TUHH
Hamburg University of Technology

Master-Arbeit

Oliver Günther

ASP Engineering

17.01.2013

supervised by:
Prof. Dr. Ralf Möller
Prof. Dr. Karl-Heinz Zimmermann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 17. Januar 2013

_____

Oliver Günther

Danksagung

Für die sehr gute Betreuung durch Prof. Dr. Ralf Möller während dieser Arbeit möchte ich mich herzlich bedanken. Ein derart hohes Maß an Zeit, konstruktiver Kritik und Anregungen hab ich selten erlebt.

Inhaltsverzeichnis

1 Einleitung

In den Anfängen der Informatik waren Anforderungen sehr genau spezifiziert und es arbeiteten Experten miteinander. Der Schwerpunkt war die Entwicklung der Hardware. Die ersten elektronischen Rechner wurden von einzelnen Personen oder Gruppen aus zwei bis drei Personen entwickelt. Als Beispiele sei hier auf die Z3 von Konrad Zuse oder den Atanasoff-Berry-Computer von John Atanasoff und Clifford Berry verwiesen. Die große gemeinsame Schnittmenge an Wissen zwischen diesen Personen reduzierte das Risiko von Missverständnissen und Fehlentwicklungen.

Mit fortschreitenden Verbesserungen der Hardware wuchsen die Errungenschaften im Bereich Software. Es konnten immer komplexere Probleme adressiert werden. Da Hardware aber zu dieser Zeit weit höhere Kosten verursachte als die Entwicklung von Software, entstanden immer größere Probleme in Softwareprojekten. Diese wurden organisatorisch vernachlässigt, oder erst gar nicht als Projekte betrachtet. Der Höhepunkt dieses Missstandes war in den 60iger Jahren die Softwarekrise. Kosten für Software überstiegen erstmals Kosten für die Hardware. Der geänderten Situation wurde mit der Einführung von Software Engineering auf der Konferenz [14] begegnet. Dies umfasste das traditionelle ingenieurmäßige Vorgehen in der Entwicklung von Software durch die Phasen Planung, Entwurf, Implementation und Test.

Mit weiterer Verbreitung der Computer-Technologie in Industrie, und später beim Endverbraucher sowie mit der Entwicklung weltumspannender Netze, stieg die Komplexität von Projekten. Fehler und Situationen, menschlicher oder technischer Natur, hatten immer größere Auswirkungen. Damit erhöhte sich der Anspruch an das traditionelle ingenieurmäßige Vorgehen im Software Engineering. In den 90iger Jahren wurde die Idee des modellgetriebenen Entwurfs stark verfolgt und Modellierungsmethoden entwickelt, um diesen Anforderungen gerecht zu werden. Zu erwähnen ist hier Grady Booch [9] mit seinen Arbeiten zur Unified Modeling Language.

Die kontinuierlich steigenden Anforderungen zeigten gegen Ende der 90iger Jahre mehr und mehr Schwächen des modellgetriebenen Entwurfs auf. Entwickler in Projekten wurden immer häufiger mit Änderungen während der Durchführung konfrontiert und diese mussten in Programmcode und Modell eingepflegt werden. Oft wurde die Pflege des Modells vernachlässigt, sodass dieses immer stärker von der Implementation abwich. Versuche, Werkzeuge zu entwickeln, die einen automatischen Abgleich von Modell und Implementation ermöglichen haben nicht den gewünschten flächendeckenden Erfolg gebracht und führen heute teilweise nur noch ein Nischendasein. Beispiel hierfür sind Borland Together oder IBM Rational Rose.

Um die Jahrtausendwende brachte Kent Beck [1] mit Extremprogrammierung einen Paradigmenwechsel: den Quellcode zum Modell zu erklären und diesen in hoher Qualität zu erhalten. Dies wurde durch Techniken wie Paarprogrammierung, testgetriebene Entwicklung und ständige Refaktorierungen erst möglich. Integrierte Entwicklungsumgebungen wurden an diese Anforderungen angepasst. Aktuelle IDEs erlauben die projektübergreifende Anpassung von Methodennamen, Parametertypen, Parameteranzahl

oder Rückgabewerten. Sie ermöglichen eine automatische Erzeugung von Delegates, Adaptern oder Buildern. Selbst Verschiebungen von Methoden in der Vererbungshirachie sind möglich.

Parallel entstanden die agilen Methoden. Durch kurze, sich wiederholende Zyklen, schnelle Reviews und starke Kommunikation ist es möglich, Projekte umzusetzen, deren Anforderungen starken Änderungen ausgesetzt sind. Als bekannte Beispiele sei hier auf Scrum in [17] und Kanban in [16] verwiesen. Dabei ergänzen sich die agilen Methoden und Extremprogrammierung sehr gut.

Nach heutiger Auffassung eignen sich die agilen Methoden besonders gut, um Projekte mit einer großen Menge von Anforderungen, welche fast in beliebiger Reihenfolge implementiert werden können, zu entwickeln. Die Anforderungen werden in der Regel nur mit Kurzbeschreibungen festgehalten und erst kurz vor Beginn von Umsetzung und Testvorbereitung ausformuliert. Herausforderungen sind hier oft, in der Industrie die Eigenschaften der agilen Vorgehensweise beim Vertragsabschluss ausreichend zu berücksichtigen. Traditionelle ingenieursmäßige Vorgehensmodelle werden weiterhin gewählt, wenn ein Projekt klar definierte Anforderungen erfüllen muss, kurze Liegezeiten hat und engen Zeit- oder Budgetvorgaben unterliegt. Agile Methoden werden aber auch hier in Teilen, zum Beispiel beim Rapid Prototyping, eingesetzt.

Bis in die 70iger Jahre waren nur imperative und funktionale Programmierung bekannt. Entsprechend wurden nur diese Konzepte von Forschung und Entwicklung des Software Engineering betrachtet. Mit Entstehung der logischen Programmierung und der Standardisierung von Prolog in [2] kam ein neues Paradigma hinzu. Logische Programmierung rückte die Fähigkeit, Fakten und Regeln zu definieren und daraus neue Fakten zu schlussfolgern, in den Vordergrund. Dies fand in der Wissenschaft für Künstliche Intelligenz genau so Anwendung wie in der Industrie. Hier seien Produktkonfiguratoren, Routenplaner oder Marktsimulation zur Preisfindung als Beispiele genannt.

Die Möglichkeiten der logischen Programmierung führten entsprechend zu Herausforderungen im Software Engineering. Besonders die Schwäche, dass Regeln ihrer Natur nach reihenfolgeunabhängig sind, aber in der Implementation sehr wohl eine Reihenfolge haben, ist hervorzuheben. Dies lässt sich besonders gut an den Formeln 1 zeigen.

$$\begin{aligned} \neg a &\rightarrow b \\ \neg b &\rightarrow a \end{aligned} \tag{1}$$

Die Interpretation bzw. die Frage ob a wahr ist oder nicht, fordert Implementationen von Prolog genau so heraus wie die philosophisch-logische Analyse.

Um eine deterministische Entscheidung zu ermöglichen, bildeten sich 2 bekannte Semantiken heraus.

- Die fundierte Semantik[1] (Well-Founded Semantics); vorgestellt im Artikel [18].
- Die stabile Modellsemantik[2] (Stable Model Semantics); vorgestellt im Artikel [6] die Answer Set Programming (ASP) zugrunde liegen.

Die Entwicklung dieser Semantiken löste nicht nur das vorgestellte Beispiel, sondern auch die Schwäche vorhandener Implementationen, Regeln in einer Reihenfolge betrachten zu müssen.

1.1 Ziel der Arbeit

In dieser Arbeit wird untersucht, mit welchen Techniken Software Engineering bei der Entwicklung und Wartung von Answer Set Programmen unterstützt werden kann. Dazu werden spezielle Techniken ausgewählt und im Kontext von ASP untersucht. Es wird geprüft, ob eine Anwendung überhaupt möglich ist, welche Rahmenbedingungen gelten müssen und unter welchen die Anwendung empfohlen wird. Auch klare Situationen, wo von einer Technik abgeraten wird, werden benannt. Die Empfehlungen werden belastbar durch praktische Beispiele untermauert. In der gesamten Arbeit wird DLV als ASP Implementation verwendet.

1.2 Aufbau der Arbeit

In Kapitel 2 werden stabile Modellsemantik, ASP und DLV im Detail vorgestellt. Um nicht nur Laborbeispiele zu untersuchen, sondern auch belastbare praktische Beispiele aufzuzeigen, konzentriert sich ein Teil der Arbeit auf die Untersuchung der Techniken in einer existierenden Implementation in der Industrie. Dazu wird in Kapitel 3 die Firma GG-Net GmbH und ihre in DLV implementierte Preis-Maschine vorgestellt. Als erste Technik wird in Kapitel 4 das naive Ausprobieren zur Verifikation von Answer Set Programmen untersucht. Hierbei wird besonders die praktische Eignung des Verfahrens auf Grund seiner Komplexität betrachtet. In Kapitel 5 wird untersucht, ob die Lösung aus Kapitel 4 mit Hilfe von Abstrakter Interpretation in ihrer Komplexität reduziert werden kann. Abduktion als Software-Engineering-Technik wird in Kapitel 6 vorgestellt. Die Fähigkeit, automatisch Fakten zu bestimmen, wird im Kontext betrachtet, eine gewünschte Lösung zu ermöglichen. Die gewonnenen Erkenntnisse aller untersuchten Techniken werden im Kapitel 7 zusammengetragen. Es wird gezeigt, dass tatsächlich jede Technik einen sinnvollen Anwendungsbereich hat.

[1] Das Beispiel in Well-Founded Semantics hat eine Lösung, in der kein Fakt wahr ist.

[2] Das Beispiel in Stable Model Semantics hat zwei Lösungen, in welcher jeweils $a \wedge \neg b$ oder $b \wedge \neg a$ wahr sind.

2 Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, auf denen diese Arbeit aufbaut. Dazu wird in Abschnitt 2.1 die stabille Modellsemantik und Answer Set Programming vorgestellt. Im darauffolgenden Abschnitt 2.2 wird DLV, die in dieser Arbeit verwendete Implementation von ASP, vorgestellt und im Detail beschrieben.

2.1 Answer Set Programming

Der Ursprung von ASP liegt im nicht-monotonen Schlussfolgern über Wissensdatenbanken. Der Begriff Answer Set Programming wurde erstmalig in den Artikeln [13] und in [15] verwendet. Unter ASP fasst man ein System zur logischen Programmierung zusammen, welches die stabile Modellsemantik implementiert. Das stabile Modell und damit ASP lösten einige der essentiellen Probleme der logischen Programmierung und des Quasi-Standards Prolog.

Mit Prolog, entwickelt von Colmerauer und Kowalski, existierte seit den 70iger Jahren bereits ein System, das es erlaubte, logische Probleme deklarativ in einer Programmiersprache darzustellen. Die Problematik ist, dass damit die prozedurale Arbeitsweise von Prolog und verwandten Systemen nur verschleiert wurde.

$$\begin{aligned} on(book, table) \\ on(x, y) \longrightarrow above(x, y) \\ \forall xy(\exists z(on(x, z) \wedge above(z, y)) \longrightarrow above(x, y)) \end{aligned} \tag{2}$$

$$\begin{aligned} on(book, table) \\ \forall xy(\exists z(on(x, z) \wedge above(z, y)) \longrightarrow above(x, y)) \\ on(x, y) \longrightarrow above(x, y) \end{aligned} \tag{3}$$

Schaut man sich die Formeln 2 und 3 an, so haben diese den selben deklarativen Inhalt. Programmiert man diese in Prolog und führt die Frage `above(book,table)?` aus, so wird in Implementation 2 die Antwort wahr lauten. In Implementation 3 wird Prolog leider in eine Endlosschleife laufen.

$$\begin{aligned} \neg a \rightarrow b \\ \neg b \rightarrow a \end{aligned} \tag{4}$$

Ein weiteres Problem sei mit den Formeln 4 gezeigt. Auf die Frage, ob `a?` wahr ist, wird Prolog wieder in einer Endlosschleife stecken bleiben. Beide Probleme haben mit der prozeduralen/Beweis suchenden Vorgehensweise von Prolog und vergleichbaren

Systemen zu tun. Hierbei versuchte ein Algorithmus, eine Abfrage auf einem logischen Programm korrekt zu schlussfolgern. Dies führt, wie gezeigt, zu großen Schwierigkeiten beim Einsatz von Negationen über nicht definierte Fakten und beachtet unerwünschter Weise die Reihenfolge der Regeln.

Die logischen Formeln in 4 sind allerdings nicht nur für Prolog ein Problem, sondern stellten auch die grundlegende logische Interpretation vor die Frage der Bedeutung. Bei der stabilen Modellsemantik, vorgestellt im Artikel [6], die ASP zugrunde liegen, handelt es sich um eine grundlegende Verschiebung in der Betrachtung logischer Programme und Probleme. Anstatt einen Beweis zu finden, wird versucht, eine Lösung für das gesamte logische Programm zu finden. Ein ASP-Solver sucht nach Mengen von Fakten, genannt *Modelle*, die alle Regeln des logischen Programms erfüllen.

ASP erlaubt somit Lösungen für Probleme zu finden, wo dies bisher nicht möglich war. Dazu werden Regeln in der Form 5 jeweils als Randbedingung an die zu findenden Modelle $\mathcal{M}$ aufgefasst.

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\ (L_k\ for\ 0 \leq k \leq n) \tag{5}$$

Dabei steht L jeweils für eine atomare Aussage über einer endlichen Domain. Wenn $L_i \in \mathcal{M}$ für $1 \leq i \leq m$ und $L_j \notin \mathcal{M}$ für $m < j \leq n$, dann ist $L_0 \in \mathcal{M}$. Logische Programme bestehen dann aus einer Menge solcher Regeln und haben keine, eine oder gar mehrere Modelle.

Unter dieser Definition hebt sich die Reihenfolgeabhängigkeit von Regeln auf. Eine Implementation der Formeln 2 und 3 in einem ASP System wird auf die Frage `above(book,table)?` immer wahr antworten. Auch die Formeln 4 führt nicht zu einer Endlosschleife, sondern es werden die zwei Modelle $a = wahr$ und $b = wahr$ gefunden.

Die Herausforderung für die Wissenschaft besteht seit Definition der stabilen Modellsemantik im Erforschen von effizienten Suchalgorithmen für gültige Modelle. In dieser Arbeit findet die Implementation DLV Verwendung. Diese wird im folgenden Kapitel vorgestellt.

2.2 DLV

In diesem Kapitel wird DLV vorgestellt und eine Einführung in das ausführbare Programm und die Sprache betrieben. Im Artikel [12] wird DLV als ein deduktives Datenbank-System präsentiert. Es basiert auf disjunktiver Logikprogrammierung und ist eine vollständige Implementation von Answer Set Programming. Es wurde von einem italienisch-österreichischen Forschungsteam (von der Universität von Kalabrien und der Technischen Universität Wien) konzipiert.

Das System unterstützt eine Sprache auf logischen Formalismen mit sehr hoher Ausdruckskraft, die es ermöglicht, Programme für relevante praktische Probleme in der Umgebung von unvollständigem oder widersprüchlichem Wissen zu formulieren. Die erste Version wurde nach mehreren Jahren der theoretischen Forschung 1997 freigegeben. Seit dieser Zeit wurden deutliche Verbesserungen, neue Funktionen und relevante Optimierungstechniken in allen Modulen realisiert. Das Laufzeitsystem ist effizient. Durch die mehrstufige Analyse des logischen Programms und der Eingabemerkmale ist das System in der Lage, Techniken, die für die jeweilige Komplexität des Problems geeignet sind, auszuwählen. So lassen sich *einfache* Probleme schnell lösen, während für *schwierigere* Probleme Methoden mit höherem Rechenaufwand verwendet werden.

Im folgenden findet eine Einführung in die ASP Implementation von DLV statt. DLV hat noch weitere Mechanismen, die aber im Verlauf dieser Arbeit nicht weiter betrachtet werden. Die Beispiele und der Aufbau wurden inspiriert durch die Online-Dokumentation[3] und den Artikel [12].

2.2.1 Der Aufruf von DLV

DLV ist ein Programm auf der Kommandozeile. Dem Aufruf von DLV folgen Optionen und eine oder mehrere Textdateien, die das logische Programm enthalten.

```
DLV {Optionen} [Dateiname [Dateiname [...]]]
```

Durch den Aufruf von DLV wird geprüft, ob die übergebenen Optionen und Dateinamen korrekt sind, und der Solver wird mit diesen Informationen gestartet. Sollten Eingabeparameter fehlerhaft sein, bricht der Ablauf ab und eine Fehlermeldung wird ausgegeben. Bei erfolgreichem Aufruf löst der Solver das übergebene logische Programm und liefert kein, ein oder mehrere Modelle als Ergebnis an der Konsole. Sollte DLV kein Modell liefern, ist das Programm inkonsistent oder widersprüchlich. Ein leeres Modell ist eine Lösung für ein konsistentes Programm, in dem kein Fakt wahr ist.

[3] http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html

2.2.2 Die Sprache DLV

Die grundlegenden Elemente von DLV sind Konstanten. Konstanten müssen mit einem Kleinbuchstaben beginnen und dürfen aus Buchstaben, Unterstrichen und Ziffern bestehen. Zusätzlich sind alle Zahlen Konstanten. Ausgenommen ist das Schlüsselwort `not`. Im Gegensatz zu Konstanten müssen Variablen mit einem Großbuchstaben beginnen und dürfen Buchstaben, Unterstriche und Ziffern enthalten.

