weiteres Wissen über die Löserschnittstelle selbst. Den Zugriff auf die Löser erlangt der MetaSolver über eine Fabrik-Entwurfsmuster [siehe Gamma, 2004, S. 107ff] bei der sich die Erzeuger eintragen. Diese beinhalten dann die nötigen Parameter-Interpreter-Implementierungen, um die Konfiguration des jeweiligen Lösers verstehen zu können.

Der MetaSolver arbeitet sich schrittweise durch die gegeben DSL-Konstrukte, die er erhalten hat. Er liest den Kontext bis zum Lösertyp ein und weiß dadurch, welcher Erzeuger für die weiteren Inhalte der Konfiguration zuständig ist. Dies könnte beispielsweise durch eine Abbildung von Lösertyp-ID auf eine in den Erzeugern implementierte Grammatik beschrieben werden. Diese Grammatik wird dann durch eine Erzeugermethode aufgerufen, die den relevanten Parameterkontext übergeben bekommt. Somit würde sich die Hauptgrammatik in weitere löserbezogene Subgrammatiken unterteilen.

Die Klassen-Hierarchie der Erzeugerklassen in Abbildung 5.2 entsprechen der Hierarchie der Löserklassen, um auch die Konfiguration hierarchisch zu struk-

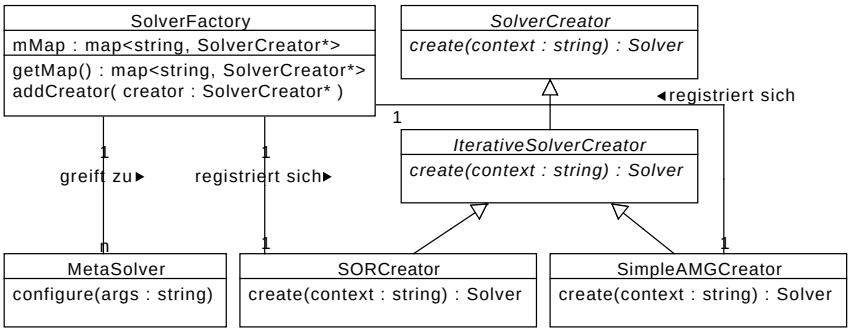

Abbildung 5.2.: Erzeugerklassen, welche die Löser erzeugen und konfigurieren.

turieren können, um Redundanzen zu vermeiden (siehe Featurediagramm 3.3, S. 31). Durch die Erzeugerklassen werden die `create`-Methoden implementiert. Der MetaSolver benötigt für die Umsetzung mit den Erzeugerklassen durch die lose Kopplung lediglich den Zugriff auf das assoziative Datenfeld, welches in der Factory-Klasse gehalten wird.

Für die Erzeugung der Haltekriterien und Logger werden nur einzelne Erzeugerklassen implementiert, da es hier keine Hierarchie an Konfigurationsparametern gibt.

5.5. Instanzverknüpfung

Wie bereits am Anfang dieses Kapitels im Abschnitt 5.1 erwähnt, wird der Identifikator, der nach dem Typen eingelesen wird, zur Verknüpfung der Instanzen verwendet. Da in dem aktuellen Löser-Framework bereits ein Konstruktorparameter ID je Löserinstanz angegeben werden muss, kann dieses Feld für die Löserverknüpfung verwendet werden. Aktuelle Löserverknüpfungen sind die Präkonditionierung von iterativen Lösern, Glätter und Grobgitterlöser bei Mehrgitterverfahren.

Abhängig von der Konfigurationen lassen sich alle Instanzen durch einen Baum abbilden, welcher durch die Verknüpfungen entsteht. Ein Beispiel für einen

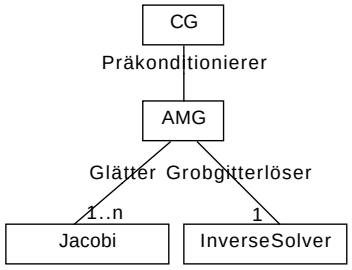

Abbildung 5.3.: Beispiel eines einfachen Löserbaums, aufbauend auf einem typischen Anwenderszenario.

Löserbaum wird in Abbildung 5.3 gezeigt.

Da nur auf den Wurzellöser (hier CG) die Initialisierungs- und Lösungsaufrufe ausgeführt werden, ist nur dieser für den MetaSolver relevant. In dem Beispiel benötigt der MetaSolver also nur den Zugriff auf die CG-Instanz. Dieser ist über die Präkonditionierung mit dem SimpleAMG verknüpft und leitet die Initialisierungs- und Löseraufrufe an diesen weiter. Wird bei einem AMG die *initialize*-Methode aufgerufen, so wird das AMG-Setup ausgeführt, welches die Gitter bzw. Level des Mehrgitterverfahren erzeugt. Die Glätter und der Grobgitterlöser erhalten ebenfalls die Aufrufe durch ihren Elternlöser (der AMG-Instanz). In Abschnitt 5.6 wird aufgrund der nicht vorhandenen Konfigurierbarkeit des AMG gesondert eingegangen.

Wenn zur Parserzeit die Verknüpfungen zwischen den Lösern wie bisher üblich über Zeiger auf Löserinstanzen hergestellt werden, muss die Reihenfolge der Konfigurationen durch den Nutzer beachtet werden. In dem Beispiel in Abbildung 5.3 muss die Instanz des AMGs erzeugt werden, bevor auf der CG-Instanz die Methode `setPreconditioner( Solver* preconditioner )` aufgerufen werden kann.

Ansonsten müssten die Abhängigkeiten zwischen den Konfigurationen zuvor ermittelt werden, um die richtige Reihenfolge – angefangen mit den Blättern des Löserbaums und endend mit der Wurzel – herzustellen. Eine Alternative dazu ist, dass Löser eine weitere Methode wie `setPreconditioner( string id )`

anbieten und sich den angegebenen Löser über die ID, verzögert über eine Instanzregistry holen. Jedoch müsste dann jede Löserinstanz auf diesen Zugriff haben, wofür ein statisches Objekt der Instanzregistry nötig wäre. Diese Instanzregistry benötigt also entweder die Löser, die sich mit anderen verknüpfen lassen oder die Lösererzeuger, wenn diese die Löser über Zeiger auf Löserinstanzen direkt zu Parserzeit verknüpfen.

Für die Implementierung wird der Ansatz gewählt, bei dem zur Parserzeit die Löser durch die Lösererzeuger verknüpft werden, da so keine Abhängigkeiten der Löser zu Instanzregistry entsteht. Zudem müssen keine zusätzlichen Methoden mit Zeichenketten als Verknüpfungsparameter hinzugefügt werden.

Neben der Erzeugerregistry ist nun also auch eine Registry für die Abbildung von Löser-IDs auf Löserinstanzen nötig, welche durch andere Lösererzeuger abgefragt werden können. Die Instanzregistry kann entweder statisch bzw. global implementiert werden oder es gibt eine Instanzregistry je MetaSolver. Wird die Instanzregistry MetaSolver-gebunden implementiert, ist es möglich, dass MetaSolver zweimal mit der gleichen Konfiguration ausgeführt werden können. Mit einer globalen Instanzregistry ist es dagegen möglich, MetaSolver-übergreifend die erzeugten Löserinstanzen abzurufen. Für beide Möglichkeiten gibt es keine Anforderungen, die die Auswahl dieser Alternativen beeinflussen. Anwendungsfälle, dass MetaSolver-übergreifend Instanzen ausgetauscht oder Konfigurationen mehrfach angewandt werden müssen, gibt es nicht. Der Ansatz der globalen Registry wird gewählt, da der Zugriff auf die statische Registry-Instanz hier zwar ein Aufruf zur Löschung der Instanzen auf des Registry nötig ist, falls eine Konfiguration öfters aufgerufen werden soll, jedoch auch der Vorteil des MetaSolver-übergreifenden Zugriffs gegeben ist.

Für die Erzeugerklasse der Haltekriterien sind keine Sub-Grammatiken wie bei den Lösern notwendig. Da die Verknüpfung der Haltekriterien über logische Operatoren (UND und ODER) geschieht ist diese anders zu behandeln. Wie bereits in Listing 8 angedeutet, ist die Haltekriteriendeklaration in der Konfigurationssprache eine Zuweisung. Darauf folgt dann eine Expression, die die Verknüpfung der Instanzen beschreibt. Listing 9 zeigt die Grammatikbe-

```
criterion_config ← 'Criterion' id '=' node ';'
id ← [a-zA-Z_][a-zA-Z_0-9]*
node ← ( '(' node logical_connective node ')' ) / leaf
logical_connective ← 'AND' / 'OR'
leaf ← iteration_count / residual_threshold / residual_stagnation

iteration_count ← 'IterationCount' '(' int ')'
residual_threshold ←
    'ResidualThreshold' '(' norm ',' double ',' check_mode ')'
residual_stagnation ←
    'ResidualStagnation' '(' norm ',' double ',' int ')'

norm ← 'L1Norm' / 'L2Norm' / 'MaxNorm'
check_mode ← 'Relative' / 'Absolute'
```

Listing 9: Grammatikbeschreibung für Haltekriterien-Konfigurationen im Detail.

schreibung für Haltekriterien vollständig. Die Startregel für Haltekriterien ist `criterion_config`, dieser wird durch MetaSolver aufgerufen und liefert einen einzigen Zeiger auf eine Haltekriterien-Instanz zurück. Die Regel besteht aus dem Einlesen des Schlüssels und ruft dann die Regel `node` auf. Diese interpretiert entweder ein Blatt (Regel `leaf`) oder zwei Kinder-Knoten verbunden über eine logische Verknüpfung. Ein Blatt entspricht einem konkreten Haltekriterium (IterationCount, ResidualThreshold oder ResidualStagnation). Die logische Verknüpfung zusammen mit einem linken und einem rechten Kind erzeugen eine Instanz der Klasse `BooleanCondition`. Die Regel, die die Schlüsselwörter für logische Verknüpfungen beschreibt, bildet auf den in den Grundlagen gezeigten Aufzählungstyp (engl. *Enumeration*) `BooleanOperator` in der BooleanCondition-Klasse ab (siehe in Abbildung 2.6, S. 11). Die Regel `check_mode` bildet auf die Aufzählung `CheckMode` in der `ResidualThreshold`-Klasse ab. Die Regel `norm` erzeugt anhand der Schlüssel Instanzen der Klassen, die von der abstrakten Klasse `Norm` ableiten. Diese werden für die Residuumsberechnung in den Haltekriterien ResidualStagnation und ResidualThreshold verwendet.

Auf das Konzept zur LoggerCreator-Klasse wird in diesem Fall verzichtet, da

diese nicht untereinander verknüpft werden und entsprechend analog zu einem Blatt in der Grammatikbeschreibung der Haltekriterien deklariert werden. Die Parameter sind dem Featurediagramm in Abbildung 3.5, S. 32 zu entnehmen.

Für die Verknüpfung mit den Lösern werden auch die Instanzen der Logger und Haltekriterien in einer Instanzregistry gehalten. Jedoch sollten diese getrennt gehalten werden. Um sie unterscheiden zu können, wird die Registry-Klasse templatebasiert umgesetzt. Das Template entspricht dem Instanztyp, also Logger, Kriterium oder Löser. Es werden im Falle einer globalen Instanzregistry-Implementierung drei statische Objekte mit jeweiligem Templateargument erzeugt. Die Klasse wird in Abbildung 5.4 als Klassendiagramm dargestellt.

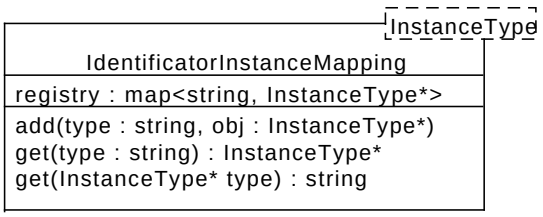

Abbildung 5.4.: Registryklasse für Instanzen, welche statisch erzeugt werden, um einen globalen Zugriff zu erlauben.

5.6. Konfigurierbarkeit des Mehrgitterlösers

Da die Anzahl der Gitter von der verwendeten Matrix abhängig ist und sich dies erst nach der Konfigurationszeit ergibt, ist es schwer, den SimpleAMG zu konfigurieren. Der aktuelle AMG bietet nicht die Möglichkeit, auf bestimmten oder allen Gittern Löser zu setzen. In diesem Abschnitt soll herausgefunden werden, wie der AMG konfigurierbar werden kann, um die Anforderung 11 auf Seite 27 zu erfüllen.

Eine Möglichkeit ist die Implementierung zweier Zuweisungsmethoden, die nur Löser-Zeiger in Membervariablen der Löser-Klasse speichern. Diese werden dann wie die anderen Parameter bei dem Initialisierungsaufruf an das dyna-

misch gebundene Setup weitergereicht. Wenn zur Initialisierungszeit klar ist, wie viele Glätter benötigt werden, werden Kopien des gesetzten Glättungslösers erzeugt, die dann für die Glättung der jeweiligen Gitter zuständig sind. Aktuell sind Löser nicht kopierbar, da sie Laufzeit-Informationen beinhalten, wie z.B. vergangene Iteration oder einen Zeiger auf die verwendet Matrix. Diese dürfen nicht kopiert werden, da jeder Glätter seine eigene Matrix hat und Iterationszahlen zurückgesetzt werden müssten. Die Konfigurationsparameter müssen deshalb von den Laufzeit-Informationen getrennt werden.

```
DefaultJacobi jacobiSolver {...}
CG cgSolver { preconditioner = jacobiSolver; }
InverseSolver inverseSolver {...}
SimpleAMG simpleAMG {
    smoother = jacobiSolver;
    coarseLevelSolver = inverseSolver;
    maxLevel = 3;
    minValuesPerRow = 10;
}
SimpleAMG root {
    smoother = cgSolver;
    coarseLevelSolver = simpleAMG;
    maxLevel = 5;
    minValuesPerRow = 100;
}
```

Listing 10: Beispielkonfiguration zweier aneinander geketteter SimpleAMGs.

