HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Constraint-basierte Generierung von Testdaten für datenbankbasierte Anwendungen

Jens Küblbeck

Konstanz, 02.07.2015

MASTERARBEIT

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Master of Science Informatik

Thema: **Constraint-basierte Generierung von Testdaten für datenbankbasierte Anwendungen**

Masterkandidat: Jens Küblbeck

1. Prüfer: Prof. Dr.-Ing. Jürgen Wäsch
2. Prüfer: Dr. Thomas Fox

Zusammenfassung (Abstract)

Thema:	Constraint-basierte Generierung von Testdaten für datenbankbasierte Anwendungen
Masterkandidat:	Jens Küblbeck
Firma:	Seitenbau GmbH
Betreuer:	Prof. Dr.-Ing. Jürgen Wäsch Dr. Thomas Fox
Abgabedatum:	02.07.2015
Schlagworte:	Constraint-basierte Generierung von Testdaten, Valide Testdaten, Modellierung von Constraints, Constraint Problem Solving, Tiefensuche

Um datenbankbasierte Anwendungen zu testen, werden in der Regel Testdaten benötigt, die in die zugrundeliegende Datenbank eingespeist werden. Da das manuelle Erstellen dieser Testdaten aufwendig und ein immer wiederkehrendes Problem des Softwareentwicklungszyklus ist, erleichtert eine automatisierte Generierung von Testdaten den Aufwand für die Qualitätsprüfung von Software enorm.

In dieser Arbeit wird deshalb untersucht, wie valide Testdaten generiert werden können. Es wird insbesondere Wert auf die Einhaltung von Constraints in den generierten Daten gelegt. Zuerst werden verschiedene Arten von Constraints untersucht und anschließend werden diese nach Arten von Abhängigkeiten zwischen den verschiedenen zu generierenden Daten kategorisiert. Nachfolgend werden unterschiedliche Strategien und Algorithmen analysiert, die Testdaten unter Beachtung dieser Constraints generieren können.

Ein weiterer Schwerpunkt dieser Arbeit ist die Implementierung dieser Algorithmen. Dazu wird die vorhandene Test-Bibliothek STU um den Ansatz deklarativer Constraints erweitert, indem zum einen die Modellierung von Constraints ermöglicht und zum anderen geeignete Algorithmen zur Auflösung implementiert werden. Die verschiedenen Lösungsansätze werden zuletzt anhand eines Beispielmodells verglichen und bewertet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Um datenbankbasierte Anwendungen zu testen, werden in der Regel Testdaten benötigt, welche in die zugrundeliegende Datenbank eingespeist werden. Da das manuelle Erstellen dieser Testdaten aufwendig und ein immer wiederkehrendes Problem des Softwareentwicklungszyklus ist, erleichtert eine automatisierte Generierung von Testdaten den Aufwand für die Qualitätsprüfung von Software erheblich.

Als Grundlage für die Generierung der Testdaten dient das Datenbankschema. Aus dem Datenbankschema sind zum einen die unterschiedlichen Relationen und zum anderen die verwendeten Datentypen auslesbar. Für eine geeignete Testdatenmenge müssen jedoch weitere Anforderungen und Bedingungen beachtet werden. Diese werden entweder vom Datenbankschema in Form von Constraints definiert oder durch die Anwendungslogik vorgegeben. In dieser Arbeit wird deshalb untersucht, wie diese zusätzlichen Anforderungen an die Testdaten in den Generierungsprozess eingebunden werden können. Dabei wird vor allem betrachtet, wie mithilfe von deklarativen Constraints, welche durch den Tester definiert werden, qualitativ hochwertige Testdaten generiert werden können.

Weiterer Schwerpunkt der Arbeit ist die Analyse von Strategien und Algorithmen zur Auflösung der Constraints und der durch die Constraints erzeugten Abhängigkeiten zwischen den einzelnen Datenwerten. Hierbei werden verschiedene Ansätze untersucht und geeignete Lösungen implementiert.

Basis für die Implementierung von geeigneten Lösungen ist die von der Firma Seitenbau entwickelte STU-Bibliothek[1]. Die als Teil der Master-Arbeit [Mol13] von Nikolaus Moll implementierten Funktionalitäten zur

[1]STU (Simple Testing Utils) ist ein Open-Source-Bibliothek der Firma Seitenbau zur Vereinfachung des Testprozesses in Java-Anwendungen. Die Bibliothek ist unter https://github.com/Seitenbau/stu frei zugänglich.

Testdatengenerierung werden dabei übernommen und weiterentwickelt.

Zu Beginn der Arbeit wird der vorgefundende Stand der zugrundeliegenden STU-Bibliothek in Kapitel 2 beschrieben. Anschließend werden in Kapitel 3 die Funktionen vorgestellt, mit denen die STU-Bibliothek erweitert werden soll. Im nächsten Schritt wird in Kapitel 4 mithilfe einer Anforderungsanalyse ermittelt, welche funktionalen und nicht-funktionalen Anforderungen an eine Implementierung gestellt werden. Nachfolgend wird in Kapitel 5 konzeptionell untersucht, wie die gestellten Anforderungen möglichst vollständig erfüllt werden können. In Kapitel 6 wird anschließend beschrieben, wie die in Kapitel 5 vorgestellten Strategien und Algorithmen konkret umgesetzt und in die STU-Bibliothek implementiert werden können. Anschließend werden die verschiedenen implementierten Strategien und Algorithmen zum Auflösen von Constraint-bedingten Abhängigkeiten in Kapitel 7 anhand eines Beispiels verglichen und bewertet. Im letzten Schritt wird das Ergebnis dieser Arbeit in Kapitel 8 zusammengefasst und zusätzliche mögliche Erweiterungen werden betrachtet.

Kapitel 2

Vorgefundener Stand

Teil dieser Arbeit ist die Weiterentwicklung der bestehenden STU-Bibliothek. STU bedeutet Simple Test Utils und ist eine von der Seitenbau GmbH entwickelte Testbibliothek für JUnit und DBUnit. Der für diese Arbeit zentral betrachtete Teil der Bibliothek ist die Modellierung und Generierung von Testdaten für datenbankbasierte Anwendungen. Hierfür wird das Modell aus dem Datenbankschema extrahiert und in Form von Builder-Klassen dargestellt. Danach werden mit dem aus [HTW06] abgeleiteten und in [Mol13, 60-70] vorgestellten Algorithmus anhand der Beziehungen unter den Tabellen versucht, eine Testdatenmenge zu generieren, die die Grenzen der einzelnen Beziehungen zwischen den Tabellen abdeckt. Mithilfe von Wertgeneratoren werden Werte für die verschiedenen Datentypen generiert. Weiterführend sind spezielle Wertgeneratoren für verschiedene domänenspezifische Werte definiert, sodass nicht nur dem Datentyp entsprechende Werte, sondern auch aus fachlicher Sicht valide Daten generiert werden. Die generierten Testdaten werden mithilfe einer speziell entwickelten DSL[1] modelliert.

Die Wertgeneratoren generieren zwar Datentyp- und Domänenspezifisch-valide Daten, jedoch ist es nicht möglich Daten zu generieren, die voneinander abhängig sind. Dies rührt daher, dass die Wertgeneratoren keine Funktionalität unterstützten um Werte abhängig von anderen Werten zu generieren. Einzige Ausnahme ist hierbei die Generierung von `Primary Keys` und `Foreign Keys`, da diese nicht mit den Wertgeneratoren generiert werden. In diesem Kapitel wird zunächst in Abschnitt 2.1 der genaue Ablauf der Testdatengenerierung beschrieben. Anschließend werden in Abschnitt 2.2 alle bisher unterstützten und in Abschnitt 2.3 die wünschenswerten, jedoch nicht unterstützten Funktionalitäten, aufgeführt. In Abschnitt 2.4 werden die bestehenden Funktionalitäten anhand eines Beispiels dargestellt.

[1] Eine DSL (Domain Specific Language) ist eine formale Sprache die für eine bestimmte Domäne entwickelt wird.

2.1 Ablauf der Testdatengenerierung

Bei der Generierung von Testdaten für eine datenbankbasierte Anwendung werden in STU nacheinander mehrere Schritte durchgeführt:

1. **Modell aus Datenbankschema ableiten:** Das Modell wird zuerst aus dem Datenbankschema der Datenbank extrahiert und in Form von Builder-Klassen dargestellt.

2. **Modell anpassen:** Das Modell kann anschließend angepasst und mit domänenspezifischen Generatoren und Eigenschaften angereichert werden.

3. **Testdaten generieren:** Anschließend werden die Testdaten anhand der Builder-Klassen generiert. Das Ergebnis der Generierung ist eine Tabelle in Form einer für diesen Zweck entwickelten DSL.

4. **Testdaten anreichern:** Es ist nun möglich, die Testdaten anzupassen, indem sie um weitere Sonderfälle oder zusätzliche Datensätze erweitert werden.

5. **Testdaten in die Datenbank einspielen:** Im letzten Schritt werden die Testdaten in die Datenbank eingespielt.

2.2 Unterstütze Funktionalitäten

Um einen genaueren Überblick über die STU-Bibliothek zu erhalten, werden die bisher unterstützten Funktionalitäten für die im vorherigen Abschnitt beschriebenen Schritte betrachtet. In diesem Abschnitt wird deshalb nochmals detaillierter auf die bereits unterstützten Funktionalitäten der STU-Bibliothek in Bezug auf die Testdatengenerierung eingangen.

1. **Modell aus Datenbankschema ableiten:** Unter Angabe der Zugangsdaten einer Datenbank kann mithilfe der Anwendung ein bestehendes Datenbankschema als Modell in Form von Java-Builder-Klassen erstellt werden. Alternativ kann das Modell mithilfe von Builder-Klassen manuell definiert werden. Dabei werden sowohl alle gängigen Datentypen des Datenbankschemas als auch die unterschiedlichen möglichen Beziehungen von relationalen Datenbanken unterstützt. Es kann genau eine Spalte einer Tabelle als `Primary Key` deklariert werden. Als `Foreign Key` sind beliebig viele Spalten deklarierbar. Mit Hilfe von assoziativen Tabellen werden N:M-Beziehungen dargestellt, die genau über zwei als `Foreign Key` deklarierte Spalten verfügen.

2. **Model anpassen:** Das zuvor automatisch aus einem Datenbankschema generierte Modell kann anschließend manuell angepasst werden. Für alle Spalten ist ein Datentyp definiert. Anhand dieses Datentyps werden jeder Spalte automatisch dem Datentyp entsprechende Wertgeneratoren zugewiesen. Wird für eine Spalte ein domänenspezischer Wert benötigt, so kann dieser Spalte explizit ein entsprechender Wertgenerator zugewiesen werden.

3. **Testdaten generieren:** Anhand der definierten Builder-Klassen werden mithilfe des in [Mol13] vorgestellten Algorithmus Testdaten für alle Tabellen generiert und in Form einer eigens hierfür entwickelten DSL dargestellt. Mit Hilfe des Generierungsalgorithmus werden vier Typen von binären Beziehungen unterstützt: 1:1, 1:N, N:1 und N:M. Der Algorithmus verwendet dabei das Konzept der Äquivalenzklassenbildung und der Grenzwertanalyse (siehe [Mol13]). Die Werte für die als `Primary Key` und `Foreign Key` deklarierten Spalten werden durch Iteration von aufeinanderfolgenden natürlichen Zahlen erzeugt. Alle sonstigen Werte werden mithilfe der Datentyp entsprechenden Wertgeneratoren bzw. der explizit definierten domänenspezifischen Wertgeneratoren erzeugt.

4. **Testdaten anreichern:** Die generierten Testdaten werden in Form einer DSL dargestellt. Dies ermöglicht eine übersichtliche und verständliche Ansicht der generierten Testdaten. In der erzeugten Datei können die einzelnen Werte manuell geändert und angepasst werden.

5. **Testdaten in die Datenbank einspielen:** Auf Grundlage der in Form der DSL präsentierten Testdaten, kann nun ein Modell der Testdaten in Form von Java-Klassen instanziert werden. Im letzten Schritt kann das erzeugte Modell mithilfe des STU DatabaseTester in ein DBUnit IDatabase Model transformiert und von DBUnit in die Datenbank eingespielt werden.

2.3 Nicht-unterstützte Funktionalitäten

In diesem Abschnitt wird erörtert, welche Funktionalitäten wünschenswert sind, aber nicht unterstützt werden. Der Schwerpunkt wird hierbei auf den Schritten **Modell aus Datenbankschema ableiten**, **Model anpassen** und **Testdaten generieren** gelegt. Die zwei Schritte **Testdaten anreichern** und **Testdaten in die Datenbank einspielen** werden an dieser Stelle nicht weiter betrachtet, da diese beiden Schritte keinen Einfluss auf die Constraint-basierte Generierung von Testdaten haben.

1. **Modell aus Datenbankschema ableiten:** Bisher werden aus dem Datenbankschema alle Tabellen mit den zugehörigen Spalten, die Da-

tentypen und die Beziehungen zwischen den Tabellen in ein Modell abgeleitet. Constraints, die für einzelne Spalten definiert sind, sind nicht im Modell abbildbar.

2. **Model anpassen:** Es ist bisher nicht möglich, einen `Primary Key` aus mehreren Spalten zusammenzusetzen. Ebenfalls werden bisher keine `Primary Key`-Spalten unterstützt, die nicht vom Datentyp `Integer` oder `Long` sind. Des Weiteren sind bisher nur assoziative Tabellen, die genau zwei Spalten, die als `Foreign Key` deklariert sind, modellierbar. Assoziative Tabellen mit mehr als zwei `Foreign Key`-Spalten werden nicht unterstützt. Constraints, wie das `Unique`-Constraint, das `Not Null`-Constraint oder das `Check`-Constraint, die im Datenbankschema deklariert werden können, sind nicht im Modell definierbar. Ferner beinhalten die Builder-Klassen zur Modellierung des Modells keine Möglichkeit zur Modellierung von Constraints, die bestimmte Charakteristiken von Anwendungen abdecken.

3. **Testdaten generieren:** Zirkuläre, reflexive und nicht-binäre Beziehungen, wie z.B. Baumstrukturen, werden nicht ausreichend unterstützt. Diese Strukturen sind zwar mithilfe der Builder-Klassen modellierbar, werden jedoch beim Generierungsprozess nicht befriedigend umgesetzt. Darüber hinaus ist es nicht möglich, Werte abhängig von anderen generierten Werten der Datenbank, zu generieren.

2.4 Beispiel der aktuellen Umsetzung

In dem Beispiel aus Listing 2.1 ist zu sehen, wie zum einen das Datenbankschema, zum anderen die Anreicherung durch Generatoren mit Hilfe der `TableBuilder`-Klassen abgebildet sind. Wird nun die Methode zur Testdatengenerierung ausgeführt so erhält man als Ergebnis die in Listing 2.2 mit Hilfe einer DSL dargestellten Testdaten. Anhand dieses Beispiels lässt sich sehr gut erkennen, dass sich mit den bisher zur Verfügung stehenden Funktionen das Datenbankschema mit Hinblick auf Datentypen und relationalen Abhängigkeiten gut abbilden lässt. Um jedoch im Datenbankschema definierte Constraints oder durch die Anwendungslogik festgelegte Charakteristiken in den Testdaten abzubilden, fehlen bisher die notwendigen Möglichkeiten, um plausible Testdaten erzeugen zu können.

Deshalb ergeben sich mit hoher Wahrscheinlichkeit generierte Testdaten, die zum Teil dem Datenbankschema oder der Anwendungslogik wiedersprechen. In Listing 2.2 ist beispielsweise erkennbar, dass die Spalte `email` in Tabelle `user` nicht `Unique` ist. Wenn diese Spalte jedoch im Datenbankschema als `Unique` definiert ist, so sind die generierten Testdaten nicht Datenbankschema-konform. Wird dieser Testdatensatz unangepasst in

```
buch
  .column("id", DataType.BIGINT).defaultIdentifier().autoInvokeNext()
  .column("name", DataType.VARCHAR).generator(new BuchNameGenerator())
  .column("preis", DataType.INTEGER).generator(new IntegerGenerator(0, 99))
  .column("verlag", DataType.BIGINT).reference.local
  .foreign(verlag.ref("id")).multiplicity("0..*")
  .build();

verlag
  .column("id", DataType.BIGINT).defaultIdentifier().autoInvokeNext()
  .column("name", DataType.VARCHAR).generator(new VerlagNameGenerator())
  .build();

autor
  .column("id", DataType.BIGINT).defaultIdentifier().autoInvokeNext()
  .column("vorname", DataType.VARCHAR).generator(new VornameGenerator())
  .column("nachname", DataType.VARCHAR).generator(new NachnameGenerator())
  .column("geschlecht", DataType.VARCHAR).generator(new GeschlechtGenerator())
  .column("email", DataType.VARCHAR).generator(new EmailGenerator())
  .build();

associativeTable("buch_autor")
  .column("buch_id", DataType.BIGINT).reference.foreign(buch)
  .multiplicity("0..*")
  .column("autor_id", DataType.BIGINT).reference.foreign(autor)
  .multiplicity("1..*")
  .build();
```

Listing 2.1: Beispiel für die API zur Modellierung des Modells zur Generierung von Testdaten

die Datenbank eingespeist, so schlägt das Einspeisen aufgrund der nicht eindeutigen Werte der Spalte `email` fehl.