Eine Besonderheit ist die anonyme Variable. Sie wird durch einen Unterstrich „_" gekennzeichnet. Jedes Auftreten von „_" stellt eine neue und einzigartige Variable dar, die nirgendwo sonst in derselben Regel/Bedingung auftritt. Eine anonyme Variable wird verwendet, wenn ein Argument ignoriert werden kann oder es keine Auswirkungen auf die aktuelle Regel/Bedingung hat. Dies ist eine Alternative zu immer neuen Namen für Variablen, die sonst keine Verwendung hätten. Im Quelltext 1 und 2 sind Beispiele für Variablen und Konstanten dargestellt.

```
1 a1, 1, 9862, aBC1, cC__
```

Quelltext 1: Konstanten

```
1 A, V2F, Vi_X3
```

Quelltext 2: Variablen

Ein Funktionsterm ist ein Bezeichner, gefolgt von einer eingeklammerten Liste von Konstanten, Variablen, Zahlen und/oder Funktionstermen. Der Bezeichner unterliegt den selben Beschränkungen wie eine Konstante. Er muss mit einem Kleinbuchstaben beginnen und darf aus Buchstaben, Unterstrichen und Ziffern bestehen. Beispiele für Funktionsterme sind in Quelltext 3 zu sehen.

```
1 f(1), g_1(a), fUN(1,a,f(5)), f(X), f(a,f(gibon(X),1,Y),Zulu)
```

Quelltext 3: Funktionsterme

Ein Literal ist eine Konstante, eine Zahl oder ein Funktionsterm. Diesem kann ein explizites „-" oder schwaches `not` Negationssymbol vorangestellt sein. Beispiele für Literale sind in Quelltext 4 gezeigt.

```
1 -a
  not ~b(8,K)
3 not weight(X,1,kg)
```

Quelltext 4: Literale

Fakten sind Konstanten oder Funktionsterme, die nur Konstanten enthalten, die durch einen „." beendet werden. Damit wird im logischen Programm ein Wert als wahr definiert. Ein Fakt kann durch Voranstellen von „-" explizit negiert werden. Einige Fakten sind in Quelltext 5 zu sehen.

```
weight(apple,100,gram).
-valid(1,equals,0).
```

Quelltext 5: Fakten

Ein Kommentar in DLV wird durch ein Prozentzeichen „%" eingeleitet. Alle Zeichen nach einem Kommentar bis zum nächsten Zeilenumbruch werden vom Solver ignoriert.

2.2.3 Regeln und Bedingungen

Regeln legen die Beziehungen zwischen Fakten fest. Grundsätzlich haben Regeln die folgende Form h :- $b_1, ..., b_n$.. Hierbei muss h ein Literal sein[4]. Es wird als der Kopf der Regel bezeichnet und repräsentiert die Schlussfolgerung. $b_1, ..., b_n$ sind der Rumpf[5] einer Regel und repräsentieren die Prämissen. Auch dies sind Literale. Sie sind durch Kommata getrennt. Ein Komma hat hier die Eigenschaft eines logischen Und. Der Rumpf einer Regel kann leer sein, womit der Kopf immer wahr ist. Er wird durch einen Punkt „." abgeschlossen.

Eine Regel kann wie folgt verstanden werden: Wenn alle Elemente des Rumpfes wahr werden, dann muss auch der Kopf wahr werden. Regeln, die wahr werden führen zur Erweiterung der Modelle um die Elemente des Kopfes. Quelltext 6 zeigt einige Beispiele für Regeln.

```
online :- got_internet.
starving :- hungry,thirsty.
person(X) :- men(X), alive(X).
true :- .
ok :- not hazard.
employee(P) :- personnel_info(_,P,_,_,_).
```

Quelltext 6: Regeln

[4]Es wird darauf hingewiesen, das DLV im Kopf auch Disjunktionen zulässt. Dieses ist aber für die Arbeit unerheblich.
[5]Englisch: Body.

Eine Regel mit leerem Kopf ist ein Sonderfall und wird *Bedingung* genannt. Dieses Konstrukt ist zur Beschränkung für das gesamte logische Programm gedacht. Im einzelnen können dadurch Elemente in Modellen reduziert oder ganze Modelle ausgeschlossen werden. Der leere Kopf wird logisch betrachtet als nicht wahr angesehen. Somit darf der Rumpf auch nie wahr werden. Beispiele sind in Quelltext 7 gezeigt.

```
:- color(apple,red), color(apple,green).
:- -healthy(X), not sick(X).
```

Quelltext 7: Bedingungen

2.2.4 Prädikate

DLV verfügt über eingebaute *arithmetische* und *vergleichende* Prädikate. Konstanten können mittels der eingebauten Prädikate `<, >, <=, >=, =, !=` verglichen werden. Diese können in Präfix- oder Infixnotation verwendet werden. Alle Arten von Konstanten und Zahlen können frei miteinander verglichen werden. Wenn zwei Zahlen verglichen werden, ist die Semantik wie erwartet. Alle anderen Vergleiche garantieren nur eine feste Reihenfolge über alle Konstanten. Beispiele für Vergleiche sind in Quelltext 8 dargestellt.

```
in_range(X,A,B) :- X>=A, <(X,B).
pair(X,Y) :- Y>X, color(X,green), color(Y,green).
```

Quelltext 8: Vergleiche

Aussagen und Berechnungen über eine endliche Menge von natürlichen Zahlen sind mit den Prädikaten `#int, #succ, #prec, #mod, #absdiff, +, *, -, /` möglich. Diese sind *nur* definiert, wenn Integer-Arithmetik aktiv ist. Durch Angeben einer oberen Grenze wird die Integer-Arithmetik aktiviert. Dies wird mit mit dem Kommandozeilenparameter `-N=Grenze` oder mit dem Schlüsselwort `#maxint(Grenze)` erreicht. DLV kennt keine Zahlen, die kleiner 0 oder größer als die angegebene Grenze sind. Ein arithmetisches Prädikat wird nie Zahlen außerhalb des bekannten Bereichs generieren. Wenn Konstanten außerhalb dieses Bereichs im Programm auftreten, wird eine Warnung beim Aufruf ausgegeben.

Die Definition der arithmetischen Prädikate aus der Online-Dokumentation ist in Quelltext 9 abgebildet.

```
#int(X) is true, iff X is a known integer (i.e. 0<=X<=N).
#succ(X, Y) is true, iff X+1=Y holds.
#prec(X, Y) is true, iff X-1=Y holds.
#mod(X, Y, Z) is true, iff X%Y=Z holds.
#absdiff(X, Y, Z) is true, iff abs(X-Y)=Z holds.
+(X,Y,Z), or alternatively: Z=X+Y is true, iff Z=X+Y holds.
*(X,Y,Z), or alternatively: Z=X*Y is true, iff Z=X*Y holds.
-(X,Y,Z), or alternatively: Z=X-Y is true, iff Z=X-Y holds.
/(X,Y,Z), or alternatively: Z=X/Y is true, iff Z=X/Y holds.
```

Quelltext 9: Definition der arithmetischen Prädikate

Jedes arithmetische Prädikat hat eine Reihe von Eingabeargumenten und genau ein Ausgabeargument. Das Ausgabeargument ist immer das letzte Argument und dessen Wert wird aus den Werten der Eingabeargumente berechnet. Beispiele für arithmetische Prädikate sind in Quelltext 10 aufgezeigt.

```
fib(N,F) :- #succ(N2,N1), #succ(N1,N), fib(N1,F1), fib(N2,F2
   ), +(F1,F2,F).
previousSec(Y) :- sec(X), #prec(X,Y).
even(X) :- #int(X), #mod(X,2,0).
odd(X) :- #int(X), not #mod(X,2,0).
weight(X,KG,kilogram) :- weight(X,G,gram), *(G,1000,KG).
product(X) :- #int(P), #int(Q), X=P*Q.
productOfPrimes(X) :- #int(P), #int(Q), X=P*Q, P>1, Q>1.
prime(A) :- #int(A), not productOfPrimes(A).
netWeight(X,N) :- fullweight(X,W), tare(X,T), N=W-T.
monthlyFee(Y) :- fee(X), Y=X/12.
```

Quelltext 10: Beispiele für arithmetischen Prädikate

2.2.5 Negation

DLV implementiert zwei unterschiedliche Arten von Negation, *explizite* und *schwache*. Die schwache Negation ist nach folgendem Konzept realisiert: Wenn in einem Modell ein Fakt nicht wahr ist, dann kann die Negation des Faktes als wahr angenommen werden. Ein einfaches Beispiel ist in Quelltext 11 zu sehen.

```
open_windows :- not raining.
```

Quelltext 11: Schwache Negation

Die schwache Negation unterstützt keine explizite Behauptung des Falschen. Es gibt aber Situationen, in denen mit absoluter Sicherheit gewährleistet sein muss, dass ein Fakt nicht wahr ist. Hier ist schwache Negation ungeeignet und die explizite Negation wird verwendet. Ein Literal oder Fakt wird explizit negiert, in dem ein „-" oder „∼" vorangestellt wird.

Um die Auswirkungen zu verdeutlichen, stelle man sich Quelltext 12 und 13 in einem agentenbasierenden System vor.

```
1 cross_railroad:- not train_approaches.
```

Quelltext 12: Schwache Negation

```
1 cross_railroad:- -train_approaches.
```

Quelltext 13: Starke Negation

Agent I würde ein Kreuzen der Bahnschienen vornehmen, wenn er keine Informationen über die Ankunft eines Zuges hat. Dies könnte aber auch der Fall sein, wenn Agent I eine Fehlfunktion hat, z.b. Sensorausfall. Agent II würde nur dann die Bahnschienen kreuzen, wenn er valide Beweise hat, dass kein Zug kommt. Kurz könnte man sagen: Agent I kreuzt, wenn er der Meinung ist, dass kein Zug kommt, während Agent II dies nur tut, wenn er es weiß. In diesem Zusammenhang wären die offenen Fenster bei Regen aus Quelltext 11 wohl nicht ganz so schlimm.

2.2.6 Schwache Bedingungen

Zusätzlich zur Bedingung existiert die schwache Bedingung. Dieses Merkmal ermöglicht es, mehrere Optimierungsprobleme in einer einfachen und natürlichen Art und Weise zu formulieren. Während Bedingungen immer erfüllt sein müssen, sollten schwache Bedingungen soweit möglich erfüllt werden. Ihre Verletzung führt aber nie dazu, dass ein Modell als Lösung nicht möglich bzw. nicht gefunden wird.

Die Lösungen eines Programms P mit einer Menge W von schwachen Bedingungen sind die Lösungen, die die Zahl der verletzten schwachen Bedingungen minimieren. Sie

werden die *besten Modelle* von (P, W) genannt. Ein Programm kann mehrere beste Modelle haben. In diesem Fall verletzt jedes Modell unterschiedliche schwache Bedingungen, diese aber in gleicher Anzahl.

Schwache Bedingungen können *gewichtet* werden. Je höher das Gewicht, desto wichtiger wird die Bedingung. Sollten Gewichte angegeben sein, minimiert der Solver die Summe der Gewichte der verletzten schwachen Bedingungen. Schwache Bedingungen können auch *priorisiert* werden. Hier minimiert der Solver die Verletzung der Nebenbedingungen nach der Prioritätsstufe. Er beginnt mit der höchsten zuerst und berücksichtigt die unteren Prioritätsstufen nacheinander in absteigender Reihenfolge.

Syntaktisch werden schwache Bedingungen wie in Quelltext 14 angegeben. Schwache Bedingungen ohne Gewicht oder Priorität werden mit eins gewertet.

```
1 :~ Bedingung [Gewicht:Prioritaet]
```

Quelltext 14: Schwache Bedingungen

Das folgende Beispiel zweigt schwache Bedingungen mit Gewichten und Prioritäten anhand eines Planungsproblems. Ziel ist die Zuordnung einer bestimmten Gruppe von Mitarbeitern zu zwei Projekten. Es ist gewünscht, dass sich die Mitglieder der gleichen Gruppe bereits kennen, dies wird aber nicht als am Wichtigsten angesehen (Priorität 1). Mehr Wert wird darauf gelegt, dass die Kompetenzen jeder Gruppe heterogen sind. Genauso wichtig wird die Tatsache gesehen, dass verheiratete Mitarbeiter nicht im gleichen Team sind. In Quelltext 15 ist dieses als Programm formuliert.

```
1  employee(a). employee(b). employee(c). employee(d). employee
       (e).
   know(a,b). know(b,c). know(c,d). know(d,e).
3
   same_skill(a,b).
5  married(c,d).

7  member(X,p1) v member(X,p2) :- employee(X).

9  :~ member(X,P), member(Y,P), X != Y, not know(X,Y). [1:1]
   :~ member(X,P), member(Y,P), X != Y, married(X,Y). [1:2]
11 :~ member(X,P), member(Y,P), X != Y, same_skill(X,Y). [1:2]
```

Quelltext 15: Planungsproblem

Als Resultat liefert DLV zwei Modelle, die in Quelltext 16 zu sehen sind.

```
Best model:
{member(a,p2), member(b,p1), member(c,p1), member(d,p2),
   member(e,p2)}
Cost ([Weight:Level]): <[6:1],[0:2]>

Best model:
{member(a,p1), member(b,p2), member(c,p2), member(d,p1),
   member(e,p1)}
Cost ([Weight:Level]): <[6:1],[0:2]>
```

Quelltext 16: Planungsproblem Lösungen

2.2.7 Sicherheit und endliche Domain Prüfung

Implizite Sicherheitsbedingungen für Variablen in Regeln werden beim Aufruf von DLV automatisch geprüft. Diese garantieren, dass eine Regel dem logischen Äquivalent der Menge seiner Herbrand-Instanzen entspricht.

Eine Variable X in einer Regel ist sicher, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- X tritt in einem positiven Prädikat im Rumpf der Regel auf.
- X tritt in einem explizit negierten Prädikat im Rumpf der Regel auf.
- X tritt als letztes Argument eines arithmetischen Prädikates auf. Alle anderen Argumente des Prädikates sind sicher.

Eine Regel ist sicher, wenn alle ihre Variablen sicher sind. Jedoch sind zyklische Abhängigkeiten unzulässig. Zum Beispiel ist die Regel `:- #succ (X, Y), #succ (Y, X)` nicht sicher. In Quelltext 17 sind Beispiele für sichere Regeln gezeigt. Quelltext 18 zeigt Regeln, die nicht sicher sind.

```
a(X) :- not b(X), c(X).
a(X) :- X > Y, node(X), node(Y).
a(Y) :- number(X), #prec(X,Y).
a(Z) :- number(X), #succ(X,Y), Z=X+Y.
:- number(X), number(Y), #mod(X,Y,2).
:- -a(Y), not b(Y), not c(Y).
```

Quelltext 17: Sichere Regeln

Bei Programmen mit arithmetischen Prädikaten oder Funktionstermen wird eine endliche Domain-Prüfung durchgeführt, die sicherstellt, dass diese Programme terminieren. Aus Regeln mit arithmetischen Prädikaten im Rumpf können neue Konstanten (Zahlen)

```
a(X) v -a(X).
a(X) :- not b(X).
a(X) :- number(Y), X=Y*Z.
a(X) :- number(Y), #succ(X,Y).
:- not number(X), #succ(X,Y).
:- not -b(Y).
:- X <= Y, node(X).
```

Quelltext 18: Nicht sichere Regeln

geschlussfolgert werden. Mit Hilfe von Rekursion ist es möglich, einen Generator für eine unendliche Menge von Konstanten zu beschreiben. Ein Aufruf von DLV würde ohne explizite Grenze nicht terminieren. Deshalb muss bei Programmen mit Integer-Arithmetik, wie in Abschnitt 2.2.4 auf Seite 9 beschrieben, eine obere Grenze mit angegeben werden. Ein Beispiel für einen Generator ist in Quelltext 19 aufgeführt.

```
p(0).
p(Y) :- p(X), #succ(X,Y).
```

Quelltext 19: Generator

Programme mit Funktionstermen im Kopf einer Regel können unter Umständen nicht terminieren. Diese Programme sind nur sicher, wenn alle Argumente eines Funktionsterms durch *beschränkende* Literale zusätzlich definiert sind. In Quelltext 20 ist ein nicht terminierendes Programm gezeigt, welches in Quelltext 21 durch ein weiteres Literal sicher wird.

```
p(0).
p(f(X)) :- q(X).
q(X) :- p(X).
```

Quelltext 20: Funktionsterm im Kopf, nicht terminierend

```
p(0). r(0).
p(f(X)) :- r(X), q(X).
q(X) :- p(X).
```

Quelltext 21: Funktionsterm im Kopf, terminierend

3 Die Firma GG-Net

Die GG-Net GmbH wurde im Dezember 2006 gegründet und hat drei Unternehmensschwerpunkte: das Systemhaus, die Acer-Reparatur-Annahme und den Bereich Sonderposten. Der Bereich Sonderposten wird detailliert im nächsten Kapitel beschrieben.

Im Systemhaus werden hauptsächlich Geschäftskunden betreut, wie z. B. öffentliche Institutionen, Handelsfirmen, Arztpraxen, Kanzleien, Speditionen und auch Privatkunden. Der Tätigkeitsbereich erstreckt sich von der kompetenten und individuellen Beratung für optimierte IT-Infrastruktur über die Beschaffung von Hard- und Software, Dienstleistungen und Wartungen der Infrastruktur. Zusätzlich wird individuelle Programmierungen, Hosting, Gestaltung und Pflege von Webseiten betrieben.

Die Reparaturannahme bietet die Möglichkeit defekte Geräte der Acer Gruppe, also Acer, Packard Bell, eMachines und Gateway im Ladengeschäft abzugeben und nach erfolgter Reparatur wieder abzuholen. Die Reparatur selbst erfolgt im Acer Reparatur Zentrum in Ahrensburg. Der Transfer der defekten und reparierten Geräte erfolgt mehrfach am Tag über einen exklusiven Transport Service von GG-Net.

Seit Gründung der Firma wurden Prozesse optimiert und kreative Lösungen für ungewöhnliche Situationen gesucht. Dazu gehört unter anderem der Einsatz von DLV in der internen Software.