Diese Umsetzungsmöglichkeit lässt dem Nutzer die Option offen als Grobgitterlöser einen weiteren AMG zu nutzen, der eine gesonderte Konfiguration hat. Mit dieser Option können durch die Abbruchkriterien maxLevel und minValuesPerRow viele Zusammenstellungen von Lösern konfiguriert werden. Es können beispielsweise mehrere AMGs aneinandergekettet werden, um auf den feineren Gittern andere Glätter nutzen zu können als auf den Gröberen. Eine Beispielkonfiguration wird in Listing 10 anhand zweier verketteter Mehrgitterlöser mit den IDs *root* und *simpleAMG* gezeigt. Der Löser mit der ID *simpleAMG* erhält als Glätter einen Jacobi und als Grobgitterlöser einen InverseSolver. Dieser AMG wird als Grobgitterlöser des AMG mit der ID *root* definiert. *root* hat

zusätzlich anders als *simpleAMG* einen CG als Glätter. Diese Konfiguration sorgt bei dem Setup des AMGs dafür, dass root abhängig von der Eingabe-Matrix maximal vier Glätter-Instanzen vom Typ CG erzeugt oder falls Kriterium minValuesPerRow durch die Restriktion früher eintritt, entsprechend weniger Instanzen durch Kopieren erzeugt werden müssen.

Eine weitere Möglichkeit wäre die Übergabe von mehreren Löserinstanzen, die für die Glättung auf den Gittern zuständig sind. Ein `std::vector<Solver>` könnte beispielsweise an den AMG-Löser übergeben werden. Die Anzahl der Vektor-Elemente müsste dabei dem Wert des Kriteriums maxLevel entsprechen. Diese Umsetzungsmöglichkeit hat den Nachteil, dass Löserinstanzen erzeugt werden, die ggf. nicht verwendet werden, weil die Gitter-Erzeugung durch das Kriterium minValuesPerRow gestoppt wurde.

Da die erste Umsetzungsmöglichkeit die Funktionalität gegenüber der Vektor-Möglichkeit nicht einschränkt und diese einfacher zu implementieren und zu konfigurieren ist, wird diese für die Implementierung gewählt.

Der AMG bietet, wie im Grundlagenkapitel beschrieben, üblicherweise die Option, unterschiedliche Zyklen zu verwenden. Diese sind jedoch aktuell nicht implementiert. Deshalb wird die Konfiguration der Zyklen hier übersprungen. Allerdings wird bei der Implementierung darauf geachtet, dass das Nachziehen der Konfigurationsmöglichkeit für die LAMA-Entwickler keine große Hürde darstellt. Dies wurde ebenfalls in den Anforderungen gefordert.

5.7. Matrix-Vektor-Domäne

Wie in Abschnitt 3.2.2, S. 34 des Analyse-Kapitels festgelegt wurde, werden die Features Matrix, Vektor, Speicherbereiche, Berechnungsort und Verteilung in einer getrennten Domäne untergebracht, da Löser und dessen bezogene Features nur wenige funktionale Gemeinsamkeiten haben. Die Konfigurationseinstellungen je Vektor und Matrix sind *Speicherformat*, *Berechnungsort* und *Verteilung*. Eine Konfiguration einer Matrix soll beispielsweise, wie in folgendem

Listing formuliert werden können.

```
CSR { context=CUDA; distribution=Block(MPI); }
```

Mit `CSR` wird der konkrete Speicherbereichtyp CSR für beide Speicherbereiche (halo und local) der Matrix vorgegeben.

Bevor für diese Domäne die Grammatik beschrieben wird, wird genauer untersucht, wie die Datenstrukturen erzeugt werden können und welche Informationen für die Konfiguration bereits zur Verfügung stehen oder erforderlich sind. Für die Erzeugung einer Matrix oder eines Vektors gibt es zwei typische Anwedungsszenarien.

1. Der Anwender liefert die Daten eines Daten-Containers in Form von Arrays im CSR oder *dense*-Format, um daraus eine Matrix im Format seiner Wahl zu erstellen.
2. Der Anwender möchte die Daten aus einer Datei einlesen lassen. Hierfür werden verschiedene Datei-Speicherformate, wie z.B. das Matrix-Market-Format durch LAMA unterstützt [siehe Matrix-Market, 2012]. Diese geben vor, wie der Inhalt einer Matrix gespeichert wird und somit ausgetauscht oder persistent gespeichert werden kann. Im Fall des Matrix-Market -Format wird das COO-Format verwendet. Die dazugehörigen Arrays werden zusammen mit nötigen Header-Informationen, wie Anzahl der Elemente, Spalten und Zeilen in einem einheitlichen Format gespeichert.

Abbildung 5.5 zeigt die Methoden und Konstruktoren, die für die Instanziierung, Datenzuweisung und Konfiguration der Matrix relevant sind. Vektoren bieten die gleichen Schnittstellen, werden hier aufgrund der geringeren Komplexität zunächst nicht weiter betrachtet, da alle Schwierigkeiten, die für die Matrix gelöst werden, zugleich auch den Vektor abgedecken. Der Vektor erhält statt der zwei Verteilungen und zwei Berechnungsorte nur jeweils Eine(n) und es gibt bezüglich des Speicherformates aktuell nur eine Implementierung für dichbesetzte Vektoren.

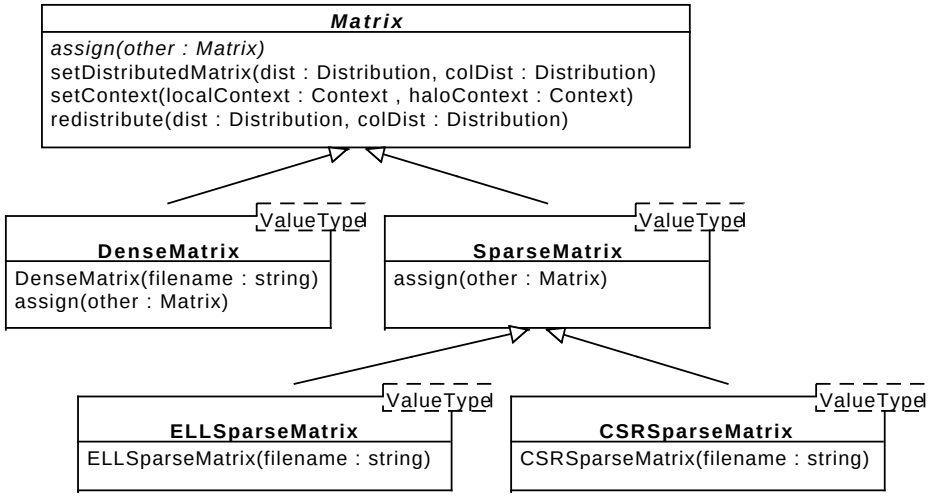

Abbildung 5.5.: Schnittstellen, die die aktuelle Matrix-Implementierung bietet, um die Matrix mit Daten zu füllen.

Um den o.g. ersten Fall zu implementieren, ist der Nutzer gezwungen, die Matrix zunächst in eine LAMA-CSRSparseMatrix oder DenseMatrix umzuwandeln. Diese kann anschließend über die assign-Methode oder einen Kopierkonstruktor in das gewünschte Speicherformat konvertiert werden, das durch die Konfiguration vorgegeben wird. Für den zweiten Fall gibt es Matrixkonstruktoren, die einen Dateinamen entgegennehmen und selbst ermitteln, welches Dateispeicherformat gewählt wurde. Wird dies beispielsweise auf eine ELLSparseMatrix mit einer Matrixdatei im MatrixMarket-Format aufgerufen, so werden die Daten zunächst vom COO-Format, wie sie für das Matrix-Market-Format abgespeichert werden, in das ELL-Format umkonvertiert.

Nachdem nun geklärt wurde, welche Möglichkeiten es gibt, um Instanzen von Matrizen und Vektoren von unterschiedlichem Typ zu erzeugen, wird dies nun für die zwei noch fehlenden Features *Verteilung* und *Berechnungsort* geklärt.

Bei den Verteilungsimplementierungen der LAMA-Bibliothek werden matrixbezogene Informationen benötigt. Um Instanzen von der Blockverteilungsklasse oder der Klasse der zyklischen Verteilung zu erzeugen, wird die Anzahl der

Elemente für die Zeile bzw. Spalte benötigt, je nachdem, wie die Verteilung eingesetzt wird. Diese Informationen müssen daher schon für die Grammatik vorliegen. Eine weitere Angabe, die die Matrizen benötigen sind die s.g. Kommunikatoren (engl. Communicator), welche die unterschiedlichen Schnittstellen aktuell von MPI und Partitioned Global Address Space (PGAS) vereinheitlichen [PGAS, 2012]. Zugewiesen werden die Verteilungen entweder über die Methode `setDistributedMatrix`, falls noch keine Daten in der Matrix vorhanden sind, oder über die Umverteilungsmethode `redistribute`, bei der die bestehenden Daten unter den Rechenknoten erneut ausgetauscht werden müssen.

Instanzen eines Contextes werden über eine Fabrikklasse abgerufen, bei der sich diese statisch registriert haben. Die Grammatik muss diese deshalb nicht zuerst erzeugen. Bezogen auf die Daten-Container benötigen die Context-Klassen keine Informationen.

Genau wie für die Löserdomäne benötigten die Daten-Container eigene Benutzerschnittstellen, um ihre Konfiguration einlesen zu können. Hierfür werden die Klassen *MetaMatrix* und *MetaVector* erstellt, die, wie in Abbildung 5.6 gezeigt, von den abstrakten Klassen Matrix und Vektor ableiten und deren rein virtuellen Methoden implementieren (für die genaue Einordnung siehe Abbildung 2.2 im Grundlagenkapitel auf Seite 6). Jede dieser Klassen behandelt immer nur die Konfiguration einer Datenstruktur, da hier keine Instanzverknüpfung möglich ist.

Es gibt entsprechend der zwei zuvor vorgestellten Anwendungsfälle jeweils einen Konstruktor. Der Erste nimmt die Konfiguration und die Matrix entgegen, die durch den Nutzer erstellt wurde. Die Konfiguration wird in dem Konstruktor durch den Parseraufruf interpretiert und die Matrixinstanz erzeugt, dessen Zeiger in der MetaMatrix-Klasse gespeichert wird. Die übergebene Matrix wird durch die Grammatik an die Konstruktoren der zu erzeugenden Matrix weitergeleitet und gegebenenfalls umgewandelt. Die Konfigurationen für die Berechnungsorte und die Verteilungen werden durch die zuvor beschriebenen Methoden auf die erzeugte Matrix angewandt.

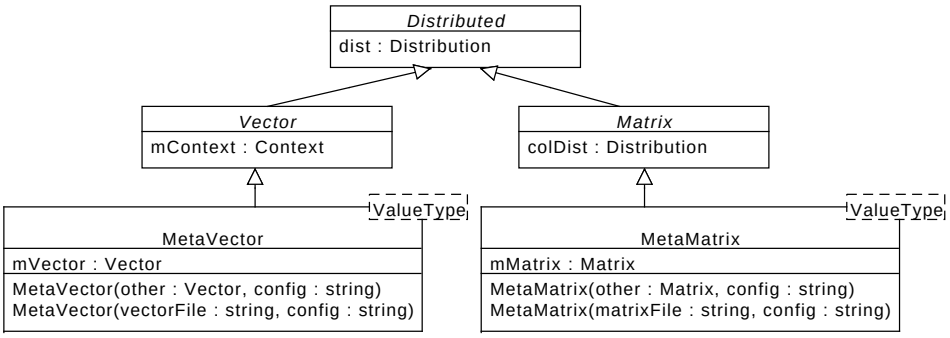

Abbildung 5.6.: Einordnung der MetaMatrix und MetaVector-Klassen in das bestehende LAMA-Projekt.

Für den anderen Fall, dass nur ein Dateiname angegeben wird, wird eine alternative Startregel benötigt, welche statt einer Matrix den Dateinamen in Form einer Zeichenkette entgegen nimmt. Über die intern implementierte Grammatik wird dieser an den Konstruktor der zu erzeugenden Matrix weitergereicht, welche dann die Daten direkt selbst einliest ohne eine Kopie erstellen zu müssen.

Um das zu Beginn dieses Abschnittes vorgestellte Beispielsprachkonstrukt einlesen zu können, wird nun beschrieben, wie die Grammatiken der MetaVector und MetaMatrix-Klassen aufgebaut werden. Es werden keine Erzeugerklassen je Speicherformat benötigt, da Matrizen und Vektoren eine einheitliche Schnittstelle bieten und keine Unterschiede in ihrer Konfigurierbarkeit aufweisen. Deshalb können die Instanzen direkt anhand des eingelesenen Typs, wie 'CSR' oder 'Dense' erzeugt werden. Ein Berechnungsort kann entweder je Speicherbereich einer SparseMatrix gesetzt werden oder es wird nur ein Einziger angegeben, der im Falle einer dünnbesetzten Matrix für beide Speicherbereiche übernommen wird. Für dichtbesetzte Matrizen oder Vektoren wird immer nur ein Berechnungsort übernommen. Listing 11 zeigt hierfür eine vereinfachte Variante, mit nur jeweils einem Berechnungsort und einer Verteilung, was abgesehen vom Container-Typ auch der Grammatikdefinition des MetaVectors entspricht.

```
1 matrix_configuration ←
2     type '\{' context_def ',' distribution_def '\}'
3 type ← 'CSR' / 'ELL' / 'Dense'
4 context_def ← 'context=' context ';'
5 context ← 'Host' / 'CUDA'
6 distribution_def ← 'distribution=' distribution ';'
7 distribution ← ( 'Block(' communicator ')' )
8              | ( 'Cyclic(' communicator ',' block_size ')')
9 communicator ← 'MPI' / 'PGAS' / 'none'
```

Listing 11: Grammatikbeschreibung für die MetaMatrix, die Konfigurationen einer Matrix einliest.

Durch die semantische Aktion von der Regel `type` – Zeile 3 – werden die Konstruktoren der zu erzeugenden Matrizen aufgerufen, dessen Parameterangabe, wie oben beschrieben entweder der Dateiname oder die durch den Nutzer erzeugte Matrix ist. In dem Implementierungskapitel 6.5 wird die umgesetzte Grammatik genauer vorgestellt.

5.8. Entscheidungsstrategien

Der vorherige Abschnitt gab einen Überblick darüber, wie der MetaSolver und die Meta-Klassen für die Daten-Container implementiert werden sollen. Nun werden die konzeptionellen Überlegungen zu den Strategierealisierungen aus den Anforderungen 15 bis 20 vorgestellt.

Die Schwierigkeit bei der Implementierung von Strategien ist, dass sie unterschiedliche Parameter benötigen, anhand derer die nötigen Entscheidungen getroffen werden. Beispielsweise soll eine Strategie anhand des definierten Berechnungsortes und der Matrixgröße entscheiden, welches der Speicherformate geeignet ist. Die Definition einer Matrixkonfiguration, deren Typ durch eine Strategie implementiert wird, wird im folgenden Listing gezeigt.

```
Strategy1 {
    contextLocal = CUDA;
```

```
    contextHalo = Host;
    distribution = Block( MPI );
    colDistribution = Cyclic( MPI, 10 );
}
```

`Strategy1` ist der Bezeichner, der auf die Strategie abbildet, die eingesetzt werden soll. Um diese abzufangen wird die zuvor beschriebene Regel `matrix_type` erweitert.