Betrachtet man die Spalten `vorname` und `geschlecht` der Tabelle `Autor`, so ist ersichtlich, dass die generierten Werte zwar dem richtigen Datentyp und somit dem Datenbankschema entsprechen, jedoch in domänenspezifischer Hinsicht nicht zusammenpassen. Diese domänenspezifischen Fehler können vor allem für Tester, welche die Anwendung mit den generierten Testdaten testen, zu Irritationen führen. Dem Tester ist dann möglichweise nicht ersichtlich, ob dieser Fehler Folge der zu testenden Anwendung oder ein Fehler in den generierten Testdaten ist.

```
buchTable.rows() {
      REF      | name                  | preis | verlag
      BUCH_1   | "Erzählungen"         | 47    | VERLAG_1
      BUCH_2   | "Germinal"            | 5     | VERLAG_1
      BUCH_3   | "Der Steppenwolf"     | 93    | VERLAG_2
    }

    verlagTable.rows() {
      REF        | name
      VERLAG_1   | "Neumann Verlag"
      VERLAG_2   | "E. S. Mittler & Sohn"
    }

    autorTable.rows() {
      REF      | vorname    | nachname     | geschlecht   | email
      AUTOR_1  | "Eva"      | "Vogt"       | "männlich"   | "eva.v@mail.de"
      AUTOR_2  | "Adam"     | "Dietrich"   | "weiblich"   | "adam@net.com"
      AUTOR_3  | "Adam "    | "Winter"     | "männlich"   | "adam@net.com"
    }
```

Listing 2.2: Beispiel für die DSL zur Darstellung der generierten Daten

Kapitel 3

Grundlagen

In diesem Kapitel wird zunächst in Abschnitt 3.1 auf die Grundidee eingegangen, Testdaten für datenbankbasierte Anwendungen abhängig von deklarativen Constraints zu generieren. Im Anschluss werden in Abschnitt 3.2 mögliche Arten von Constraints und Funktionen aufgeführt und beschrieben, welche die zu generierenden Werte in irgendeiner Weise beschränken. Anschließend wird in Abschnitt 3.3 analysiert, welche Abhängigkeitsarten durch die zuvor beschriebenen Constraints und Funktionen entstehen. Anhand von Beispielen werden dann die verschiedenen Arten von Abhängigkeiten veranschaulicht.

3.1 Datengenerierung durch deklarative Constraints

Um synthetisch Daten zu generieren, welche für datenbankbasierte Anwendungen bestimmt sind, bietet es sich an, diese anhand von deklarativ formulierten Constraints zu generieren [AKL11].Mit den deklarativ formulierten Constraints soll es möglich sein, sowohl datenbankspezifische Charakteristiken als auch die durch die Anwendungslogik vorgegebenen Charakteristiken bei der Generierung der Testdaten zu berücksichtigen. Man kann zwischen vier verschiedenen Ebenen unterscheiden [Hal10]:

Datentyp-konform Die generierten Testdaten sind nach dem in der Datenbank festgelegten Datentyp konform.

Datenbankschema-konform Die generierten Testdaten sind Datenbankschema-konform, falls sie für alle in der Datenbank definierten Constraints konform sind.

Anwendungs-konform Die generierten Testdaten sind Anwendungskonform, falls Testdaten die gleiche Charakteristik aufweisen wie Daten, die nur durch die Benutzeroberfläche erzeugt wurden.

Testpfad-konform Die generierten Testdaten sind für einen bestimmten Test konform.

Mit dem bestehenden Funktionsumfang der STU-Bibliothek können bereits Testdaten generiert werden, die Datentyp-konform und zum Teil Datenbankschema-konform sind. Das Hauptaugenmerk dieser Arbeit liegt deshalb an der Erweiterung der Funktionen um Testdaten zu generieren, die zum einen möglichst vollständig Datenbankschema-konform und zum anderen Anwendungs-konform sind. Des Weiteren sollen möglichst viele Testpfade mit einer möglichst geringen Menge an Testdaten abgedeckt werden. Ziel ist dabei nicht Testdaten zu generieren, die vollständig Testpfad-konform sind, da in diesem Fall meist eine größere Menge an Testdaten entsteht. Eine größere Menge an Testdaten würde jedoch bedeuten, dass der Testprozess aufwendiger und unübersichtlicher machen würde.

3.2 Übersicht über verschiedene Arten von Constraints

Im Folgenden werden die verschiedenen Arten von Constraints aufgeführt. Die Constraints lassen sich im Wesentlichen aus den funktionalen Abhängigkeiten ableiten. Unter funktionalen Abhängigkeiten versteht man ein Konzept des relationalen Datenmodells. Funktionale Abhängigkeiten beschreiben, wie einzelne Attribute in einer Relation zusammenhängen und wie Schlüssel definiert werden. Zusätzlich lässt sich die Datenintegrität einer relationalen Algebra ableiten. Unter Datenintegrität versteht man die Korrektheit der Daten. Man kann zwischen folgenden Typen von Datenintegrität unterscheiden [FW07, 145-151]:

Entity-Integrität Jede Relation einer Menge an Relationen besitzt nur einen Primary-Key.

Referenzielle Integrität Jeder Wert eines Foreign-Key einer Relation ist Wert eines Primary-Key einer anderen Relation.

Semantische Integrität Die Menge an Relationen gewährleistet die Korrektheit der Eingaben der Benutzer.

3.2.1 Datentyp-Constraint

Durch einen in der SQL-Spezifikation [LR13, 580-602] spezifizierten Datentyp wird der Wertebereich eines zu generierenden Wertes eingeschränkt. Der Datentyp einer Spalte kann beispielsweise vom Typ `Boolean` sein. Somit erhält man die mögliche Wertemenge `true, false` für diese Spalte.

3.2.2 Range-Constraint

Ein Range-Constraint kann den Wertebereich eines zu generierenden Wertes auf einen Bereich festlegen. Die Werte für die untere bzw. die obere Grenze können entweder konstant sein oder sie werden aus anderen Werten in der Datenbank vorgegeben. Ein Range-Constraint kann auch durch zwei Prädikats-Constraints (siehe Abschnitten 3.2.8) abgebildet werden. Dieser Constraint kann in SQL durch den in der SQL-Spezifikation [LR13, 580-602] spezifizierten `Check`-Constraint beschrieben werden.

3.2.3 Not-Null-Constraint

Der `Not-Null`-Constraint legt fest, dass kein `Null`-Wert generiert werden darf. Dieser Constraint bezieht sich in jedem Fall nur auf einen Wert und besitzt deshalb keine Abhängigkeiten zu anderen Werten der Datenbank. Dieser Constraint ist in der SQL-Spezifikation [LR13, 580-602] spezifiziert.

3.2.4 Unique-Constraint

Eine häufig gestellte Anforderung von zugrundeliegenden Datenmodellen ist, dass Werte innerhalb einer Tabellenspalte einzigartig (engl. Unique) sein müssen. Durch ein gesetztes Unique-Constraint wird erzwungen, dass die Werte einer Spalte einzigartig sind. Hierbei handelt es sich in jedem Fall um einen Constraint-Typ, der Abhängigkeiten zu anderen Werten der Datenbank besitzt. Dieser Constraint ist in der SQL-Spezifikation [LR13, 580-602] spezifiziert.

3.2.5 Primary-Key-Constraint

Der Primärschlüssel ist eine Kombination aus dem `NOT NULL` und dem `UNIQUE` Constraint. Er gibt an, dass eine Spalte oder eine Kombination von zwei oder mehr Spalten eindeutig und kein `Null`-Wert ist. Er wird genutzt, um schnell und einfach auf Datensätze zugreifen zu können. Hierbei handelt es sich in jedem Fall um einen Constraint, der Abhängigkeiten zu anderen Werten der Datenbank-Tabelle besitzt. Dieser Constraint ist in der SQL-Spezifikation [LR13, 580-602] spezifiziert.

3.2.6 Foreign-Key-Constraint

Mit dem Fremdschlüssel wird sichergestellt, dass die referenzielle Integrität der Daten in einer Tabelle mit Werten einer anderen Tabelle entsprechen. Der Foreign-Key-Constraint besitzt in jedem Fall Abhängigkeiten zu anderen Werten der Datenbank. Diese Constraint-Art ist in der SQL-Spezifikation [LR13, 580-602] als `Foreign-Key` spezifiziert.

3.2.7 Domain-Constraint

Mit Hilfe des Domain-Constraint soll die semantische Integrität von Daten gewährleistet werden. Unter semantischer Integrität der Daten versteht man die Korrektheit von Daten im Sinne ihrer domänenspezifischen Charakteristik [FW07, 149-151]. So sollten beispielsweise die generierten Werte für die Spalte `Stadt`, nicht nur vom Datentyp `String` sein, sondern sollten ausschließlich Werte, wie „Konstanz“, „Allensbach“ oder „Radolfzell“ enthalten. Stehen die Daten in einer Beziehung zu anderen domänenspezifischen Daten einer anderen Spalte, so soll gewährleistet werden, dass diese Abhängigkeit ebenfalls bei der Generierung berücksichtigt wird. Wird in einer zweiten Spalte zum Beispiel die `Postleitzahl` generiert, so soll die Kombination aus `Postleitzahl` und `Stadt` korrekt sein. In Tabelle 3.1 ist ein Ausschnitt einer domänenspezifischen Beziehungstabelle dargestellt. Diese Constraint-Art ist in der SQL-Spezifikation [LR13, 580-602] nicht abgebildet.

Postleitzahl	**Stadt**
78315	Radolfzell
78462 78464 78465 78467	Konstanz
78476	Allensbach

Tabelle 3.1: Beispiel für domänenspezifische Daten und deren Beziehung untereinander

3.2.8 Prädikats-Constraint

Werte eines Datenmodells können verschiedenen Regeln unterliegen, welche festlegen, ob ein Wert valide ist oder nicht. Die Regeln werden mithilfe der Prädikaten-Logik definiert und können sich dabei entweder auf andere generierte Werte oder auf konstante Werte beziehen. Operatoren der Prädikaten-Logik sind: gleich, ungleich, größer, größer gleich, kleiner, kleiner gleich, und, oder, nicht. Falls das Prädikats-Constraint keine Abhängigkeiten oder nur Abhängigkeiten innerhalb einer Zeile einer Tabelle besitzt, so entspricht dieser Constraint-Typ dem `Check`-Constraint der SQL-Spezifikation [LR13, 580-602].

3.2.9 Funktions-Constraint

Mithilfe von Funktions-Constraints sind funktionale Abhängigkeiten, wie mathematische Operationen zwischen verschiedenen Werten abbildbar. Dabei kann zwischen mathematischen Rechenoperationen und Aggregatsfunktionen unterscheiden. Mathematische Rechenoperationen, wie z.B. Addieren, Subtrahieren, Multiplizieren, Dividieren und Potenzieren, stellen typischerweise eine funktionale Abhängigkeit zwischen verschiedenen Werten des gleichen Datensatzes dar und können in der Form Vara mod b = c angegeben werden. Aggregatsfunktionen, wie beispielsweise Summe, Anzahl, Maximalwert, Minimalwert, Durchschnitt, beziehen sich auf alle Werte einer bestimmten Spalte. Funktions-Constraints werden in Abschnitt 3.3 in Hinsicht auf ihre Abhängigkeiten nochmals genauer betrachtet.

3.2.10 Kardinalitäts-Constraint

Unter Verwendung von Kardinalitäts-Constraints lässt sich für einzelne Spalten festlegen, wie häufig einzelne Werte pro Spalte auftreten dürfen. Man kann zwischen drei Kategorien von Constraints unterscheiden [LR13, 200]:

Hohe Kardinalität: Werte einer Spalte sind Unique oder es treten nur wenige Duplikate auf.

Niedrige Kardinalität: Es treten sehr viele Duplikate auf.

Normale Kardinalität: Spalten, die weder eine hohe Kardinalität noch eine niedrige Kardinaliät aufweisen.

3.3 Analyse der Arten von Abhängigkeiten

Bei der Generierung von Werten, die einem oder mehreren Constraints unterliegen, sind je nach Art des Constraints (siehe Abschnitt 3.2), verschiedene Arten von Abhängigkeiten innerhalb der Datenbank denkbar. Die verschiedenen Arten von Abhängigkeiten werden mithilfe einer Baumstruktur dargestellt. Dabei wird in Abbildung 3.1 zuerst eine Übersicht der grundlegenden Abhängigkeitsarten dargestellt.

Die Anzahl der Grundarten von Abhängigkeiten ist, wie in Abbildung 3.1 erkennbar ist, in fünf Grundarten unterteilbar. In den folgenden Abschnitten wird nun detaillierter auf die fünf Abhängigkeitsarten eingegangen.

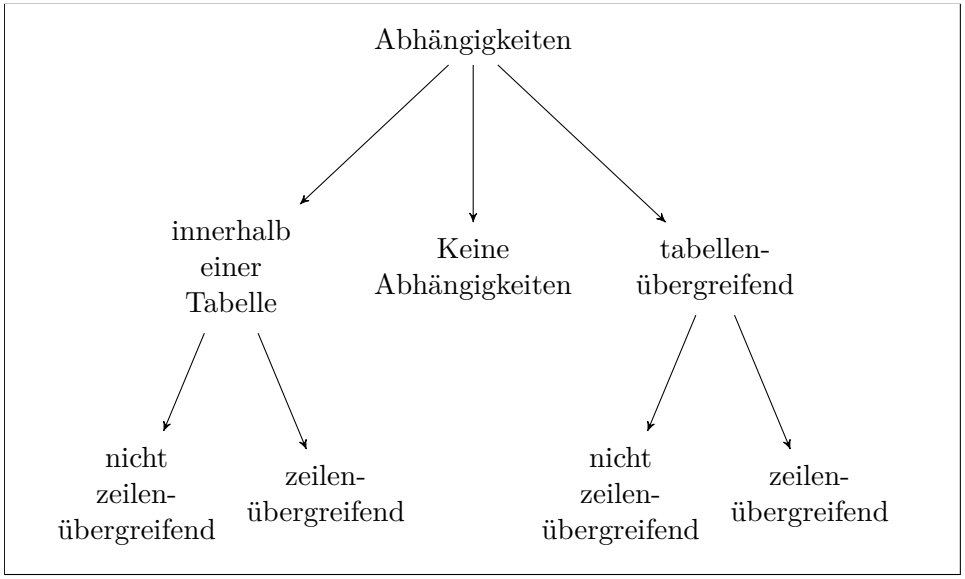

Abbildung 3.1: Übersicht aller Grundarten von Abhängigkeiten

3.3.1 Keine Abhängigkeiten von anderen Werten

Der einfachste Fall ist der, dass ein Constraint über keine Abhängigkeiten zu anderen Werten in der Datenbank verfügt. Der Wert kann deshalb jederzeit und unabhängig von anderen Werten der Datenbank generiert werden. In Abbildung 3.2 werden alle Constraint-Arten aufgeführt, die über keine Abhängigkeiten verfügen. Zuerst wird ein Wert über einen Datentyp eingeschränkt. Weiterführend kann der Wert durch einen domänenspezifischen Datentyp, welcher eine Teilmenge des Datentyps ist, weiter eingeschränkt werden. Für bestimmte Datentypen ist es zusätzlich sinnvoll, den Wertebereich über Grenzen einzuschränken.

Als Beispiel wird ein Prädikats-Constraint betrachtet. Der Wert einer Spalte `Jahr` soll durch ein Modulo-Constraint der Form `a mod b == c` eingeschränkt werden. In diesem Beispiel soll die Variable `a` das zu generierende Feld `Jahr` sein. Die Variablen `b` und `c` werden als Konstanten mit `b` gleich 4 und `c` gleich 2 angegeben. Somit ergibt sich die Bedingung `Jahr mod 4 == 2` die für jede Zelle der Spalte `Jahr` gelten muss und beispielsweise die in Abbildung 3.3 Tabelle erzeugt.