3.1 Sonderposten

Bei dem Bereich Sonderposten handelt es sich um den Verkauf in Kommission von gebrauchter IT-Hardware. Jeder Lieferant stellt GG-Net Ware, welche aufbereitet und verkauft wird. Nach dem Verkauf, zahlt GG-Net den erreichten Preis abzüglich einer Bearbeitungsgebühr an den Lieferanten. Der Verkaufspreis kann von GG-Net weitgehend frei gewählt werden, teilweise haben Lieferanten aber Schranken festgelegt. Durch Verträge ist es GG-Net zusätzlich möglich, diese Ware mit mindestens einem Jahr Herstellergarantie zu vermarkten.

Die Ware wird direkt von den Lieferanten auf Palette angeliefert. Hierbei ist hervorzuheben, dass eine Palette nicht viele Stückzahlen eines Produktes (z.B. 40 Acer Aspire One Netbooks) enthält, sondern viele Produkte mit kleineren Stückzahlen (2 Notebooks Typ A, 3 Notebooks Typ B, 3 PCs, 2 Bildschirme, etc.). Dies wird gemischte Ware genannt.

Die Arbeit des Sonderpostens wird in drei Prozesse aufgeteilt:

1. Die Aufnahme
2. Das Festlegen der Verkaufspreise
3. Der Verkauf

Während der Aufnahme erhält jedes Gerät eine eindeutige Sonderpostennummer (SopoNr), auch RefurbishId genannt. Jedes Gerät wird registriert und auf Funktionalität

und Vollständigkeit geprüft. Die Geräte werden optisch bewertet und schließlich eingelagert. Die SopoNr ist numerisch und wird in den folgenden Prozessen zur Identifikation eines einzelnen Gerätes verwendet. Die Aufnahme erfolgt über eine unternehmensinterne Software und enthält Produktdetails wie CPU, Hauptspeicher, Festplattengrösse und Details einzelner Geräte wie „hat Kratzer“, Produktionsdatum 01.03.2011 oder „Verpackung: Originalkarton“. Bei der Aufnahme der Geräte werden Testprogramme verwendet, um die Funktionsfähigkeit zu prüfen und technische Details zu identifizieren. Des weiteren wird geprüft ob das Zubehör vollständig ist (z.B. Netzkabel, Akku, Netzteil). Schließlich werden die Geräte im Lager nach SopoNr einsortiert.

Der Verkaufspreis für jedes einzelne Gerät, wird automatisch durch einen Algorithmus, die Preis-Maschine (PM), festgelegt. Zur Optimierung werden externe Information verwendet (z.B. stellen einige Lieferanten ihre Einkaufspreise zur Verfügung). Die Ergebnisse werde in einer für Menschen lesbaren Form ausgegeben. Der Anwender ist in der Lage, einige der Entscheidungen der PM zu überschreiben und einen expliziten Preis zu hinterlegen. Die Preise werden dann in die Verkaufsplattform übergeben.

Der Verkauf der Geräte an Händler und Endverbraucher geschieht über eine Verkaufsplattform, welche im Ladengeschäft und im Internet verwendet wird. Nach einem Verkauf wird der endgültige Verkaufspreis an den jeweiligen Lieferanten für die Abrechnung gemeldet.

3.2 Preis-Maschine

Wie im vorherigen Kapitel beschrieben wird die PM verwendet um die Verkaufspreise festzulegen und Vertriebsinformationen zu liefern. Diese verwendet als Eingabe die Merkmale eines Gerätes und liefert als Ausgabe den Preis für dieses Gerät und optional Vertriebsinformationen. Die PM existiert als ASP, welches in DLV implementiert wurde. Im Folgenden wird beschrieben, wieso die die PM in DLV implementiert ist, welche Merkmale von einzelnen Geräten erfasst sind und wie diese in logischen Fakten dargestellt werden. Im Anschluss wird darauf eingegangen, welche Schwerpunkte in der PM modelliert sind. Abschließend werden beispielhafte *Modelle* und die enthaltenen Informationen betrachtet.

Ursprünglich war die PM in Java implementiert. In ihrer kontinuierlichen Entwicklung wurde festgestellt, das sich die in Ihr enthaltenen Regeln in einer erschwerte Umgebung befinden. Ein Beispiel: Es existiert eine Regel, die bei neueren Geräten (jünger als ein Jahr) eine untere Schranke für den Verkaufspreis festlegt. Die untere Schranke gestaltet sich aus einem festgelegten Prozentsatz von den Produktionskosten. Jetzt ergibt sich die Situation, daß die definierte untere Schranke so hoch ist, daß sich einzelnen Geräte nicht verkaufen. Nach dem die Geräte aber ein Jahr alterten greift diese Regel nicht mehr und ein viel niedrigerer Preis wird vergeben (Tag 365 = 300 €; Tag 366 = 190 €). An diesem Tag werden die Geräte für die Kunden sehr attraktiv und sofort verkauft. Da es sich aber immer nur um einzelne Geräte handelt, fällt dies im ersten Moment nicht auf. Erst durch sehr aufmerksame Mitarbeiter und statistische Revision wird ein solcher Umstand

erkannt. Um der erschwerten Umgebung Rechnung zu tragen wurde die PM als Answer Set Programm in DLV reimplementiert um auf Möglichkeiten wie Bedingungen zugreifen zu können.

Merkmal	Gerät 1	Gerät 2	Gerät 3
Brand	Acer	Packard Bell	Acer
Klasse	Mittel	Einstieg	Mittel
Name	Aspire 3810TM	iMedia 3411	B223w
SonderpostenNr	10001	10002	10003
Warengruppe	Notebook	Desktop	Monitor
Produktionsdatum	2011-01-03	2010-07-12	2010-03-03
Festplatte	320 GB	1000 GB	-
Arbeitsspeicher	4 GB	6 GB	-
Kostpreis	340,00€	231,45€	110,00€
CPU	Intel Core i5-450M	AMD A4-3305M	-
optisches Laufwerk	Bluray Combo	DVD-ROM	-
Betriebssystem	Window 7 (32 Bit)	Linux	-
Display	15"WSXGA	-	23"Full HD
Verpackung	original Karton	neutraler Karton	neutraler Karton
optischer Zustand	neuwertig	stark gebraucht	leicht gebraucht

Tabelle 1: Merkmale und Geräte mit Beispieldaten

Bei der Aufnahme werden die Merkmale von jedem einzelnen Gerät erfasst. Hierbei wird zwischen notwendigen Merkmalen, abhängigen Merkmalen und optionalen Merkmalen unterschieden. Notwendige Merkmale müssen hinterlegt werden und existieren an jedem Gerät (z.B. Seriennummer). Abhängige Merkmale hängen von dem Wert eines notwendigen Merkmals ab und sind entsprechend nur bei dessen Auftreten zwingend zu hinterlegen (z.B. Notebook → CPU, GPU, Display und Festplatte müssen hinterlegt werden). Optionale Merkmale können gesetzt sein, sind aber nicht verpflichtend. (z.B. Gerät hat Kratzer). In Tabelle 1 sind einige Merkmale[6] nach Kategorie aufgelistet sowie einige Geräte mit Beispieldaten.

```
1 unit(acer, notebook, entry, "Acer Aspire 3810T", 10001, 220,
    50000, 34000, core_i5, i5_450M, bluray_combo, 4096, 320,
    windows_7, 15, wsxga, originalBox, almostNew).
```

Quelltext 22: Gerät 1

Um die Merkmale eines einzelnen Gerätes in der PM zu verwenden, müssen sie im DLV Syntax dargestellt werden. Dabei werden einige Merkmale konvertiert. Preise werden in Cent konvertiert, da DLV keine Gleitkomma-Arithmetik beherrscht. Datumsangaben werden zum Alter in Tagen konvertiert, wobei das Delta zwischen Datum und

[6]Die hier gezeigten Merkmale werden in dieser Arbeit weiter betrachtet.

aktuelle Zeitangabe gebildet wird. Die PM erwartet einen einzelnen Fakt der alle Merkmale enthält. In Quelltext 22 ist Gerät 1 aus Tabelle 1 als DLV Fakt dargestellt. Des weiteren wird in der PM im Zuge des Lösens einzelne Fakten für jedes Merkmal gebildet. Der hierfür verwendete Code wird in Quelltext 23 beispielhaft gezeigt.

```
brand(X)        :- unit(X,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
productGroup(X):- unit(_,X,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
class(X)        :- unit(_,_,X,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
name(X)         :- unit(_,_,_,X,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
refurbishId(X) :- unit(_,_,_,_,X,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
mfgDelta(X)     :- unit(_,_,_,_,_,X,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
lastPrice(X)    :- unit(_,_,_,_,_,_,X,_,_,_,_,_,_,_,_,_,_,_,_,_).
costPrice(X)    :- unit(_,_,_,_,_,_,_,X,_,_,_,_,_,_,_,_,_,_,_,_).
cpu(X,Y)        :- unit(_,_,_,_,_,_,_,_,X,Y,_,_,_,_,_,_,_,_,_,_).
odd(X)          :- unit(_,_,_,_,_,_,_,_,_,X,_,_,_,_,_,_,_,_,_,_).
ram(X)          :- unit(_,_,_,_,_,_,_,_,_,_,X,_,_,_,_,_,_,_,_,_).
hdd(X)          :- unit(_,_,_,_,_,_,_,_,_,_,_,X,_,_,_,_,_,_,_,_).
os(X)           :- unit(_,_,_,_,_,_,_,_,_,_,_,_,X,_,_,_,_,_,_,_).
display(X,Y)    :- unit(_,_,_,_,_,_,_,_,_,_,_,_,_,X,Y,_,_,_,_,_).
packaging(X)    :- unit(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,X,_,_,_,_).
quality(X)      :- unit(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,X,_,_,_).
```

Quelltext 23: Merkmale als einzelne Fakten

Bei der Implementation der PM wurden verschiedene Schwerpunkte abgebildet. Hierbei handelt es sich um das Prüfen den Eingabedaten auf Fehler, das Ermitteln der Preise und das Erzeugen von Vertriebshinweisen. Um Fehler im System zu vermeiden werden die Eingabedaten geprüft. Dazu existieren Bedingungen die zur korrekten Lösung der PM eingehalten werden müssen. Dies betrifft unter anderem die Validität der Eingabewerte und den Ausschluss von fehlerhaften oder unzulässigen Fakten. In Quelltext 24 werden einige Bedingungen präsentiert.

```
:- price(X), price(Y), Y != X.
:- costPrice(X), X < 1.
:- ram(X), X < 1.
costPrice_set :- costPrice(_).
:- not costPrice_set.
vhdd(320).  vhdd(500).  vhdd(640). vhdd(750).
vhdd(1000). vhdd(1500). vhdd(2000).
valid_hdd_size :- hdd(X), vhdd(Y), X = Y.
:- hdd(_), not valid_hdd_size.
```

Quelltext 24: Validation

In Zeile 1 ist eine Bedingung zu sehen, die sicherstellt, daß nie mehr als ein Fakt `price(_)` existiert. In Zeile 2 und Zeile 3 ist definiert, daß die Merkmale `costPrice(X)` und `ram(X)` nie den Wert 0 haben. Die Zeilen 6 bis 9 zeigen Bedingungen für explizit zugelassene Werte für das Merkmal `hdd(X)` .

Die *Ermittlung der Preise* ist die Hauptaufgabe der PM. Im Kern steht eine lineare Funktion die verschiedene Gewichte kombiniert. Die Gewichte werden aus einzelnen Regeln gebildet und schlussendlich durch die Merkmale eines Gerätes erzeugt. Weiterhin umfasst dieser Schwerpunkt Regeln, die unerwünschtem Verhalten entgegenwirken oder einen spezifischen Kurzzeiteffekt modellieren (z.B. 2% Rabat auf alle Bildschirme). Dies wird über zusätzliche Regeln umgesetzt. In Quelltext 25 sind beispielhafte Auszüge aus der PM dargestellt, die die Funktionalität verdeutlichen sollen.

In den Zeilen 1 bis 9 sind einige Gewichte für das Merkmal CPU zu sehen. Die Zeilen 11 bis 13 stellen Gewichte für das Merkmal optisches Laufwerk dar. Von Zeile 15 bis 19 sind Gewichte für das Merkmal Betriebssystem zu sehen. Gewichte für das Merkmal Display sind in den Zeilen 21 bis 34 sichtbar. Ein Display liefert hier zwei Gewichte. Die Implementation verwendet speziell in Zeile 34 die Variante, daß eine nicht explizite gesetzte Assoziation ein Standardgewicht liefert. Die lineare Funktion die alle Gewichte verwendet wird in Zeile 36 gezeigt. Im Logischen ist diese schlecht lesbar, auf Grund der beschränkten arithmetischen Fähigkeiten von DLV. In der Zeile 40 wurde der oben beschriebene Kurzzeiteffekt dargestellt, einen Rabat auf den ermittelten Preis für alle Bildschirme zu geben. Ein zu niedrigen Preis, also ein unerwünschtes Verhalten, wird in Zeile 42 verhindert.

Wie in Abschnitt 2.2 auf Seite 6 beschrieben findet DLV kein, ein oder mehrere Modelle. Jedes dieser Modelle entspricht der stabilen Modellsemantik und ist eine Lösung für das gegebene logische Programm. Im Fall der PM sind die Lösungen *kein Modell* und *mehrere Modelle* als Fehlerfälle definiert. Eine korrekte Lösung hat also ein einziges Modell. Exemplarisch sieht ein Aufruf von DLV wie folgend aus: `dlv unit.dl global.dl convert.dl pm.dl`. Die Datei `pm.dl` enthält alle Regeln der PM. In der Datei `convert.dl` sind all Regeln hinterlegt die aus dem Fakt eines Gerätes mehrere Fakten schlussfolgern. Tagesaktuelle Fakten (z.B. verfügbare Lagerkapazität) befinden sich in der Datei `global.dl`. Diese Datei wird täglich automatisch aktualisiert. Die Datei `unit.dl` enthält die Merkmale des jeweiligen Gerätes welches bewertet werden soll. Sie wird vor jedem Aufruf neu erstellt.

In Quelltext 26 ist die Lösung für das oben vorgestellte Gerät gezeigt. Hier sind die Fakten zu jedem Merkmal zu sehen, die durch die Konvertierung des Gerätes entstanden sind. Es sind die verschiedenen Gewichte enthalten die aus den Fakten der Merkmale geschlussfolgert wurden. Bei dem hier gezeigten Modell ist für die weitere Verarbeitung nur der Fakt `price(83000)` interessant. Er repräsentiert einen ermittelten Preis von 830,- €.

```
w_cpu(50)  :- cpu(core_i5, i5_450M).
w_cpu(50)  :- cpu(core_i5, i5_2500K).

w_cpu(120) :- cpu(core_i7, i7_2330).
w_cpu(140) :- cpu(core_i7, i7_2670QM).

w_cpu(40)  :- cpu(amd_a, a4_3300M).
w_cpu(40)  :- cpu(amd_a, a4_3305M).
w_cpu(45)  :- cpu(amd_a, a4_3400).

w_odd(3)   :- odd(dvd_rom).
w_odd(7)   :- odd(dvd_super_multi).
w_odd(47)  :- odd(bluray_super_multi).

w_os(5)  :- os(linux).
w_os(5)  :- os(android).
w_os(25) :- os(windows_xp).
w_os(30) :- os(windows_7).
w_os(50) :- os(windows_8).

w_display_size(30) :- display(X,_), X <= 12.
w_display_size(45) :- display(X,_), X > 12, X <= 14.
w_display_size(5)  :- display(X,_), X > 14, X <= 16.
w_display_size(15) :- display(X,_), X > 16, X <= 17.
w_display_size(30) :- display(X,_), X > 17.

w_display_res_pre(30) :- display(_,wuxga).
w_display_res_pre(20) :- display(_,full_hd).
w_display_res_pre(10) :- display(_,hd).
w_display_res_pre(10) :- display(_,wsxga).

w_display_res_set :- w_display_res_pre(_).
w_display_res(X)  :- w_display_res_pre(X).
w_display_res(5)  :- not w_display_res_set.

price(P) :- w_cpu(A), w_os(B), w_display_size(C),
            w_display_res(D), w_odd(E),
            P = P4 * 1000, P4 = P3 + A, P3 = P2 + B, P2 = P1
                + C, P1 = D + E.

rabatPrice(R) :- productGroup(monitor), price(P), R1 = P *
   98, R = R1 / 100.

:- price(P), costPrice(C), L1 = C * 75, L = L1 / 100, P < L.
```

Quelltext 25: Preisermittlung

```
brand(acer),
productGroup(notebook),
name("Acer Aspire 3810T"),
refurbishId(16),
mfgDelta(300),
lastPrice(62000),
costPrice(40100),
cpu(core_i5,i5_450M),
odd(dvd_rom),
ram(2048),
hdd(750),
os(linux),
display(15,full_hd),
w_cpu(50),
w_odd(3),
w_os(5),
w_display_size(5),
w_display_res_pre(20),
w_display_res_set,
w_display_res(20),
price(83000)
```

Quelltext 26: Faktenmodell

4 Naive Answer Set Programm Verifikation

Verifikationstechniken sind das jüngste Mitglied einer Familie von Techniken zur Unterstützung des Qualitätsmanagements im Software Engineering. Mit der Einführung von Software Engineering in den Entwicklungsprozess wurden auch Möglichkeiten der Qualitätssicherung und des Qualitätsmanagements untersucht. Am Anfang wurden Techniken wie das explorative manuelle Testen verwendet. Ihnen folgte in den 90iger Jahren das automatische Testen, Modultest oder Unittest genannt. Ein Schwerpunkt dieser Technik ist, einmal gefundene und behobene Fehler in Applikationen nicht durch die Implementation von neuen Funktionen wieder auftreten zu lassen.

In der Wissenschaft wurde aber schnell erkannt, dass es äußerst schwierig bis unmöglich ist, eine Software vollständig mit Modultests zu prüfen. Hier bilden Verifikationstechniken die nächste Stufe. Unter Verifikation wird ein Prozess verstanden, der für ein Programm oder ein System sicherstellt, dass es einer Spezifikation entspricht. Es wird also nicht nur eine begrenzte Menge von Fällen geprüft, sondern die gesamte Software oder ganze Bereiche. Während Modultests heutzutage Bestandteil von fast jedem aktuellen Softwareprojekt sind, werden Verifikationstechniken erst allmählich in der Industrie eingesetzt.