Sollte die Strategie jedoch nun die Angabe zum Berechnungsort (Context), welcher erst später in der Konfiguration definiert wird, benötigen, so kann die Instanz der Matrix an dieser stelle noch nicht erzeugt werden, da Angaben für die Entscheidung durch die Strategie fehlen und sobald eine Matrix eines bestimmten Typs erzeugt wurde, nicht in eine andere Matrix umgewandelt werden kann.

Um die Erzeugung verzögert zu implementieren, wird eine Klasse benötigt, die wie eine übliche Matrix erzeugt wird. Intern speichert sie einen weiteren Matrixzeiger und sammelt zunächst alle Konfigurationsinformationen ein und wird durch einen gesonderten Aufruf endgültig erzeugt. Dieser gesonderte Methodenaufruf enthält alle Entscheidungsroutinen, die dann über den Typ der intern gehaltnen Matrix entscheiden können. Die Strategieklasse wird dann weiter als Matrikklasse eingesetzt und leitet die Aufrufe an die intern gespeicherte Matrix weiter.

Die Implementierung von Strategien ist sehr komplex, da hier sehr genau untersucht werden muss, welche Parameter in die Entscheidung für eine Gruppe von Strategien einfließen. Die Parameter müssen dabei, wie im vorherigen Beispiel, nicht immer direkt verfügbar sein, weshalb Strategieklassen erzeugt werden müssen, die eine verzögerte Erzeugung der Zielinstanz übernehmen. Welche Entscheidungsparameter eine Strategie wirklich benötigt, wird in dieser Arbeit nicht genau geklärt, da zuvor typische Anwendungsfälle untersucht werden müssten, um herauszufinden, an welchen Punkten dem Nutzer Entscheidungen schwer fallen und was genau für diese Entscheidungen nötig ist.

In diesem Kapitel wurde zunächst die konkrete Syntax definiert. Die Syntax wurde dabei an die der OpenFOAM-Konfigurationssprache angelehnt, um eine der Zielgruppe geläufige Sprache zu unterstützen und somit die Einarbeitungszeit für den Nutzer zu erleichtert. Des Weiteren wurde bei der Wahl der Syntax darauf geachtet, dass sie geringen Schreibaufwand erfordert und dennoch verständlich ist. Anschließend wurden die Aufgaben des MetaSolvers beschrieben, der als Benutzerschnittstelle für die Löserdomäne verwendet wird. Hierbei wurde primär erläutert, wie die im MetaSolver implementierte Grammatik umgesetzt werden soll und an welcher Stelle die Löserparameter interpretiert werden. Hierfür soll eine Fabrikklasse umgesetzt werden, die den Zugriff auf Lösererzeugerklassen gewährt. Für die Instanzverknüpfung werden zusätzlich Instanzregistries benötigt, die den Lösererzeugerklassen die Möglichkeit bieten die Verknüpfung der Instanzen zu ermöglichen. Darauf wurde untersucht, welche Möglichkeit es gibt das AMG-Verfahren konfigurierbar zu machen, obwohl zur Konfigurationszeit nicht bekannt ist, wie viele Instanzen für die Glättung benötigt werden. Hier wurde entschieden, dass eine konfigurierte Instanz erzeugt wird und in der Setupphase entsprechend kopiert wird. Neben dem MetaSolver werden auch die Klassen MetaMatrix und MetaVector implementiert, welche eigene Sprachen interpretieren, um den jeweiligen Daten-Container instanziieren und konfigurieren zu können. Da in der Konfigurationssprache Entscheidungsstrategien eingesetzt werden sollen, wurde in den Folgeabschnitten beschrieben, wie für diese eine verzögerte Instanziierung umgesetzt werden kann.

6. Implementierung des Steuerungsframeworks

Nachdem nun in dem vorherigen Kapitel das Konzept zur Implementierung erstellt wurde, wird aufbauend auf den UML-Diagramme und den in PEG-Notation beschriebenen Grammatikausschnitten in diesem Kapitel das Vorgehen während der Implementierung beschrieben. Begonnen wird mit der Implementierung des MetaSolvers, da dieser die Schnittstelle zum Nutzer bildet. Von ihm ausgehend werden dann die Details zu Lösererzeugern, Factory-Mechanismen und weitere Implementierungen zur Daten-Container-Domäne beschrieben. Dabei werden die Details der Funktionalität von Boost.Spirit abhängig von dem jeweilig beschriebenen Implementierungsabschnitt erläutert.

6.1. Implementierung des MetaSolvers

Die Klasse MetaSolver leitet von der abstrakten Klasse `Solver` ab und bietet damit die gewünschte einheitliche Schnittstelle nach außen, sodass der Nutzer den MetaSolver als gewöhnlichen Löser nutzen kann. Der MetaSolver implementiert zusätzlich eine Methode `configure`, welche in Listing 12 gezeigt wird. Diese Methode erzeugt intern eine Grammatikinstanz und übergibt sie zusammen mit den aus der Konfigurationszeichenkette erzeugten Iteratoren an die Parser-Funktion `phrase_parse`. Die Grammatik wird entsprechend in einer gesonderten Klasse `ConfigGrammar` implementiert.

Wie im Konzept-Abschnitt 5.2 beschrieben, benötigt der MetaSolver den Zugriff auf den Wurzellöser. Dieser ergibt sich aus den Löser-Verknüpfungen, die sich wiederum aus den Konfigurationen ergeben und durch die Erzeuger gesetzt werden. Der Wurzellöser wird in der Grammatik über einen Methodenaufruf

```
void MetaSolver::configure( std::string configuration )
{
    ConfigGrammer configReader;

    StringIterator begin = configuration.begin();
    StringIterator end = configuration.end();

    bool r = phrase_parse(
        begin,                  //Iterator Anfang
        end,                    //Iterator Ende
        configReader,           //Die Grammatik
        ascii::space);          //Separator

    ...                         //Fehlerbehandlung
}
```

Listing 12: Aufruf des Parsers mit der Grammatik ConfigGrammar.

`setRootSolver()` gesetzt. Dieser kann dann durch den MetaSolver abgefragt werden. Einen direkten Rückgabewert hat die Grammatik des MetaSolvers also nicht.

Die Sequenz von Konfigurationen wird anhand der Iteratoren und der in der Grammatik beschriebenen Regeln durchlaufen. Die Konfigurationssequenz beinhaltet Konfigurationen für die Typen Löser, Logger oder Haltekriterien. Die Implementierung der Grammatik wird in Listing 13 gezeigt.

Löser-, Logger- und Kriterienkonfigurationen teilen sich immer Typ, ID und Parameter auf. Für den MetaSolver ist nur der Typ relevant, um die im Konzept vorgestellten Erzeuger aufrufen zu können. Von den erzeugten Lösern benötigt der MetaSolver lediglich den Zeiger auf die Wurzellöserinstanz. Die ID und die Parameter werden durch den Grammatikteil der Erzeugerklassen eingelesen.

Für die Abbildung vom Typ auf den jeweiligen Erzeuger benötigt der MetaSolver den Zugriff auf die im Konzept vorgestellte Erzeugerregistry (siehe Abschnitt 5.4, S. 57). In diesem Abschnitt wurde eine Implementierungsmöglichkeit für den Zugriff auf die Erzeuger vorgestellt, bei der durch Methoden-Aufrufe auf die Erzeuger eine Sub-Grammatik aufgerufen wird, die die zu in-

```
SolverConfigGrammar() :
        base_type( mRConfigurationSequence )
{
    mRConfigurationSequence = *mRConfiguration;

    mRConfiguration = mRSolverConfiguration
                    | mRLoggerConfiguration
                    | mRCriteriaConfiguration;
    ...
}
typedef qi::rule<Iterator, void(), ascii::space_type> RuleType;
RuleType mRConfigurationSequence;
RuleType mRConfiguration;
...
```

Listing 13: Einlesen einer beliebig großen Folge von Konfigurationen.

terpretierenden Parameter als Zeichenkette bekommt. Diese wird dann durch die internen Grammatikregeln ausgewertet und auf die intern erzeugte Löserinstanz angewandt.

In einer zweiten Version der Implementierung wurde auf die Sub-Grammatiken verzichtet. Stattdessen beinhalten die Erzeugerklassen nur noch eine Startregel. Diese wird dann in der Erzeugerregistry gehalten und kann von der jeweiligen MetaSolver-Instanz abgerufen werden, um die Regel des Erzeugers anschließend anhand der Typ-ID dynamisch zur Parserzeit in die eigene Grammatik einzubetten. Hierfür ist ebenfalls eine Abbildungsdatenstruktur nötig, die von Typ-ID auf die damit verbundene Erzeugerregel abbildet. Boost.Spirit bietet eine `qi::symbol`-Datenstruktur, die sich für diese Problemstellung sehr gut eignet, da sie direkt in die Grammtik eingebettet werden kann, ohne zusätzliche semantische Aktionen für die Umwandlung zu erfordern. Gefüllt wird dieses assoziative Datenfeld mit den statischen Erzeugerobjekten, die sich bei SolverFactory-Klasse registrieren.

Sobald der MetaSolver den Zugriff auf diese Datenstruktur erlangt, wird der jeweiligen Erzeugerregel der restliche Kontext übergeben, der noch zu interpretieren ist. Mit dieser Regel liest der Qi-Interpreter die Löserkonfiguration des

jeweiligen Lösers ein und fährt im Anschluss – falls vorhanden – mit der darauf folgenden Konfiguration fort. Für diese kann wieder ein anderer Erzeuger nötig sein, dessen Regel auch wieder dynamisch in die Grammatikregel eingebettet wird. Diese dynamische Aufnahme von Funktionalität in der eigenen Grammatik wird als *Nabialek-Trick* bezeichnet [mehr dazu siehe de Guzman und Kaiser, 2012]. Listing 14 zeigt die Implementierung des Nabialek-Tricks in der Löser-Grammatikregel.

```
1  typedef qi::rule<Iterator, SolverPtr(), ascii::space_type>
2      SolverRuleType;
3  qi::symbols<char, SolverRuleType>& creatorMap =
4      SolverFactory::getFactory().getSolverCreatorRules();
5
6  mRSolverConfiguration =
7          creatorMap [ _a = _1 ]
8       >> lazy( _a )
9             [ if_( phoenix::bind( &Solver::getId, _1 ) != "root" )
10              [
11                  phoenix::bind(
12                      &SolverFactory::addSolver,
13                      phoenix::ref( factory ),
14                      _1 )
15              ].else_[
16                phoenix::bind(
17                    &ConfigGrammar::setRootSolver,
18                    *this,
19                    _1 )
20              ]
21            ];
```

Listing 14: Implementierung des Nabialek-Tricks in der MetaSolver-Klasse.

In den Zeile 3 und 4 wird eine Referenz auf das assoziative Datenfeld geholt. Dies geschieht über die statische Instanz der SolverFactory-Klasse. Das assoziative Datenfeld wird in Zeile 7 in die Regel eingebettet. Hinter dem Datenfeld wird in der semantischen Aktion `_a` der Rückgabewert zugewiesen, der sich hinter dem Platzhalter `_1` verbirgt. `_a` ist dabei eine für die Regel lokale Variable, dessen Typ wie folgt in der Regeldeklaration festgelegt wird:

```
qi::rule<Iterator, void(),
         qi::locals< SolverRuleType >, // Lokale Variable
         ascii::space_type> mRSolverConfiguration;
```

Anmerkung: Für eine zweite lokale Variable `_b` würde `qi::locals` einfach einen weiteren Templateparameter erhalten.

`_a` speichert also die zurückgegebene Regel vom Typ `SolverRuleType`. Diese Regel wird in der darauf folgenden Zeile ausgeführt und wurde somit aus der Semantik-Beschreibung in die Grammatikbeschreibung integriert. Der Spirit.Qi-Parser `lazy` sorgt dafür, dass der Übergabeparameter nicht direkt ausgeführt wird. Hinter dem Platzhalter `_a` steht ein Boost.Phoenix-Funktionsobjekt, das durch geschickte Operatorüberladung erst dann aufgerufen wird, wenn der Parser die in `_a` gespeicherte Regel zusammen mit dem zu interpretierenden Kontext aufruft.

Der Rückgabewert der aufgerufenen Erzeugerregel, wird durch die Aufrufe der Funktion `phoenix::bind` entweder in die Instanzregistry durch den Aufruf `addSolver` eingefügt oder an die oben bereits erwähnte Methode `setRootSolver` weitergegeben. Das *Binden* einer Funktion, was durch den Phoenix-Aufruf geschieht, ist ebenfalls ein Verfahren, um konkrete Funktionen an Argumente zu koppeln und hiermit aufgeschobene (*lazy*) Aufrufe zu ermöglichen. Dafür werden der Funktionszeiger (z.B. `$ConfigGrammar::setRootSolver`), die Instanz (z.B. `*this`) auf der die Funktion aufgerufen werden soll und ggf. die Argumente (z.B. `_1`), die an die aufzurufende Funktion übergeben werden soll, benötigt.

Jede Erzeugerregel gibt nur dann einen Zeiger auf eine Löserinstanz zurück, falls diese ein Löser ist, der die ID *root* hat. Es wird für die Speicherung in der Instanzregistry kein normaler Zeiger der Form `Solver*` verwendet, sondern ein `SolverPtr`. Dies ist eine C++-Typdefinition (`typedef`) auf einen `boost::shared_ptr<Solver>`. Diese gemeinsamen Zeigerklassen, verpacken Zeiger so dass mehrere Zugriffe von unterschiedlichen Objekten möglich sind, ohne dass eins der Objekte die Verantwortung für die Speicherfreigabe (Garbage-Collection) übernimmt. Diese Zeiger werden ebenfalls für Logger, Haltekriteri-

en, Matrizen und Vektoren verwendet, da die Grammatik zwar die Instanzen erzeugt, jedoch nicht weiß, wie lange diese zur Laufzeit existieren müssen ohne, dass Speicherzugriffsfehler entstehen.

Für Haltekriterien werden die Erzeugerklassen direkt aufgerufen. Die Haltekriterien-Erzeugerklasse hat nur eine Regel, die direkt mit der Hauptgrammatik des MetaSolvers verbunden ist. Mit dieser und ihren Subregeln werden die Kriterienbäume durch logische Verknüpfungen der Kriterien erzeugt. Die Implementierung der Haltekriterienregel wird in Abschnitt 6.2 beschrieben.