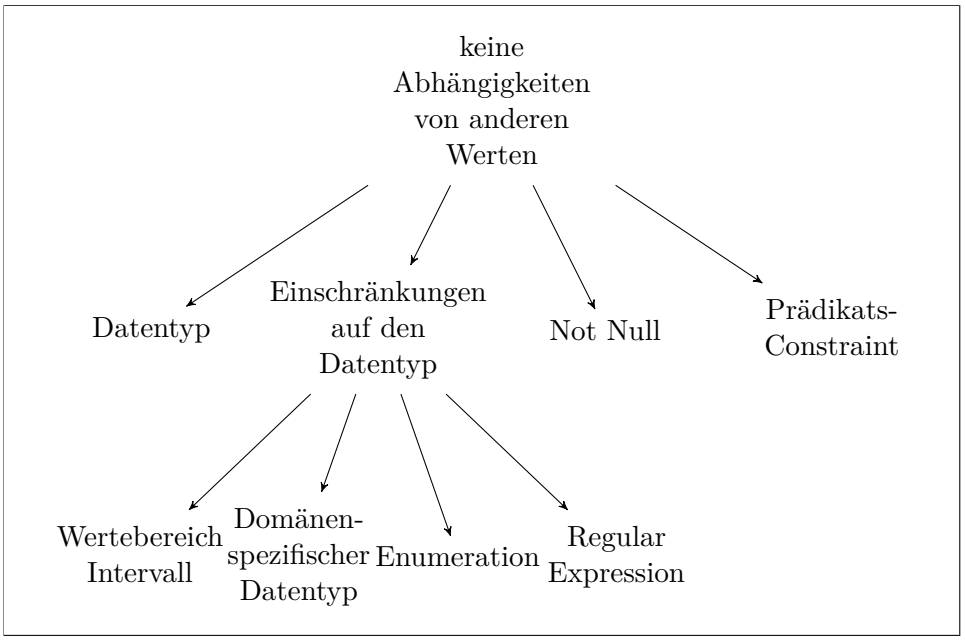

Abbildung 3.2: Keine Abhängigkeiten von anderen Werten

REF	**Land**	**Ort des Finales**	**Weltmeister**	**Jahr**
WM_1	Deutschland	Berlin	Italien	2006
WM_2	Südafrika	Johannesburg	Spanien	2010
WM_3	Brasilien	Rio de Janeiro	Deutschland	2014

Abbildung 3.3: Beispiel für Werte, die über keine Abhängigkeiten zu anderen Werten im Modell verfügen

3.3.2 Nicht-zeilenübergreifende Abhängigkeiten innerhalb einer Tabelle

Zuerst werden die Abhängigkeiten innerhalb einer Tabelle betrachtet, die nicht zeilenübergreifend sind. Man kann bei den Abhängigkeiten zwischen verschiedenen Spalten eines Datensatzes zwischen verschiedenen Arten unterscheiden. Die Einschränkung der Wertemenge kann entweder durch Domain-Constraints, durch Prädikats-Constraints oder durch Funktions-Constraints erfolgen.

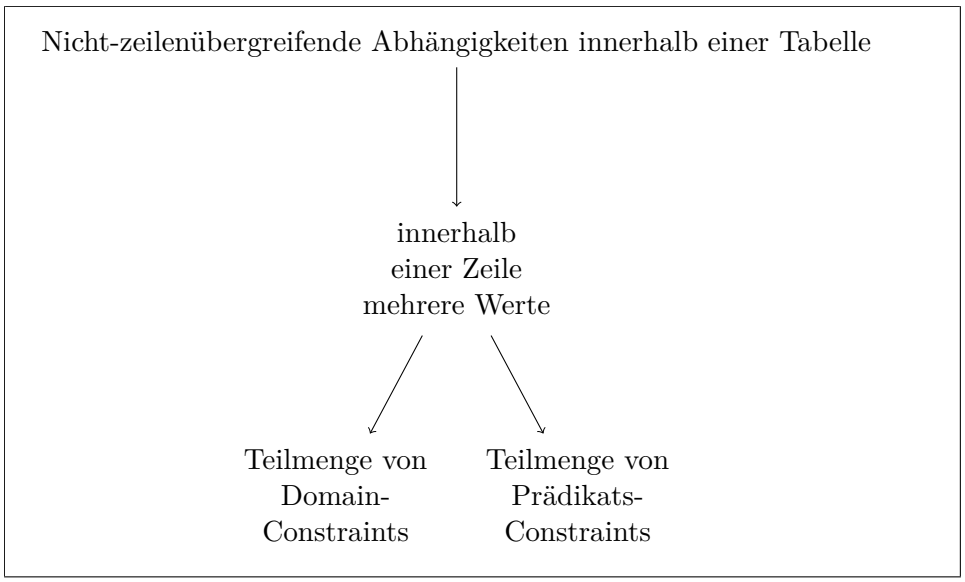

Abbildung 3.4: Nicht-zeilenübergreifende Abhängigkeiten innerhalb einer Tabelle

In dem in Abbildung 3.5 dargestellten Beispiel ist sowohl die Spalte `geschlecht` als auch die Spalte `sprache` abhängig von der Spalte `vorname`, da Vornamen dem Geschlecht und Sprachen zugeordnet werden können. Der erzeugte Datensatz gilt nur dann als valide, falls die Werte der Spalten `geschlecht` und `sprache` passend zu dem Wert der Spalte `vorname` generiert werden.

3.3.3 Zeilenübergreifende Abhängigkeiten innerhalb einer Tabelle

In datenbankbasierten Anwendungen werden jedoch oftmals auch Daten benötigt, die durch Constraints eingeschränkt sind, welche zeilenübergreifenden Abhängigkeiten unterliegen. Dabei handelt sich oftmals um im Datenbankschema definierten Indizes, wie `Primary Key` oder Spalten, die

REF	vorname	geschlecht	sprache	land
AUTOR_1	Adalie	weiblich	französisch	Frankreich
AUTOR_2	Alan	männlich	englisch	USA
AUTOR_3	Peter	männlich	deutsch	Deutschland

Abbildung 3.5: Beispiel für domänenspezifische Daten mit Abhängigkeiten innerhalb einer Zeile

als `Unique` spezifiziert sind. Zusätzlich sind jedoch auch zeilenübergreifende Abhängigkeiten denkbar, die durch funktionale Abhängigkeiten oder durch Kardinalitäts-Constraints beschrieben sind. In Abbildung 3.6 werden die verschiedenen Möglichkeiten von zeilenübergreifenden Abhängigkeiten innerhalb einer Tabelle grafisch dargestellt.

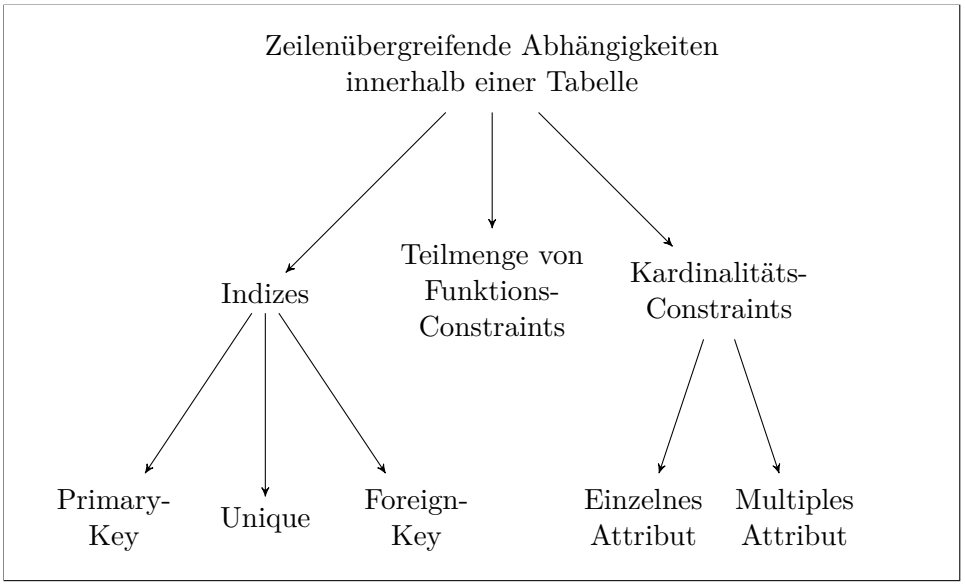

Abbildung 3.6: Zeilenübergreifende Abhängigkeiten innerhalb einer Tabelle

In Abbildung 3.7 wird ein Beispiel für spaltenübergreifende Abhängigkeiten gezeigt. Die Spalte `email` ist im Datenbank-Schema als Unique definiert. Somit dürfen die Werte dieser Spalte keine Übereinstimmung besitzen.

3.3.4 Tabellen- und nicht-zeilenübergreifende Abhängigkeiten

Da in datenbankbasierten Anwendung Daten nicht nur Abhängigkeiten innerhalb von Tabellen besitzen können, werden in Abbildung 3.8 mögliche

REF	vorname	nachname	email
AUTOR_1	Adalie	Dubois	adalie.herrman@mail.com
AUTOR_2	Alan	Willis	alan67@inter.net
AUTOR_3	Peter	Meyer	peter-m@mail.de
AUTOR_4	Robin	Schuster	robin@schuster.de
AUTOR_5	Adam	Bauer	adam-bauer@inter.net

Abbildung 3.7: Beispiel für zeilenübergreifende Abhängigkeiten innerhalb einer Spalte

tabellenübergreifende Abhängigkeitsarten dargestellt.

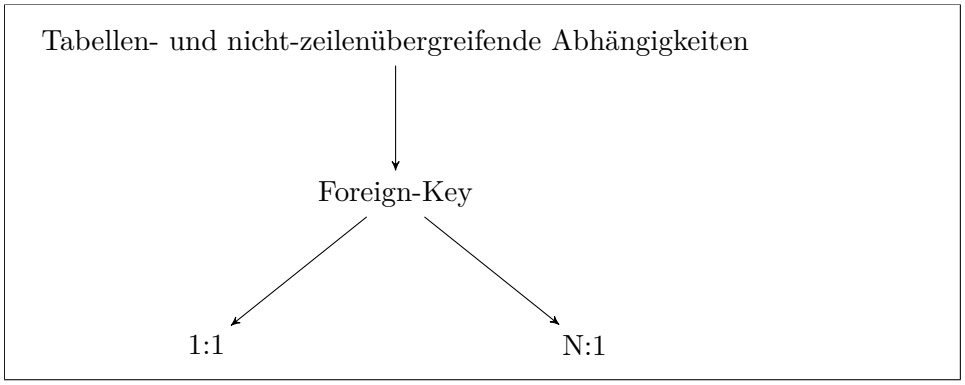

Abbildung 3.8: Tabellen- und nicht-zeilenübergreifende Abhängigkeiten

Der einfachste Fall einer tabellenübergreifenden Abhängigkeit ist eine 1:1-Beziehung zwischen zwei Tabellen einer Datenbank und wird in Abbildung 3.9 dargestellt. In dem dargestellten Beispiel ist der Wert der Spalte `autor.land` abhängig von der Spalte `adresse.land`. In diesem einfachen Fall sollen die Werte dieser beiden Spalten identisch sein.

3.3.5 Tabellen- und zeilenübergreifende Abhängigkeiten

Als letzten Fall werden in Abbildung 3.10 Constraint-Arten dargestellt, die tabellen- und zeilenübergreifende Abhängigkeiten besitzen. Es handelt sich hierbei um funktionale Abhängigkeiten oder Kardinalitäts-Constraints mit einem oder mehreren Attributen.

In dem in Abbildung 3.11 dargestellten Beispiel wird die Spalte `autor.anzahlbuecher` anhand der Anzahl der Einträge in der Tabelle `buch` berechnet, wenn der Primärschlüssel des Autors mit dem Fremdschlüssel in der Tabelle Buch übereinstimmt.

REF	vorname	land	adresse_id
AUTOR_1	Adalie	Frankreich	ADRESSE_1
AUTOR_2	Alan	USA	ADRESSE_2
AUTOR_3	Peter	Deutschland	ADRESSE_3

REF	plz	stadt	land
ADRESSE_1	75015	Paris	Frankreich
ADRESSE_2	10001	New York	USA
ADRESSE_3	78467	Konstanz	Deutschland

Abbildung 3.9: Beispiel für tabellenübergreifende Abhängigkeiten bei 1:1 Beziehungen

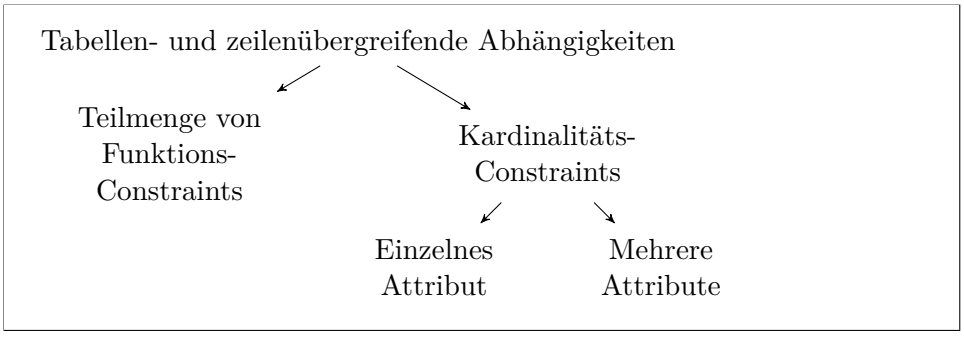

Abbildung 3.10: Tabellen- und zeilenübergreifende Abhängigkeiten

REF	**vorname**	**nachname**	**anzahl_buecher**
AUTOR_1	Adalie	Dubois	1
AUTOR_2	Alan	Willis	2
AUTOR_3	Peter	Meyer	2

buch_id	**autor_id**
BUCH_1	AUTOR_2
BUCH_2	AUTOR_1
BUCH_2	AUTOR_2
BUCH_2	AUTOR_3
BUCH_3	AUTOR_3

REF	**name**	**preis**
BUCH_1	Tristan	10
BUCH_2	Das Nibelungenlied	15
BUCH_3	Erzählungen	20

Abbildung 3.11: Beispiel für beliebige tabellenübergreifende Abhängigkeiten

Kapitel 4

Anforderungsanalyse

In diesem Kapitel wird untersucht, welche Anforderungen an die Umsetztung gestellt werden. Dazu wird in Abschnitt 4.1 zuerst auf die funktionalen Anforderungen eingegangen. Anschließend werden die nichtfunktionalen Anforderungen im Abschnitt 4.2 aufgeführt.

4.1 Funktionale Anforderungen

Im diesem Abschnitt werden alle funktionalen Anforderungen beschrieben, die umgesetzt werden sollen. Die funktionalen Anforderungen werden entsprechend ihrer Wichtigkeit unterschiedlich priorisiert. Es wird zwischen den Prioritäten `hoch`, `mittel` und `niedrig` unterschieden. Übersichtshalber werden die funktionalen Anforderungen in die Abschnitte

- Allgemeine Anforderungen
- Unterstützte Constraint-Arten
- Unterstützte Abhängigkeiten innerhalb des Datenbankmodells
- Anforderungen an die Modellierung

unterteilt.

4.1.1 Allgemeine Anforderungen

Beschreibung	**Priorität**
Realistische und plausible Testdaten durch Generierung von Testdaten anhand von domänenspezifischen Wertgeneratoren: E-Mail-Adressen, Telefonnummern, ISBN.	hoch
Das Datenbankschema wird vollständig im Modell abgebildet.	hoch

Der Algorithmus liefert bei gleichen Eingabeparametern immer die gleichen Wertkombinationen als Ergebnisse.	mittel
Alle Generatoren sind mit allen Constraint-Arten kompatibel und verfügen über kein explizites Wissen über Art der Constraints.	niedrig

4.1.2 Unterstütze Constraint-Arten

Beschreibung	**Priorität**
Datentyp-Constraint Der Datentyp kann explizit für jede Tabellenspalte angegeben werden.	hoch
Range-Constraint Für numerische Datentypen kann eine obere und untere Grenze angegeben werden.	hoch
NotNull-Constraint Spalten sind als NotNull deklarierbar.	niedrig
Unique-Constraint Spalten sind als Unique definierbar.	hoch
PrimaryKey Spalten sind als PrimaryKey definierbar.	hoch
ForeignKey Spalten sind als ForeignKey definierbar.	hoch
Domain-Constraint Domänenspezifische Abhängigkeiten zwischen Spalten sind mithilfe von Domain-Constraints modellierbar.	hoch
Prädikats-Constraint Mit dieser Constraint-Art lassen sich verschiedene Prädikate modellieren.	hoch
Funktions-Constraint Funktionale Abhängigkeiten lassen sich mithilfe des Funktions-Constraint modellieren.	hoch
Kardinalitäts-Constraints Kardinalitäten lassen sich für Spalten festlegen.	niedrig

4.1.3 Unterstützte Abhängigkeiten innerhalb des Datenbankmodells

Beschreibung	**Priorität**
Keine Abhängigkeiten von anderen Werten der Datenbank.	hoch
Abhängigkeiten innerhalb einer Zeile einer Tabelle.	hoch
Abhängigkeiten innerhalb einer Spalte einer Tabelle.	hoch
Tabellen- und nicht-zeilenübergreifende Abhängigkeiten.	hoch
Tabellen- und zeilenübergreifende Abhängigkeiten.	mittel

4.1.4 Anforderungen an die Modellierung

Beschreibung	**Priorität**
Der Tester kann Constraints bei der Modellierung entweder direkt an die Spalte oder an das Modell anhängen.	hoch
Constraints haben mindestens eine Quellspalte, für die der Constraint definiert wird.	hoch
Je nach Constraint-Art kann zusätzlich zur benötigten Quellspalte eine Spalte oder mehrere Spalten, von denen der zu generierende Wert abhängig ist, angegeben werden.	hoch
Die API zur Modellierung von Constraints unterstützt das Builder-Pattern für eine intuitive Benutzung.	mittel

4.2 Nichtfunktionale Anforderungen

In diesem Abschnitt werden nachfolgend die nichtfunktionalen Anforderungen aufgelistet und die Priorität ihrer Umsetzung festgelegt.