In diesem Kapitel wird das naive Ausprobieren als Verifikationstechnik mit ASP vorgestellt. Dazu wird zunächst eine Klassifikation von Verifikationstechniken in Abschnitt 4.1 vorgestellt und Bedingungen als in ASP enthaltene Verifikationstechnik klassifiziert. In Abschnitt 4.2 wird eine Hypothese für ein Verfahren aufgestellt, welches dann in Abschnitt 4.3 beschrieben und im Anschluss analysiert wird. In Abschnitt 4.4 wird das vorgestellte Verfahren in der Preis-Maschine implementiert und die praktische Anwendbarkeit analysiert. Abschließend wird in Abschnitt 4.5 eine Bewertung und Einschätzung des Verfahrens getätigt.

4.1 Klassifikation von Verifikationstechniken

Die meisten Logiken in der Konzeption, Spezifikation und Verifikation von Computersystemen, befassen sich grundlegend mit der Relation

$$\mathcal{M} \models \phi$$

wobei $\mathcal{M}$ eine Art von Situation oder Modell eines Systems ist, und ϕ eine Spezifikation, eine Beschreibung in Logik, die ausdrückt, was in Situation $\mathcal{M}$ wahr sein soll. Die Verifikationstechniken enthalten dann als Herzstück einen oder mehrere Algorithmen, die $\models$ ermitteln.

Im Buch [7] ab Seite 172 werden formale Verifikationstechniken als eine Kombination von drei Teilen beschrieben:

- eine *Modellierung*, also eine Möglichkeit zur formalen Beschreibung von Systemen, typischerweise eine Domain-spezifische Sprache
- eine *Spezifikation* zur Beschreibung von Merkmalen, die überprüft werden sollen, typischerweise eine weitere Domain-spezifische Sprache
- ein *Verifikationsverfahren*, um festzustellen, ob das Modell des Systems die Spezifikation erfüllt.

Weiterhin werden folgende Merkmale zur Klassifikation von *Verifikationsverfahren* vorgestellt:

Beweis-basierend vs. Modell-basierend. In einem Beweis-basierten Ansatz ist die formale Beschreibung des Systems eine Menge logischer Formeln Γ und die Spezifikation eine weitere Formel ϕ. Das Verifikationsverfahren versucht, einen Beweis zu finden, der zeigt, dass $\Gamma \vdash \phi$. In der Regel erfordert dieses Verfahren manuelle Unterstützung.

In einem Modell-basierten Ansatz wird die formale Beschreibung des Systems durch ein Modell $\mathcal{M}$ repräsentiert. Die Spezifikation ist wiederum durch eine Formel ϕ dargestellt. Das Verifikationsverfahren versucht zu prüfen, ob das Modell $\mathcal{M}$ durch ϕ Gültigkeit erhält ($\mathcal{M} \models \phi$). Für endliche Modelle ist diese Berechnung in der Regel automatisch möglich.

Grad der Automatisierung, in dem sich die verschiedenen Verfahren unterscheiden. Dieser Grad kann von vollautomatisch bis absolut manuell reichen. Die meisten existierenden Implementationen liegen irgendwo dazwischen.

Vollständige Verifikation oder Verifikation von Merkmalen. Die Spezifikation kann nur ein oder einzelne Merkmale enthalten, oder das gesamte Verhalten des Systems. Das gesamte Verhalten zu verifizieren, ist oft sehr aufwändig.

Anwendungsbereich, welcher zum Beispiel Hardware oder Software sein kann und sich in sequentieller oder paralleler Umgebung befinden kann. Weitere Einschränkungen bzw. Spezifikationen des Anwendungsbereiches sind möglich und oft gegeben.

Vor der Entwicklungsphase oder nach der Entwicklungsphase. Verifikationsverfahren früh im Verlauf der Systementwicklung zu verwenden, ist oft von großem Vorteil. Korrekturen von Fehlern, die zu Beginn einer Entwicklung entdeckt werden, sind oft mit weniger Aufwand verbunden.

4.1.1 ASP Integrierte Verifikation

ASP enthält bereits ein Verifikationsverfahren. Dieses ist praktisch durch die Funktionsweise und Eigenschaften von ASP unausweichlich gegeben. Hiermit ist die Möglichkeit, in ASP Bedingungen zu formulieren, gemeint.

Die Klassifikation erfolgt nach der im vorherigen Kapitel vorgestellten Auswahl. Die Modellierung und die Spezifikation finden beide in ASP statt, sind das ASP Programm selbst. Das Verfahren liefert der Solver und ist somit Modell-basierend. Der Grad der Automatisierung kann als voll automatisch betrachtet werden, da keine manuellen Eingriffe während des Lösens, also auch der Verifikation, notwendig sind. Es finde nur eine Verifikation von Merkmalen statt, da sich Bedingungen lediglich auf gegebene oder deduzierte Fakten beziehen können. Der Anwendungsbereich kann als Software im sequenziellen Betrieb bestimmt werden, wobei diese Klassifikation nicht sehr klar zu spezifizieren ist. Grundlegend ist der Anwendungsbereich nur durch die Einsatzmöglichkeiten von ASP beschränkt. Die Verifikation findet nach der Entwicklungsphase statt, da der Solver im Falle einer Verletzung von Bedingungen keine Lösung liefert.

Als Beispiel sei hier noch einmal auf Quelltext 24 auf Seite 18 verwiesen. Die Bedingung in Zeile 1 darf zu keiner Zeit erfüllt sein. Sollte sie wahr werden, wird die Lösung als unzulässig betrachtet und der Solver liefert sie entsprechend nicht.

4.2 Hypothese

Nach Ansicht des Autors ist es möglich, ein Answer Set Programm durch Ausprobieren zu verifizieren. Dafür werden folgende Definitionen eingeführt:

Definition 1 *Eine Verifikation durch Ausprobieren ist ein Verfahren, mit dem für ein Answer Set Programm unter Anwendung der Permutation aller Mengen der Eingabemerkmale die Menge aller Mengen der Ausgabemerkmale bestimmt wird und diese Mengen gegen eine Spezifikation geprüft werden.*

Definition 2 (Geschlossenheit) *Eine Verifikation durch Ausprobieren ist nur möglich, wenn das gegebene Answer Set Programm ein geschlossenes Problem ist.*

Unter einem geschlossenen Problem wird verstanden, dass das Problem auf keine externen Quellen mit veränderlichen Daten zugreifen darf. DLV als ASP Implementation bietet zum Beispiel die Möglichkeit, Fakten direkt aus einer relationalen Datenbank zu laden. Eine solche Funktion darf nicht verwendet werden.

Definition 3 (Beschränkte Eingabemerkmale) *Eine Verifikation durch Ausprobieren ist des weiteren nur möglich, wenn alle Mengen der Eingabemerkmale zu dem gegebenen Problem einen geschlossenen Raum beschreiben.*

Es ist offensichtlich klar, dass alle Parameter, die als Eingabe betrachtet werden, bekannt sein müssen.[7]

Beispiele für *nicht* geschlossene Räume von Merkmalen:

- Instanzen vom Typ Object (Java)
- Strings
- Arrays, Listen und andere Collections.

Beispiele für geschlossene Räume von Merkmalen:

- Enumerationen: (Sonne, Mond); (Frühling, Sommer, Herbst, Winter)
- Boolesche Werte: true, false
- Integer: -2^{31} *bis* 2^{31} (Java).

Der Erfolg einer Verifikation ist wie in 4.1 Klassifikation von Verifikationstechniken auf Seite 22 vorgeschlagen, durch das Erfüllen der Spezifikation vom Modell des Systems gegeben.

Definition 4 (Erfolgreiche Verifikation) *Eine Verifikation durch Ausprobieren ist erfolgreich, wenn die Permutation aller Mengen der Eingabemerkmale und die Menge aller Mengen der Ausgabemerkmale gegen gegebene Spezifikationen geprüft wurden und diese erfüllen.*

4.3 Beschreibung des Verfahrens

Im folgenden wird eine Technik vorgestellt, die die vorgestellten Definitionen erfüllt.

Aus Definition 3 ist abzuleiten, dass ein Automatismus notwendig ist, der es ermöglicht, alle Eingabemerkmale zu erzeugen.

Definition 5 *Eine Verifikation durch Ausprobieren ist nur möglich, wenn ein Automatismus verfügbar ist, der die Permutationsmenge der Mengen aller Eingabemerkmale erzeugt.*

Aus Definition 4 ist abzuleiten, dass eine Möglichkeit der Speicherung der Ausgabemerkmale in Assoziation mit den dazugehörigen Eingabemerkmalen existieren muss.

[7] Wie soll etwas ausprobiert werden, das nicht bekannt ist ?

Definition 6 *Eine Verifikation durch Ausprobieren ist nur möglich, wenn ein geeigneter Speicher existiert, der es erlaubt, Ausgabemerkmale in Assoziation mit Eingabemerkmalen zu speichern.*

Des weiteren ist aus Definition 4 abzuleiten, dass der Speicher nicht nur sequenziell gelesen werden kann. Dies ist notwendig, um eine performante Auswertung von Spezifikationen zu erlauben.

Definition 7 *Der Speicher muss einen Zugriff auf Assoziationsgruppen erlauben, der durch eine Teilmenge der Ein- oder/und Ausgabemerkmale identifiziert wird.*

Erschwert wird diese Auswertung, wenn Spezifikationen so formuliert sind, dass sie sich auf mehrere Assoziationsgruppen beziehen. Ein Beispiel ist in Abschnitt 4.4.2 auf Seite 30 zu sehen. Aus Definition 7 folgt eine Abhängigkeit der Spezifikationen vom gewählten Speicher.

Definition 8 *Die Spezifikationen müssen in einem zur Zusammenarbeit mit dem Speicher geeigneten Format vorliegen.*

Die Verifikation durch Ausprobieren ist eine Zwei-Phasentechnik. In der ersten Phase werden alle Lösungen ermittelt. Dazu muss ein Generator existieren bzw. implementiert werden, der alle notwendigen Eingabedaten in einer Reihe zur Verfügung stellt. Mit Hilfe dieses Generators und des Solvers (DLV) werden nun jeweils alle Modelle für die entsprechenden Eingabemerkmale ermittelt. Diese werden in einen der Definition entsprechenden Speicher geschrieben. Da im Falle der PM alle Eingabemerkmale und alle interessanten Ausgabemerkmale bekannt sind[8], ist eine relationale Datenbank hierfür geeignet. In der zweiten Phase werden die ermittelten Lösungen gegen gegebene Spezifikationen geprüft. Die Spezifikation werden direkt in SQL verfasst. Sollten alle Spezifikationen erfolgreich geprüft sein, kann das Answer Set Programm als verifiziert betrachtet werden.

4.3.1 Klassifikation

Die Modellierung findet in ASP statt, die Spezifikation findet in SQL statt. Die Zwei-Phasentechnik ist modell-basierend. Der Grad der Automatisierung ist voll automatisch[9]. Es findet eine vollständige Verifikation statt, da entsprechend alle Eingabemerkmale geprüft werden. Der Anwendungsbereich ist nur durch die Einsatzmöglichkeiten von ASP beschränkt. Die Verifikation findet vor bzw. während der Entwicklungsphase statt.

[8] Zur Erinnerung: die Ausgabemerkmale sind als Funktionsterme realisiert. Ein inferierter Fakt *price*(80000) lässt sich somit als Attribut vom Type Integer modellieren.

[9] Es wird als klar angenommen, das ein Framework existieren muss, das die automatischen Aufrufe von Phase eins und zwei realisiert.

4.3.2 Grenzen

Die Grenze der Verifikationstechnik liegt klar in der Größe der Permutationsmenge der Eingabeparameter. Da sich naives Ausprobieren im Raum der NPC-Probleme befindet, wächst der zu prüfende Lösungsraum extrem schnell an. Im folgenden werden im Rahmen der Implementation in der PM Analysen und Tests durchgeführt, um festzustellen, in welchen Bereichen dieser Ansatz mit heutiger Rechentechnik nutzbar ist. Hierbei wird auch gezeigt, wie stark der praktische Kontext eine Beschränkung der Eingabeparameter erlaubt.

4.4 Implementation in der Preis-Maschine

Um zu evaluieren, ob die vorgestellte Technik auch in einer realistischen Umgebung einsetzbar ist, wurde sie an der Preis-Maschine angewendet. Dazu musste geprüft werden, ob die PM (a) für das Ausprobieren geeignet und (b) diese Technik auch praktisch anwendbar ist.

Kriterien für (a)

1. Entspricht die PM den Definitionen 2 und 3 der Hypothese.
2. Lässt sich für die PM die Definition 5 erfüllen, also ein Generator für alle Eingabemerkmale implementieren.
3. Lässt sich für die PM ein geeignetes Datenmodell in SQL realisieren, was die Definitionen 6 und 7 erfüllt.
4. Lassen sich geeignete Spezifikationen in SQL formulieren und damit der Definition 8 entsprechen.

Nach Analyse der PM kann bestätigt werden, dass diese Kriterium 1 erfüllt. Um Kriterium 2 zu erfüllen, wurde ein Generator aller Eingabemerkmale implementiert. Der Generator verwendet das vorhandene Datenmodell und dessen Konverter der PM. Für Kriterium 3 wurde ein passendes Datenmodell entworfen und implementiert. Dieses wird in Abschnitt 4.4.1 im Detail vorgestellt. Zur Erfüllung von Kriterium 4 wurden Spezifikationen von praktischem Nutzen in SQL formuliert, welche in Abschnitt 4.4.2 gezeigt werden.

Kriterien für (b) Um eine praktische Anwendbarkeit zu betrachten, muss die Lösung innerhalb eines gegebenen Zeitraums auf definierter Hardware terminieren. Es wird eine aktuelle Hardware (2 CPUs mit je 8 Kernen, 2GHz, 16GB Ram) als Referenzsystem definiert, da dieses der Firma GG-Net bereits zur Verfügung steht. Im Rahmen dieser Arbeit wird ein Rechencluster[10] aus 1024 Kernen als Obergrenze angesehen. Die Lösung sollte

[10] Ein solches System ist als Amazon EC2 für einen Tagespreis von 3200,- € (3. November 2012) erhältlich und erscheint dem Autor somit relevant finanzierbar.

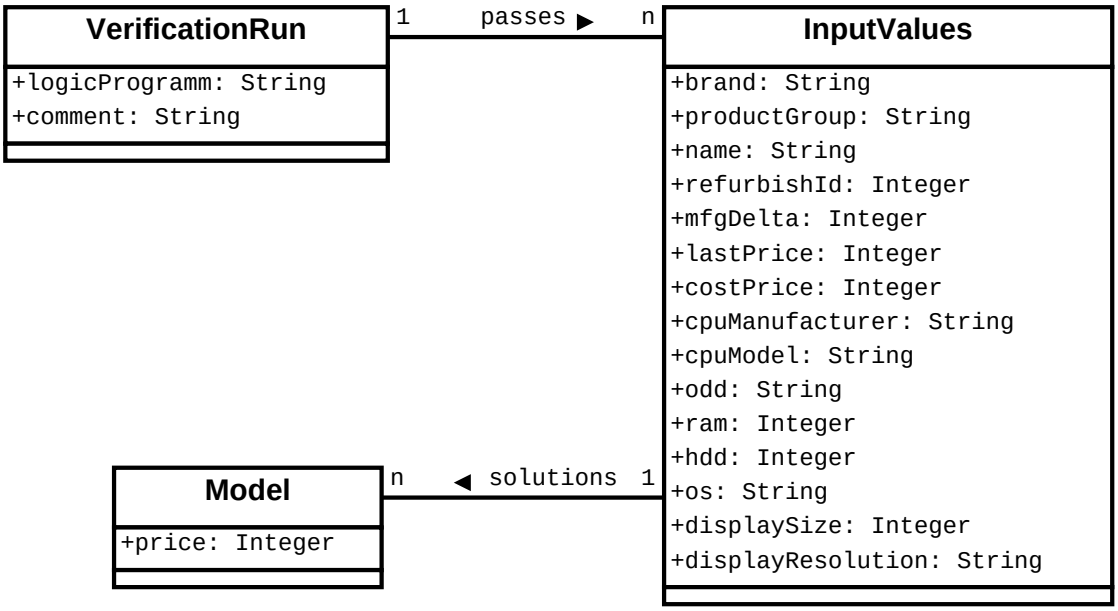

Abbildung 1: Datenmodel

nach 24 Stunden terminieren. Ist dies nicht erreichbar, sollte untersucht und diskutiert werden, in welchem Rahmen Teilergebnisse relevant sind.

Dass die Technik nach den vorgestellten Kriterien terminiert, erscheint auf Grund folgender Aspekte als vielversprechend:

1. Die Eingabewerte sind durch die Realität im einzelnen beschränkt (z.B. es gibt nur eine gewisse Menge von CPUs)
2. Die Kombinationen der Eingabewerte sind beschränkt (z.B. alle Geräte vom Typ Desktop haben kein Display)
3. Die Kombinationen sind durch die Realität eingeschränkt (z.B. gewisse CPUs wurden nie in Notebooks verbaut)
4. Ein Durchlauf der PM liegt bei wenigen Millisekunden.
5. Die PM ist voll funktional und damit leicht parallelisierbar.

4.4.1 Datenmodell

Um Punkt 2 der Kriterien für (a) zu erfüllen, wurde ein passendes Datenmodell entworfen. Das Datenmodell muss für jede eindeutige Menge von Eingabemerkmalen einen Eintrag enthalten. Es muss weiterhin dem Verhalten von ASP entsprechen, dass jeder Aufruf von DLV kein, ein oder mehrere Modell(e) als Ausgabemerkmale erzeugen kann. Dies wurde über ein 1-n Assoziation realisiert. Das Datenmodell ist in Abbildung 1 dargestellt.