6.2. Implementierung der Erzeugerregeln

Nachdem nun weitestgehend alle MetaSolver-bezogenen Implementierungen beschrieben wurden, wird in diesem Abschnitt die Umsetzung der vom MetaSolver entkoppelten Erzeugerregeln beschrieben. Begonnen wird zunächst mit der Implementierung der Lösererzeuger, darauf folgen die Haltekriterien- und Loggererzeuger.

Die Lösererzeugerregeln haben keinen Übergabewert und liefern einen `Solver*` zurück. Anhand der im Konzeptabschnitt 5.4 ab Seite 57 vorgestellte Hierarchie der Erzeugerklassen, werden die Lösererzeugerregeln zusammengestellt, um eine bestmögliche Wiederverwendung der Subregeln zu ermöglichen. Die Implementierung der Lösererzeugerregel für die GMRES-Methode wird in Listing 15 gezeigt.

`mRGMRES` ist dabei die Startregel, mit der die Instanz erzeugt und anschließend konfiguriert wird. `mRId` ist eine Regel, die eine ID einliest und diese als `string` zurückgibt. Mit diesem Identifikator wird die Löserinstanz in der semantischen Aktion erzeugt. Die Regel wird dabei in der SolverCreator-Klasse implementiert, auf die am Ende dieses Abschnittes eingegangen wird. `new_` ist eine durch Boost.Phoenix bereitgestellte Funktion, die das Löserobjekt *lazy* also erst zur Parserzeit erzeugt. Über den Templateparameter der Funktion wird der Typ der zu erzeugenden Instanz angegeben.

```
1  mRGMRES = mRId    [ _val = new_<GMRES>( _1 ) ]
2      > '{'
3      > mRIterativeSolver( _val )
4     >> -( lit("krylovDim") > '='
5          > int_     [ phoenix::bind(
6                             &GMRES::setKrylovDim,
7                             *dynamic_cast_<GMRES*>( _val ) ),
8                             _1 )
9                     ]
10         > ';' )
11     > '}';
```

Listing 15: Implementierung der Löser-Erzeuger-Klasse der GMRES-Methode.

Nachdem nun die Löserinstanz erzeugt wurde, werden die Parameter eingelesen. Diese werden durch geschweifte Klammern vom Rest der Konfiguration getrennt. In Zeile 3 kommt die zuvor angesprochene Hierarchie der Erzeugerregeln zum Einsatz. `mRIterativeSolver` ist eine Regel, die in der Klasse IterativeSolverCreator definiert ist und durch alle Lösererzeuger in die eigene Regel integriert wird, die auch einen konkreten iterativen Löser erzeugen. Diese Regel deckt alle Parameter ab, die iterative Löser bereitstellen. Um den Löser konfigurieren zu können, benötigt diese Regel den Zugriff auf dessen Instanz. Deshalb wird der Regel die Instanz über den Platzhalter `_val` als Funktionsparameter übergeben. Auf die Implementierung dieser Regel wird eingegangen, sobald die Erklärung der GMRES-Erzeugerregel abgeschlossen wurde.

Die GMRES-Löserklasse hat einen Konfigurationsparameter *Krylov-Dimension* in Form eines Integerwertes (siehe Anforderung 13). In Zeile 4 wird der dafür definierte Schlüssel und das Zuweisungszeichen eingelesen und in Zeile 5 der Integerwert. In der darauf folgenden semantischen Aktion wird dieser Wert der Löserinstanz über den Methodenaufruf `setKrylovDim` gesetzt. Bevor auf die Instanz des Lösers zugegriffen werden kann, muss diese erst durch eine dynamische Umwandlung (engl. dynamic cast) des Zeigers von der der Klasse `Solver` auf die GMRES-Klasse geschehen, um die Funktionalität der GMRES-Klasse und somit die der Zuweiungsmethode zu erlangen. Dieser Umwandlungsschritt ist nötig, da `val_` als `Solver`-Zeiger in der Regel gespeichert werden muss,

da sonst die Erzeugerregeln einen unterschiedlichen Typ hätten. Dieser muss jedoch gegeben sein, um die Regeln in der Abbildungsdatenstruktur der Hauptgrammatik (`qi::symbol<char, SolverRuleType>`) halten zu können, da diese nur einen Ziel-Datentyp bzw. in diesem Fall einen Regeltyp für alle Einträge erlaubt.

Da die Krylov-Dimension der einzige Parameter ist, der in der `GMRES`-Klasse konfigurierbar ist, wird nun auf die Implementierung der Erzeugerregel für iterative Löser eingegangen, welche in Listing 16 ab Zeile 5 gezeigt wird.

```
1  mRPreconditioner = lit("preconditioner")
2      > '='
3      > mRSolverReference [ _val = _1 ];
4
5  mRIterativeSolver = mRSolver( _r1 )
6     >> -(
7        > mRPreconditioner
8              [ bind( &IterativeSolver::setPreconditioner,
9                      *dynamic_cast_<IterativeSolver*>( _r1 ),
10                     _1 )
11             ]
12       > ';'
13     )
14    >> -( CriteriaCreator::getSolverBoundRule()
15             [ bind( &IterativeSolver::setStoppingCriterion,
16                     *dynamic_cast_<IterativeSolver*>( _r1 ),
17                     _1 )
18             ]
19         );
```

Listing 16: Implementierung der Löser-Erzeuger-Klasse für iterative Löser.

Da die Klasse `IterativeSolver` des Löserframeworks von `Solver` ableitet, muss die Regel `mRIterativeSolver` auch die Parameter einlesen, die in der `Solver`-Klasse definiert sind. Deshalb wird in Zeile 5 die Regel `mRSolver` in die Regel eingebunden, um die Parameter der nächst höheren Hierarchieebene zu unterstützen. Da diese Regel ebenfalls die Löserinstanz benötigt, wird diese durch den Platzhalter `_r1` übergeben. Der Platzhalter entspricht dem ersten –

und einzigen – Argument der aktuellen Regel `mRIterativeSolver`, dem Zeiger auf die GMRES-Instanz in Form eines `Solver`-Zeigers.

Zunächst wird jedoch auf die Implementierung der Präkonditionierung und der lösergebundenen Haltekriteriendefinition eingegangen. Die Regel für die Präkonditionierung wird in Zeile 1 definiert. Sie repräsentiert die Funktionssignatur `SolverPtr()` und hat nur die Aufgabe, den Schlüsselwert *preconditioner* und das Zuweisungszeichen zu interpretieren. Der Zugriff auf die Löser-Instanzregistry wird durch die Regel `mRSolverReference` abgedeckt. Die Implementierung dieser Regel befindet sich in der nächsthöheren Klassenhierarchieebene, der SolverCreator-Klasse, deren Regeln in Listing 17 gezeigt werden. Die Regel `mRSolverReference` benötigt für den Registry-Zugriff die angegebene ID, welche wie bereits in der `mRGMRES`-Regel über die Regel `mRId` eingelesen wird. Diese wird dann für den Aufruf auf die Löser-Instanzregistry verwendet, welche ebenfalls in der SolverFactory implementiert wurde. Dieser liefert anhand der ID den passenden Löser in Form eines `SolverPtr`s zurück. Dieser wird weiter über die Regeln für Löser-Referenzen und Präkonditionierung an die Iterative-Löser-Regel zurückgegeben. Diese setzt den Löser als Präkonditionierer über den Aufruf in Listing 16 in den Zeilen 8-10.

```
1  /* SolverCreator */
2  mRId = char_("a-zA-Z_")      [ _val = _1 ]
3      >> *char_("a-zA-Z_")     [ _val += _1 ];
4
5  SolverFactory& factory = SolverFactory::getFactory();
6
7  mRSolverReference = mRId
8      [ _val = phoenix::bind( &SolverFactory::getSolver,
9                   ref(factory),
10                  _1 )
11     ];
12
13 mRSolver = -( LoggerCreator::getSolverBoundRule()
14     [ phoenix::bind( &Solver::setLogger, *_r1, _1 )]
15             );
```

Listing 17: Implementierung der Basis-Löser-Erzeuger-Klasse.

Nachdem der Präkonditionierer eingelesen wurde, behandelt die Grammatikregel die lösergebundene Haltekriteriendefinition. Der Erzeuger wird in der *Singleton*-Klasse CriteriaCreator implementiert [siehe Gamma, 2004, S. 157ff]. Hier wird die gegebene Konfiguration, wie im Konzept beschrieben, durch die in Listing 9, Seite 61 vorgestellte Grammatik interpretiert. Der MetaSolver greift über die statische Instanz auf die Methode `getIndependentRule()` der Erzeuger-Klasse zu. Die zurückgegebene Regel entspricht der Startregel, die im Konzept beschrieben wurde. Sie liefert nichts an den MetaSolver zurück und organisiert sich selbst über die eigene Instanzregistry, mit der sie auf Instanzen zugreifen und registrieren kann. Abgerufen werden die Instanzen durch die Klasse `IterativeSolverCreator`, die die Haltekriterien mit den iterativen Lösern verknüpft. Die Instanzregistry ist im Wesentlichen ebenfalls nur eine Abbildungsdatenstruktur vom Typ

```
qi::symbols<char, CriterionPtr > mCriterionInstanceMap;
```

welche in der Erzeugerklasse für Haltekriterien gespeichert wird. Diese bietet Methoden, um die Abbildungsdatenstruktur zu füllen und abzufragen.

Eine weitere Möglichkeit, um Haltekriterienvorgaben festzulegen, ist, diese direkt in der Löserkonfiguration zu definieren, ohne dass sie über die ID verknüpft werden. Dies ist sinnvoll, falls das Haltekriterium nicht wiederverwendet werden muss und simple Haltekriteriendefinitionen, wie `IterationCount(10)` mit weniger Schreibaufwand deklariert werden. Hierfür wurde eine zweite Startregel deklariert, die die erzeugte Instanz nicht registriert, sondern direkt an den iterativen Löser zurückgibt und entsprechend keine ID einliest. Diese Regel wird durch die Methode `getSolverBoundRule()` an die Erzeugerklasse für iterative Löser zurückgeliefert.

Die im Konzept angesprochene Abbildung von Schlüsselwörtern auf Aufzählungsdatenstrukturen wird hier nur kurz an dem Beispiel für den Abfragemodus des Residuums in folgendem Listing gezeigt.

```
CriteriaCreator::CriteriaCreator()
{
```

```
    mResidualCheckMode.add
        ( "Relative"    , ResidualThreshold::Relative )
        ( "Absolute"    , ResidualThreshold::Absolute );

   /* Weitere Regel- und Symboldefinitionen */
}
```

Diese Abbildungsdatenstruktur `mResidualCheckMode` ist vom Typ

```
qi::symbols<char, ResidualThreshold::ResidualThresholdCheckMode>
```

Auf die detaillierte Beschreibung der Implementierung der Klasse für die Erzeugung von Loggerinstanzen wird an dieser Stelle verzichtet, da diese der Implementierung der Haltekriterienerzeugerklasse sehr ähnelt und statt der einzulesenden Ausdrücke nur Routinen für direkte Instanzerzeugung von CommonLoggern oder eines FileLoggers bereitstellt. Die Parameter, die die Logger benötigen, 1. die Id, die beim Logging zur wiedererkennung mit ausgegeben wird, 2. das LogLevel, 3. das Schreibverhalten, die ebenfalls durch eine Abbildungdatenstruktur implementiert wurden. Der 4. Parameter ist eigentlich ein Timer, welcher jedoch nicht angegeben werden muss, da hier auf die einzig verfügbare Timerimplementierung OpenMPTimer standardmäßig verwendet wird.

Da eine Konfiguration eines Loggers ähnlich, wie eine Haltekriterienkonfiguration definitiert wird, werden durch die Erzeugerklasse ebenfalls zwei Methoden angeboten, die die Regeln für die unabhängige und die lösergebundene Loggerdefinition zurückgeben.

6.3. Konfigurierbarkeit des SimpleAMGs

Um dem Nutzer des AMG-Lösungsverfahrens eine Möglichkeit zu bieten, die Implementierung zu konfigurieren, werden, wie im Konzept Abschnitt 5.6 auf Seite 62 beschrieben, neben den Kriterien für das Setup auch Zuweisungsmethoden für den Glätter und den Grobgitterlöser implementiert. Die Implementierung des SimpleAMG reicht in der Initialisierung die Instanzen weiter an das

AMGSetup bzw. eine der konkret abgeleiteten Implementierungen, die über eine Umgebungsvariable vorgegeben wurde.

Von der durch `setSmoother(SolverPtr smoother)` gesetzten Glätterinstanz werden im Anschluss Kopien zu den aus dem Setup entstandenen Gittern erzeugt. Die Glätter haben dabei die gleiche Konfiguration, jedoch unterschiedliche Laufzeit-Informationen, wie z.B. die aktuelle Iteration und das Residuum der vorherigen Iteration. In dem Klassendiagramm in Abbildung 6.1 werden die aktuellen Löser-Membervariablen aufgeführt und klarer voneinander getrennt.

Die zustandsabhängigen Informationen werden in separaten `structs` untergebracht, welche in den Klassen der Löser gehalten werden, jedoch nicht mitkopiert werden. Die *SolverRuntime* genannten `structs` haben eigene Konstruktoren, welche die Instanziierung der eigenen Member-Variablen vornehmen und ebenfalls für die Freigabe des Speichers durch die Destruktoren verantwortlich sind. Die Löserklassen beinhalten eigene `structs`, welche entsprechend der Klassenhierarchie des Frameworks voneinander ableiten. Die Instanz der jeweiligen *SolverRuntime* wird in der konkreten Löserklasse, wie z.B. *CG*, *DefaultJacobi* gehalten und nicht in den abstrakten Klassen wie *IterativeSolver* oder Solver. Diese Klassen greifen über eine virtuelle/abstrakte Methode `SolverRuntime* getRunime()` auf die SolverRuntime-Instanz zu, um die eigenen Parameter ändern oder abfragen zu können.

Eine Alternative ist es die Konfigurationsinformationen der Löser so zu trennen, dass Löser-Instanzen nur noch Konfigurationsobjekte über ihren Konstruktor erhalten und sich darüber konfigurieren. So könnte eine Konfigurationsinstanz für mehrere Löser bestehen. Die Löser würden nur noch zustandsabhängige Informationen beinhalten. Diese Implementierung wurde jedoch nicht gewählt, da dies die Konstruktoren und somit die bestehende API des Löser-Frameworks ändern würde.