Beschreibung	**Priorität**
Die Laufzeit von einigen Minuten soll für die Generierung der Testdaten nicht überschritten werden.	hoch
Es soll mindestens ein Algorithmus angeboten werden, welcher in jedem Fall terminiert.	hoch
Der Algorithmus muss für den Fall, dass eine realistische Menge an Constraints modelliert ist, eine Lösung finden, falls diese existiert.	hoch
Falls keine Lösung existiert, soll der Algorithmus terminieren.	mittel

Kapitel 5

Konzeption

In diesem Kapitel werden Lösungen konzeptionell erarbeitet, welche die in Kapitel 4 gestellten Anforderungen möglichst vollständig erfüllen. Ausgangspunkt ist dabei ein Modell, das bisher über alle definierten Tabellen, Spalten und die mithilfe des in [Mol13] beschriebenen Algorithmus enthaltenen Datensätzen verfügt. Die Datensätze sind jedoch bis auf Spalten, die entweder einen `Primary Key` oder einen `Foreign Key` enthalten, nicht mit Daten gefüllt.

REF	**Name**	**Geschlecht**
AUTOR_1	$Name_1$	$Geschlecht_1$
AUTOR_2	$Name_2$	$Geschlecht_2$
AUTOR_3	$Name_3$	$Geschlecht_3$

Tabelle 5.1: Beispiel für eine ungefüllte Tabelle

Zur Veranschaulichung der Problemstellung betrachtet man zunächst die generierte Tabelle 5.1, die über die Spalten `REF`, `Name` und `Geschlecht` verfügt und mit drei Datensätzen gefüllt ist. Die Datensätze sind bis zu diesem Zeitpunkt noch nicht gefüllt und verfügen nur nur über je einen Primärschlüssel für jeden Datensatz. Man nehme an, dass für diese Tabelle einerseits ein Domain-Constraint für die Spalten `Name` und `Geschlecht` und andererseits ein Unique-Constraint für die Spalte `Name` modelliert ist. Dadurch ergibt sich folgender in Abbildung 5.1 dargestellter Abhängigkeits-Graph, welcher die Abhängigkeiten zwischen den einzelnen Zellen veranschaulicht. Eine Zelle des Graphen ist hierbei über die drei Parameter Tabelle, Spalte und Zeile (auch Datensatz) definiert. Die Kanten werden aus den modellierten Constraints gebildet, wobei pro modelliertem Constraint mehrere Kanten entstehen können. Jede Kante entspricht einer Abhängigkeit zwischen zwei Zellen im Graphen. Im Beispiel werden aus dem Unique-Constraint, welches für die Spalte

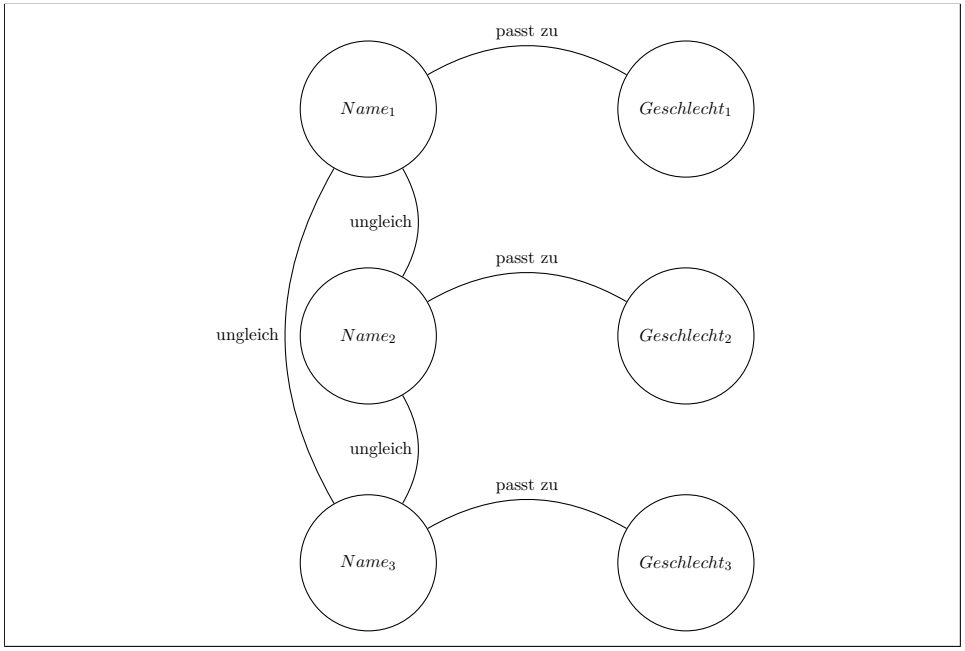

Abbildung 5.1: Beispiel für einen durch Constraints-Abhängigkeiten gebildeten Graph mit sechs beteiligten Zellen

`Name` modelliert wurde, drei Kanten gebildet. Aus dem Domain-Constraint für die Spalten `Name` und `Geschlecht` entsteht je eine Kante pro Datensatz.

In Abschnitt 5.1 wird zunächst eine Schnittstelle beschrieben, mit der es möglich ist, verschiedene Arten von Constraints zu formulieren und diese anschließend dem Modell hinzuzufügen. Anschließend werden in Abschnitt 5.2 Strategien und Algorithmen zur Generierung von Testdaten unter Berücksichtigung der definierten Constraints vorgestellt und auf verschiedene Kriterien verglichen. Darauf folgend wird in Abschnitt 5.3 die Konzeption einer Umsetztung vorgestellt. Dazu werden zuerst die für die Umsetzung benötigten Komponenten und anschließend die Verwendung der Komponenten in den bei der Umsetzung verwendeten Algorithmen beschrieben.

5.1 Modellierung von Constraints

Die in Abschnitt 3.2 vorgestellten Constraint-Arten und in Abschnitt 3.3 beschriebenen Abhängigkeitsarten sollen über eine gemeinsame und einheitliche Schnittstelle formuliert werden können. Für die verschiedenen Arten von Constraints werden dem Benutzer entsprechenden Klassen mit unterschiedlichen Konstruktoren bereitgestellt. Jeder Konstruktor erwartet mindestens eine Referenz auf eine Zelle des Datenbankmodells.

Zusätzlich ist es möglich, einem Constraint konstante Werte zu übergeben. Anschließend kann der Benutzer das definierte Constraint-Objekt dem Datenbankmodell hinzufügen. Im Datenmodell wird für diesen Zweck die Methode `constraint()` zur Verfügung gestellt. Als Übergabeparameter der Methode wird das zuvor durch den Benutzer definierte Constraint-Objekt übergeben.

Alternativ zu der Methode, die Constraints über die verschiedenen Konstruktoren zu erstellen und die Objekte an das Modell anzuhängen, kann noch eine zusätzliche Methode angewendet werden. Anstelle der vielen unterschiedlichen Konstruktoren wird die Parameterübergabe mithilfe des Builder-Patterns[1] realisiert. Die Übergabe der Parameter an das Constraint-Objekt geschieht hierbei nicht über den Konstruktor, sondern wird über geeignete Methoden bereitgestellt. Die Anwendung des Builder-Patterns führt in vielen Fällen zu einer höheren Übsichtlichkeit im Vergleich zur erstgenannten Methode. Wird diese Methode nicht auf das Modell, sondern direkt auf die Spalten angewandt, so kann bei der Definition eines Constraints jeweils auf den Parameter, der einen Constraint an eine Spalte bindet, verzichtet werden.

5.2 Beschreibung und Vergleich verschiedener Strategien zur Auflösung

In diesem Abschnitt werden verschiedene Algorithmen und Strategien vorgestellt und untersucht, welche eine Lösung für die durch Constraints modellierte Problemstellung bieten. Dazu wird jeweils zuerst grob die Funktionsweise beschrieben und nachfolgend anhand eines Beispiels veranschaulicht. Anschließend werden alle Strategien und Algorithmen hinsichtlich nachfolgend aufgelisteter Kriterien untersucht:

Implementierungsaufwand Es wird untersucht, wie viel Implementierungsaufwand nötig ist, um den Algorithmus in die STU-Bibliothek einzubauen.

Garantierte Lösungsfindung Der Algorithmus findet eine Lösung, falls eine Lösung existiert und erkennt andernfalls, dass keine Lösung existiert.

Laufzeit und Skalierung der Laufzeit Es wird untersucht, ob der Algorithmus in einer für das zugrundeliegende Modell annehmbarer Laufzeit

[1] Unter Builder-Pattern versteht man die Trennung von Konstruktion und Repräsentation eines Objekts [Gam07, 119-130].

terminiert und wie der Algorithmus bei steigender Constraint-Zahl, Modellgröße und Abhängigkeitsanzahl skaliert.

Unterstützte Constraint-Arten Es wird untersucht, welche Constraint-Arten aus Abschnitt 3.2 mithilfe des Algorithmus aufgelöst werden können. Im Idealfall sind alle Constraint-Arten mithilfe des Algorithmus auflösbar.

Unterstützte Abhängigkeitsarten Es wird untersucht, ob alle Abhängigkeitsarten vom Algorithmus unterstützt werden.

5.2.1 Generate-and-Test-Methode

Als einfacher und primitiver Ansatz wird zuerst die Generate-and-Test-Methode untersucht. Hierbei werden die einzelnen Werte mithilfe der Generatoren generiert. Nach jeder Generierung der Werte wird überprüft, ob alle Constraints, die diesem Wert zugeordnet sind, gültig sind. Schlägt die Überprüfung eines Constraints fehl, so werden so lange zufällige Werte für diese einzelne Zelle generiert, bis die Prüfung erfolgreich ist.

Diese Vorgehensweise ist hinsichtlich des Implementierungsaufwandes schnell umsetzbar. Der Algorithmus ist zwar für alle Constraint-Arten und alle fünf Abhängigkeitsarten anwendbar, jedoch erhöht sich die Laufzeit sehr stark mit der Anzahl der durch Constraints entstehenden Abhängigkeiten. Umso kleiner die Lösungsmenge im Vergleich zu den vorhandenen möglichen Kombinationen wird, desto länger wird die Laufzeit. Bei ungünstigen Kombinationen von Constraints besteht die Möglichkeit, dass keine Lösung gefunden wird, obwohl eine Lösung existiert. Ebenfalls ist es unmöglich, mithilfe dieser Methode herauszufinden, ob eine Lösung oder ob keine Lösung existiert. Da durch den Algorithmus nicht ermittelt werden kann, ob für die aktuelle Constraint-Kombination und die bereits generierten Werte noch Lösungen existieren, verwirft der Algorithmus bereits generierte Werte nicht.

5.2.2 Lösung durch Aufstellen eines linearen Ungleichungssystem

Unter linearen Ungleichungssystemen versteht man eine Menge von mathematischen Ungleichungen. Die einzelnen Ungleichungen bestehen aus beliebig vielen Variablen und mathematischen Operationen. Für die Lösung von linearen Ungleichungssystemen existieren verschiedene Bibliotheken. Eine Bibliothek, welche in einer Java-Umgebung verwendet werden kann, ist beispielsweise das LPSolve-Framework [B+07].

Bei diesem Ansatz wird versucht, die durch Constraints enstehenden Abhängigkeiten mithilfe eines linearen Ungleichungssystems zu lösen. Dabei

müssen die verschiedenen Arten von Constraints und Datentypen auf ein lineares Ungleichungssystem abgebildet werden. Nachdem ein lineares Ungleichungssystem aufgebaut wurde, kann dieses mathematisch aufgelöst werden. Ist das lineare Ungleichungssystem nicht lösbar, so bedeutet dies, dass für die aufgestellten Constraints keine Lösung existiert. Wurde das lineare Ungleichungssystem erfolgreich aufgelöst, so ist es möglich, gültige Werte zu generieren, die alle Constraints befolgen.

Da für lineare Ungleichungssysteme Algorithmen zur Lösung existieren, terminiert der Algorithmus in jedem Fall und findet eine Lösung, falls eine Lösung existiert bzw. erkennt, dass keine Lösung existiert. Das Aufstellen eines linearen Ungleichungssystems ist jedoch nur für einige Constraint-Arten und nur für numerische Datentypen möglich. Für andere Constraint-Arten wie dem Domain-Constraint ist dieser Ansatz nicht anwendbar, da die Constraint-Arten nicht als Lineares Ungleichungssystem abbildbar sind.

5.2.3 Lösung mithilfe von Constraint-Programmierung

Ein weiterer möglicher Ansatz ist die Verwendung von Algortithmen aus dem Bereich der Constraint-Programmierung. Unter Constraint-Programmierung versteht man Programmierparadigmen, bei denen man die Beziehungen zwischen verschiedenen Variablen durch bestimmte Ausdrücke formulieren kann [HW07]. Für alle unbekannten Variablen wird anschließend anhand schon bekannter Werte und den formulierten Ausdrücken eine Lösung gesucht. Die Aufstellung und Lösung solcher Probleme wird Constraint-Solving-Problem genannt [HW07].

Bei der Verwendung von Constraint-Programmierung wird ähnlich vorgegangen wie bei der Lösung mithilfe eines linearen Ungleichungssystems. Für den Einsatz der Constraint-Programmierung in Java-Umgebungen sind verschiedene Bibliotheken verfügbar. Beispiele für Bibliotheken, die in Java-Umgebungen eingesetzt werden können, sind das JaCoP-Framework [KS14] oder das Choco-Framework [CP14]. Durch den Einsatz der Constraint-Programmierung kann gewährleistet werden, dass eine Lösung für ein formuliertes Problem gefunden wird, falls eine Lösung für dieses Problem existiert.

Bei der Verwendung eines Constraint-Solving-Frameworks werden im ersten Schritt die definierten Constraints in die von der verwendeten Bibliothek angebotene Form transformiert. Die Bibliothek löst im nächsten Schritt das Constraint Solving Problem. Ähnlich wie beim Ansatz der Lösung durch ein lineares Ungleichungssystem unterstützen die verschiedenen Bibliotheken nur eine Teilmenge der vorhandenen Constraint-Arten und Datentypen. Somit kann dieses Konzept nicht grundsätzlich für diesen Einsatzbereich eingesetzt

werden, sondern könnte nur für einige Constraint-Arten und Datentypen verwendet werden. Da jedoch für diesen Einsatzbereich ein Lösungsansatz gesucht wird, der für alle benötigten Constraint-Arten funktioniert, wird kein Constraint-Programmierungs-Framework eingesetzt.

5.2.4 Tiefensuche

Mithilfe von Tiefensuche sollen möglichst alle Wertkombinationen systematisch durchlaufen und überprüft werden. Bei der Tiefensuche werden alle Knoten und alle Kanten in einem Baum durchsucht. Der Tiefensuchealgorithmus durchläuft den Graphen indem er entlang der Kanten von Knoten zu Knoten wandert [SWW14, S.566-576]. Wird nun jede Wertkombination als Knoten in einem Baum gesehen, dann kann dieser Baum mithilfe der Tiefensuche durchlaufen werden. Eine Wertkombination kann als Liste von Indizes gesehen werden, wobei jeder Index der Liste für den Index eines generierten Wertes steht. Der Aufbau des Graphen wird hierbei rekursiv durchgeführt. Jeder Knoten generiert dazu, die Anzahl an neuen Knoten, wie es Werte in der Wertkombination gibt. Neue Knoten erhält man dadurch, dass die neu generierten Wertkombinationen jeweils über nur einen geänderten Index im Vergleich zum Ursprungsknoten verfügen.