Jeder Durchlauf einer Verifikation enthält das logische Programm ohne Eingabemerkmale in Tabelle VerifactionRun. Ein Durchlauf ist mit allen Permutationen von Eingabemerkmalen (Tabelle InputValues) assoziiert. Jede Entität der InputValues hat einen 1-n Assoziation mit Modellen (Tabelle Model). Hier finden sich die interessanten Fakten der Modelle. Auch wenn die PM per Definition alle Fälle mit leerem und mit mehreren Modellen als Fehlerfall definiert, lassen sich diese dennoch mit diesem Datenmodell erfassen. Quelltext 27 zeigt den generierenden SQL Code zum Erzeugen des Datenmodells.

```
CREATE TABLE VerificationRun (
  id INT NOT NULL AUTO_INCREMENT,
  logicProgramm TEXT DEFAULT NULL,
  comment TEXT DEFAULT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE InputValues (
  id INT NOT NULL AUTO_INCREMENT,
  VerificationRun_Id INT NOT NULL,
  brand VARCHAR(255), productGroup VARCHAR(255),
  name VARCHAR(255), refurbishId INT,
  mfgDelta INT, lastPrice INT,
  costPrice INT, cpuManufacturer VARCHAR(255),
  cpuModel VARCHAR(255), odd VARCHAR(255),
  ram INT, hdd INT, os VARCHAR(255), displaySize INT,
  displayResolution VARCHAR(255),
  PRIMARY KEY (id),
  FOREIGN KEY (VerificationRun_Id) REFERENCES
     VerificationRun(id)
);

CREATE TABLE Model (
  id INT NOT NULL AUTO_INCREMENT,
  InputValues_Id INT NOT NULL,
  price INT,
  PRIMARY KEY (id),
  FOREIGN KEY (InputValues_Id) REFERENCES InputValues(id)
);
```

Quelltext 27: Datenmodel DDL

4.4.2 Spezifikation

Zur Erfüllung von Punkt 4 der Kriterien für (a) müssen Spezifikationen von praktischem Nutzen formalisiert und in SQL transformiert werden.

Als Beispiel wird hier ein Verhalten, das als Preissprung bezeichnet wird, formalisiert. Es wurde bereits in Abschnitt 3.2 auf Seite 16 vorgestellt. Ein Preissprung ist eine starke Änderung des Verkaufspreises in Abhängigkeit einer kleinen Änderung eines Eingabemerkmals, in diesem Fall des Produktionsdatums `mfgDelta`.

Folgende formale Beschreibung wird gewählt:

- Eingabemerkmal: `mfgDelta`
- Unter Betrachtung zweier bei einander liegender Werte von `mfgDelta` (Vorgänger, Nachfolger) und Gleichheit aller weiteren Eingabemerkmale darf in allen gefundenen Modellen die Änderung des Ausgabemerkmals `price` maximal 10,-€ betragen.

```
SELECT
  iv1.id,iv2.id
FROM
  InputValues iv1 JOIN Model iv1m
  ON iv1.id = iv1m.InputValues_Id,
  InputValues iv2 JOIN Model iv2m
  ON iv2.id = iv2m.InputValues_Id
WHERE
  iv1.VerificationRun_Id = 1 AND
  iv2.VerificationRun_Id = 1 AND
  iv1.brand = iv2.brand AND
  iv1.productGroup = iv2.productGroup AND
  iv1.name = iv2.name AND
  iv1.refurbishId = iv2.refurbishId AND
  iv1.lastPrice = iv2.lastPrice AND
  iv1.costPrice = iv2.costPrice AND
  iv1.cpuManufacturer = iv2.cpuManufacturer AND
  iv1.cpuModel = iv2.cpuModel AND
  iv1.odd = iv2.odd AND
  iv1.ram = iv2.ram AND
  iv1.hdd = iv2.hdd AND
  iv1.os = iv2.os AND
  iv1.displaySize = iv2.displaySize AND
  iv1.displayResolution = iv2.displayResolution AND
  iv1.mfgDelta = (iv2.mfgDelta+1) AND
  abs(iv1m.price-iv2m.price) > 1000;
```

Quelltext 28: Abfrage Preissprung

Die dazu gehörige SQL Abfrage ist in Quelltext 28 abgebildet. Die Abfrage verwendet zwei Entitäten von InputValues, verbunden mit den assoziierten Modellen. Der Durchlauf ist in Zeile 9 und 10 für dieses Beispiel fest hinterlegt[11]. In Zeile 11 bis 24 wird sichergestellt, dass nur Entitäten mit gleichen Eingabemerkmalen außer `mfgDelta` verglichen werden. Die Vorgänger-Nachfolger-Relation ist durch Zeile 25 gegeben. Die eigentliche Bedingung des maximalen Preisunterschiedes von 10,- € ist in Zeile 26 gezeigt.

Die vorgestellte Abfrage liefert alle Paare von Durchläufen, die einen Preissprung beinhalten. Es ist somit möglich, zu verifizieren, dass die Preis-Maschine unter gar keinen Umständen einen Preissprung abhängig von `mfgDelta` erzeugt.

4.4.3 Laufzeit

Um zu prüfen, ob die Kriterien für (b) erfüllt sind, wird die Laufzeit der Technik untersucht. Dazu wird wie folgt vorgegangen:

1. Feststellung der maximalen Anzahl der Durchläufe
2. Reduktion auf eine realistische Anzahl der Durchläufe
3. Messung der Laufzeit eines einzelnen Durchlaufs
   - Erzeugung einer eindeutigen Menge an Eingabemerkmalen
   - Ausführen der PM mit den Eingabemerkmalen
   - Speichern der Ausgabemerkmale
4. Schätzung der Laufzeit über alle Mengen an Eingabemerkmalen
5. Test mit gewählten Mengen an Eingabemerkmalen.

Damit wird die Laufzeit aller Durchläufe (Aufruf von DLV mit eindeutigen Eingabemerkmalen) gemessen. Die Laufzeit der SQL-Abfragen wird nicht weiter betrachtet. Es wird angenommen, dass diese im Verhältnis zur Laufzeit aller Durchläufe sehr klein ist. Zur Messung wird ein aktuelles System, das den Anforderungen entspricht, verwendet.

[11] In der Implementation von GG-Net wird dieser Wert aus einem Userinterface ausgelesen.

4.4.4 Maximale Anzahl der Durchläufe

Um die maximale Anzahl der Durchläufe R_{max} zu ermitteln, wird die Mächtigkeit jeder Menge der Eingabemerkmale benötigt. Diese Mächtigkeiten sind in Tabelle 2 dargestellt.

Name	Individuen, Wertebereich, Details	Mächtigkeit
brand	Acer, Packard Bell, eMachines, Gateway	4
productGroup	Notebook, Desktop, Server, Monitor, Tablet, Tv, Netbook, Misc	8
name	String (max Length = 255, UTF-8)	$255 * 2^{20}$
refurbishId	Integer (DLV maxint)	10^6
mfgDelta	Integer (DLV maxint)	10^6
lastPrice	Integer (DLV maxint)	10^6
costPrice	Integer (DLV maxint)	10^6
cpuManufacturer	Intel, AMD, nVidia, Null	4
cpuModel	String (max Length = 255, UTF-8)	$255 * 2^{20}$
odd	DVD, Bluray-Combo, Bluray-Writer, Null	4
ram	Integer (DLV maxint)	10^6
hdd	Integer (DLV maxint)	10^6
os	Linux, Android, Windows ...	21
displaySize	Integer (DLV maxint)	10^6
displayResolution	String (max Length = 255, UTF-8)	$255 * 2^{20}$
Summe		$R_{max} \approx 2^{236}$

Tabelle 2: Mächtigkeit der Parametermengen

Die maximale Anzahl der Durchläufe hat also den Wert $R_{max} \approx 2^{236}$.

4.4.5 Reduzierte Anzahl der Durchläufe

Um die Anzahl der Durchläufe zu reduzieren, werden im ersten Schritt die einzelnen Eingabemerkmale durch Bedingungen aus der Realität eingeschränkt. Einige Beispiele hierfür sind:

- name und refurbishId sind irrelevant
- lastPrice und costPrice sind irrelevant
- mfgDelta: 1 Element/Woche, Bereich -3 Jahre bis heute : $10^6 \overset{red}{\rightarrow} 3 * 52 = 156$
- cpuManufacturer, cpuModel, odd, ram, hdd, os, displaySize, displayResolution sind alles Mengen, die bekannt sind.

Die gesamte Gegenüberstellung der maximalen und reduzierten Mächtigkeiten ist in Tabelle 3 aufgeführt.

Name	Mächtigkeit	red. Mächtigkeit
brand	4	4
productGroup	8	8
name	$255 * 2^{20}$	1
refurbishId	10^6	1
mfgDelta	10^6	156
lastPrice	10^6	1
costPrice	10^6	1
cpuManufacturer	4	4
cpuModel	$255 * 2^{20}$	221
odd	4	4
ram	10^6	8
hdd	10^6	12
os	21	5
displaySize	10^6	24
displayResolution	$255 * 2^{20}$	15
Summe	$R_{max} \approx 2^{236}$	$R_{red} \approx 2^{42}$

Tabelle 3: Mächtigkeit der reduzierten Parametermengen

In einem zweiten Schritt werden kombinatorische Einschränkungen betrachtet. Es gibt viele Kombinationen, die in der Realität nicht existieren:

- Ein Monitor hat keine Hdd, Ram, Cpu und OS Merkmale.
- Ein Desktop hat kein Display.
- Der Brand Gateway hat keine Monitore, Tablets oder Tvs.
- Notebooks haben nur 5 definierte Displaygrößen.
- Jede Displaygröße eines Notebooks hat nur 3 mögliche Auflösungen.

Durch den zweiten Schritt wurde die Anzahl von Durchläufen auf $R_{red} \approx 2^{34}$ reduziert. Im Zuge der Reduktion wurde auch der aktuelle Datenbestand untersucht. Aktuell existieren 14834 Artikelbeschreibungen im System. In diesem Fall sind nur noch `mfgDelta` veränderlich. Sollte man also die Verifikation nur auf den vorhandenen Artikelbestand beschränken, ist $R_{min} \approx 2^{21}$.

4.4.6 Schätzung und Messung der Laufzeit

Um die Zeiten eines einzelnen Durchlaufes zu bestimmen, wurde ein mehrfaches Aufrufen simuliert und die Zeit gemittelt. Die Messergebnisse sind in Tabelle 4 dargestellt. Hierbei ist zu erwähnen, dass die Laufzeit für das Speichern eines Datensatzes errechnet wurde, da das Speichern über einen Stapelvorgang von je 1000 Datensätzen erfolgt.

Schritt	min. Laufzeit	max. Laufzeit	avg. Laufzeit
eindeutige Eingabemerkmale erzeugen	0,11 ms	0,37 ms	0,25 ms
Aufruf der Preis-Maschine	0,8 ms	12 ms	2,5 ms
Datensatz speichern	-	-	2 ms
Summe			$L_1 \approx 5$ ms

Tabelle 4: Ergebnisse der einzelnen Laufzeitmessungen

Nachdem die Laufzeit eines einzelnen Durchlaufes ermittelt ist, wird diese nun in Kombination von R_{red} betrachtet. Des weiteren werden Vorgaben eines parallelen Systems wie in Abschnitt 4.4 in die Schätzung mit aufgenommen. Hierbei wird ein Verlust von 10% zu einer idealen Parallelisierung aus Erfahrung angenommen. Die geschätzte Laufzeit der reduzierten Anzahl der Durchläufe mit 16 und 1024 Kernen wird in Gleichung 6 gezeigt.

$$\begin{aligned} L_{linear,red} = R_{red} * L_1 = 2^{34} * 5ms = 8,59 * 10^{10} ms = 23861,92h \\ c16 = 16 * 0,9 = 14,4 \\ L_{c16,red} = \frac{L_{linear,red}}{c16} = \frac{23861,92h}{14,4} = 1657h = 69,04d \\ c1024 = 1024 * 0,9 = 921,6 \\ L_{c1024,red} = \frac{L_{linear,red}}{c1024} = \frac{23861,92h}{921,6} = 25,89h = 1,07d \end{aligned} \tag{6}$$

Dabei wird eine lineare Laufzeit von $\approx$ 23862 Stunden geschätzt. Aufgeteilt auf 16 Kerne würde dies $\approx$ 69 Tagen entsprechen. Bei der maximalen Größe von 1024 Kernen wird das Ziel von $\approx$ 1 Tag erreicht. Zusätzlich ist in Gleichung 7 die geschätzte Laufzeit der minimalen Durchläufe dargestellt.

$$\begin{aligned} L_{linear,min} = R_{min} * L_1 = 2^{21} * 5ms = 1 * 10^7 ms = 3,25h \\ L_{c16,min} = \frac{L_{linear,min}}{c16} = \frac{3,25h}{14,4} = 0,23h \ll 1d \end{aligned} \tag{7}$$

Hierbei liegt bereits die lineare Laufzeit bei 3,25 Stunden. Diese auf 16 Kerne parallelisiert, führt zu einer Laufzeitschätzung von unter einer Stunde. Die geschätzte Laufzeit

bestätigt die Vermutung, dass die in Abschnitt 4.4 auf Seite 27 gestellten Anforderungen, erfüllbar sind.

Um die tatsächliche Laufzeit zu bestimmen, wurde nur R_{min} untersucht. Dazu wurde die von GG-Net zur Verfügung gestellte Hardware verwendet. Die Ergebnisse dieser Messung sind in Tabelle 5 aufgezeichnet.

Messung	Laufzeit
1	0,22 h
2	0,27 h
3	0,19 h
4	0,22 h
5	0,25 h

Tabelle 5: Gemessene Laufzeit von R_{min}

Hiermit bestätigt sich die Laufzeitschätzung zumindest für R_{min}. Damit ist gezeigt, dass naives Ausprobieren eine geeignete Verifikationstechnik ist, die im Rahmen der PM von GG-Net praktisch anwendbar ist.

4.5 Einschätzung der ASP Verifikation

In diesem Kapitel wurde Verifikation und die Klassifikation von Verifikationstechniken vorgestellt. Im Rahmen von ASP wurde ein Verfahren entworfen und beschrieben, das naives Ausprobieren als Verifikationstechnik realisiert. Dieses Verfahren wurde klassifiziert und in einem Experiment auf praktische Anwendbarkeit untersucht. Dazu wurde die zuvor vorgestellte Preis-Maschine als Versuchsaufbau verwendet. Es wurde die Laufzeit analysiert. Als Ergebnis kann dieser Versuch als erfolgreich betrachtet werden, da die Technik in einem gegebenen Zeitfenster terminiert.

Als Erkenntnis kann festgehalten werden, dass naives Ausprobieren unter geeigneten Bedingungen eine zielführende Verifikationstechnik ist, auch wenn nicht sehr elegant. Die Herausforderung liegt in der geeigneten Reduktion der Eingabemerkmale. Es ist auf jeden Fall bei der Auswahl von Verifikationstechniken zu empfehlen, genau zu recherchieren, ob sich die Eingabemerkmale auf eine geeignete Größe reduzieren lassen. Abschließend ist hervorzuheben, dass die Technik auf Grund ihres einfachen Charakters geringe Risiken einer fehlerhaften Implementation mit sich bringt.

5 Abstrakte Interpretation und Datenflussanalyse

Abstrakte Interpretation ist eine allgemeine Theorie für die Approximation der Semantik von diskreten dynamischen Systemen. Sie wurde ursprünglich Ende der 70iger Jahre von Patrick und Radhia Cousot, vorgestellt in den Artikeln [3] und [4], als eine Methode zur statischen Programmanalyse entwickelt. Ziel der abstrakten Interpretation ist es, Informationen über das Verhalten von Programmen zu bekommen, indem Teile von Programmen abstrahiert werden. Dabei wird eine abstrakte Semantik gebildet, die eine Annäherung an die konkrete Semantik ist, wobei exakte (konkrete) Eigenschaften durch approximierten Eigenschaften ersetzt werden. Abstrakte Interpretation ist geeignet, die Hierarchien von Semantiken zu untersuchen und den Datenfluss oder Typensysteme zu rekonstruieren. Sie kann aber auch verwendet werden, um die Korrektheit eines Analysealgorithmus zu beweisen.

Bei der Datenflussanalyse handelt es sich um eine statische Quelltextanalyse. Diese verwendet ein Programm als Untersuchungsgegenstand, um festzustellen, zwischen welchen Teilen des Programms Daten weitergegeben werden und ob Abhängigkeiten daraus resultieren. Datenflussanalysen arbeiten meist auf dem Kontrollflussgraphen, einem gerichteten Graphen des zu analysierenden Programms. Die Knoten des Graphen sind Blöcke mit einer oder mehreren Anweisungen als Inhalt. Datenflussanalysen untersuchen, wie sich Daten durch einen Knoten verändern. Betrachtet man folgenden den Quelltext in Java:

```
z = 7;
```

Dieser Ausdruckt wird einem Knoten im Kontrollflussgraphen zugeordnet. An diesem Punkt findet eine Veränderung im Datenbestand des Programmes statt. Die Variable z hat nach diesem Knoten den Wert 7, unabhängig davon, wie ihr Zustand zuvor war. Durch den Graphen wird weiterhin der Zusammenfluss von Daten untersucht. Dieses geschieht, wenn mehrere Kanten in einem Knoten enden. Die Behandlung dieses Falls ist immer abhängig von der jeweiligen Analyse.

In diesem Kapitel wird Datenflussanalyse und Abstrakte Interpretation untersucht. Dazu werden in Abschnitt 5.1 die Grundlagen der Datenflussanalyse vorgestellt und erörtert. In Abschnitt 5.2 wird die klassische aktive Variablenanalyse dann als Vertreter der Datenflussanalyse vorgestellt und an einem Beispiel präsentiert. Aufbauend darauf wird in Abschnitt 5.3 die aktive Merkmalanalyse für ASP entwickelt. Dieser liegt die Motivation zu Grunde, die Lösung in Kapitel 4 zu optimieren, indem die Eingabemerkmale reduziert werden. Diese Analyse wird dann beispielhaft an einem Auszug der PM in Abschitt 5.4 verwendet. Abschließend findet in Abschnitt 5.5 eine Einschätzung der Analyse statt.