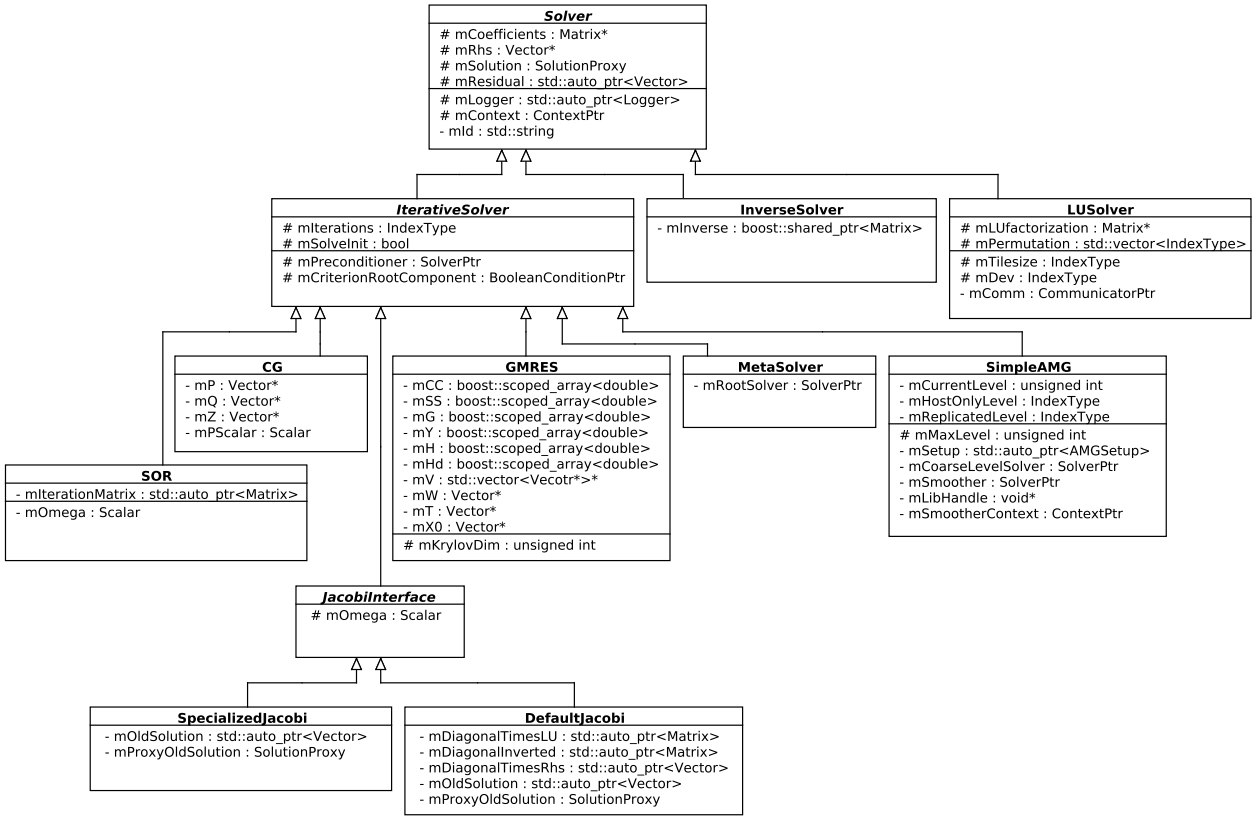

Abbildung 6.1.: Trennung der Member-Variablen in Konfigurationsinformationen und zustandsabhängige Informationen.

6.4. Bereitstellung der Parameter

Laut Anforderung 3 soll besonders auf die Erweiterbarkeit durch neue Parameter oder vollständig neue Löser geachtet werden, ohne dass der Entwickler sich mit der DSL-Entwicklung besonders gut auskennen muss. Hierfür wird in diesem Abschnitt geklärt, wie die Grammatikbeschreibung so vereinfacht werden kann, dass diese Anforderung erfüllt wird. Dazu werden zunächst die wichtigen Teile einer Regel für die Interpretation von Schlüssel-Datenwert-Paaren hervorgehoben. Eine Parameterzuweisung besteht im Detail, wie im Konzeptabschnitt 5.1 definiert, aus einem Schlüssel, einem Zuweisungsoperator, einem Datenwert und einem abschließenden Semikolon, das die aktuelle Parameterbeschreibung abschließt. Die Grammatikregel für die Präkonditionierer und Omega-Zuweisung werden in Listing 18 gezeigt.

```
mRPreconditioner =
    "preconditioner" >> '='
 >> mRSolverReference
        [ bind( &IterativeSolver::setPreconditioner,
                *dynamic_cast_<IterativeSolver*>( _r1 ),
                _1 ) ]
 >> ';';

mROmegaSolver =
    "omega" >> "="
 >> double_
        [ bind( &OmegaSolver::setOmega,
                *dynamic_cast_<OmegaSolver*>( _r1 ),
                _1 ) ]
 >> ';';
```

Listing 18: Zwei Beispiele für Parameterzuweisungen.

Die Informationen der Regeldefinition werden nun in variable und konstante Informationen unterteilt. Variable Bestandteile der Regel sind:

- Der *Schlüsselwert* in Form einer Zeichenkette.
- Der *Datenwert*, der den Typ des Datenwertes beschreibt. Dafür gibt es

zwei Unterscheidungen:
  - Ein *Terminal* wie z.B. `double_`, `int_` oder `char_`.
  - Ein *Nicht-Terminal* in Form einer Regel, wie `mRSolverReference`, die von einer ID auf einen `SolverPtr` über die Instanzregistry abbildet.
- Der *Funktionszeiger*, der auf die Zuweisungsmethode zeigt.
- Die *Löser-Klasse*, auf die die Löser-Instanz zunächst umgewandelt (durch `dynamic_cast_<T>`) werden muss.

Diese variablen Bestandteile der Regel sind für die Schnittstellenbeschreibung relevant, da sie durch den LAMA-Entwickler definiert werden müssen, falls dieser Änderungen an dem Löser-Framework und dessen Konfigurierbarkeit vorgenommen hat. Nun werden noch die Informationen gesammelt, die den Entwickler vorbehalten werden sollen bzw. mit denen er sich nicht beschäftigen muss. Damit ist hauptsächlich die Definition der semantischen Aktion gemeint, die durch die komplizierten, verzögerten Aufrufe schnell zu Fehlern in der Syntax-Beschreibung führen können.

Hierfür wurden Makros geschrieben, die durch den Präprozessor Regeldefinition zusammenstellen lässt. Listing 19 zeigt ein Präprozessor-Makro, was anhand der Angaben zu den variablen Bestandteilen der Regeldefinition eine vollständige Regel generiert. Die zu dem Beispiel in Listing 18 passende Implementierung mit Makro-Aufrufen wird in Listing 20 zum Vergleich gezeigt. Hier wird ersichtlich, dass Makros durch geeignete Datenkapselung (engl. encapsulation oder information hiding) den Code deutlich übersichtlicher für den Entwickler machen können und somit auch Fehler in der Regelbeschreibung vermieden werden.

```
#define LAMA_RULE                                                    \
( key, grammarInput, className, setterFunction )                     \
    key                                                              \
 >> '='                                                              \
 >> grammarInput                                                     \
        [ phoenix::bind(&className::setterFunction,                  \
              *dynamic_cast_<className*>( _r1 ), _1 ) ]              \
 >> lit(';')
```

Listing 19: Ein Präprozessor-Makro zur Generierung der vollständigen Regeldefinition.

```
mRIterativeSolver =
    mRSolver( _r1 )
 >> LAMA_RULE( "preconditioner", mRSolverReference, _r1,
        IterativeSolver, setPreconditioner );
mROmegaSolver =
    mRIterativeSolver( _r1 )
 >> LAMA_RULE( "omega", double_, _r1, OmegaSolver, setOmega );
```

Listing 20: Präprozessor-Makro-Aufrufe, welche Grammatikregeln zur Parameterinterpretation generieren und somit die genauen Aufrufe der Regelbesschreibung vor dem Entwickler verstecken.

6.5. MetaMatrix-, MetaVector- und Strategie-Implementierung

In diesem Abschnitt werden die Implementierungen bezogen auf die Matrix-Vektor-Domäne vorgestellt. Hierbei werden vor allem die Details zur im Konzept vorgestellten Grammatik in Listing 11 auf Seite 69 beschrieben.

Die Klassen MetaMatrix und MetaVector leiten von einer gemeinsamen Klasse ab, die die Regeln für die Interpretation der Features Berechnungsort und Verteilung implementieren. Die MetaKlassen implementieren ihre Startregeln und solche, die anhand der Speichertypangabe eine Instanz dieses Typs selbst erzeugt. Diese Typangabe kann ebenfalls zu einer Strategieklasse passen, wie in Konzeptabschnitt 5.8 auf Seite 69 beschrieben und in folgendem Listing gezeigt.

```
mRType = lit("CSR")    [_val = new_<CSRSparseMatrix<T> >(_r1)]
       | lit("ELL")    [_val = new_<ELLSparseMatrix<T> >(_r1)]
       | lit("DIA")    [_val = new_<DIASparseMatrix<T> >(_r1)]
       | lit("JDS")    [_val = new_<JDSSparseMatrix<T> >(_r1)]
       | lit("COO")    [_val = new_<COOSparseMatrix<T> >(_r1)]
       | lit("Dense") [_val = new_<DenseMatrix<T> >(_r1)]
       | lit("Strategy1") [ _val = new_<Strategy1<T> >(_r1)];
```

Hinter dem Platzhalter `_r1` verbirgt sich die Referenz auf die Matrix, die durch den Nutzer bereitgestellt wird. Damit wird nur der erste im Konzept vorgestellte Anwendungsfall abgedeckt. Für den zweiten Anwendungsfall, bei dem der Nutzer nur einen Dateinamen angibt, muss er zunächst eine LAMA-Matrix einlesen, die dann als Eingabematrix für den MetaSolver verwendet wird. Die Fließkommazahlgenauigkeit, die bei der Erzeugung durch den Templateparameter `T` vorgegeben wird, entspricht der der Eingabematrix.

Als nächsten Schritt werden, die Berechnungsorte, wie im Konzeptkapitel beschrieben, eingelesen. Die dazugehörigen Implementierungen werden anhand des Listings 21 genauer beschrieben. Der Nutzer hat durch `mRContextDef` die

Wahl, ob er zwei Berechnungsorte oder nur Einen für beide Speicherbereiche (halo oder lokal) vorgeben möchte.

Nachdem der Schlüssel und das Zuweisungszeichen interpretiert wurden, kommt die Regel `mRContext` zu Einsatz, welche dazu verwendet wird die untergeordnete Regel `mRContextManager` aufzurufen. Die auf einem Rechenknoten verfügbaren Contextinstanzen werden in ContextManager-Klassen gehalten. Jeder ContextManager kann mehrere Context-Instanzen eines Typs haben, da beispielsweise mehrere Grafikkarten je Rechenknoten verfügbar sind. Abgerufen werden die konkreten Berechnungsorte über ihre *Gerätenummer* auf den ContextManagern über die Methode `getContext`. Falls keine Gerätenummer angegeben wird, wird über eine in der LAMA-Bibliothek definierte Präprozessorvariable `LAMA_DEFAULT_DEVICE_NUMBER` das Standardgerät verwendet.

ContextManager können über die ContextFactory-Klasse anhand des Berechnungsorttyps abgerufen werden (siehe semantische Aktion zu `mRContextManager`). Der Berechnungsorttyp wird durch einen C++-Aufzählungstyp repräsentiert, auf den durch eine `qi::symbols`-Datenstruktur abgebildet wird (siehe in Zeile 26).

Die Implementierung der Verteilungsangabe(n) wurde, wie im Konzept beschrieben, mit Angabe der Kommunikator-Klassen und der optional anzugebenden Blockgröße für die blockzyklische Verteilung umgesetzt. Auf die Beschreibung der Implementierung wird in diesem Abschnitt verzichtet, da hier keine Abweichungen zum Konzept bestehen und die Regelimplementierungen, wie in Listing 11 auf Seite 11 beschrieben realisiert wurden.

Eine SimpleStorageStrategie-Klasse wurde für die Implementierung einer Beispielstrategie implementiert, die anhand des später gesetzten Berechnungsortes entscheidet, welche Matrix intern erzeugt wird. Ein einfacher Entscheidungsmechanismus, der in der Klasse implementiert wurde, legt fest, dass die intern erzeugte Matrix vom Typ ELLSparseMatrix ist, wenn der Berechnungsort auf `Context::CUDA` gesetzt wurde und ansonsten eine CSRSparseMatrix verwendet, um Matrixtypen einzusetzten, die für den jeweiligen Berechnungsort

```
mRContextDef =
 ( lit("context") > lit('=') > mRContext > ';' )
        [ phoenix::bind( &Matrix::setContext, _r1, _1 ) ]
 |
 ( lit("localContext") > lit('=') > mRContext > ';'
 > lit("haloContext") > lit('=') > mRContext > ';' )
        [ phoenix::bind( &Matrix::setContext, _r1, _1, _2 ) ];

mRContext =
    mRContextManager
        [ _val = phoenix::bind(
                    &ContextManager::getContext, _1,
                    LAMA_DEFAULT_DEVICE_NUMBER )
        ]
 | ( mRContextManager > '(' > int_ > ')' )
        [ _val = phoenix::bind( &ContextManager::getContext,
                    _1, _2 )
        ];

mRContextManager = mRContextMap
        [ _val = phoenix::bind(
                    &ContextFactory::getContextManager,
                    ContextFactory::getFactory(), _1 )
        ];

mRContextMap.add
        ("CUDA", Context::CUDA)
        ("Host", Context::Host)
```

Listing 21: Implementierung der berechnungsortbezogenen Regelimplementierungen.

üblich sind. Die Strategieklasse kann wie eine gewöhnliche SparseMatrix eingesetzt werden, da sie die Methodenaufrufe, wie eine MetaMatrix an ihre intern erzeugte Matrix weiterleitet.