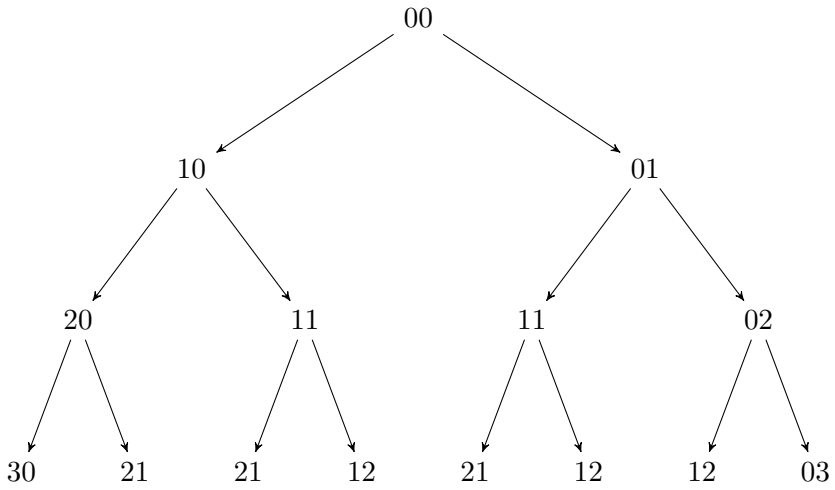

Abbildung 5.2: Kombinationsmöglichkeiten, die durch Tiefensuche untersucht werden

In Abbildung 5.2 ist in einem Baum bis zu einer Tiefe von zwei beispielhaft dargestellt, wie die Wertkombinationen bestehend aus zwei Variablen erstellt werden. Es wird mit der Wertkombination `00` begonnen. Die beiden entstehenden Kindknoten aus diesem Knoten sind die Wertkombinationen `10` und `01`. Wie man unschwer erkennen kann, generiert diese Vorgehensweise

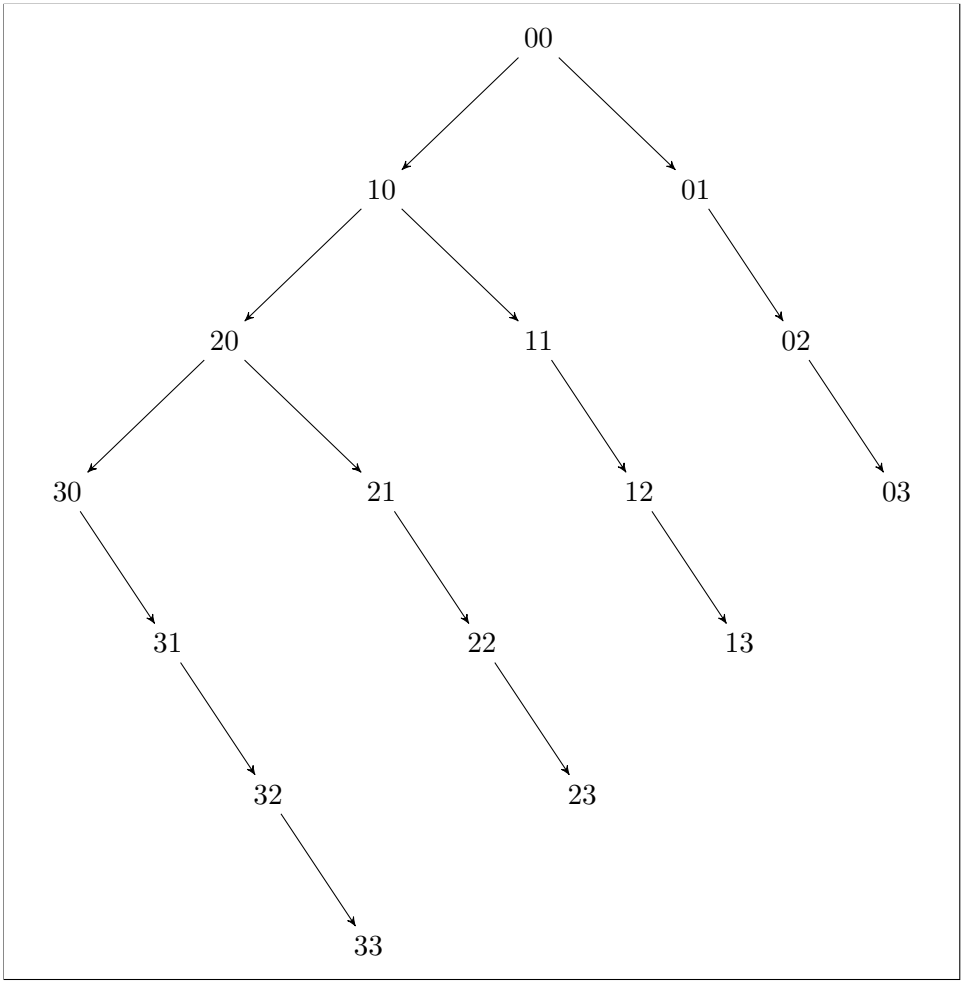

Abbildung 5.3: Kombinationsmöglichkeiten bis zu vier unterschiedlichen Werten pro Variable, die durch Tiefensuche untersucht werden

überflüssigerweise einige Wertkombinationen mehrfach an verschiedenen Stellen des Baumes. Um dies zu verhindern muss die Tiefensuche bei Erkennung einer bereits überprüften Wertkombination an diesem Knoten nicht mehr weitergeführt werden. Werden bereits untersuchte Wertkombinationen im Baum nicht mehr besucht so ergibt sich der Baum aus Abbildung 5.3. Dieser Baum wird nicht nur bis zu einer Tiefe von zwei dargestellt, sondern zeigt alle Wertkombinationen, die durch zwei Variablen und vier möglichen Werten pro Variable entstehen.

Man nehme an, es stehen zwei Zellen `a` und `b` durch das Constraint `b > a` und das Constraint `b mod 4 = 0` in Abhängigkeit. Es soll nun eine gültige Wertkombination zwischen den zwei Zellen gefunden werden, wobei für jede

Zelle in Tabelle 5.2 vier unterschiedliche Werte zur Verfügung stehen. In Abbildung 5.4 werden nun alle möglichen Kombinationen in einem Graphen mit Baumstruktur dargestellt, die mit der Tiefensuche durchlaufen werden. Die erste valide Kombination für die beide Constraints gültig sind, sind die Werte `a = 2004` und `b = 2008`. Für dieses Beispiel würde die Tiefensuche somit erst im Worst-Case eine Lösung finden.

Index	Zelle a	Zelle b
0	2004	1996
1	2006	2000
2	2005	2004
3	2007	2008

Tabelle 5.2: Beispiel für die ersten vier generierten Werte bei zwei Variablen

5.2.5 Reduzieren der Wertemenge

Ein möglicher Ansatz zur Optimierung ist die Reduzierung der Wertemenge anhand der zugehörigen Regeln, bevor der Wertgenerator ausgeführt wird. Dabei ist eine unterschiedliche Herangehensweise für die verschiedenen Constraint-Arten aus Abschnitt 3.2 und der zugrundeliegenden Wertemenge erforderlich. Man kann zwischen zwei Arten von Wertemengen unterscheiden:

Endliche Wertemenge Alle möglichen Werte sind als eine Menge von Werten darstellbar.

Unendliche Wertemenge Mögliche Werte sind durch eine obere und eine untere Grenze darstellbar.

5.2.5.1 Reduzieren einer endlichen Wertemenge

Bei der Reduzierung einer endlichen Wertemenge sind zwei Vorgehensweisen denkbar. Entweder alle ungültigen Werte werden aus der Wertemenge entfernt und es wird aus der resultierenden Menge gültiger Werte ein Wert ausgewählt oder es werden zur Generierung ausschließlich gültige Werte angegeben aus denen ein Wert ausgewählt wird. Als Ergebnis beider Vorgehensweisen erhält man im Idealfall eine Wertemenge aus ausschließlich gültigen Werten, sodass nach einem Aufruf des Wertgenerators ein gültiger Wert für eine Zelle ermittelt wird.

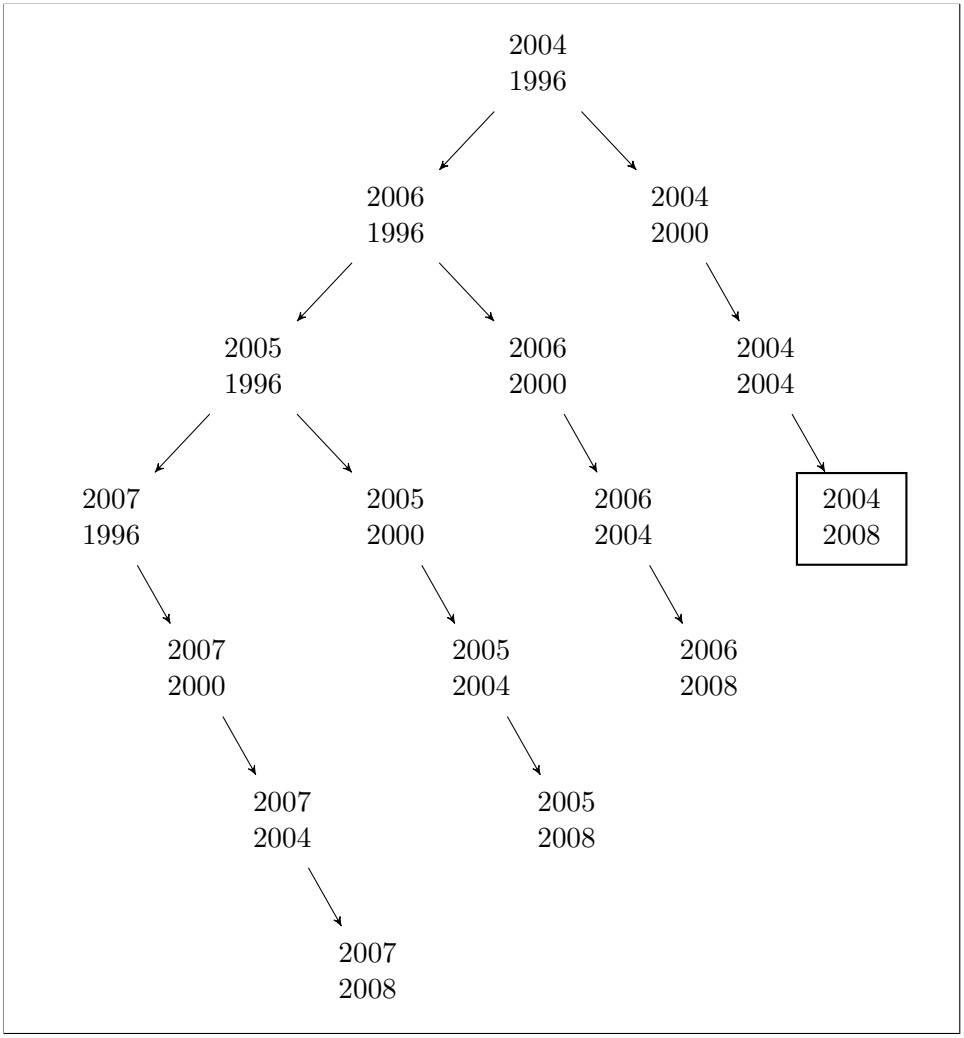

Abbildung 5.4: Beispiel für Kombinationsmöglichkeiten, die durch Tiefensuche untersucht werden

5.2.5.2 Eingrenzen von unendlichen Wertemengen

Um bei der Auswahl eines Wertes aus einer unendlichen Wertemenge die Wahrscheinlichkeit zu erhöhen einen gültigen Wert zu erhalten, ist es möglich die obere und untere Grenze der Wertemenge einzuschränken. Dadurch wird eine Teilmenge an Werten, die nicht Teil einer gültigen Lösung sind, ausgeschlossen.

5.3 Konzeption der Umsetzung in STU

Nachdem die durch den Benutzer definierten Constraints an das Datenbankmodell gekoppelt wurden, müssen diese bei der Generierung der Testdaten berücksichtigt werden. Dazu werden verschiedene Strategien und Algorithmen eingesetzt, die die Generierung von Testdaten ermöglichen, welche durch Constraints definierte Abhängigkeiten verfügen. Aus den zuvor vorgestellten Ansätzen wird daher nicht nur ein einziger Ansatz verwendet, sondern es werden verschiedene Algorithmen und Vorgehensweisen kombiniert, um möglichst viele Anforderungen zu erfüllen.

Für die Umsetzung in der STU-Bibliothek wird als Basis der Tiefensuchealgorithmus eingesetzt werden. Zusätzlich werden verschiedene Strategien zur Reduzierung der Wertemenge verwendet werden. Für die Umsetzung werden hierzu einige Komponenten benötigt, die in Abschnitt 5.3.1 eingeführt und beschrieben werden. Anschließend werden die verwendeten Algorithmen und Strategien in Abschnitt 5.3.2 anhand dieser Komponenten erläutert.

5.3.1 Beschreibung der Komponenten

Für die Generierung der im Datenbank-Modell beschriebenen und durch Constraints eingeschränkten Werte sind unterschiedliche Komponenten vorgesehen. In der Abbildung 5.5 werden alle beteiligten Komponenten und deren Zusammenarbeit dargestellt. In den nachfolgenden Abschnitten wird anschließend auf die einzelnen Komponenten und deren Aufgaben eingegangen.

5.3.1.1 Value-Generator

Dem Benutzer stehen verschiedene Wertgeneratoren zur Verfügung, die für die Generierung einzelner Werte verwendet werden. Jeder Wertgenerator liefert als Ergebnis einen Wert eines bestimmten Typs. Der Typ des Rückgabewertes kann entweder ein elementarer Datentyp oder ein domänenspezifischer Datentyp sein. So liefert ein elementarer Wertgenerator beispielseweise Integer-Werte. Ein bestimmter domänenspezifischer Wertgenerator liefert dagegen ausschließlich E-Mail-Adressen.

Man kann zwischen zwei verschiedenen Vorgehensweisen beim Generieren der Werte unterscheiden. Der Wertgenerator wählt entweder aus einer endlichen Menge von vorgegebenen Werten oder aus einem vorgegebenen Intervall einen Wert aus. Die Rückgabewerte werden in einer Value-Klasse gekapselt, die im Abschnitten 5.3.1.5 beschrieben wird.

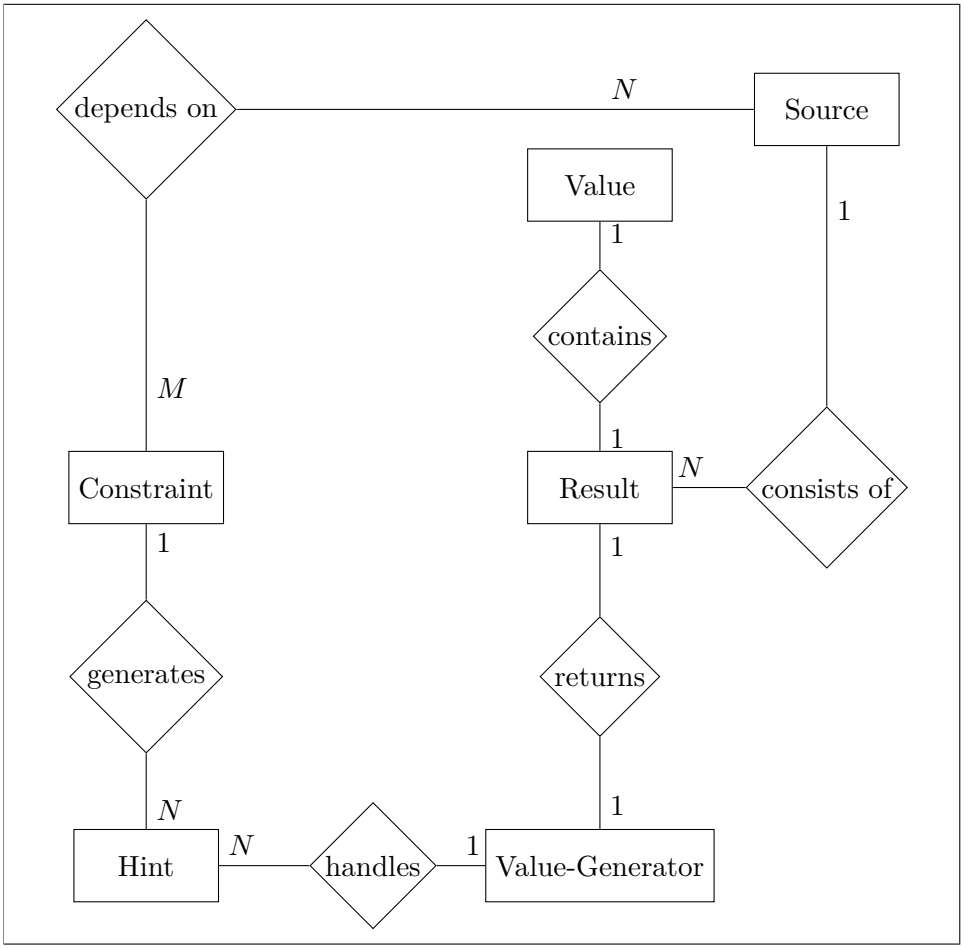

Abbildung 5.5: Entity-Relationship-Diagramm der für die Wertgenerierung beteiligten Komponenten

5.3.1.2 Constraint

Unter Constraint versteht man eine Einschränkung auf den Wertebereich einer Zelle im Modell. Constraints schränken mindestens eine Zelle des Modells ein. Theoretisch soll es jedoch möglich sein, dass alle Werte einer Datenbank durch Constraints voneinander abhängen können. Deshalb bezieht sich ein Constraint immer auf eine Menge von Quellen, welche in Abschnitt 5.3.1.3 beschrieben ist. Alle Constraints stellen eine Schnittstelle bereit, über die die anderen Komponenten auf die Funktionalität von Constraints zugreifen können. Zu den Hauptfunktionen gehören die Prüfung, ob ein Constraint erfüllt ist, sowie die Erzeugung von Hint-Objekten anhand der bereits generierten Werte.

5.3.1.3 Source

In Abschnitt 3.3 wurden die verschiedenen Arten von Abhängigkeiten, die durch Constraints gebildet werden können, vorgestellt. Mithilfe von Source-Objekten werden diese verschiedenen Arten modelliert. Dazu enthält das Source-Objekt einerseits die Informationen über den Typ der Quelle und andererseits Referenzen auf alle Result-Objekte, die diese Quelle einschließt.