5.1 Grundlagen der klassischen Bit Vektor Datenflussanalyse

Die klassische Bit Vektor Datenflussanalyse wird sehr detailliert im Buch [11] gezeigt, woraus dieser und der folgende Abschnitt gespeist werden. Der Name hat seinen Ursprung in der Tatsache, dass nicht nur der Datenfluss in Bit Vektoren dargestellt werden kann, sondern auch die Operationen auf dem Datenfluss lassen sich auf Bit Vektor Operationen abbilden. Die Datenflussanalyse eines Programms erfolgt, für eine gegebene Programmkomponente wie einen Ausdruck, in zwei Schritten:

1. Feststellen der Wirkung einzelner Anweisungen auf den Ausdruck,
2. Feststellen der Abhängigkeiten und Effekte dieser Wirkung auf andere Anweisungen im Programm.

Schritt Eins ist die *lokale* Datenflussanalyse und wird nur einmal für einen Knoten durchgeführt. Schritt Zwei repräsentiert die *globale* Datenflussanalyse und erfordert möglicherweise wiederholte Durchläufe über die Knoten des Kontrollflussgraphen. Um die Abhängigkeiten und Effekte auf andere Anweisungen im Programm festzustellen, müssen Datenflussinformationen aus einem Knoten zu einem anderen weitergegeben werden. Dies erfolgt in oder entgegen der Richtung des Kontrollflusses.

Die globalen Datenflussinformationen werden mit den Ein-und Ausgangspunkten eines Knotens assoziiert, genannt $Entry(n)$ und $Exit(n)$. Diese repräsentieren die möglichen Zustände des Programms kurz vor der Ausführung der ersten Anweisung und kurz nach der Ausführung der letzten Anweisung in dem Knoten n. Assoziierte Datenflussinformationen werden In_n und Out_n genannt, während die lokalen Datenflussinformationen Gen_n und $Kill_n$ genannt werden. Während Gen_n die Informationen enthält, die in einem Knoten generiert werden, beschreibt $Kill_n$ Informationen, die ungültig oder unwirksam werden.

Die Beziehungen zwischen dem lokalen und globalen Datenfluss für einen Knoten (Gen_n, $Kill_n$, In_n und Out_n) und zwischen globalen Datenflussinformationen über verschiedene Knoten werden durch ein System von linearen Gleichungen realisiert. Diese werden Datenflussgleichungen genannt.

Kanten in Kontrollflussgraphen stellen eine Vorgänger-Nachfolger-Relation dar. Bei einer möglichen Kante $n_1 \rightarrow n_2$ ist n_1 ein Vorgänger von n_2 und n_2 ist ein Nachfolger von n_1. Vorgänger und Nachfolger eines Knotens n werden durch $pred(n)$ und $succ(n)$ gekennzeichnet. Es wird davon ausgegangen, dass der Kontrollflussgraph zwei spezielle Knoten hat, den Start-und den Endknoten. Der Startknoten verfügt über keine Vorgänger, während der Endknoten keine Nachfolger hat. Weiterhin wird angenommen, dass es für jeden Knoten n mindestens einen Weg vom Start- und zum Endknoten gibt.

5.2 Klassische aktive Variablenanalyse (Live Variables Analysis)

Die aktive Variablenanalyse wurde erstmals in [10] vorgestellt und untersucht im Wesentlichen, ob eine Variable in Zukunft verwendet wird.

Definition 9 *Eine Variable* $\boldsymbol{x}$ *ist an einem Programmpunkt* $\boldsymbol{u}$ *aktiv, wenn ein Weg von* $\boldsymbol{u}$ *zum Punkt* ***End*** *eine Verwendung von* $\boldsymbol{x}$ *enthält, aber keine Definition.*

Die Datenflussgleichungen für aktive Variablenanalyse sind:

$$In_n = (Out_n - Kill_n) \cup Gen_n \tag{8}$$

$$Out_n = \begin{cases} Bl & n \text{ ist Endknoten} \\ \bigcup\limits_{s \in succ(n)} In_s & \text{sonst} \end{cases} \tag{9}$$

wobei Gen_n, $Kill_n$, In_n, Out_n und Bl Mengen von Variablen sind.

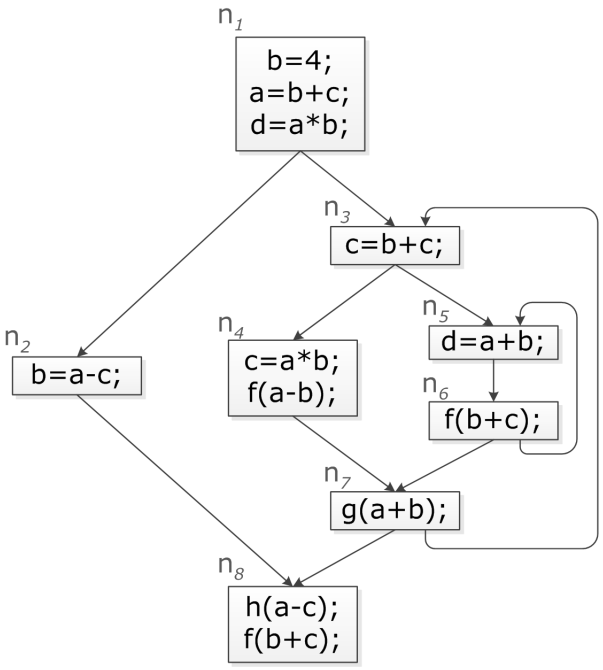

Abbildung 2: Beispiel eines Kontrollflussgraphen

Ob Variablen am Punkt $Exit(End)$ aktiv sind, wird durch den Inhalt der Menge Bl dargestellt. Lokale Variablen sind im Punkt $Exit(End)$ nicht aktiv, während Rückgabewerte, globale Variablen und referenzierte Variablen in Abhängigkeit des aufrufenden Kontextes aktiv sein können. Da hier nur die Analyse für lokale Variablen vorgestellt

wird, ist Bl somit leer. Die Menge Gen_n enthält die Variablen, welche durch den Knoten n aktiv werden, während $Kill_n$ Variablen enthält, die durch diesen Knoten inaktiv werden. Dies sind die Variablen, die auf der linken Seite einer Zuweisung im Knoten n auftreten.

Eine Beispielanwendung von aktiver Variablenanalyse wird in Abbildung 2 gezeigt. Da die Analyse eine entgegen der Richtung des Kontrollflussgraphen ist, wird dies in der tabellarischen Darstellung berücksichtigt. Es wird die leere Menge zur Initialisierung verwendet. Weiterhin wird, wie bereits erwähnt, eine lokale Analyse durchgeführt und daher Bl als leer definiert. Es ist zu erkennen, dass die errechneten Werte des ersten Durchlaufs identisch mit den Werten des zweiten Durchlaufs sind. Dies weist auf Konvergenz hin. Die Gen_n und $Kill_n$ Mengen der Knoten, ebenso die Out_n und In_n Mengen mit zwei Durchläufen, sind in Abbildung 3 zu sehen.

Block	Local Information		Global Information			
			Iteration # 1		Iteration # 2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
n_8	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\{a,b,c\}$	$\emptyset$	$\{a,b,c\}$
n_7	$\{a,b\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_6	$\{b,c\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_5	$\{a,b\}$	$\{d\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_4	$\{a,b\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b\}$	$\{a,b,c\}$	$\{a,b\}$
n_3	$\{b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_2	$\{a,c\}$	$\{b\}$	$\{a,b,c\}$	$\{a,c\}$	$\{a,b,c\}$	$\{a,c\}$
n_1	$\{c\}$	$\{a,b,d\}$	$\{a,b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{c\}$

Abbildung 3: Aktive Variablenanalyse am Beispiel

Für eine ausgewählte Variable x entdeckt die aktive Variablenanalyse eine Menge an *aktiven Wegen* im Graph. Jeder dieser Wege ist eine Folge von Knoten $(b_1, b_2, \cdots, b_k)$, die einen möglichen Ausführungspfad beschreiben, beginnend in b_1, so dass:

- b_k enthält eine nach oben sichtbare Nutzung von x, und
- b_1 ist entweder der Startknoten oder enthält eine Zuweisung an x, und
- kein anderer Weg enthält eine Zuweisung an x.

Einige aktive Wege der Variablen c im Beispiel sind : (n_4, n_7, n_8), $(n_3, n_5, n_6, n_7, n_8)$, $(n_3, n_5, n_6, n_5, n_6, n_7, n_8)$, und (n_1, n_2, n_8).

5.3 Angepasste aktive Merkmalanalyse für ASP

Im Gegensatz zur klassischen aktiven Variablenanalyse die aktive Wege für eine Variable findet, untersucht die aktive Merkmalanalyse für Answer Set Programming, welche expliziten Merkmale oder Teilmengen überhaupt Auswirkungen auf das logische Programm haben.

Ein Merkmal im Kontext logischer Programme ist entweder eine Konstante, eine Zahl oder ein Funktionsterm. Funktionsterme haben hierbei einen besonderen Charakter. Während der äußere Teil eines Funktionsterms, also der alphanumerische Teil, der sich außerhalb der äußersten Klammern befindet, eine kategorisierende Eigenschaft hat, repräsentieren alle inneren Elemente die individuellen Eigenschaften der Kategorie. Es ist auch möglich, diesen Zusammenhang als Abbildung von einer Menge von Mengen von Elementen zu Merkmalen zu interpretieren.

Definition 10 *Die aktive Merkmalanalyse für Answer Set Programming benötigt eine definierte Menge von beschränkten Mengen von Eingabe- und Ausgabemerkmalen.*

Die Notwendigkeit der Definition zeigt sich sofort unter Betrachtung des Verhaltens eines Answer Set Programms. Da alle Fakten Bestandteil der Modelle sind, also Eingabemerkmale automatisch Ausgabemerkmale werden, ist in diesem Fall eine Analyse überflüssig. Sind Fakten nicht Bestandteil des Modells, möglich durch Verwendgung des DLV Parameters `--nofacts`, könnte eine Analyse eines ungünstigen logischen Programms zu unendlichen Mengen von Eingabemerkmalen führen. Als Beispiel sei die Regel `f(X) :- f(f(X))` gegeben. Diese würde bei einer entgegengesetzten Analyse immer neue Eingabemerkmale erzeugen. Weiterhin wäre eine aussagekräftige und nützliche Analyse ohne Beschränkung schwierig, da jede Regel mögliche Elemente zur Menge der Eingabe- und Ausgabemerkmale hinzufügt. Im folgenden Abschnitt wird dies noch deutlich. Es sei noch einmal darauf hin gewiesen, dass die aktive Merkmalanalyse für ASP mit der Motivation entwickelt wurde, die Menge der Eingabemerkmale für die in Kapitel 4 auf Seite 22 vorgestellte Lösung weiter zu reduzieren.

5.3.1 Konstruktion des Kontrollflussgraphen

Um eine statische Code-Analyse durchführen zu können, wird ein Kontrollflussgraph für das logische Programm benötigt. Die Konstruktion des Graphen gestaltet sich etwas komplizierter als die Konstruktion für ein prozedurales Programm, da die Regeln eines Answer Set Programms keiner Reihenfolge unterliegen. Im Folgenden wird ein Algorithmus vorgestellt, der einen solchen Graphen erstellt.

1. Für jede Regel des logischen Programms wird ein Knoten erstellt und diese mit n_1 *bis* n_i benannt.
2. Es wird ein leerer Knoten[12] n_{start} hinzugefügt. Dies ist der Startknoten.

[12]Ein Knoten ohne Regel als Inhalt

3. Es wird ein Knoten n_{end} hinzugefügt, der die möglichen Mengen der Ausgabemerkmale enthält. Dies ist der Endknoten. Hierbei wird $\top$ als Platzhalter in Funktionstermen verwendet werden[13]..
4. Für jeden Knoten, der ein Eingabemerkmal im Rumpf verwendet, füge diesem Knoten eine Kante von n_{start} hinzu.
5. Für jeden verbundenen Knoten n_i, für jedes Merkmal p_{ij} im Kopf der Regel des Knoten n_i, für alle Knoten n_k, wenn n_k das Merkmal p_{ij} im Rumpf verwendet, verbinde n_i mit n_k.
6. Wiederhole den vorangegangenen Punkt so lange, bis sich der Graph nicht mehr ändert.
7. Für jeden verbundenen Knoten, der ein Ausgabemerkmal im Kopf hat, verbinde diesen mit dem Endknoten.
8. Entferne jeden Knoten und alle Kanten die zu diesem führen, der keinen Weg zum Endknoten hat.

Es sei erwähnt, dass unter der Verwendung eines Merkmals bei Funktionstermen nicht nur der identische Wert gemeint ist, sondern auch Teile, die durch Variablen repräsentiert werden. Dies wird in Quelltext 29 einmal im Detail gezeigt.

```
a(b(1)).

q :- a(b(1)).
p :- a(_).
r(X) :- a(X).
```

Quelltext 29: Verwendung von Merkmalen

In Zeile 1 wird ein Fakt definiert, der das zu betrachtende Merkmal repräsentiert. In Zeile 3 wird dieses Merkmal direkt im Rumpf der Regel verwendet. Die Regeln in Zeile 4 und 5 enthalten auch Verwendungen des Merkmals, auch wenn es hier nicht Konstant ist, sondern mit einem variablen Anteil.

[13]Natürlich macht ein reines $\top$ keinen Sinn, da es Definition 10 widersprechen würde.

Um den Algorithmus zu präsentieren, wird Quelltext 30 verwendet. Dabei sind folgende Merkmale für Ein-und Ausgabe definiert:

- Eingabemerkmale:
  - $c(A) \mid A \in \{a, b, c, d\}$
  - $n(B) \mid B \in \mathbb{N}^{\#maxint}$
  - $w(C) \mid C \in [1; 3]$
- Ausgabemerkmal:
  - $z(Q) \mid Q \in \mathbb{N}^{\#maxint}$

```
1  p :- c(a).
   q :- c(b).
3  r(X) :- c(X),n(Y), Y > 100.
   t(10) :- n(Y), Y < 5.
5  s(U) :- n(Y), U = Y + 2.
   p :- r(X), X = a.
7  z(1) :- p.
   z(2) :- q.
9  z(3) :- t(Y), Y = 10.
   z(4) :- s(A), A > 2000.
11 u(X) :- r(X).
   r(X) :- u(X).
13 z(Z) :- w(Z).
```

Quelltext 30: Beispiel für Graphenkonstruktion

Nach Anwendung des Algorithmus erhalten wir einen Graphen, der in Abbildung 4 zu sehen ist. Zur Übersichtlichkeit wurden Knoten, die im letzten Schritt entfernt würden, grau markiert.

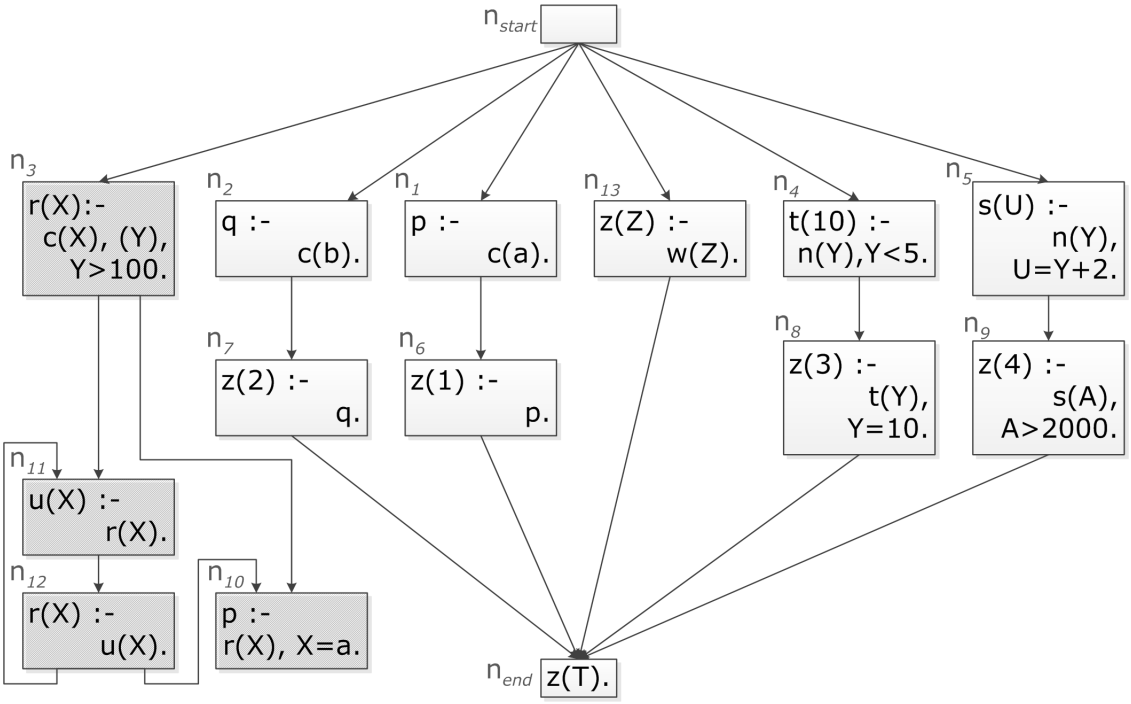

Abbildung 4: Beispielgraph

5.3.2 Die Analyse

Die aktive Merkmalanalyse für ASP untersucht, welche Individuen der Mengen der Eingabemerkmale tatsächlich Auswirkungen auf Ausgabemerkmale haben. Dabei ist festzuhalten, dass die Analyse in der im folgenden vorgestellten Form die schwache Negation nicht vollständig erfasst.

Es wird die Menge $Trans_n$ eingeführt.