Durch die verzögerte Erzeugung der Matrixinstanz, muss immer unterschieden werden, ob ein Aufruf auf die Strategieklasse konfigurierend und somit noch wichtig für die Entscheidung ist, oder ob ein Aufruf für eine Berechnung verwendet wird. Falls ein Aufruf mathematischen Bezug hat, muss an dieser Stelle die Entscheidung über den Typen anhand der aktuellen Konfiguration getroffen werden und anschließend die interne Matrix erzeugt und konfiguriert werden. Zu den konfigurierenden Methoden gehören beispielsweise `setContext` und `redistribute`. Mathematische Methoden sind beispielsweise `matrixTimesVector`, die eine Matrix-Vektor-Multiplikation berechnet. Findet also ein solcher Berechnungsaufruf statt, so wird geprüft, ob die innere Matrix existiert. Ist dies nicht der Fall, so wird die Matrix in der Methode `applyStrategy` erzeugt und im Anschluss der Aufruf weitergeleitet. Diese Methode wird öffentlich (*public*) implementiert, sodass sie auch manuell durch den Anwender aufgerufen werden kann und er somit die Kontrolle darüber hat, wann die Matrix erzeugt wird.

Da Strategien auf diese Weise nur in der `mRType`-Regel eingebunden werden und ansonsten unabhängig von Boost.Spirit und der damit verbundenen Grammatikdefinition sind, ist die Erweiterbarkeit im Hinblick auf weitere komplexe Strategien gegeben.

In diesem Kapitel wurde beschrieben, wie bei der Implementierung vorgegangen wurde. Zunächst wurde vorgestellt, wie die Erzeugerklassen der Löser an den MetaSolver gebunden werden können, ohne dass dieser wissen muss, wie die Parameter zu interpretieren sind. Bewerkstelligt wurde dies zunächst durch die Implementierung von kleinen Sub-Grammatiken. In einer zweiten Version wurde dies jedoch so umgeschrieben, dass die Löser-Erzeugerklassen nur Grammatikregeln bereitstellen, die zur Laufzeit in die Hauptgrammatik des MetaSolvers eingebunden werden können und somit der zu interpretierende

Kontext direkt weitergegeben werden kann, ohne ihn zuvor in eine gesonderte Zeichenkette umwandeln zu müssen. Dabei wurden die Details der Grammatikbeschreibungen erläutert. Vor allem verzögerte Funktionsaufrufe und Attributauswertungen sind dabei Techniken, die in der üblichen Programmierung eher selten Anwendung finden, hier jedoch eine wichtige Rolle spielen. Neben diesen Mitteln werden Spirit Funktionsobjekte, die zur Laufzeit erzeugt und auf den Eingabekontext angewandt werden können, verwendet. Hierzu musste zu jeder Regel die entsprechende Funktionssignatur vorgegeben werden, dessen Attribute dann innerhalb der Regel abgerufen und gesetzt werden konnten.

Für die Konfigurierbarkeit des AMG-Verfahrens, musste ein Weg gefunden werden, die bestehenden Löser der LAMA-Bibliothek kopierbar zu machen. Dazu wurde genau ermittelt, welche Informationen sich während der Lösungsphase eines Löser ändern. Diese wurden dann von den Konfigurationsangaben getrennt ohne dabei die bestehenden Schnittstellen der Löser zu ändern. Durch die Kopierbarkeit der Löser ist es nun möglich einen bereits konfigurierten Glätter zu definieren, der dann in der Setup-Phase der AMG-Implementierungen für jedes erzeugte Gitter kopiert werden kann.

In dem letzten Abschnitt dieses Kapitels wurde die Implementierung der Matrix-Vektor-Domäne vorgestellt, bei der die Strategieimplementierung im Vordergrund stand. Hierfür musste ein Weg gefunden werden, zunächst alle für die Strategieentscheidung relevanten Informationen zu sammeln und sie dann zum richtigen Zeitpunkt anzuwenden.

7. Analyse der Ergebnisse

7.1. Testen der Implementierung

In diesem Abschnitt wird beschrieben, wie die Implementierungen der DSL getestet werden. Die Bestandteile der LAMA-Bibliothek werden in einem gesonderten Framework getestet, welches mit Boost.Test entwickelt wurde [Rozental, 2012]. Der MetaSolver wird getestet indem die Ergebnisse von nativen Implementierungen – also herkömmlicher Instanziierung und Konfiguration – von Löser-Verknüpfungen und Haltekriterien mit den nachgebauten Konfigurationen in der DSL verglichen wurden. Erstellt wurden Konfigurationen für alle Löser und deren Konfigurationsmöglichkeiten, für Haltekriterien-Verknüpfung und Löser-Verknüpfung. Die Löser-Verknüpfung wird durch die Konfigurationen des CG und des SimpleAMG abgedeckt. Die Testklasse `MetaSolverTest` beinhaltet mehrere Methoden, welche einzelne Testphasen beschreiben.

Zuerst werden alle nötigen Vorbereitungen im Konstruktor der Testklasse vorgenommen. Hier werden die nötigen Matrizen und die zwei Vektoren für die Lösung und die rechte Seite erzeugt und initialisiert. Der Konstruktor verlangt die Angabe der Kofigurationseingabe und des Zeigers auf die Löserinstanz, der nativen Implementierung, um für die darauf folgenden Phasen zur Verfügung zu stehen. Die nächste Phase schließt einen Konstruktortest und einen Konfigurationstest (Interpretation) ein. Hier wird die Instanz des MetaSolvers erzeugt und über die Konfigurationsmethode die Konfiguration gelesen und die Löser anhand der Konfigurationsbeschreibung erzeugt. Darauf folgen in den zwei darauf folgenden Phasen die Initialisierung und der Lösungsprozess des MetaSolvers. In der letzten Phase wird die native Implementierung ausgeführt (Initialisierung und Lösungsphase) und das Ergebnis über die L^2-Normen der

zwei Lösungsvektoren verglichen.

Ein Beispiel für die Implementierung eines einzigen Testfalls wird in Listing 22 gezeigt. Es stellt den Testaufruf des MetaSolvers für eine SOR-Implementierung mit dem Parameter *omega* vor. Die Variable `configuration` beinhaltet die Konfigurationsbeschreibung. Hinter der Konfigurationsbeschreibung wird die native Implementierung definiert und beide Werte dem Konstruktor der Testklasse übergeben.

```
BOOST_AUTO_TEST_CASE( metaSolverTestWithSOR )
{
    std::string configuration = "SOR root{omega=1.5;}";
    SOR* sor = new SOR( "SOR_Native" );
    sor->setOmega( Scalar( 1.5 ) );
    SolverPtr nativeImpl( sor );
    MetaSolverTestImpl metaSolverTestWithSOR(
        configuration, nativeImpl );
    metaSolverTestWithSOR.runTests();
}
```

Listing 22: Aufruf des MetaSolvers in den LAMA-Tests.

Um innerhalb der Methoden die Implementierungen zu testen, wurden verschiedene LAMA_ASSERT-Makros eingesetzt, die Mechanismen zur Fehlerbehandlung durch Bedingungen implementieren, wie sie in der Java-Entwicklung bekannt sind. Die getesteten Methoden gehören der Fabrikklasse, den Meta- und Erzeugerklassen an.

7.2. Interpreter-Performanz

Besonders interessant für die LAMA-Bibliothek ist die Interpretationsgeschwindigkeit, da diese für performantes Rechnen entwickelt wurde und dieses Kriterium ein wichtiger Faktor für das Ergebnis dieser Arbeit ist.

Szenario\Zeit	MetaSolver	Nativ	Matrix 1	Matrix 2	Matrix 3
1	44,61 µs	2,08 µs			
2	80,64 µs	6,52 µs	64,19 ms	18,26 ms	90,27 ms
3	93,11 µs	8,36 µs	14,17 ms	42,9 ms	129,76 ms
4	152,28 µs	14,89 µs	22,11 ms	61,34 ms	310,71 ms

Tabelle 7.2.: Laufzeiten der Instanziierungen und Konfigurationen von vier verschieden Szenarien.

	Intel Xeon 5650
Kerne	6
Taktung	2.67 GHz
Speichergeschwindigkeit	800 MHz
Speicherbandbreite	32 GB/s

Tabelle 7.1.: Spezifikation für den Intel Xeon 5650 [Intel, 2012].

Die Geschwindigkeitsmessungen werden auf einem gesonderten leistungsstarken Rechner mit einer ausgeführt, da dies dem typischen Anwendungsfall entspricht. Dessen zusammengefasste Spezifikation findet sich in Tabelle 7.1. Für die Performanztests wurden die folgenden vier Konfigurationen für den MetaSolver getestet:

1. Ein CG ohne Parameter.
2. Ein CG mit einem SOR als Präkondtionierer. Das SOR-Verfahren erhält einen Omegawert.
3. Ein CG mit dem Haltekriterium (IterationCount(10) AND ResidualThreshold(L2Norm, 0.0001, Absolute)).
4. Ein CG präkonditioniert mit vier über den Grobgitterlöser aneinandergeketteten SimpleAMGs, die jeweils unabhängige Glätter (SOR) haben. Der letzte SimpleAMG erhält als Grobgitterlöser einen InverseSolver.

Von diesen vier Szenarien wird immer die gemessene Konfigurationszeit bzw. Parserzeit exklusive den Dateioperationen und zum Vergleich werden die Laufzeit der dazu passenden nativen Implementierungen angegeben.

Des Weiteren wurden Messungen mit den konfigurierten Lösern durchgeführt. Hierbei wurde die Lösungphase gemessen. Für diese Messung wurden drei verschieden große dünnbesestzte Matrizen im CSR-Speicherformat verwendet. Die Ergebnisse werden in Tabelle 7.2 zusammengefasst. Die für die Messung verwendeten Matrizen 1-3 haben dabei die folgenden Eigenschaften:

Matrix 1 ist eine 500x500 große Matrix mit circa 1000 Nicht-Null-Elementen.
Matrix 2 ist eine 1000x1000 große Matrix mit circa 5000 Nicht-Null-Elementen.
Matrix 3 ist eine 2000x2000 große Matrix mit circa 10000 Nicht-Null-Elementen.

Die Instanziierungs- und die Konfigurationszeit wird durch den MetaSolver im Verhältnis stark vergrößert. Allerdings befinden sich diese Werte im Mikrosekundenbereich und wirken sich kaum auf die Gesamtlaufzeit aus. Gegenüber der Laufzeit der Lösungsphasen fällt dies bereits bei kleinen Matrizen mit nur wenigen Nicht-Null-Elementen auf. Hier liegt der Anteil der Konfigurationszeit für das Szenario 4 bei unter 0 7%. Für Szenario 2 mit der kleinsten Matrix liegt der Konfigurationsanteil bei unter 0 13%. Insgesamt ist fällt das Ergebnis der Performanzmessungen positiv aus.

8. Zusammenfassung & Ausblick

Ziel der Arbeit war es eine Konfigurationssprache zu entwickeln, die es dem Anwender ermöglicht Löser ausgerichtet auf seinen Anwendungsfall zu konfigurieren und untereinander verknüpfen zu können. Ein weiteres wichtiges Ziel der Arbeit war es die Konfigurationsbeschreibung zur Laufzeit einlesen und anwenden zu können, sodass für den Nutzer kein zeitaufwändiges Neukompilieren nötig ist. Diese Sprache sollte innerhalb seines eingegrenzten Problemfeldes möglichst einfach erweiterbar, verständlich und benutzerfreundlich gestaltet werden.

Zunächst wurden die wichtigsten Elemente und Zusammenhänge des Bestands-Projektes LAMA erläutert, um die Basis und somit das Umfeld der Domäne für die folgende Arbeit zu vermitteln. Anschließend wurden Ansätze und Grundlagen zum Thema DSL-Entwicklung vorgestellt, um einen Überblick über die Vorgehensweise in den Folgekapiteln zu verschaffen. Die darauf folgende Anforderungenerhebung diente als Basis der Domänenanalyse, bei der das Problemfeld genau abgegrenzt und untersucht wurde. Kriterien, die sich ebenfalls aus den Anforderungen ergaben, wurden aufgestellt, um Realisierungsmöglichkeiten miteinander vergleichen und eine Auswahl treffen zu können. Dabei wurden drei Möglichkeiten aus unterschiedlichen Implementierungsmustern verglichen. Antlr als Parsergenerator-Variante, XML-Parser als COTS-Variante und Boost.Spirit mit einer DSEL-Variante. Gewählt wurde Boost.Spirit, da sich die Bibliothek gut in den aktuellen Entwicklungsprozess integrieren ließ, keine Einschränkungen für das LAMA-Projekt entstanden und sie sich damit gegenüber der Alternativansätze absetzen konnte.

Boost.Spirit verwendet Techniken, wie Expression-Templates und Template-Meta-Programmierung, um die Grammatikdefinitionen in einer EBNF ähnli-

chen Notation eingebettet in C++ beschreiben und Attribute dieser Beschreibungen behandeln zu können. Die Bibliothek Boost.Phoenix wurde in den zu den Nicht-Terminalen und Terminalen passenden semantischen Aktionen der Grammatik eingesetzt, um verzögerte (lazy) Funktions- und Methodenaufrufe durchzuführen, wie beispielsweise Zuweisungsoperationen und Instanzerzeugungen.

Nach der Auswahl, wurde zunächst die konkrete Syntax für die Konfigurationssprache festgelegt, die im weiteren Verlauf des Konzeptes zusammen mit den semantischen Aspekten ausgebaut wurde. Diese Syntax lehnt entsprechend des Zielgruppenwissens an die Syntax der OpenFOAM-Konfigurationssprache Software an. Als Benutzerschnittstelle wurde dazu eine Klasse MetaSolver entwickelt, die wie ein gewöhnlicher Löser eingesetzt werden kann, jedoch zusätzlich Methoden bereitstellt, Konfigurationsbeschreibungen entgegenzunehmen und durch die Grammatik interpretieren zu lassen. Implementiert wurde die Grammatik so, dass sie im Falle von Änderungen oder Erweiterungen des Löser-Frameworks einfach anzupassen ist. Für die Erweiterbarkeit wurden vom MetaSolver entkoppelte Erzeugerklassen implementiert, die sich bei einer Fabrikklasse registrieren und somit durch den MetaSolver je nach Bedarf dynamisch in die Grammatik eingebettet werden können. Damit sich die Entwickler der LAMA-Bibliothek bei Änderungen nicht mit den Details der Grammatikbeschreibung befassen müssen, wurden Makros entwickelt, die ebenfalls wie eine DSL nur die variablen Angaben zur Schnittstellenbeschreibung entgegennehmen und zur Übersetzungszeit die Grammatikbeschreibung erzeugen, ohne Laufzeiteinschränkungen mit sich zu bringen.

Um Löser, Haltekriterien und Logger untereinander verknüpfen zu können, wurden Instanzregistries implementiert. Die Erzeugerklassen können die erzeugten Instanzen der Registry hinzufügen und ggf. über ihre ID abrufen, um die Verknüpfungen herzustellen.