5.3.1.4 Result

Jede Zelle in der Datenbank ist mit den Parametern Tabelle, Spalte und Zeile eindeutig identifizierbar und wird mithilfe der in Abschnitt 5.3.1.1 vorgestellten Wertgeneratoren erzeugt. Das Ergebnis aller Wertgeneratoren ist ein Result-Objekt. Das Ergebnis enthält neben dem eigentlichen erzeugten Wert (siehe Abschnitt 5.3.1.5) auch Informationen über die Position im Modell und über den für dieses Ergebnis zuständigen Wertgenerator. Zusätzlich kann ein Wert im Modell beliebig vielen vom Benutzer definierten Constraints unterliegen, welche ebenfalls im Result-Objekt in einer Liste gespeichert sind.

5.3.1.5 Value

Der Datentyp und der Wert der Zelle wird mithilfe der Value-Klasse gekapselt. Es handelt sich hierbei um eine abstrakte Template-Klasse bei der der Datentyp des Wertes in der Unterklasse festgelegt wird. Somit wird ermöglicht, dass Werte von Zellen, die nicht vom gleichen Datentyp sind, allgemein verglichen werden können.

5.3.1.6 Hint

Für eine optimierte Lösungsfindung werden Hint-Objekte eingeführt. Hint-Objekte werden von Constraint-Objekten anhand der bereits generierten Werte erzeugt. Die erzeugten Objekte werden dann wiederum von Wertgeneratoren verarbeitet und interpretiert um ungültige Kombinationen vor der Wertgenerierung bereits auszuschließen. In Abschnitt 5.3.2.3 wird das Vorgehen näher erläutert.

5.3.2 Strategien und Algorithmen

In diesem Abschnitt wird beschrieben, wie durch die in den vorherigen Abschnitten vorgestellten Komponenten der Tiefensuchealgorithmus mit dem zusätzlichen Hint-Konzept zur Wertebereich-Einschränkung in die STU-Bibliothek integriert werden kann. Dazu wird zuerst in Abschnitt 5.3.2.1 der Aufbau der durch Constraints entsehenden Abhängigkeits-Graphen erläutert. Anschließend wird in Abschnitt 5.3.2.2 die Tiefensuche und in Abschnitt 5.3.2.3 die Optimierung des Algorithmus mithilfe der Einschränkung der Wertebereiche durch das Hint-Konzept beschrieben.

5.3.2.1 Aufbau der durch Constraints entstehenden Abhängigkeits-Graphen

Als Ausgangslage kann man sich eine drei-dimensionale Matrix vorstellen, wobei jede Zelle in der Matrix über die Parameter Tabelle, Zeile und Spalte definiert ist. Für einzelne Zellen dieser Matrix können nun beliebig viele Constraints definiert sein, welche wiederum über Abhängigkeiten auf beliebig viele andere Zellen verfügen können. Das Ziel des Algorithmus ist nun das Finden von gültigen Werten für alle Zellen, welche durch die Constraints miteinander verknüpft sind, sodass alle Constraints eingehalten werden.

Der Aufbau des Graphen beginnt dadurch, dass zuerst alle Zellen des Modells nacheinander durchlaufen werden. Wird eine Zelle erreicht, die über Constraint-bedingte Abhängigkeiten zu anderen Zellen verfügt, in eine Liste aufgenommen. Alle Zellen, die abhängig von der ersten sind, werden ebenfalls in diese Liste aufgenommen. Danach werden alle abhängigen Zellen weiterhin dahingegend untersucht, ob diese ebenfalls zu anderen Zellen Abhängigkeiten besitzen. Durch dieses rekursive Vorgehen erhält man eine Liste, welche über alle Zellen verfügt, die miteinander in Abhängigkeit stehen. Gleichzeitig wird eine zweite Liste angelegt, welche alle Constraints speichert. Der durch die zwei Listen beschriebene Graph kann nun systematisch auf eine valide Lösung durchsucht werden. Dazu wird die in Abschnitt 5.3.2.2 beschriebene Tiefensuche verwendet.

5.3.2.2 Tiefensuche

Nachdem der Graph mit allen beteiligten Zellen erstellt wurde, sucht der Algorithmus für alle sich in der Liste befindlichen Zellen gültige Werte, welche alle Constraints erfüllen. Als grundsätzlicher Algorithmus wird der Tiefensuchealgorithmus eingesetzt. Beim Tiefensuchealgorithmus werden alle möglichen Kombinationen nacheinander durchprobiert. Wird eine gültige Lösungskombination gefunden, so terminiert der Algorithmus. In der Theorie kann somit in jedem Fall eine Lösung gefunden werden, falls eine Lösung existiert. In der Praxis ist es jedoch häufig nicht möglich eine Lösung in annehmbarer Zeit zu finden. Deshalb ist es notwendig zusätzlich Strategien einzusetzen, welche eine schnellere Lösungsfindung ermöglichen.

5.3.2.3 Wertebereiche des Wertgenerators mithilfe von Hints eingrenzen

Damit die Wahrscheinlichkeit erhöht wird, dass Wertgeneratoren für die aktuelle Constraint-Kombination gültige Werte zurückliefert, werden dem Generator vor der Generierung Hints übergeben. Man kann zwischen verschiedenen Strategien unterscheiden um mithilfe von Hints die Wertemenge bzw. den Wertebereich innerhalb von Wertgeneratoren einzuschränken.

Gültige Werte vorschlagen Eine einfache Strategie um zu gewährleisten, dass ein Generator nur gültige Werte zurückgibt, ist die Vorgabe von einem oder mehreren gültigen Werten. Der Wertgenerator sucht dann einfach zufällig einen Wert aus der gültigen Wertemenge aus und gibt diesen zurück.

Ungültige Werte ausschließen Eine ähnliche Strategie, ist die umgekehrte Vorgehensweise. Dem Generator werden ausschließlich ungültige Werte per Hint übergeben. Der Generator darf nun keinen Wert aus dieser Wertemenge zurückliefern.

Grenzen des Wertebereichs festlegen Handelt sich bei dem Wertgenerator um einen Generator der nicht aus einer endlichen Wertemenge einen Wert auswählt, sondern einen Wert aus einer unendlichen Menge auswählt, so ist es möglich über entsprechende Hints den Wertbereich so zu beschränken, dass der Generator einen Wert aus einem Wertebereich mit ausschließlich gültigen Werten auswählt.

Kapitel 6

Umsetzung

In diesem Kapitel wird beschrieben, wie die in Kapitel 5 dargestellten Konzepte innerhalb der STU-Bibliothek umgesetzt werden. Dazu wird in Abschnitt 6.1 zuerst auf Modellierung der verschiedenen umgesetzten Constraints-Arten eingegangen. In Abschnitt 6.2 werden anschließend die eingesetzten Algorithmen und Strategien zur Lösung der durch Constraints modellierten Abhängigkeiten eingegangen.

6.1 Modellierung von Constraints

Für die Modellierung von Constraints werden zwei unterschiedliche Möglichkeiten unterstützt. Ein Constraint kann entweder an das Datenbankmodell oder direkt an eine Spalte einer Tabelle angehängt werden. In den folgenden zwei Abschnitten werden die Möglichkeiten der API anhand von Beispiel-Code veranschaulicht.

6.1.1 API zur Definition der Constraints im Modell

Das Hinzufügen von Constraints im Datenbankmodell wird mit dem Schlüsselwort „constraint“ ermöglicht. Zum Hinzufügen eines Constraints muss zum einen der Constraint-Typ und zum anderen die Tabellenspalte, auf die der Constraint angewendet werden soll, angegeben werden.

```
constraint(new UniqueConstraint("autor.email"));
constraint(new ModConstraint("turnier.jahr", new IntValue(4), new IntValue(2))
    ;
constraint(new DomainConstraint("autor.vorname", "autor.geschlecht"));
constraint(new GreaterEqualConstraint("autor.mitgliedseit", "autor.geburtsjahr"
    ));
```

Listing 6.1: Beispiel für die Schnittstelle zur Formulierung von Constraints im Modell

```
column("isbn", DataType.VARCHAR).constraint().unique();
column("vorname", DataType.VARCHAR).constraint().domain("autor.geschlecht");
```

Listing 6.2: Beispiel für die Schnittstelle zur Formulierung von Constraints direkt an der Spalte

6.1.2 API zur Definition der Constraints direkt an der Spalte

Um die Benutzerfreundlichkeit der Bibliotheks-API zu erhöhen, wird für essentielle Constraints das Builder-Pattern implementiert. Wie in dem Beispiel aus Listing 6.2 zu sehen ist, wird ein Constraint mit Hilfe der Methode `constraint()` direkt an eine Spalte angehängt. Die Methode liefert ein `ConstraintBuilder` zurück, dass wiederum verschiedene Constraints-Arten als Methode bereitstellt. Die Methode der `ConstraintBuilder`-Klasse liefert daraufhin das `ColumnBuilder`-Objekt zurück. Dies ermöglicht es mehrere Constraints pro Spalte zu definieren und gleichzeitig eine übersichtliche Form beizubehalten.

6.2 Detailierte Beschreibung des Algorithmus

Als eine Alternative zur Verwendung eines einfachen Generate-and-Test-Algorithmus wird der Tiefensuchealgorithmus implementiert. Der Vorteil der Verwendung eines Tiefensuchealgorithmus, ist die garantierte Findung einer Lösung, falls diese existiert. Da im Allgemeinen nicht alle Werte über Constraints verfügen und die Constraints-Abhängigkeiten sich nicht über die gesamte Modell-Struktur erstrecken, ist es nicht sinnvoll den Tiefensuchealgorithmus über die gesamte Modell-Struktur laufen zu lassen. Dies heißt, dass der Tiefensuchealgorithmus nur alte Werte wieder verwerfen soll, die direkt mit dem aktuellen Wert in Verbindung stehen.

6.2.1 Implementierung der Tiefensuche

Für den Algorithmus werden die in Listing 6.3 globalen Variablen verwendet. In drei Listen werden alle im aktuellen Graphen berücksichtigten Constraints, Ergebnisse, sowie alle bereits geprüften Kombinationen gespeichert. In den globalen Variablen `maxDepth` und `maxDuration` werden die maximale Tiefe die der Tiefensuchealgorithmus durchläuft und die maximale Laufzeit bis die Generierung abgebrochen wird angegeben.

Nachdem das Modell mithilfe des in der Arbeit [Mol13] beschriebenen Algorithmus erzeugt wurde, wird die Methode `generate`, die in Listing 6.4 zu sehen ist, aufgerufen. Diese durchläuft die gesamte Struktur und somit

```
ArrayList<Constraint> constraintList;
ArrayList<Result> resultList;
ArrayList<Combination> combinationList;

Integer maxDepth = 5;
Integer maxDuration = 60;
```

Listing 6.3: Globale Variablen des Algorithmus

```
void DataGenerator.generate(){
  for(Table table: tables){
    for(Entity entity: entities){
      for(Column column: columns){
        Result result = getResult(table, entity, column);
        if(result.isGenerated())
          continue;

        addResult(result);   // 1. Schritt: Graph erstellen
        generateValues();    // 2. Schritt: Gültige Werte finden

        // Aufräumen aller globalen Listen
      }
    }
  }
}
```

Listing 6.4: generateValues()-Methode

jede Zelle des Datensatzes. Ist für die Zelle noch kein Wert generiert, so wird die Methode `addResult()` aufgerufen. Diese Methode baut einen Graphen aller Results auf, die durch Constraints voneinander abhängig sind. Anschließend wird für alle Zellen im zuvor erstellten Graphen ein gültiger Wert gesucht. Nachdem die Werte für den aktuellen Graphen generiert wurden, werden die drei globalen Listen geleert und zur nächsten Zelle in der Datenbankmodell gesprungen.

Mithilfe der Methode `addResult()` aus Listing 6.5 wird eine Zelle, die noch keinen generierten Wert enthält, zur Result-Liste hinzugefügt. Unterliegt diese Zelle einem oder mehreren Constraints, so wird die Methode `addConstraint()` aufgerufen, die in Listing 6.6 beschrieben ist.

Die Methode `addConstraint()` aus Listing 6.6 fügt das übergebene Constraint zur globalen Constraint-Liste hinzu. Anschließend werden alle Zellen, die durch dieses Constraint in Abhängigkeit stehen, mit Hilfe

```
void addResult(Result result){
  if(resultList.contains(result))
    return;

  for(Constraint constraint: result.getConstraints()){
    addConstraint(constraint);
  }
}
```

Listing 6.5: addResult(Result result)-Methode

```
// Füge Constraint zur constraintList hinzu und rufe addResult für alle
     beteiligten results auf
void addConstraint(Constraint constraint){
  if(constraintList.contains(constraint))
    return;

  for(Source source: constraint.getSources()){
    for(Result result: source.getResult()){
      if(!resultList.contains(result))
        addResult(result);
    }
  }
}
```

Listing 6.6: addConstraint(Constraint constraint)-Methode

```
// Generiere gültige Werte für alle Werte in der resultList
boolean generateValues(){
  Combination combination = new Combination(resultList.length);
  maxDepth = calcMaxDepth(resultList.length); // Maximale Tiefe wird anhand der
       Anzahl der beteiligten Zellen berechnet

  Combination combination = generateValue(0, combination);
  if(constraintList.size() > 0 && combination == null){
    setAllResultsNull();
    return false;          // Werte nicht erfoglreich generiert.
  }
  return true;             // Werte erfolgreich generiert.
}
```

Listing 6.7: generateValues()-Methode

der in Listing 6.5 vorgestellten Methode `addResult()` zur Result-Liste hinzugefügt.

Nachdem nun der erste Schritt mit der Erstellung des Graphs fertig gestellt wurde, werden nun im zweiten Schritt die Werte für alle Zellen im aktuellen Graph erstellt. Dieser Vorgang wird mit der Methode `generateValues()` aus Listing 6.7 gestartet. Zuerst wird maximale Tiefe des Tiefensuchealgorithmus, anhand der Anzahl der abhängigen Zellen, berechnet. Anschließend wird mit Hilfe der Methode `generateValue()`, die nachfolgend in Listing 6.8 dargestellt ist, die erste mögliche Kombination an Werten generiert. Wurde keine gültige Kombination gefunden, so werden alle Werte des Graphen auf `null` gesetzt und `false` zurückgegeben, sodass sichtbar wird, dass keine Werte gefunden wurden die alle Constraints erfüllen konnten. Andernfalls wurde eine gültige Wertkombination gefunden und es kann `true` zurückgegeben werden.

Nachdem die Methode `generateValue()` durch die Methode `generateValues()` aus Listing 6.7 einmalig aufgerufen wurde, prüft diese rekursiv solange neue Wertkombinationen bis eine gültige Wertkombination für alle Constraints in der Constraint-Liste gefunden wurde. Dazu wird zuerst mit Hilfe der globalen Liste `combinationList` abgefragt, ob die aktuelle Wertkombination in einem vorherigen Aufruf schon geprüft wurde. Als nächstes wird ermittelt, ob entweder die maximal festgelegte Tiefe für

```
// Generiere einen Wert für eine Zelle
Combination generateValue(int depth, Combination combination){

  if(combinationList.contains(combination))
    return null;
  combinationList.add(combination);

  if(depth > maxDepth || duration > maxDuration)
    return null;

  // Generiere Werte für aktuelle Kombination
  for(int i = 0; i < resultList.size(); i++){

    for(int j = 0; j < i; j++){
      for(Hint hint: resultList.get(j).getHints()){
        resultList.get(i).addHintToGenerator(hint);
      }
    }
    resultList.get(i).generateValue(combination.get(i));
  }

  if(checkConstraints() == null)
    return combination;

  // Rekursiver Aufruf
  for(int i = 0; i < combination.length(); i++){
    Combination newCombination = combination.getNext(i);

    Combination combination = generateValue(depth+1, newCombination);
    if(combination != null)
      return combination;      // Gültige Kombination gefunden
  }
  return null;
}
```

Listing 6.8: generateValue(int depth, Combination combination)-Methode

Tiefensuche oder die maximal festgelegte Laufzeit, die der Algorithmus benötigen darf, erreicht wurde. Ist dies der Fall, so wird `null` zurückgegeben und der Algorithmus damit abgebrochen.

Nun werden alle Elemente der Result-Liste durchlaufen. Für jedes Element wird zuerst versucht, anhand von bekannten schon generierten Werten, Hints zu erzeugen. Die Hints werden anschließend dem Wertgenerator mitgeteilt, sodass dieser bereits vor der Generierung ungültige Werte ausschließt. Die Wahrscheinlichkeit, dass der Generator damit einen gültigen Wert liefert, wird somit um ein Vielfaches erhöht.

Nach der Wertgenerierung wird nun über die Methode `checkConstraints()` aus Listing 6.9 geprüft, ob alle Constraints erfüllt werden. Ist dies nicht der Fall, so wird ein nicht erfülltes Constraint zurückgegeben. Werden alle Constraints erfüllt, so wird von der Methode `checkConstraints()` `null` und die gültige Kombination wird als Ergebnis dieser Methode zurückgeliefert.

Ist die aktuelle Kombination nicht erfolgreich, so muss eine neue Wertkombination ausprobiert werden. Dazu werden soviele neue mögliche Kombinationen erzeugt, wie es Werte in der Liste Results gibt. Alle neuen Kombinationen werden dann mit einem rekursiven Aufruf der Methode

```
Constraint checkConstraints() {
  for (Constraint constraint : constraintList) {

    if(!constraint.allResultsGenerated())
      return constraint;

    if(!constraint.isValid())
      return constraint;
  }
}
```

Listing 6.9: checkConstraints()-Methode

`generateValues()` erzeugt und geprüft.