Definition 11 *Eine Menge* $\boldsymbol{Trans_n}$ *des Knoten* $\boldsymbol{n}$ *enthält für jede Variable, die im Kopf der Regel des Knoten auftritt, die inverse Konstruktion.*

Die Menge $Trans_n$ wird, wie Gen_n und $Kill_n$, einmal für jeden Knoten erzeugt. Beispiele für $Trans_n$ sind in Tabelle 6 gezeigt.

Regel	$Trans_n$
f(X) :- a(X).	$f(X) \rightarrow a(X)$
f(X,Y) :- a(X),b(Y).	$f(X,\_) \rightarrow a(X), f(\_,Y) \rightarrow b(Y)$
f(X) :- a(Y),X = Y + 100.	$f(X) \rightarrow a(X - 100)$

Tabelle 6: Beispiele für $Trans_n$

Weiterhin wird eine Relation $\circ$ eingeführt, die zwei Parameter benötigt, einmal $Trans_n$ und eine Menge an Merkmalen. Diese Menge wird später Out_n sein.

Definition 12 *Die Relation $\circ$, angewandt auf eine Menge $\boldsymbol{Trans_n}$ und eine Menge an Merkmalen $\boldsymbol{V_1}$, erzeugt eine neue Mengen an Merkmalen $\boldsymbol{V_2}$, die alle transformierten Elemente, die eine Entsprechung in $\boldsymbol{Trans_n}$ haben, enthält.*

Beispiele für die Anwendung von $\circ$ sind in Tabelle 7 gezeigt.

V_1	$Trans_n$	V_2
$f(\{1,2\})$	$f(X) \to a(X)$	$a(\{1,2\})$
$f(\{1,2\},\{5 \leq i \leq 10\})$	$f(X,\_) \to a(X)$, $f(\_,Y) \to b(Y)$	$a(\{1,2\})$; $b(\{5 \leq i \leq 10\})$

Tabelle 7: Beispiele für die Anwendung von $\circ$

Nach der Definition von $Trans_n$ und $\circ$ ist noch zu erwähnen, dass die Menge $Kill_n$ keine Anwendung findet.

Definition 13 *Ein Merkmal $\boldsymbol{m}$ ist an einem Programmpunkt $\boldsymbol{u}$ aktiv, wenn es Auswirkungen auf mindestens ein Ausgabemerkmal hat.*

Die Datenflussgleichungen für die Analyse sind:

$$In_n = (Out_n \circ Trans_n) \cup Gen_n \tag{10}$$

$$Out_n = \begin{cases} Bl & n \text{ ist Endknoten} \\ \emptyset & succ(n) = \emptyset \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \tag{11}$$

wobei Gen_n, In_n, Out_n und Bl Mengen von Merkmalen sind und $Trans_n$ eine Menge von Abbildungen ist. Auch hier ist die Eigenschaft eines Merkmals am Punkt $Exit(End)$ abhängig vom Inhalt der Menge Bl. Die Menge Gen_n enthält die Merkmale, welche durch den Knoten n aktiv werden. Ein Sonderfall ist hier eine anonyme Variable als Teil eines Funktionsterms. Diese wird mit $\perp$ erfasst, da ein willkürliches Individuum des Merkmals diesen Teil der Regel wahr werden lässt.

Es wird die leere Menge zur Initialisierung verwendet, während Bl mit der Menge der Mengen aller Ausgabemerkmale initialisiert wird. In diesem Fall wird das Symbol $\top$ für alle Elemente einer Kategorie verwendet. Als Beispiel sei hier die Kategorie a mit den Elementen $1, 2, 3$ genannt. Diese entspricht den Merkmalen $a(1), a(2), a(3)$ und würde für alle Elemente kurz $a(\top)$ geschrieben.

Für eine Beispielanwendung von aktiver Merkmalanalyse wird der Graph aus Abbildung 4 von Seite 43 verwendet. Da auch diese Analyse eine entgegen der Richtung des Kontrollflussgraphen ist, wird dies in der tabellarischen Darstellung berücksichtigt.

Die Gen_n und $Trans_n$ Mengen der Knoten, sowie die Out_n und In_n Mengen mit zwei Durchläufen, sind in Tabelle 8 und Tabelle 9 zu sehen. Auch hier ist zu erkennen, dass die errechneten Werte des ersten Durchlaufs identisch sind mit den Werten des zweiten Durchlaufs, was wiederum auf Konvergenz hinweist.

Knoten	Lokale Informationen		Globale Informationen	
			Durchlauf 1	
	Gen_n	$Trans_n$	In_n	Out_n
n_{end}	-	-	$z(\top)$	$z(\top)$
n_{13}	-	$z(X) \to w(X)$	$w(\top)$	$z(\top)$
n_9	s(i $>$ 2000)	-	$s(\{i > 2000\})$	$z(\top)$
n_8	t(10)	-	$t(10)$	$z(\top)$
n_7	q	-	q	$z(\top)$
n_6	p	-	p	$z(\top)$
n_5	-	$s(X) \to n(X+2)$	$n(\{i > 1998\})$	$s(\{i > 2000\})$
n_4	n(i $<$ 5)	-	$n(\{i < 5\})$	$t(10)$
n_2	c(b)	-	$c(\{b\})$	p
n_1	c(a)	-	$c(\{a\})$	q
n_{start}	-	-	$w(\top)$, $c(\{a,b\})$, $n(\{i > 1998\})$	

Tabelle 8: Aktive Merkmalanalyse

Knoten	Globale Informationen	
	Durchlauf 2	
	In_n	Out_n
n_{end}	$z(\top)$	$z(\top)$
n_{13}	$w(\top)$	$z(\top)$
n_9	$s(\{i > 2000\})$	$z(\top)$
n_8	$t(10)$	$z(\top)$
n_7	q	$z(\top)$
n_6	p	$z(\top)$
n_5	$n(\{i > 1998\})$	$s(\{i > 2000\})$
n_4	$n(\{i < 5\})$	$t(10)$
n_2	$c(\{b\})$	p
n_1	$c(\{a\})$	q
n_{start}	$w(\top)$, $c(\{a,b\})$, $n(\{i > 1998\})$	

Tabelle 9: Aktive Merkmalanalyse Durchlauf 2

Die aktive Merkmalanalyse für ASP entdeckt alle Eingabemerkmale, die tatsächlich

Auswirkungen auf die Ausgabemerkmale haben. Im gegeben Beispiel wurde festgestellt, dass alle Merkmale $w(\top)$ benötigt werden. Aus den Merkmalen der Kategorie c werden nur die Elemente $c(a)$ und $c(b)$ benötigt. Die Elemente $c(c)$ und $c(d)$ haben keine Auswirkung auf die Ausgabemerkmale. Die Merkmale der Kategorie n wurden von $n(0) \ldots n(\#maxint)$ auf $n(0) \ldots n(1997)$ eingeschränkt.

5.4 Anwendung an der Preis-Maschine

Im Folgenden wird die aktive Merkmalanalyse für ASP an der PM angewandt. Dazu wird ein gekürzter Auszug aus Quelltext 25 von Seite 20 verwendet. Dieser ist in Quelltext 31 zu sehen. Hier sei noch erwähnt, dass Zeile 16 der Einfachheit halber gekürzt wurde.

```
1  w_display_size(30) :- display(X,_), X <= 12.
   w_display_size(45) :- display(X,_), X > 12, X <= 14.
3  w_display_size(5)  :- display(X,_), X > 14, X <= 16.
   w_display_size(15) :- display(X,_), X > 16, X <= 17.
5  w_display_size(30) :- display(X,_), X > 17.

7  w_display_res_pre(30) :- display(_,wuxga).
   w_display_res_pre(20) :- display(_,full_hd).
9  w_display_res_pre(10) :- display(_,hd).
   w_display_res_pre(10) :- display(_,wsxga).
11
   w_display_res_set :- w_display_res_pre(_).
13 w_display_res(X)  :- w_display_res_pre(X).
   w_display_res(5)  :- not w_display_res_set.
15
   price(P) :- w_display_size(C), w_display_res(D), P = C + D.
```

Quelltext 31: Auszug Preis-Maschine für Merkmal Display

Nach Anwendung des Algorithmus zur Konstruktion eines Kontrollflussgraphen erhalten wir Abbildung 5.

Da der Graph keinerlei Zyklen enthält, ist zu erwarten, dass bereits im ersten Durchlauf das Endergebnis erreicht wird[14]. Die Anwendung der aktiven Merkmalanalyse auf diesen Graphen ist in den Tabellen 10 und 11 gezeigt.

[14]Nur eine unglückliche Ordnung der Knoten könnte hier zu einem anderen Effekt führen.

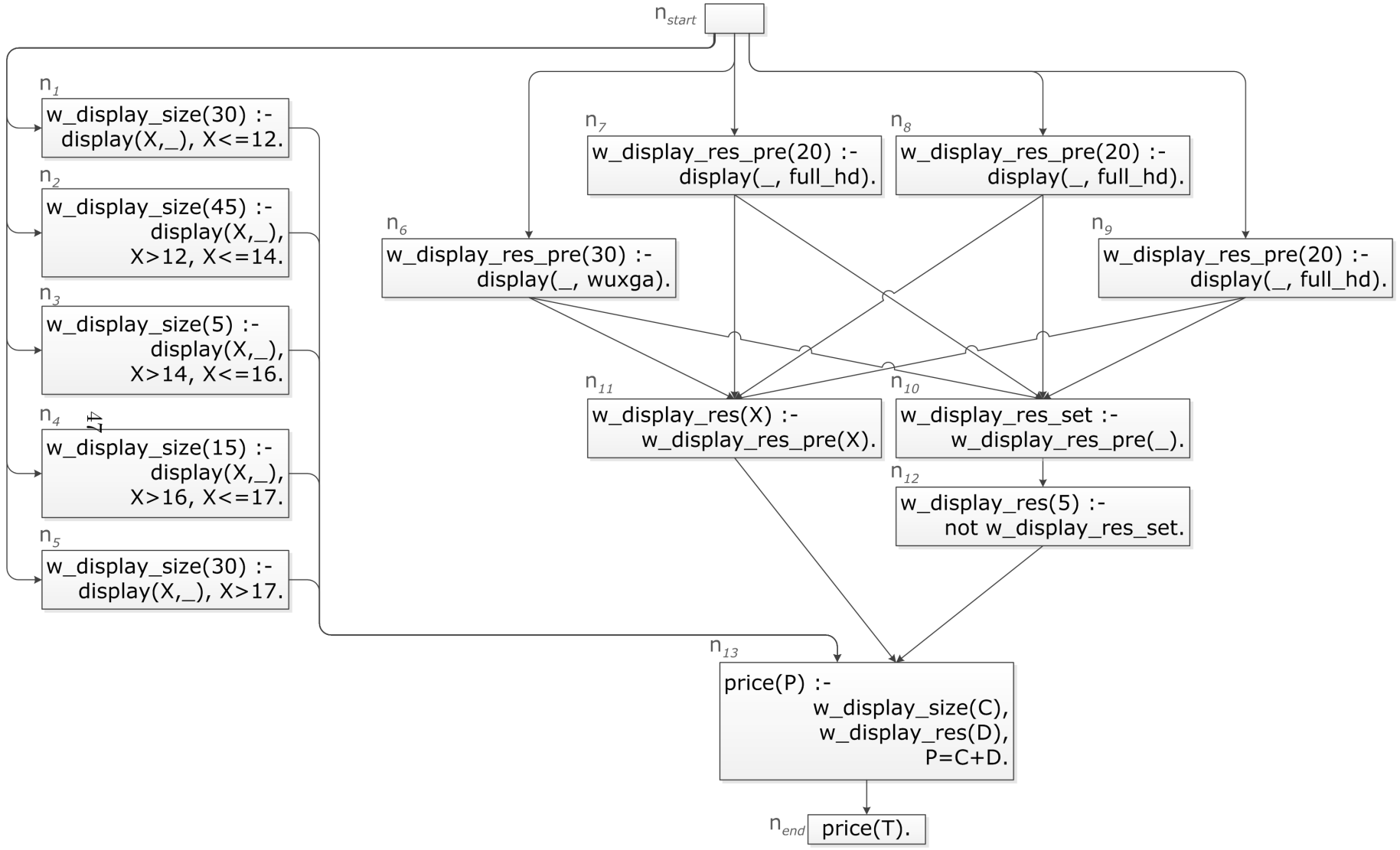

Abbildung 5: Graph für Merkmal-Display

Knoten	Lokale Informationen	
	Gen_n	$Trans_n$
n_{end}	-	-
n_{13}	-	$price(X) \rightarrow w\_display\_size(X)$, $price(X) \rightarrow w\_display\_res(X)$
n_{12}	$w\_display\_res\_set$	-
n_{11}	-	$w\_display\_res(X) \rightarrow w\_display\_res\_pre(X)$
n_{10}	$w\_display\_res\_pre(\bot)$	-
n_9	$display(\bot, wsxga)$	-
n_8	$display(\bot, hd)$	-
n_7	$display(\bot, full\_hd)$	-
n_6	$display(\bot, wuxga)$	-
n_5	$display(i > 17, \bot)$	-
n_4	$display(i = 17, \bot)$	-
n_3	$display(16 \geq i > 14, \bot)$	-
n_2	$display(14 \geq i > 12, \bot)$	-
n_1	$display(i \leq 12, \bot)$	-
n_{start}	-	-

Tabelle 10: Aktive Merkmalanalyse für Display

Knoten	Globale Informationen	
	Durchlauf 1	
	In_n	Out_n
n_{end}	$price(\top)$	$price(\top)$
n_{13}	$w\_display\_size(\top)$, $w\_display\_res(\top)$	$price(\top)$
n_{12}	$w\_display\_res\_set$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_{11}	$w\_display\_res\_pre(\top)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_{10}	$w\_display\_res\_pre(\bot)$	$w\_display\_res\_set$
n_9	$display(\bot, wsxga)$	$w\_display\_res\_pre(\{\top, \bot\})$
n_8	$display(\bot, hd)$	$w\_display\_res\_pre(\{\top, \bot\})$
n_7	$display(\bot, full\_hd)$	$w\_display\_res\_pre(\{\top, \bot\})$
n_6	$display(\bot, wuxga)$	$w\_display\_res\_pre(\{\top, \bot\})$
n_5	$display(i > 17, \bot)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_4	$display(i = 17, \bot)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_3	$display(16 \geq i > 14, \bot)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_2	$display(14 \geq i > 12, \bot)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_1	$display(i \leq 12, \bot)$	$w\_display\_size(\top)$, $w\_display\_res(\top)$
n_{start}	$display(\bot, \{wsxga, hd, full\_hd, wuxga\})$, $display(\{i \leq 12, 14 \geq i > 12, 16 \geq i > 14, i = 17, i > 17\}, \bot)$	

Tabelle 11: Aktive Merkmalanalyse für Display, Durchlauf

In Kapitel 4 Tabelle 3 auf Seite 33 wurde bereits aufgezeigt, dass es sich bei dem Merkmal `displaySize`, die erste Komponente von `display(_,_)`, nicht um den gesamten Integerbereich, sondern um 24 individuelle Werte handelt. Durch Anwendung des vorgestellten Verfahrens lässt sich dieser Wert weiter auf 5 Teilmengen reduzieren, wobei jedes Element einer Teilmenge als Repräsentant verwendet werden kann und zum selben Ergebnis führt. Gleiches gilt für das Merkmal `displayResolution`, die zweite Komponente von `display(_,_)`. Auch hierbei handelte es sich nach der Untersuchung in Kapitel 4 um eine Menge von 15 Individuen. Nach der Analyse bleiben 4 Individuen übrig.

An der PM ist leider auch die in Abschnit 5.3.2 beschriebene Grenze der Analyse zu sehen. Nach manueller Inspektion des Quelltextes 31 ist in den Zeilen 12 bis 14 eine schwache Negation in indirekter Abhängigkeit eines Eingabemerkmals zu sehen. Daraus lässt sich schlussfolgern, für das Merkmal `displayResolution` sind nicht 4 sondern 5 Individuen notwendig. Das 5 Element kann willkürlich aus der Menge der übrigen 11 Element ausgewählt werden.

5.5 Einschätzung der aktiven Merkmalanalyse für ASP

In diesem Kapitel wurde Datenflussanalyse und Abstrakte Interpretation im Kontext von ASP untersucht. Dazu wurden die Grundlagen in Abschnitt 5.1 und Abschnitt 5.2 vorgestellt und erörtert. Basierend auf diesen wurde in Abschnitt 5.3 die aktive Merkmalanalyse für ASP entwickelt und diese abschließend an einem Auszug der PM in Abschnitt 5.4 verwendet.

Grundlegend wurde gezeigt, dass mit Hilfe von Abstrakter Interpretation die Menge von Eingabemerkmalen reduziert werden kann. Damit wurde eine Lösung gefunden, die der Motivation zu Grunde lag. Auch wenn die Anwendung an der PM nur beispielhaft war, lässt sich eine große Auswirkung erahnen.

Eine interessante weiterführende Untersuchung der Analyse wäre, zu prüfen, ob sich diese noch besser durch allgemeine Methoden der Datenflussanalyse[15] repräsentieren lässt. Diese Methoden werden sehr gut im Buch [11] ab Seite 100 beschrieben. Es ist die Vermutung des Autors, das sich $Trans_n$ und die Relation $\circ$ durch die komplexeren Datenflussgleichungen ersetzen lässt.

[15] General Data Flow Frameworks

6 Abduktion

Abduktion lässt sich allgemein folgendermaßen beschreiben: „Die überraschende Tatsache B wird beobachtet; aber wenn A wahr wäre, würde B eine Selbstverständlichkeit sein; folglich besteht Grund zur Vermutung, dass A wahr ist.“

Am Anfang steht keine bekannte Regel, sondern ein überraschendes Ereignis. Diese Ereignis lässt ernsthafte Zweifel an der Richtigkeit eigener Vorstellungen aufkommen. Dann kommt es im zweiten Schritt zu einer Als-ob-Annahme: Wenn es eine Regel oder eine Fakt A gäbe, dann hätte das überraschende Ereignis seinen Überraschungscharakter verloren. Entscheidend ist nun für die Bestimmung der Abduktion, dass nicht die *Beseitigung der Überraschung* das Wesentliche an ihr ist, sondern die Beseitigung der Überraschung durch *eine neue Regel oder einen neuen Fakt A*. Im zweiten Teil des abduktiven Prozesses wird also eine bislang noch nicht bekannte Regel entwickelt. Der dritte Schritt erbringt dann zweierlei: Zum einen, dass das überraschende Ereignis ein Fall der konstruierten Regel ist, zum anderen, dass diese Regel eine gewisse Überzeugungskraft besitzt.