Für die Konfigurationsmöglichkeit des AMG-Lösers musste eine Trennung von Konfigurations- und Laufzeitparametern vorgenommen werden, was im bestehenden Design der LAMA-Bibliothek nicht vorgesehen war.

Durch die Anbindungsmöglichkeit von Strategien und der Festlegung von Standardwerten, sollen dem Nutzer schwierige Entscheidungen abgenommen werden. Strategien treffen dabei situationsabhängige Entscheidungen. Durch die Strategien für Speicherformate von Matrizen und Vektoren sowie für Verteilungen und Berechnungsorte entstand eine zweite Domäne, welche durch die Klassen MetaMatrix und MetaVector abgedeckt wird. Implementiert wurde eine Beispielstrategie, die anhand des gesetzten Berechnungsortes einen geeigneten Typ für die intern zu erzeugende Matrix auswählt.

Mit den in LAMA implementierten Expressions wurden Spracherweiterung durch die Mischung aus templatebasierter Entwicklung und Operatorüberladung umgesetzt, die die Aufrufe mathematischer Routinen vereinfacht. Mit der Entwicklung der domänenspezifischen Sprache wurde in dieser Arbeit eine Abstraktion der Löserkonfiguration geschaffen. Durch die flexible Verknüpfbarkeit der Löser untereinander werden die Möglichkeiten für den Nutzer gegenüber der starren Verknüpfung der gängigen Anwendungen PETSc und OpenFOAM stark verbessert. Gleiches gilt für die Definition der Haltekriterien, welche durch eine kompakte Schreibweise festgelegt und zusammengestellt werden können. Insgesamt obligt wesentlich weniger Implementierungsaufwand und damit benötigte API-Kenntnis beim Nutzer; zudem ist der Umgang ohne Neukompilieren für ihn gebräuchlicher.

In Zukunft ist eine Entwicklung einer benutzerfreundlichen Graphical User Interface (GUI) denkbar, mit der die einzelnen Löser durch Drag&Drop zusammengesetzt und zusätzliche Einstellungen in Parameterlisten vorgenommen werden können. Außerdem soll mit Hilfe der entwickelten DSL eine größere Logik für die Strategien entwickelt werden.

Glossar

Domänen-Modell

Das Domänen-Modell wird in der Domänen-Analyse der DSL-Entwicklung erstellt, um die Grenzen und die Domänen-Terminologie der Domäne zu spezifizieren. Die Terminologie beinhaltet das Vokabular und Regeln, die beschreiben, wie Sprachkonstrukte aufgebaut werden.

Feature

Ein Softwaremerkmal (Feature) repräsentiert nach Riebisch einen für den Kunden wertvollen Aspekt, welcher sich aus den Anforderungen ergibt [siehe Riebisch, 2003, S. 69].

Featuremodell

Ein Featuremodell ist ein Modell, das optionale und erforderliche Features in einer Domäne anhand der gesammelten Anforderungen voneinander abgrenzt. Im Detail besteht ein solches Featuremodell aus: einem Featurediagramm, welches eine hierarchische Zerlegung der Features und ihres Charakters (erforderlich, alternativ oder optional) darstellt; der semantischen Definition der Features und Feature-Kompositions-Regeln, die festlegen welche Feature-Verknüpfungen erlaubt oder verboten sind.

InfiniBand

InfiniBand ist ein Industrie-Standard verschiedener Hardware-Hersteller, der I/O-Architekturen zur Verbindung von verteilten Systemen beschreibt. Dahinter verbirgt sich eine serielle Hochgeschwindigkeitsübertragungstechnologie mit bis zu 120 Gigabit/s. InfiniBand wird vorwiegend für die Verbindung von Clustern verwendet.

Lexer

Ein Lexer (auch Scanner oder Tokenizer) hat die Aufgabe, das Eingabesprachkonstrukt in Tokens aufzuteilen. Tokens werden durch Trennzeichen – üblicherweise das Leerzeichen – gebildet. Lexer arbeiten auf Zeichenbasis und ermitteln beispielsweise, ob ein Variablenname nur aus Großbuchstaben besteht und somit die richtige Form hat.

Myrinet

Myrinet ist eine Hochgeschwindikeitsübertragungstechnologie für Computer-Cluster, die von der Firma Myricom entwickelt wurde. Sie bietet die Vorteile von kurzen Latenzzeiten, geringem Protokolloverhead und hohen Datenraten. Zu dem ist sie auf Hardwareebene kompatibel zu dem weit verbreiteten Ethernet-Standard.

Norm

Die Norm wird in der mathematischen Numerik zur Beschreibung der Größe von Vektoren oder Matrizen verwendet. Wie diese Größe ermittelt wird, ist von der gewählten Norm abhängig. In LAMA implementiert wurden die Maximumsnorm (auch Tschebyschew-Norm), die L^1-Norm und die L^2-Norm. Die L^2-Norm wird beispielsweise über die Wurzel des Skalarproduktes des Vorgabe-Vektors mit sich selbst ermittelt. Für weitere Details zu den Normberechnung wird auf die Literatur verwiesen [siehe Königsberger, 2004, S. 11ff].

Ontologie

Durch die Ontologie werden Regeln für den Aufbau von Sprachkonstrukten und somit die formalisierte Struktur der Domäne (Metamodell) beschrieben.

Parser

Parser haben die Aufgabe, die durch Lexer interpretierten Token auf ein Modell abzubilden, welches eine Instanz eines Metamodells ist. Ein Metamodell gibt eine syntaktische Struktur vor und hiermit auch, wie ein Modell auszusehen hat. Damit kann der Parser ermitteln, ob die konkrete Eingabe valide ist. Die Struktur bzw. die abstrakte Syntax wird meist durch einen AST beschrieben. Mit dem Modell wird dann weiter in der

Zielplattform gearbeitet. Die Bedeutung eines Ausdrucks wird durch die Semantik festgelegt. Nach dem Einlesen eines Ausdrucks werden abhängig von diesem semantische Aktionen ausgeführt, wie z.B. Funktionsaufrufe in der Zielsprache. Interpreter und Compiler verwenden beide einen Parser für die Überführung von konkreter Syntax in eine Objektstruktur.

Präkonditionierung

Unter Präkonditionierung eines Gleichungssystems versteht man die Links- und/oder Rechtsmultiplikation mit einer Matrix. Ziel dabei ist es ein äquivalentes Gleichungssystem mit besseren Lösungseigenschaften, d.h. einer besseren Konditionierung, zu erhalten. Im Kontext von iterativen Lösern, wie z.B. dem CG-Verfahren, entspricht diese Präkonditioniermatrix dann genau dem linearen Operator eines Lösungsschrittes mit dem entsprechenden Präkontitionierer, wie beispielsweise Jacobi oder AMG.

Residuum

Das Residuum beschreibt die Abweichung von der optimalen Lösung in der Numerik. Für ein lineares Gleichungssystem $Ax = b$ wird das Residuum folgendermaßen berechnet: $r = Ax - b$, wobei x die Lösung des zuletzt berechneten Schrittes eines iterativen Verfahrens ist.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[Abrahams und Gurtovoy 2004] Abrahams, David ; Gurtovoy, Aleksey: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004 (C++ in-depth series). – ISBN 9780321227256

[ANTLR 2012a] ANTLR: *ANTLR Parser Generator*. `http://www.antlr.org/about.html`. 24. Oktober 2012

[ANTLR 2012b] ANTLR: *ANTLR v3 Documentation*. `http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation`. 24. Oktober 2012

[Balay u. a. 2004] Balay, Satish ; Buschelman, Kris ; Eijkhout, Victor ; Gropp, William D. ; Kaushik, Dinesh ; Knepley, Matthew G. ; McInnes, Lois C. ; Smith, Barry F. ; Zhang, Hong: PETSc Users Manual Revision 3.2 / Argonne National Laboratory. 2004. – Forschungsbericht

[Barrett u. a. 1994] Barrett, R. ; Berry, M. ; Chan, T. F. ; Demmel, J. ; Donato, J. ; Dongarra, J. ; Eijkhout, V. ; Pozo, R. ; Romine, C. ; Vorst, H. V. der: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA : SIAM, 1994

[Bell u. a. 1994] Bell, Jeffrey M. ; Bellegarde, Francoise ; Hook, James ; Kieburtz, Richard B. ; Kotov, Alex ; Lewis, Jeffrey ; McKinney, Laura ; Oliva, Dino ; Sheard, Tim ; Tong, L. ; Walton, Lisa ; Zhou, Tong: Software design for reliability and reuse: a proof-of-concept demonstration. In: *TRI-Ada*, URL `http://dblp.uni-trier.de/db/conf/sigada/triada94.html#BellBHKKLMOSTWZ94`, 1994, S. 396–404

[CUDA 2012] CUDA: *Weltweiter Marktführer für Visual Computing Technologien | NVIDIA*. http://www.nvidia.de/page/home.html. 19. Oktober 2012

[Czarnecki und Eisenecker 2000] Czarnecki, Krysztof ; Eisenecker, Ulrich: *Generative Programming: Methods, Tools, and Applications*. Boston, MA : Addison-Wesley, 2000. – ISBN 978-0-201-30977-5

[EduTechWiki 2012] EduTechWiki: *XML editor - EduTech Wiki*. `http://edutechwiki.unige.ch/en/XML_editor`. 22. Oktober 2012

[Förster und Kraus 2011] Förster, Malte ; Kraus, Jiri: Scalable parallel AMG on ccNUMA machines with OpenMP. In: *Computer Science - R&D* 26 (2011), Nr. 3-4, S. 221–228. – URL `http://dblp.uni-trier.de/db/journals/ife/ife26.html#ForsterK11`

[Ford 2004] Ford, Bryan: Parsing expression grammars: a recognition-based syntactic foundation. In: Jones, Neil D. (Hrsg.) ; Leroy, Xavier (Hrsg.): *POPL*, ACM, 2004, S. 111–122. – URL `http://pdos.csail.mit.edu/~baford/packrat/popl04/`

[Fowler 2012] Fowler, M.: *SyntacticNoise*. `http://martinfowler.com/bliki/SyntacticNoise.html`. 14. Oktober 2012

[Fowler und Parsons 2011] Fowler, Martin ; Parsons, Rebecca: *Domain-Specific Languages*. Upper Saddle River, N.J. : Addison-Wesley, 2011. – URL `http://www.worldcat.org/search?qt=worldcat_org_all&q=9780321712943`. – ISBN 9780321712943 0321712943

[Gamma 2004] Gamma, E.: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. München : Addison-Wesley, 2004 (Professionelle Softwareentwicklung). – URL `http://books.google.de/books?id=-GXxUV0L6XsC`. – ISBN 9783827321992

[Ghosh 2010] Ghosh, Debasish: *DSLs in Action*. Pap/Psc. Greenwich, CT, USA : Manning Publications, Dezember 2010. – ISBN 1935182455

[Griebel u. a. 1998] Griebel, Michael ; Dornseifer, Thomas ; Neunhoeffer, Tilman: *Numerical simulation in fluid dynamics: a practical introduction*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1998. – ISBN 0-89871-398-6

[de Guzman und Kaiser 2011] Guzman, Joel de ; Kaiser, Hartmut: *Spirit 2.5 - Documentation*. `http://boost-spirit.com/dl_docs/spirit2_5.pdf`. 2011

[de Guzman und Kaiser 2012] Guzman, Joel de ; Kaiser, Hartmut: *Nabialek trick*. `http://boost-spirit.com/home/articles/qi-example/nabialek-trick/`. 17. Oktober 2012

[InfiniBand 2012] InfiniBand: *InfiniBand Trade Association: Home*. `http://www.infinibandta.org/`. 24. September 2012

[Intel 2012] Intel: *ARK | Intel Xeon Processor x5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel QPI)*. `http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-(12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI)`. 24. Oktober 2012

[ISO/IEC JTC 1 1996] ISO/IEC JTC 1: *14977 Syntactic metalanguage – Extended BNF (1^{st} Edition)*. Februar 1996. – JTC – Joint Technical Committee, `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip` – last visited 15^{th} January 2009

[Jaxe 2012] Jaxe: *Jaxe, your XML editor*. `http://jaxe.sourceforge.net/`. 13. Oktober 2012

[Kang u. a. 1990] Kang, K. C. ; Cohen, S. G. ; Hess, J. A. ; Novak, W. E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. November 1990. – Forschungsbericht

[Königsberger 2004] Königsberger, K.: *Analysis 2*. Springer, 2004 (Springer-Lehrbuch). – ISBN 9783540203896

[Kanzow 2004] Kanzow, C.: *Numerik linearer Gleichungssysteme: Direkte und iterative Verfahren*. Berlin : Springer, 2004 (Springer-Lehrbuch). – URL http://books.google.de/books?id=Y3VdWoQgIMgC. – ISBN 9783540206545

[LibAMA 2012] LibAMA: *LAMA - Library for Accelerated Mathematical Applications: Overview*. http://www.libama.org. 24. September 2012

[Libxml2 2012] Libxml2: *The XML C parser and toolkit of Gnome*. http://xmlsoft.org/. 17. Oktober 2012

[Martin-Vide 2010] Martin-Vide, C.: *Scientific Applications of Language Methods*. Imperial College Press, 2010 (Mathematics, Computing, Language, and Life: Frontiers in Matematical Linguistics and Language Theory). – ISBN 9781848165441

[Matrix-Market 2012] Matrix-Market: *Matrix Market: File Format*. http://math.nist.gov/MatrixMarket/formats.html. 16. Oktober 2012

[Maven 2012] Maven: *Maven - Welcome to Apache Maven*. http://maven.apache.org/. 17. Oktober 2012

[Mernik u. a. 2005] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), December, S. 316–344. – URL http://doi.acm.org/10.1145/1118890.1118892. – ISSN 0360-0300

[Mizushima u. a. 2010] Mizushima, Kota ; Maeda, Atusi ; Yamaguchi, Yoshinori: Packrat parsers can handle practical grammars in mostly constant space. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA : ACM, 2010 (PASTE '10), S. 29–36. – URL http://doi.acm.org/10.1145/1806672.1806679. – ISBN 978-1-4503-0082-7

[Musy u. a. 2009] Musy, Francois ; Nicolas, Laurent ; Perrussel, Ronan: Agglomeration-based algebraic multigrid for linear systems coming from

edge-element discretizations. In: *Institut Camille Jordan, Laboratoire Ampere, Universite de Lyon* (2009)