In Listing 6.9 werden mit Hilfe der Methode `checkConstraints()` alle Constraints des aktuellen Constraint-Graphs geprüft. In einer Schleife werden alle Constraints der Constraint-Liste durchlaufen. Dann wird zuerst abgefragt, ob alle beteiligten Results bereits generiert sind. Anschließend wird geprüft, ob alle Constraints mit den aktuell generierten Werten erfüllt sind.

6.2.2 Umsetzung des Hint-Konzepts

Um die Wahrscheinlichkeit zu erhöhen, dass Wertgeneratoren den Constraints entsprechende Werte liefern, wird das in Abbschnitt 5.3.1.6 beschriebene Hint-Konzept implementiert. Die Idee des Konzepts ist die Reduzierung der Wertemenge vor dem Aufruf des Wertgenerators. Zur Unterstützung des Hint-Konzepts muss die Methode `getHints(Result result)` von der Constraint-Klasse implementiert werden. Dabei werden aus der Art des Constraints und aus den für dieses Constraint schon vorliegenden Werten ein oder mehrere Hint-Objekte generiert. In Listing 6.10 ist ein Beispiel für die Implementierung der Methode `getHints(Result result)` des Unique-Constraints dargestellt. Als Übergabeparameter erwartet die Methode das Result-Objekt für das die Hint-Objekte gelten sollen. Als Rückgabewert werden die erstellten Hint-Objekte geliefert.

In Listing 6.12 wird die Implementierung der `nextValue(Integer index)`-Methode des Wertgenerators für domänenspezifische Werte dargestellt. Dem Wertgenerator wird über den Schlüssel die Domäne zugeteilt, aus welcher dieser einen Wert generieren soll. Als erstes wird die Methode `handleHints()` aufgerufen, die alle dem Wertgenerator zugewiesen Hints verarbeitet und die Liste `valueList` mit möglichen Werten füllt. Über den als Parameter übergebenen Wert `index` wird der Pseudozufallsgenerator initialisiert. Anschließend wird aus der Liste von möglichen Werten ein zufälliger Wert ausgewählt und zurückgegeben.

```
@Override
ArrayList<Hint> getHints(Result result) {
  ArrayList<Hint> hints = new ArrayList<Hint>();

  for (Source source : sources) {
    for (Result r : source.getResults()) {
      if (r.isGenerated() || result == r)
        continue;

      NotEqualHint hint = new NotEqualHint(this);
      hint.setSourceName(r.getSourceName());
      hint.setValue(r.getValue());
      hints.add(hint);
    }
  }

  return hints;
}
```

Listing 6.10: getHints(Result result)-Methode

```
protected void handleHint(NotEqualHint hint){
  Value<?> value = hint.getValue();
  if(value != null){
    if(!notAllowedValues.contains(value))
      notAllowedValues.add(value);
  }
}
```

Listing 6.11: handleHint(NotEqualHint hint)-Methode

In Listing 6.12 wird die Generierung eines Wertes unter Rücksichtnahme von Hints dargestellt. Allgemeine Hints, wie beispielsweise das Ausschließen von bestimmten Werten oder das Beachten der unteren und oberen Grenze im Fall von numerischen Werten, werden dabei in der Oberklasse aller Wertgeneratoren von der Methode `handleHints()` übernommen. Diese Methode erstellt anhand der vorliegenden Hint-Objekte für diesen Generator, je nach Hint-Art beispielsweise eine Liste an erlaubten Werten, eine Liste an nicht erlaubten Werten oder setzt die untere und obere Grenze im Fall von numerischen Generatoren. Bei einigen speziellen Constraints werden meist äquivalente Hints angeboten. So erstellt das Domain-Constraint ein Domain-Hint-Objekt anhand schon bekannter Werte. Die Verarbeitung des Domain-Hints wird ausschließlich vom Domain-Value-Generator übernommen.

In Listing 6.13 ist dargestellt, wie die domänenspezifischen Daten mithilfe der DomainData-Klasse abgelegt werden. Zunächst wird eine globale `HashMap<Key, Value>` angelegt, welche als Key das Schlüsselwort der Domäne und als Value eine Liste aller Werte in Form von DomainData-Objekten enthält, die der Domäne entsprechen. In der globalen Liste werden alle Domänen und alle domänenspezifischen Werte, die verfügbar sein sollen, abgelegt. Ein einzelnes DomainData-Objekt enthält neben dem Schlüssel und dem Wert ebenfalls eine HashMap der oben beschriebenen Form. Diese beinhaltet jedoch im Gegensatz zur globalen HashMap nicht

```
@Override
public Result nextValue(Integer index) {
  handleHints(); // Mit Hilfe aller verfügbaren Hints, die Wertemenge reduzieren.
  Random rand = new Random(index);
  Result result = new Result(null, false); // Erstellen eines Result-Objekts

  // Liste an möglichen Werten vorhanden
  if (valueList.size() > 0){
    DomainHint value = valueList.get(rand.nextInt(valueList.size())); // Zufä
        lligen Wert aus der Liste wählen
    result.setValue(value.getValue());
    result.setGenerated();
  }

  return result;
}
```

Listing 6.12: nextValue(index)-Methode

```
HashMap<String, ArrayList<DomainData>> DomainDataRoot = new HashMap<String,
    ArrayList<DomainData>>();

public class DomainData {
  private String key;
  private Value value;
  public HashMap<String, ArrayList<DomainData>> domainData = new HashMap<String,
      ArrayList<DomainData>>();

  public void addData(String key, Value value);
  public boolean appliesTo(DomainData domainData);
}
```

Listing 6.13: Speicherung der domänenspezifischen Charakteristiken mithilfe der DomainData-Klasse

alle Domänen und alle Werte, sondern ausschließlich die Domänen und die Werte, die zu dem Wert dieses Objektes passen. Die HashMap kann über die Hilfsmethode `addData(String key, Value value)` gefüllt werden. Über die Methode `appliesTo(String key, Value value)` kann geprüft werden, ob ein Schlüssel-Wert-Paar zu diesem DomainData-Objekt passt. Diese Methode durchsucht dazu die HashMap nach dem per Parameter übergebenen Schlüssel-Wert-Paar und gibt `True` im Falle eines Treffers zurück. Falls das übergebene DomainData-Objekt nicht in der HashMap vorhanden ist, wird `False` zurückgegeben.

In Listing 6.14 wird die Methode des Wertgenerators für domänenspezifische Werte dargestellt. Beim Aufruf wird zuerst die `handleHints`-Methode der Elternklasse aufgerufen. Diese bearbeitet zuerst alle allgemeinen Hints. Anschließend wird eine Liste an allen Werten, die für diese Domäne in der globalen HashMap hinterlegt sind, angelegt. Im nächsten Schritt werden alle Werte entfernt, die durch andere Hints bereits ausgeschlossen wurden. Im letzten Schritt werden die domänenspezifischen Verbindungen untersucht, indem alle zu diesem Zeitpunkt noch möglichen Werte durchlaufen und mit der Methode `appliesTo(String key, Value value)` überprüft werden. In der Liste `valueList` bleiben zum Schluss also ausschließlich Werte übrig, die zu den übergebenen domänenspezifischen Hints passen. Der

```
@Override
public void handleHints() {
  super.handleHints(); // Allgemeine Hints durchlaufen

  // Liste mit allen für diese Domäne möglichen Werte füllen
  ArrayList<DomainHint> allValues = DomainData.data.get(getKey());
  valueList = new ArrayList<DomainHint>();
  for (DomainHint entry : allValues) {
    if(!notAllowedValues.contains(entry.getValue()))
      valueList.add(entry);
  }

  // Alle Domain-Hints durchlaufen
  for (Hint hint : getHints()) {
    // Nur DomainHints bearbeiten
    if (DomainHint.class.isInstance(hint)) {
      DomainHint domainHint = ((DomainHint) hint);
      String key = domainHint.getKey();
      Value<?> value = domainHint.getValue();

      // Key des Domain-Hint mit dem Key des DomainValueGenerator abgleichen
      if (getKey().compareTo(key) == 0 && value != null) {

        // Durchlaufen aller zu diesem Zeitpunkt noch möglichen Werte für eine
            Domäne
        for (DomainData possibleValue : valueList) {
          if(!possibleValue.appliesTo(key, value))
            valueList.remove(possibleValue);          // Ungültige Werte aus der
                Liste entfernen
        }
      }
    }
  }
}
```

Listing 6.14: handleHints()-Methode

Wertgenerator kann nun einen Wert aus der Liste auswählen und erhält mit hoher Wahrscheinlichkeit einen gültigen Wert.

Kapitel 7

Bewertung

Nachdem im vorherigen Kapitel beschrieben wurde, wie die unterschiedlichen Algorithmen in der STU-Bibliothek umgesetzt wurden, werden diese nun anhand eines Beispiels auf folgende Kriterien hin verglichen:

Garantierte Terminierung Der Algorithmus terminiert für das zugrundeliegende Modell.

Garantierte Lösungsfindung Der Algorithmus findet eine Lösung, falls mindestens eine Lösung existiert und erkennt andernfalls, dass keine Lösung existiert.

Laufzeit und Skalierung der Laufzeit Es wird untersucht, ob der Algorithmus in einer für das zugrundeliegende Modell annehmbarer Laufzeit terminiert und wie der Algorithmus bei steigender Constraint-Zahl, Modellgröße und Abhängigkeits-Anzahl skaliert.

Algorithmus	**Szenario**	**Laufzeit**	**Wert-generator-Aufrufe**
Tiefensuche	1. Szenario	1072ms	19634
	2. Szenario	2951ms	55520
	3. Szenario	4218ms	88710
Tiefensuche mit Hints	1. Szenario	30ms	158
	2. Szenario	36ms	292
	3. Szenario	45ms	303

Tabelle 7.1: Ergebnisstabelle der durchgeführten Szenarien für die Tiefensuche und Tiefensuche mit Hints

```
user //
.column("id", DataType.BIGINT) //
    .defaultIdentifier() //
    .autoInvokeNext() //
    //
    .column("firstname", DataType.VARCHAR) //
    .generator(new DataGenerator("firstname")) //
    //
    .column("lastname", DataType.VARCHAR) //
    .generator(new DataGenerator("lastname")) //
    // EMail Adresse
    //
    .column("cardnumber", DataType.INTEGER) //
    .constraint().range(new IntValue(100000), new IntValue(999999)) //
    .constraint().unique() //
    //
    .column("birthday", DataType.INTEGER) //
    .generator(new IntegerGenerator(1900, 2015)) //
    //
    .column("lastlogin", DataType.INTEGER) //
    .generator(new IntegerGenerator(2000, 2015)) //
    .constraint().greater("author.birthday") //
    //
    .column("rightid", DataType.BIGINT) //
.reference.local.name("belongsTo").description("").foreign(right).name("
    hasMembers").description("") //
    //
    .build();
```

Listing 7.1: Ausschnitt aus dem verwendeten Beispiel zur Evaluation

Bei dem zur Bewertung verwendeten Beispiel handelt es sich um ein Datenbankschema mit insgesamt zehn Tabellen und ungefähr 30 Spalten. In Listing 7.1 ist ein Ausschnitt des verwendeten Modells zur Bewertung zu sehen. Das vollständige Beispiel ist in Anhang A aufgeführt. Da die zuvor genannten Kriterien, wie garantierte Terminierung, garantierte Lösungsfindung und Laufzeit vor allem von der Anzahl der definierten Constraints, sowie der Anzahl der Zellen pro aufgebautem Lösungsgraph bestimmt werden, wird zwischen drei Szenarien unterschieden:

Szenario 1: Es sind zehn Constraints definiert, wobei die durchschnittliche Anzahl an Zellen, die einen Graph bilden fünf und die maximale Anzahl an Zellen die einen Graph bilden, zehn Werte sind.

Szenario 2: Es sind 20 Constraints definiert, wobei die durchschnittliche Anzahl an Zellen, die einen Graph bilden fünf und die maximale Anzahl an Zellen die einen Graph bilden, zehn Werte sind.

Szenario 3: Es sind 20 Constraints definiert, wobei die durchschnittliche Anzahl an Zellen, die einen Graph bilden zehn und die maximale Anzahl an Zellen die einen Graph bilden, 20 Werte sind.

Im vorliegenden Beispiel werden je nach verwendetem Szenario zwischen 50-150 Datensätze generiert. Die Ergebnisse für die verschiedenen verwendeten Algorithmen und Szenarien werden in Tabelle 7.1 dargestellt. Die drei Szenarien werden mithilfe der drei Algorithmen durchgeführt. Der Generate-and-Test-Algorithmus wird jeweils nach 60 Sekunden terminiert. Die beiden

Tiefensuche-Algorithmen werden dagegen durch die maximale Tiefe limitiert um eine möglichst große Variation an Kombinationen zu überprüfen. In den folgenden Abschnitten werden die Ergebnisse, die mit den unterschiedlichen Algorithmen erzielt wurden, betrachtet.

7.1 Ergebnis des Generate-and-Test-Algorithmus

Der Generate-and-Test-Algorithmus liefert für alle drei Szenarien keine brauchbaren Ergebnisse und wurde deshalb nicht in der Tabelle 7.1 aufgeführt. Selbst für Anwendungsfälle mit vergleichsweise wenig definierten Constraints und wenigen zusammenhängenden Zellen benötigt der Algorithmus eine relativ hohe Laufzeit und viele Wertgeneratoraufrufe ohne alle Constraints aufzulösen. Wird das Szenario komplexer gewählt, so steigt die Anzahl der nicht aufgelösten Graphen und die Laufzeit erhöht sich sehr stark. Auch terminiert der Algorithmus nicht in jedem Fall, sondern wird nach einer gewissen Zeit abgebrochen. Der Grund hierfür liegt darin, dass der Algorithmus einmal generierte Werte, die bisher alle Constraints erfüllen, später nicht mehr zurücksetzt. Das Zurücksetzten und neu generieren einzelner Werte wäre jedoch notwendig, da ungünstig gewählte Werte zu einem nicht mehr auflösbaren Graphen führen können. Der Algorithmus versucht nun einen Wert für eine Variable zu generieren, für die kein gültiger Wert existiert. Es ist also ersichtlich, dass dieser Lösungsansatz nur für einfache Constraints funktioniert.

7.2 Ergebnis der Tiefensuche

Wird nun das Ergebnis bei der Verwendung des Tiefensuchealgorithmus betrachtet, so ist ersichtlich, dass generell eine bessere Laufzeit im Vergleich zum Generate-and-Test-Algorithmus erreicht wird. Da der Algorithmus theoretisch systematisch alle möglichen Kombinationen an Werten für die gegebenen Variablen durchläuft, terminiert der Algorithmus in der Theorie in jedem Fall. Dies bedeutet, dass er eine Lösung findet, falls diese existiert. Für den Fall, dass keine Lösung existiert, wird dies vom Algorithmus erkannt.

In der Praxis ist es jedoch oftmals nicht möglich, alle Wertkombinationen auf eine mögliche Lösung zu prüfen, da eine inaktzeptable Laufzeit für den Algorithmus erreicht würde. Wird der Algorithmus beispielsweise für das Szenario 3 verwendet, kann dieser nicht alle gebildeten Graphen auflösen. Der Grund hierfür liegt darin, dass im Vergleich zur Anzahl an möglichen Lösungen für den Graphen zu wenig gültige Lösungen existieren. Die Anzahl der Aufrufe in Szenario 3 steigt im Vergleich zu Szenario 2 nur nicht mehr

stark an, da die maximale Tiefe, die die Tiefensuche erreichen darf, limitiert ist.

7.3 Ergebnis der Tiefensuche mit Hint-Konzept

Die Ergebnisse, welche mit dem um das Hint-Konzept erweiterten Tiefensuchealgorithmus erreicht werden, sind als nochmals besser zu bewerten. Es zeigt sich in den gewählten Szenarien, dass sowohl alle Graphen aufgelöst, als auch die Laufzeiten stark verkürzt wurden. Der Grund hierfür lässt sich sehr gut anhand der Anzahl der Aufrufe von Wertgeneratoren erkennen. Dadurch, dass die verschiedenen Wertgeneratoren mithilfe von Hints ungültige Werte bzw. Wertebereiche schon vor der Wertgenerierung ausschließen, kann die Anzahl der Aufrufe von Wertgeneratoren um einen sehr hohen Faktor verringert werden. Dies hat zur Folge, dass auch verwandte Methoden-Aufrufe, wie z.B. das Prüfen aller Constraints weniger oft aufgerufen werden. Es lässt sich also daraus schließen, dass sich der Aufwand, der für die Hint-Generierung und Verarbeitung geleistet wird, im Vergleich zum Aufwand für das Durchprobieren aller theoretisch möglichen Kombinationen, lohnt.