[Myricom 2012] Myricom: *Home*. http://www.myricom.com/. 24. September 2012

[Netlib 2012] Netlib: *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/. 31. August 2012

[OpenFOAM 2012a] OpenFOAM: *OpenFOAM - The Open Source Computational Fluid Dynamics (CFD) Toolbox*. http://www.openfoam.com/. 24. September 2012

[OpenFOAM 2012b] OpenFOAM: *Solution and algorithm control*. http://www.openfoam.org/docs/user/fvSolution.php. 17. Oktober 2012

[OpenMP 2012] OpenMP: *OpenMP.org*. http://openmp.org/wp/. 19. Oktober 2012

[Parr 2007] Parr, Terence: *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Raleigh, NC : Pragmatic Bookshelf, 2007. – ISBN 978-0-9787-3925-6

[PETSc 2012] PETSc: *PETSc: Home Page*. http://www.mcs.anl.gov/petsc/. 19. Oktober 2012

[PGAS 2012] PGAS: *PGAS - Partitioned Global Address Space Languages*. http://www.pgas.org/. 19. Oktober 2012

[Riebisch 2003] Riebisch, Matthias: Towards a More Precise Definition of Feature Models. In: Riebisch, M. (Hrsg.) ; Coplien, J. O. (Hrsg.) ; Streitferdt, D. (Hrsg.): *Modelling Variability for Object-Oriented Product Lines*. Norderstedt : BookOnDemand Publ. Co, 2003, S. 64–76. – URL http://www.theoinf.tu-ilmenau.de/~riebisch/home/publ/06-riebisch.pdf

[Rozental 2012] Rozental, Gennadiy: *Boost Test Library*. http://www.boost.org/doc/libs/1_51_0/libs/test/doc/html/index.html. 1. Oktober 2012

[Saad und Schultz 1986] Saad, Youcef ; Schultz, Martin H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. In: *SIAM J. Sci. Stat. Comput.* 7 (1986), Juli, Nr. 3, S. 856–869. – URL `http://dx.doi.org/10.1137/0907058`. – ISSN 0196-5204

[Saad 2003] Saad, Yousef: *Iterative methods for sparse linear systems.* 2. Philadelphia : SIAM, 2003. – I–XVIII, 1–528 S. – ISBN 978-0-89871-534-7

[Spinellis 2001] Spinellis, Diomidis: Notable design patterns for domain-specific languages. In: *J. Syst. Softw.* 56 (2001), February, S. 91–99. – URL `http://dl.acm.org/citation.cfm?id=371260.371313`. – ISSN 0164-1212

[Stahl u. a. 2007] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* 2. Heidelberg : dpunkt, 2007. – ISBN 978-3-89864-448-8

[Vogel u. a. 2005] Vogel, O. ; Arnold, I. ; Chughtai, A. ; Ihler, E. ; Kehrer, T. ; Mehlig, U. ; Zdun, U.: *Software-Architektur: Grundlagen - Konzepte - Praxis.* 1. München : Spektrum Akademischer Verlag, 2005. – URL `http://books.google.de/books?id=2xsK1-YJgZ8C`. – ISBN 9783827419330

[Weiss und Lai 1999] Weiss, David M. ; Lai, Chi Tau R.: *Software product-line engineering: a family-based software development process.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0-201-69438-7

[Xerces 2012a] Xerces: *SAX2 Programming Guide.* `http://xerces.apache.org/xerces-c/program-sax2-3.html`. 13. Oktober 2012

[Xerces 2012b] Xerces: *Xerces-C++ XML Parser.* `http://xerces.apache.org/xerces-c/`. 13. Oktober 2012

[Xtext 2012] Xtext: *Xtext - Language Development Made Easy!* `http://www.eclipse.org/Xtext/`. 17. Oktober 2012

A. Anhang

A.1. Compressed Sparse Row (CSR) als Speicherformat für dünnbesetzte Matrizen

Für das CSR-Format wird im Folgenden ein kleines Beispiel für eine Matrix A gezeigt:

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 6 & 7 \end{pmatrix} \end{array}$$

$$\begin{aligned} data &= \{1; 2; 3; 4; 5; 6; 7\} \\ ja &= \{0; 1; 1; 2; 2; 2; 3\} \\ ia &= \{0; 2; 4; 5; 7\} \end{aligned}$$

Die drei Arrays *data*, *ja* und *ia* werden für die Speicherung einer Matrix im CSR-Format benötigt. *data* wird für die zeilenweise Speicherung der Nicht-Null-Elemente verwendet. Das Array *ja* speichert die Spaltenposition des jeweiligen Datenelements bezogen auf die Matrix A. Entsprechend ist die Elementanzahl von *ja* und *data* identisch. *ia* speichert die Indizes von *data* bzw. *ja* bei denen eine neue Zeile in A begonnen wird. Für die letzte Zeile wird zusätzlich noch gespeichert bei welchem Element die Matrix zu Ende ist, um bei der

Programmierung ein definiertes Ende für die zeilenweisen Schleifendurchläufe zu haben. Entsprechend kann mit *ia* die Anzahl der Elemente einer Zeile *row* durch *ia[row+1] - ia[row]* ermittelt werden.

A.2. Jacobi Lösungsverfahren

Als Beispiel wird im Folgenden das Jacobi-Verfahren vorgestellt, um die genauen Abläufe in LAMA zu verstehen. Das Jacobi-Verfahren ist ein iteratives Verfahren, welches oft in Verbindung mit Anderen genutzt wird, z.B. als Präkonditionierer oder Glätter.

Für das Beispiel wird das allgemein funktionierende Verfahren beschrieben. Das Verfahren löst lineare Gleichungssysteme der Form $Ax = b$ indem zusätzliche Matrizen in der Initialisierungs-Phase erzeugt werden. A wird dabei in eine gesonderte Diagonalmatrix D und in eine weitere Matrix C mit unterem und oberen Dreieck aufgeteilt. Die iterative Berechnung des Lösungsvektors x ergibt sich dann aus folgendem Schritt:

$$x^{(i+1)} = D^{-1}(b - C)x^{(i)}$$

Dabei ist $x^{(i)}$ der Lösungsvektor der Iteration i, welcher für die Berechnung des Folgelösungsvektors $x^{(i+1)}$ benötigt wird. Der nicht-konstante Teil dieses Berechnungsschrittes wird in der iterate-Methode implementiert. Berechnungen, die immer gleich sind, wie die Berechnung der Inversen der Diagonalmatrix, werden zusätzlich in die Initialisierung ausgelagert, um Redundanz zu vermeiden.

Die genauen Implementierungen der Verfahren sind in der folgenden Arbeit als Black-Boxes zu betrachten, da nur die nach außen verfügbaren Schnittstellen zu Konfiguration der Verfahren relevant sind.

A.3. Featurediagramme

In diesem Abschnitt werden die in dem Domänenanalyse-Abschnitt fehlenden Featurediagramme vorgestellt.

Das zu den in LAMA implementierten direkten Lösern passende Feature-Diagramm in Abbildung A.1 zeigt, dass diese Löser keine weiteren Konfigruationsmöglichkeiten und besitzen somit neben den allgemeinen Löserfeatures auch keine neu hinzukommenden Features.

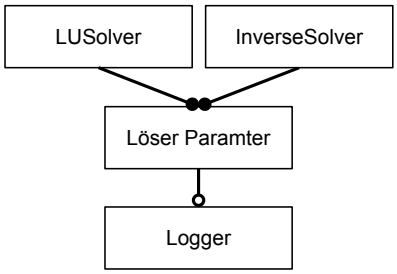

Abbildung A.1.: Featurediagramm zu den in LAMA implementierten direkten Lösern LUSolver und InverseSolver.

In Abbildung A.2 befindet sich das Featurediagramm zu den iteraitven Lösern CG und GMRES. Beide haben die Konfigurationsmöglichkeiten die jeder iterative Löser hat. Einer GMRES-Instanz kann noch eine in Anforderung 13 beschriebene Krylov-Dimension zugeordnet bekommen.

Das Featurediagramm zu den Lösern, die einen Omega-Wert zugeordnet bekommen (siehe Anforderung 12) wird in Abbildung A.3 gezeigt. Diese Wert bestimmt, wie viel von der alten Lösung aus der vorherigen Iteration in die Lösung der aktuellen Iteration einfließt. Der DefaultJacobi ist eine Implementierung des Jacobi-Verfahrens, das für alle in LAMA implementierten Matrixtypen funktioniert, da hier nur Standard-Matrix-Operationen angewandt werden. Jedoch kommt es hier dazu, dass Matrizen kopiert werden, was die Berechnung verlangsamt. Der SpecializedJacobi beinhaltet spezialisierte Implementierungen für einige Speicherformate, die keine Kopien der Datenstrukturen verlangen und auf das Format optimierte Operationen ausführen.

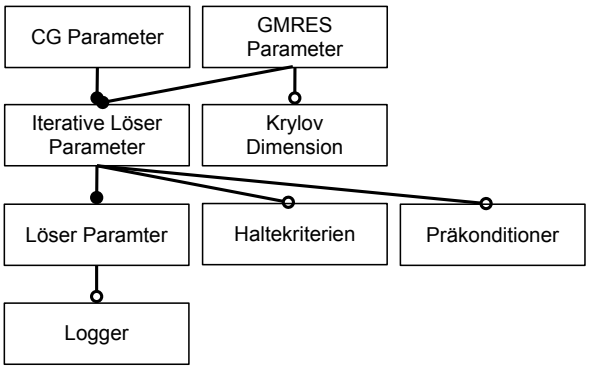

Abbildung A.2.: Featurediagramm zu den in LAMA implementierten iterativen Lösern CG und GMRES.

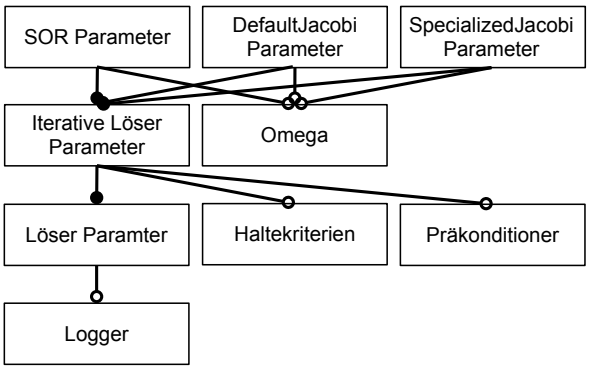

Abbildung A.3.: Featurediagramm zu den in LAMA implementierten iterativen Lösern SOR und DefaultJacobi und SpecializedJacobi.

A.4. Beispiel für eine Schemadefinition in XSD

Hier wird passend zu dem Beispiel in Listing 5 in dem Auswahlkapitel 4.2.2 auf Seite 4.2.2 für die Realisierungsmöglichkeit mit XML und XSD die Schemadefinition in Listing 23 gezeigt.

```
<xs:simpleType name="norm">
  <xs:restriction base="xs:string">
    <xs:enumeration value="L1Norm"/>
    <xs:enumeration value="L2Norm"/>
    <xs:enumeration value="MaxNorm"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="checkmode">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Absolute"/>
    <xs:enumeration value="Relative"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="ResidualThreshold">
    <xs:attribute name="norm" type="norm"/>
    <xs:attribute name="precision" type="xs:decimal"/>
    <xs:attribute name="checkmode" type="checkmode"/>
</xs:complexType>
```

Listing 23: Beispiel für eine XSD für das Haltekriterium ResidualThreshold.

A.5. Vollständige Grammatik des MetaSolvers in PEG-Notation

In den Listings 24, 25, 26 und 27 findet sich die vollständige Grammatikbeschreibung in PEG-Notation, bezogen auf die Grammatik des MetaSolvers.

```
configurationSequence ← configuration*
configuration ← solver_config / logger_config / criteria_config
solver_config ← solver_type solver_parameter
solver_type ← 'CG' / 'SOR' / 'GMRES' / 'LUSolver'
    / 'InverseSolver' / 'SimpleAMG' / 'DefaultJacobi'
solver_parameter ← cg_parameter_list / sor_parameter_list / ...
id ← [a-zA-Z_] [0-9a-zA-Z_]*
iterative_solver_parameter_list ← solver_parameter_list
    preconditioner? stopping_criterion?
solver_parameter_list ← logger?
```

Listing 24: Beschreibung der MetaSolver-bezogenen Grammatik in PEG-Notation.

```
logger ← 'logger' ( id / logger_def ) ';'
logger_def ← common_logger / file_logger
common_logger ← 'CommonLogger' common_logger_parameter
common_logger_parameter ← '(' id ',' log_level ','
    write_behaviour ',' timer ')'
log_level ← 'solverInformation' / 'noLogging'
    / 'completeInformation' / ...
write_behaviour ← 'toConsoleOnly' / ...
timer ← 'OpenMPTimer'
```

Listing 25: Beschreibung der loggerbezogenen Grammatik in PEG-Notation.

```
stopping_criterion ← 'criteria' ( id / node )  ';'
node ← ( '(' node logical_connective node ')' ) | leaf
logical_connective ← 'AND' / 'OR'
leaf ← iteration_count / residual_threshold / residual_stagnation
iteration_count ← 'IterationCount(' int ')'
residual_threshold ← 'ResidualThreshold(' norm ',' precision ','
    check_mode ')'
norm ← 'L1Norm' / 'L2Norm' / 'MaxNorm'
precision ← double
check_mode ← 'Absolute' / 'Relative'
residual_stagnation ← 'ResidualStagnation(' norm ',' lookback ','
    precision ')'
lookback ← int ';'
```

Listing 26: Beschreibung der haltekrierienbezogenen Grammatik in PEG-Notation.

```
preconditioner ← 'preconditioner' solver_reference ';'
solver_reference ← id
cg_parameter_list ← id '{' iterative_solver_parameter_list '}'
sor_parameter_list ← id '{' omega_solver_parameter_list '}'
omega_solver_parameter_list ← iterative_solver_parameter omega
omega ← double ';'
gmres_parameter_list ← iterative_solver_parameter_list krylov_dim
krylov_dim ← 'krylovDim' int
lu_solver ← solver_parameter_list
inverse_solver ← solver_parameter_list
simple_amg ← iterative_solver_parameter_list smoother
    coarse_level_solver max_level min_vars_per_row
smoother ← 'smoother' solver_reference ';'
coarse_level_solver ← 'coarseLevelSolver' solver_reference ';'
max_level ← 'maxLevel' int ';'
min_vars_per_row ← 'minVarsPerRow' int ';'
```

Listing 27: Beschreibung der lösererzeugerbezogen Grammatik in PEG-Notation.