Kapitel 8

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Generierung von validen Testdaten basierend auf der vorhanden STU-Bibliothek und der Arbeit [Mol13]. Dazu wurde zuerst konzeptionell untersucht, wie es möglich ist, den bestehenden Generierungsalgorithmus und die vorhandenen Wertgeneratoren soweit zu erweitern, dass valide Testdaten generiert werden können. Zur Lösung des Problematik wurde die Bibliothek um einen Constraint-basierten Ansatz erweitert. Dieser Ansatz erlaubt es nun für einzelne Testdatenwerte bestimmte Constraints zu formulieren, die den Wert abhängig von konstanten Werten oder anderen Werten des Datenmodells einschränken. Um die durch Constraints entstehenden Abhängigkeiten aufzulösen und gültige Wertkombinationen zur erhalten, die den definierten Constraints entsprechen, wurden zunächst verschiedene Algorithmen und Strategien untersucht. Implementiert wurde anschließend eine Lösung, die auf Basis des Tiefensuchealgorithmus nach gültigen Kombinationen sucht. Erweitert wurde der Algorithmus durch das Hint-Konzept, um auch valide Testdatenwerte bei komplexen Abhängigkeitsgraphen zu erhalten. Als Ergebnis der Implementierung können nun mit einfachen Mitteln Constraints spezifiziert werden, die valide Testdaten beschreiben. Zu guter Letzt wurden die implementierten Algorithmen mit einem geeigneten Beispiel eines Datenmodells in Hinblick auf Terminierung und Laufzeit verglichen.

Hinsichtlich nachfolgender Arbeiten sind zusätzliche Erweiterungen und Optimierungen denkbar. So kann der Algorithmus zur Lösungsfindung weiter optimiert werden, indem der Algorithmus durch weitere Konzepte ergänzt wird. Die Optimierungen würden sich einerseits auf die Generierungszeit auswirken und andererseits komplexe Constraint-basierte Abhängigkeitsgraphen ermöglichen. Zusätzlich ist ebenfalls denkbar, dass weitere domänenspezifische Wertgeneratoren und zusätzliche Constraint-Arten,

wie beispielsweise das Kardinalitäts-Constraint, implementiert werden. Dieser erweiterte Funktionsumfang erlaubt es dem Tester qualitativ noch hochwertigere Testdaten zu erzeugen.

Bis jetzt ist es zwar möglich, beliebig viele Constraints pro Spalte zu definieren, ein gültiger Wert wird jedoch nur generiert, wenn alle Constraints erfüllt werden. Die definierten Constraints sind somit im Prinzip mit einem logischen Und verknüpft. Nützlich wäre es jedoch, wenn man zusätzlich verschiedene Constraints mit anderen logischen Operatoren, wie beispielsweise dem Oder-Operator, verknüpfen könnte.

Eine Performanceverbesserung kann mithilfe von Parallelisierung des Generierungsvorgangs erreicht werden. Die Erzeugung der durch Constraints bedingten Result-Graphen kann dabei wie gehabt stattfinden. Die Auflösung der einzelnen Graphen kann jedoch auf einzelne Tasks aufgeteilt werden, da die zu generierenden Werte unabhängig von denjenigen anderer Graphen sind. Dadurch kann die Dauer der Testdatengenerierung je nach Anwendungsfall und je nach eingesetztem Prozessor stark verkürzt werden.

Weiterhin besteht die Möglichkeit die verschiedenen möglichen Kombinationen ebenfalls zu parallelisieren. Im Vergleich zur Parallelisierung der Auflösung der einzelnen Graphen muss bei bei der Implementierung dieses Ansatzes jedoch mehr Aufwand betrieben werden, da es einer Synchronisierung der einzelnen Tasks bedarf. So müssen andere Tasks von dem erfolgreichen Task benachrichtigt werden, sodass diese terminiert werden können.

Die zusätzlich in die STU-Bibliothek implementierten Funktionalitäten und die erweiterte API sind bisher ausschließlich manuell verwendbar. Die vorhandene GUI zum Extrahieren des Datenbankschemas und zur Anpassung des Modells könnte um die zusätzlichen Funktionalitäten erweitert werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[AKL11] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 685–696. ACM, 2011.

[B+07] Michel Berkelaar et al. lpsolve: Interface to lp solve v. 5.5 to solve linear or integer programs. *R package version*, 5(8), 2007.

[CP14] Xavier Lorca Charles Prud'homme, Jean-Guillaume Fages. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[FW07] Heide Faeskorn-Woyke. *Datenbanksysteme: Theorie und Praxis mit SQL2003, Oracle und MySQL*. Pearson Studium, München [u.a.], 2007. ISBN 978-3-8273-7266-6. 512 S. pp.

[Gam07] Erich Gamma. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley, München [u.a.], [nachdr.] edition, 2007. ISBN 3-8273-2199-9. XX, 479 S. pp.

[Hal10] Klaus Haller. The test data challenge for database-driven applications. In *Proceedings of the Third International Workshop on Testing Database Systems*, DBTest '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0190-9.

[HTW06] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1243–1246. VLDB Endowment, 2006.

[HW07] Petra Hofstedt and Armin Wolf. *Einführung in die Constraint-Programmierung: Grundlagen, Methoden, Sprachen, Anwendungen*. Springer-Verlag, 2007.

[KS14] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP Library. User's Guide*. http://www.jacop.osolpro.com/, 2014.

[LR13] Diana Lorentz and MB Roese. *Oracle Database SQL Language Reference, 11g Release 1 (11.1)*, 2013.

[Mol13] Nikolaus Moll. Modellierung und Generierung von Testdaten für Datenbank-basierte Anwendungen. Masterarbeit, HTWG Konstanz, 2013.

[SWW14] Robert Sedgewick, Kevin Daniel Wayne, and Kevin Daniel Wayne. *Algorithmen: Algorithmen und Datenstrukturen.* Pearson Studium, 2014.

Anhang A

Vollständiges Beispiel zur Constraint-basierten Modellierung

```
public class BookDatabaseModel extends DatabaseModel {

  public BookDatabaseModel() {
    database("BookDatabase");
    packageName("com.seitenbau.stu.bookdatabase.model");
    infinite(1);
    dataSource(new DomainSpecificData());

    TableBuilder address = table("address");
    TableBuilder author = table("author");
    TableBuilder book = table("book");
    TableBuilder copy = table("copy");
    TableBuilder publisher = table("publisher");
    TableBuilder right = table("right");
    TableBuilder section = table("section");
    TableBuilder user = table("user");

    address.column("id", DataType.BIGINT) //
        .defaultIdentifier() //
        .autoInvokeNext() //
        //
        .column("street", DataType.VARCHAR) //
        .generator(new DomainGenerator("street")) //
        .constraint().domain("address.city") //
        //
        .column("number", DataType.INTEGER) //
        .generator(new IntegerGenerator(1, 99)) //
        //
        .column("postalcode", DataType.INTEGER) //
        .generator(new IntegerGenerator(70000, 80000)) //
        .constraint().range(new IntValue(78100), new IntValue(78200)) //
        //
        .column("city", DataType.VARCHAR) //
        .generator(new DomainGenerator("city")) //
        .constraint().domain("address.country") //
        .constraint().domain("address.postalcode") //
        //
        .column("country", DataType.VARCHAR) //
        .generator(new DomainGenerator("country")) //
        //
        .build();

    author //
    .column("id", DataType.BIGINT) //
        .defaultIdentifier() //
        .autoInvokeNext() //
        //
        .column("firstname", DataType.VARCHAR) //
        .generator(new DomainGenerator("firstname")) //
```

```
        .constraint().unique() //
        //
        .column("lastname", DataType.VARCHAR) //
        .generator(new DomainGenerator("lastname")) //
        //
        .column("gender", DataType.VARCHAR) //
        .generator(new DomainGenerator("gender")) //
        .constraint().domain("author.firstname") //
        //
        .column("birthyear", DataType.INTEGER) //
        .generator(new IntegerGenerator(1900, 2100)) //
        .constraint().range(new IntValue(1900), new IntValue(2015)) //
        //
        .column("memberyear", DataType.INTEGER) //
        .generator(new IntegerGenerator(1900, 2100)) //
        .constraint().range(new IntValue(1980), new IntValue(2015)) //
        .constraint().greater("author.birthyear") //
        //
        .column("lastlogin", DataType.INTEGER) //
        .generator(new IntegerGenerator(2000, 2015)) //
        .constraint().greater("author.memberyear") //
        //
        .column("addressid", DataType.BIGINT) //
    .reference.local.name("belongsTo").description("").foreign(address).name("
        hasMembers").description("") //
        //
        .build();

    book //
    .column("id", DataType.BIGINT)
    //
        .defaultIdentifier()
        //
        .autoInvokeNext()
        //
        .column("name", DataType.VARCHAR).generator(new DomainGenerator("bookname
            "))
        //
        .column("price", DataType.INTEGER).generator(new IntegerGenerator(1, 20))
        //
        .column("publisher", DataType.BIGINT) //
    .reference.local.foreign(publisher.ref("id")).multiplicity("0..*") //
        .build();

    copy //
    .column("id", DataType.BIGINT) //
        .defaultIdentifier() //
        .autoInvokeNext() //
        //
        .column("instance", DataType.INTEGER) //
        .generator(new IntegerGenerator(1, 100)) //
        //
        .column("bookid", DataType.BIGINT).reference //
    .local.name("belongsTo").description("").foreign(book).name("hasMembers").
        description("")//
        //
        .build();

    publisher //
        .column("id", DataType.BIGINT) //
        .defaultIdentifier() //
        .autoInvokeNext() //
        //
        .column("name", DataType.VARCHAR) //
        .generator(new DomainGenerator("publisher")) //
        //
        .column("addressid", DataType.BIGINT).reference //
    .local.name("belongsTo").description("").foreign(address).name("hasMembers").
        description("")//
        .build();

    right //
    .column("id", DataType.BIGINT) //
        .defaultIdentifier() //
        .autoInvokeNext() //
        //
        .column("name", DataType.VARCHAR) //
        .generator(new DomainGenerator("rightname")) //
        .build();

    section //
    .column("id", DataType.BIGINT) //
        //
        .defaultIdentifier() //
```

```
        //
        .autoInvokeNext() //
        //
        .column("name", DataType.VARCHAR) //
        //
        .column("floor", DataType.INTEGER) //
        .constraint().range(new IntValue(0), new IntValue(3)) //
        //
        .build();

    user //
    .column("id", DataType.BIGINT) //
        //
        .defaultIdentifier() //
        //
        .autoInvokeNext() //
        //
        .column("firstname", DataType.VARCHAR) //
        .generator(new DomainGenerator("firstname")) //
        //
        .column("lastname", DataType.VARCHAR) //
        .generator(new DomainGenerator("lastname")) //
        // EMail Adresse
        //
        .column("cardnumber", DataType.INTEGER) //
        .generator(new IntegerGenerator(50000, 4999999)) //
        .constraint().range(new IntValue(100000), new IntValue(999999)) //
        .constraint().unique() //
        //
        .column("birthday", DataType.INTEGER) //
        .generator(new IntegerGenerator(1900, 2015)) //
        //
        .column("lastlogin", DataType.INTEGER) //
        .generator(new IntegerGenerator(2000, 2015)) //
        .constraint().greater(new IntValue(2012)) //
        //
        .constraint().range(new IntValue(55), new IntValue(66)) //
        //
        .column("rightid", DataType.BIGINT) //
    .reference.local.name("belongsTo").description("").foreign(right).name("
        hasMembers").description("") //
        //
        .build();

    associativeTable("bookauthor").column("bookid", DataType.BIGINT).reference.
        foreign(book).multiplicity("0..*").column("authorid", DataType.BIGINT).
        reference.foreign(author)
        .multiplicity("1..*").build();

    associativeTable("bookuser").column("bookid", DataType.BIGINT).reference.
        foreign(book).multiplicity("0..*").column("userid", DataType.BIGINT).
        reference.foreign(user)
        .multiplicity("1..*").build();
  }
}
```

Listing A.1: Vollständiges Beispiel zur Modellierung eines Modells

Anhang B

Vollständiges Beispiel für die DSL zur Darstellung der generierten Testdaten

```
package com.seitenbau.stu.bookdatabase.model;

import static com.seitenbau.stu.bookdatabase.model.BookDatabaseDataSetRefs.*
import static com.seitenbau.stu.bookdatabase.model.AddressTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.AuthorTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.BookTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.CopyTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.PublisherTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.RightTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.SectionTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.UserTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.BookauthorTableAdapter.*
import static com.seitenbau.stu.bookdatabase.model.BookuserTableAdapter.*

class BookDatabaseDataSetDefault extends BookDatabaseBuilder
{
  def tables() {
    addressTable.rows() {
      REF        | street          | number | postalcode | city      | country
      ADDRESS_1 | Hauptstrasse    | 57     | 10115      | "Berlin" | "Deutschland"
      ADDRESS_2 | Rathausstrasse | 33     | 3000       | "Bern"   | "Schweiz"
      ADDRESS_3 | Spitalstrasse   | 85     | 4000       | "Basel"  | "Schweiz"
    }

    authorTable.rows() {
      REF       | firstname | lastname   | gender     | birthyear | memberyear |
           lastlogin |  addressid
      AUTHOR_1 | "Tim"      | "Vogt"     | "männlich"| 1909      | 1980       |
           2012      | ADDRESS_1
      AUTHOR_2 | "Adalie"   | "Dietrich"| "weiblich"| 1954      | 2009       |
           2013      | ADDRESS_2
      AUTHOR_3 | "Tilly"    | "Winter"   | "weiblich"| 1974      | 1991       |
           2000      | ADDRESS_3
      AUTHOR_4 | "Adam"     | "Hahn"     | "männlich"| 1930      | 1984       |
           2009      | ADDRESS_2
      AUTHOR_5 | "Robin"    | "Stein"    | "weiblich"| 1987      | 1991       |
           2008      | ADDRESS_3
    }

    bookTable.rows() {
      REF     | name                                          | price | publisher
      BOOK_1 | "Erzählungen"                                 | 10    | PUBLISHER_2
      BOOK_2 | "Germinal"                                    | 2     | PUBLISHER_3
      BOOK_3 | "Der Steppenwolf"                             | 15    | PUBLISHER_3
      BOOK_4 | "Traurige Tropen"                             | 16    | PUBLISHER_3
      BOOK_5 | "Phantastische Erzählungen"                   | 9     | PUBLISHER_3
      BOOK_6 | "Verlorene Illusionen"                        | 12    | PUBLISHER_3
      BOOK_7 | "Alice im Wunderland"                         | 5     | PUBLISHER_3
      BOOK_8 | "Die Verwirrungen des Zöglings Törless"| 18    | PUBLISHER_3
    }
```

```
    copyTable.rows() {
      REF      | instance | bookid
      COPY_1 | 70       | BOOK_5
      COPY_2 | 2        | BOOK_1
      COPY_3 | 55       | BOOK_2
      COPY_4 | 56       | BOOK_3
      COPY_5 | 69       | BOOK_4
      COPY_6 | 32       | BOOK_6
      COPY_7 | 85       | BOOK_7
      COPY_8 | 38       | BOOK_8
    }

    publisherTable.rows() {
      REF          | name                          | addressid
      PUBLISHER_1 | "Neumann Verlag"              | ADDRESS_1
      PUBLISHER_2 | "E. S. Mittler & Sohn"        | ADDRESS_2
      PUBLISHER_3 | "Ellert & Richter Verlag"     | ADDRESS_3
    }

    rightTable.rows() {
      REF      | name
      RIGHT_1 | "Admin"
      RIGHT_2 | "Superuser"
      RIGHT_3 | "User"
      RIGHT_4 | "Guest"
      RIGHT_5 | "Root"
    }

    sectionTable.rows() {
      REF        | name    | floor
      SECTION_1 | "Beta" | 2
    }

    userTable.rows() {
      REF      | firstname    | lastname    | cardnumber | birthday | lastlogin |
           rightid
      USER_1 | "Tim"        | "Vogt"      | 2395069    | 1905     | 2015      |
           RIGHT_1
      USER_2 | "Thomas"     | "Dietrich" | 1573509    | 1929     | 2015      |
           RIGHT_2
      USER_3 | "Jürgen"     | "Winter"    | 4268687    | 1906     | 2015      |
           RIGHT_3
      USER_4 | "Jürgen"     | "Hahn"      | 3447127    | 2007     | 2014      |
           RIGHT_4
      USER_5 | "Adalgis"    | "Stein"     | 3477345    | 2008     | 2014      |
           RIGHT_5
    }

    bookauthorTable.rows() {
      bookid | authorid
      BOOK_1 | AUTHOR_2
      BOOK_2 | AUTHOR_3
      BOOK_3 | AUTHOR_4
      BOOK_4 | AUTHOR_5
    }

    bookuserTable.rows() {
      bookid | userid
      BOOK_5 | USER_2
      BOOK_6 | USER_3
      BOOK_7 | USER_4
      BOOK_8 | USER_5
    }
  }
}
-------------------------------
address:  3
author: 5
book: 8
bookauthor: 4
bookuser: 4
copy: 8
publisher:  3
right:  5
section:  1
user: 5

Verification Loop Iterations: 17
```

Listing B.1: Vollständiges Beispiel für die DSL zur Darstellung der generierten Testdaten