Abduktion lässt sich auch aus einem anderen Blickwinkel betrachten. Das Ereignis am Anfang ist nicht überraschend, sondern gewünscht, sprich ein überraschendes Ereignis wird herbeigesehnt. Somit stellt sich dann die Frage, welche Regel gebildet werden muss, um dieses Ereignis zu erklären. Dieser Ansatz führt, motiviert durch die PM, zu der Fragestellung, ob ein solches Verhalten für Answer Set Programming realisierbar ist: Kann zu einem gegebenen logischen Programm und einem gewünschten Modell herausgefunden werden, welche Fakten notwendig sind, um dieses Modell als Lösung zu erhalten?

In diesem Kapitel wird der aktuelle Stand der Entwicklung von Abduktion in ASP vorgestellt. Dazu werden in Abschnitt 6.1 die Grundlagen gelegt. Darauf aufbauend wird in Abschnitt 6.2 Abduktion in ASP untersucht. In Abschnitt 6.3 wird eine Anwendung an der PM durchgeführt und bewertet. Abschließend wird eine Einschätzung in Abschnitt 6.4 gegeben.

6.1 Abduktion in der Logik

Im Artikel [5] wird eine Übersicht von Abduktion in der logischen Programmierung präsentiert. Dabei wird zuerst Abduktion in der Logik wie folgt formal beschrieben: Es existiert eine logische Theorie T und eine Reihe von Beobachtungen O. Mit Hilfe des Verfahrens Abduktion werden Erklärungen von O in T gefunden. Eine dieser Erklärungen wird ausgewählt.

Definition 14 *Sei E eine Erklärung von O in T, so gilt:*

$$T \cup E \models O$$
$$T \cup E \textit{ ist konsistent}$$

Es muss also die Beobachtung O aus der Erklärung E und der Theorie T folgen. Dabei muss O konsistent in T sein. In der formalen Logik sind O und E jeweils Mengen von Literalen.

Neben diesen beiden Bedingungen für E werden normalerweise weitere Bedingungen verwendet, um Minimalität zu gewährleisten. Dabei werden irrelevante Tatsachen, die nicht zur Folgerung aus O notwendig sind, vermieden.

Darauf aufbauend ist Abductive Logic Programming. Diese erweitert die Abduktion in der Logik wie folgt: Ein abduktives logisches Programm hat drei Komponenten $< P,\ A,\ IC >$.

- P ist ein logisches Programm.
- A ist eine Menge an Erklärungsprädikaten bzw. Hypothesen.
- IC ist eine Menge von Bedingungen.

In der Gegenüberstellung ist das logische Programm P das Äquivalent zur Theorie T. Dabei hat P die Eigenschaft, keine Konstrukte und Regeln zuzulassen, die nicht auch in ASP beschrieben werden können. Die Menge A ist verwandt mit der Erklärung E. Neu ist die Menge IC, welche Integritätsbedingungen enthält. Diese stellen sicher, dass kein Atom $p \in A$ im Kopf einer Regel von P auftritt.

Definition 15 *Sei ein abduktives logisches Programm $< P, A, IC >$ gegeben, dann ist eine abduktive Erklärung für die Frage Q eine Menge $\triangle \subseteq A$ von erklärenden Atomen, so dass:*

$$P \cup \triangle \models Q$$
$$P \cup \triangle \models IC$$
$$P \cup \triangle \textit{ ist konsistent}$$

Diese ist die allgemeine Definition für Abductive Logic Programming (ALP).

6.2 Abduktion in ASP

Am Anfang der wissenschaftlichen Untersuchung von ALP in ASP existierte die Ansicht, dass Abduktion eine Erweiterung ist. In [8] wird aber gezeigt, dass sich Abduktion vollständig in ASP beschreiben lässt. Darauf basierend wird im folgenden ein abduktives logisches Programm in ein Answer Set Programm transformiert.

Definition 16 *Sei $< P, A, IC >$ ein abduktives logisches Programm:*

1. *Für jedes erklärende Atom $p \in A$ wird ein zusätzliches Atom $\tilde{p} \notin A \cup P$ erstellt.*
2. *Es werden die folgenden Regeln zu P für jedes Atom $p \in A$ hinzugefügt:*
   - $p \leftarrow \neg\tilde{p}$
   - $\tilde{p} \leftarrow \neg p$
3. *Die hinzugefügten Regeln werden in der Menge $\Gamma(A)$ referenziert, so das $\Gamma(A) = \{p \leftarrow \neg\tilde{p} | p \in A\} \cup \{\tilde{p} \leftarrow \neg p | p \in A\}$*
4. *Die Bedingungen in IC werden fallengelassen.*

Das resultierende Answer Set Programm $\Gamma(A) \cup P$ ist eine vollständige Transformation des abduktiven logischen Programms $< P, A, IC >$.

Die obigen Paare von Regeln bringen zum Ausdruck, dass sich p und $\tilde{p}$ gegenseitig ausschließen. Wird $\tilde{p}$ übergeben, so bedeutet dies, dass p nicht geglaubt wird. Die erste Regel $p \leftarrow \neg\tilde{p}$ entspricht der Annahme, p sei wahr, während die zweite Regel $\tilde{p} \leftarrow \neg p$ die Annahme, p sei falsch beschreibt. Sollte die Annahme von p einen Widerspruch auslösen, verhindert die zweite Regel eine Schlussfolgerung von p aus der ersten Regel.

Dass die Bedingungen in IC fallengelassen werden müssen ist offensichtlich klar, da die hinzugefügten Regeln per Definition diesen Bedingungen widersprechen. Dies stellt aber keinen Widerspruch dar, da das Ziel eine Transformation eines abduktiven logischen Programms in ein Answer Set Programm darstellt. Eine Implementation könnte in einem Schritt null das vorhandene logische Programm P prüfen, ob es den Bedingungen IC entspricht. Damit wäre dies a priori umgesetzt.

Definition 17 *Sei $< P, A, IC >$ ein abduktives logisches Programm, $\Gamma(A) \cup P$ das resultierende transformierte Answer Set Programm und $\triangle \subseteq A$, dann ist $\mathcal{M}(\triangle)$ das stabile Modell genau dann wenn ein stabiles Modell $\mathcal{M}'$ für $\Gamma(A) \cup P$ existiert, so dass $\mathcal{M}' = \mathcal{M}(\triangle) \cup \tilde{\bigtriangledown}$ mit $\tilde{\bigtriangledown} = \{\tilde{p} | p \in (A - \triangle)\}$.*

Zur Erläuterung wird das folgende abduktive logische Programm $< P, A, IC >$, gegeben in den Formeln 12 und 13, als Beispiel vorgestellt. Dabei ist IC[16] entsprechend gegeben.

$$A = \{a, b\} \tag{12}$$

$$P = \left\{ \begin{array}{c} p \leftarrow b \\ q \leftarrow a \\ r \leftarrow \neg b \\ \bot \leftarrow q, b \\ \bot \leftarrow \neg q, r \end{array} \right\} \tag{13}$$

Für dieses Beispiel kann gezeigt werden, dass $\mathcal{M}_1(\triangle_1) = \{b, p\}$ mit $\triangle_1 = \{b\}$ und $\mathcal{M}_2(\triangle_2) = \{a, q, r\}$ mit $\triangle_2 = \{a\}$ Lösungen sind.

Führt man eine Transformation zu einem Answer Set Programm durch, so werden die Regeln $\Gamma(A)$, gezeigt in den Formeln 14, hinzugefügt.

$$\begin{array}{c} a \leftarrow \neg\tilde{a} \\ \tilde{a} \leftarrow \neg a \\ b \leftarrow \neg\tilde{b} \\ \tilde{b} \leftarrow \neg b \end{array} \tag{14}$$

Damit wird ersichtlich, dass die beiden Aussagen der Formeln 15, 16 tatsächlich stabile Modelle von $\Gamma(A) \cup P$, und somit Lösungen für das abduktive logische Programm, sind.

$$\mathcal{M}'_1 = \mathcal{M}_1(\triangle_1) \cup \tilde{\bigtriangledown}_1 = \{b, p\} \cup \{\tilde{a}\} = \{b, p, \tilde{a}\} \tag{15}$$

$$\mathcal{M}'_2 = \mathcal{M}_2(\triangle_2) \cup \tilde{\bigtriangledown}_2 = \{a, q, r\} \cup \{\tilde{b}\} = \{a, q, r, \tilde{b}\} \tag{16}$$

Somit wurde ASP mit Bezug auf Hypothesen zur Erläuterung verwendet.

Definition 18 *Seit $< P, A, IC >$ ein abduktives logisches Programm, $\Gamma(A) \cup P$ das resultierende transformierte Answer Set Programm und $\boldsymbol{q}$ eine Beobachtung. Die Beobachtung $\boldsymbol{q}$ hat eine Erklärung mit einer Menge an Hypothesen $\triangle$ genau dann, wenn ein stabiles Modell $\mathcal{M}$ für $\Gamma(A) \cup P \cup \{\bot \leftarrow \neg q\}$ mit $\triangle = \mathcal{M} \cap A$ existiert.*

[16] Die Bedingungen sind entsprechend formuliert, dass weder a noch b im Kopf einer Regel von P auftreten dürfen.

Um dies zu erläutern, wird noch einmal das logische Programm aus den Formeln 12 und 13 verwendet. Unter Annahme einer Beobachtung q existiert eine eindeutige Erklärung mit der Menge von Hypothesen $\{a\}$. Es wird die Bedingung

$$\bot \leftarrow \neg q$$

die der Beobachtung entspricht, zum Answer Set Programm $\Gamma(A) \cup P$ hinzugefügt. Dabei ist $\mathcal{M}'_2 = \{a, q, r, \tilde{b}\}$ das stabile Modell für $\Gamma(A) \cup P \cup \{\bot \leftarrow \neg q\}$. Somit zeigt sich, dass $\mathcal{M}'_2 \cap A = \{a\}$ äquivalent zu der Menge der Hypothesen ist.

6.2.1 Integration in DLV

Da gezeigt ist, dass sich ALP in ASP ausdrücken lässt, wurde dieses Wissen bereits in Implementationen von ASP Solvern angewandt. Die für diese Arbeit verwendete Implementation DLV ist hier keine Ausnahme, sondern verfügt über explizite Parameter und Konfigurationseinstellungen, die eine direkte Auswertung eines abduktiven Problems erlauben. Die richtige Benennung der Dateien ist hier für den korrekten Ablauf der Abduktionskomponente von DLV notwendig. Es werden mindestens drei einzelne Dateien benötigt, deren Eigenschaft über die Endung bzw. dem Typ festgelegt werden. Hierbei ist folgende Assoziation für $< P, A, IC >$ festgelegt:

- Das logische Programm $P \mapsto$ `dl` (dies kann sich auch auf mehrere Dateien verteilen)
- Die Hypothesen $A \mapsto$ `hyp`
- Die Frage bzw. Beobachtung $Q \mapsto$ `obs`

Ein abduktives logisches Programm `prg` in DLV besteht entsprechend aus den Dateien `prg.dl`, `prg.hyp` und `prg.obs`.

Zur Demonstration von Abduktion in DLV wird das bereits vorgestellte Beispiel der Formeln 12 und 13 von Seite 53 verwendet. Das logische Programm P ist in Quelltext 32 in DLV Syntax dargestellt.

```
p :- b.
q :- a.
r :- not b.

:- q,b.
:- not q,r.
```

Quelltext 32: Beispiel für Abduktion, P

Die Fakten für A sind in Quelltext 33 zu sehen, während die Beobachtung in Quelltext 34 gezeigt wird.

```
a.
b.
```

Quelltext 33: Beispiel für Abduktion, A

```
q.
```

Quelltext 34: Beispiel für Abduktion, Beobachtung

Abduktion wird in DLV als analytisches Frontend betrachtet und aktiviert, indem die Option `-FD` zum Aufruf hinzufügt wird. Der Aufruf ist dann wie folgt:

- `dlv -FD prg.dl prg.hyp prg.obs`

Nach Aufruf am Beispiel liefert DLV wie erwartet folgende Ausgabe :

```
DLV [build BEN/Dec 21 2011   gcc 4.6.1]

Diagnosis: a
```

Quelltext 35: Abduktion Lösung

Es sei noch erwähnt, dass DLV beim Aufruf mit `-FD` alle möglichen Lösungen für das abduktive Problem liefert. Beim dem gezeigten Beispiel gibt es aber nur eine einzige Lösung. Sind nur minimale Lösungen gesucht, so lässt sich dies mit dem Parameter `-FDsingle` erreichen.

6.3 Anwendung an der Preis-Maschine

Auch bei der Untersuchung von Abduktion war die Preis-Maschine eine Motivation. Neben der Preisbestimmung liefert die PM unterstützende Informationen für den Vertrieb. Eine dieser Informationen ist die Auswahl eines geeigneten Verkaufskanals für jedes Gerät. Im Normalfall ist ein Gerät nicht für jeden Kunden sichtbar. Die Geräte werden nach Endkunden und Händlern aufgeteilt. Dabei ist das Ziel, Endkunden immer eine gleichmäßig verteilte Palette an Geräten zu liefern. Diese müssen sich in Merkmalen unterscheiden, die von einem Endkunden gut bewertbar sind. Geräte, die den Zustand dieser für den Endkunden gewählten Palette verschlechtern, werden an Händler weiterverkauft. Kriterien für die Auswahl des Endkundenkanals sind:

Brand Es sollte Ware von jeder Marke verfügbar sein.

Warengruppe Es sollten aus jeder Warengruppe Geräte verfügbar sein.

Klasse Geräte werden je nach Ausstattung in die Klassen Einstieg, Mitte und Top eingeteilt. Für jede Klasse sollten Geräte vorhanden sein.

Preisunterschiede der Klassen Geräte innerhalb einer Klasse sollten sich um eine Preispunkt scharen, damit bereits aus diesem Kriterium die Klasse sichtbar wird.

Farbe Existierten von einem Gerät Modelle mit verschiedenen Farben, so sollte jede im Endkundenkanal verfügbar sein.

Gebrauchsspuren Die Geräte sollten so wenig Gebrauchsspuren wie möglich haben.

Verpackung Geräte im Originalkarton sind zu bevorzugen.

Alter und Ausstattung Geräte, die sehr alt sind, ältere Komponenten oder ein veraltetes Betriebssystem haben, sollten nicht im Endkundenkanal sein.

Diese Kriterien wurden in der PM berücksichtigt und entsprechend in DLV implementiert. Ein Auszug von diesem relevanten Bereich ist in den Quelltexten 36, 37 und 38 dargestellt. Dabei zeigt Quelltext 36 einen Auszug von definierten Grenzen für Preise nach Warengruppe und Klasse.

```
1 priceBorder(notebook,entry,0,30000).
  priceBorder(notebook,middle,35000,55000).
3 priceBorder(notebook,top,65000,200000).

5 priceBorder(desktop,entry,0,10000).
  priceBorder(desktop,middle,12000,31000).
7 priceBorder(desktop,top,35000,200000).
```

Quelltext 36: Auszug aus der PM - Preisgrenzen

In Quelltext 37 sind Regeln zu sehen, die es erlauben, aus den festgelegten Grenzen für Preise und dem ermittelten Preis eine Konstante für diesen Zustand zu schlussfolgern.

```
inPriceBorder :- priceBorder(G1,C1,B,T), productGroup(G2),
   class(C2), price(P), G1 = G2, C1 = C2, P > B, P < T.
outOfOtherPriceBorder :- priceBorder(G1,middle,_,T),
   productGroup(G2), class(top), price(P), G1 = G2, P > T,
   not inPriceBorder.
outOfOtherPriceBorder :- priceBorder(G1,middle,B,_),
   productGroup(G2), class(entry), price(P), G1 = G2, P < B,
    not inPriceBorder.
outOfOtherPriceBorder :- priceBorder(G1,entry,_,T),
   productGroup(G2), class(middle), price(P), G1 = G2, P > T
   , not inPriceBorder.
outOfOtherPriceBorder :- priceBorder(G1,top,B,_),
   productGroup(G2), class(middle), price(P), G1 = G2, P < B
   , not inPriceBorder.
inOtherPriceBorder :- not inPriceBorder, not
   outOfOtherPriceBorder.

newUnit :- mfgDelta(X), X <= 100.
avgUnit :- mfgDelta(X), X > 100, X < 250.
oldUnit :- not newUnit, not avgUnit.
```

Quelltext 37: Auszug aus der PM - Regeln

Die Ermittlung einer Empfehlung für den Verkaufskanal Endkunde ist in Quelltext 38 zu sehen. Es sei noch erwähnt, dass diese Empfehlung nur aus den individuellen Merkmalen eines Gerätes resultiert. Im Einsatz existiert noch eine zweite Komponente, die basierend auf den individuellen Empfehlungen, dem Brand, der Warengruppe, der Klasse und der Farbe eine Optimierung durchführt. Daraus resultiert letztlich die Empfehlung, die dem Vertrieb für die Auswahl der Verkaufskanäle zur Verfügung gestellt wird.

Im Einsatz stellt sich bei der Betrachtung aller Geräte und ihrer Verteilung in die Verkaufskanäle die Frage, was notwendig wäre, damit ein Gerät im Endkundenkanal und nicht im Händlerkanal landet.

Wählt man als Eingabemerkmale noch einmal das Gerät aus Quelltext 22 von Seite 17 und einen ermittelten Preis von 200,-€ (Fakt `price(20000)`) wird eine schwache Empfehlung für den Verkaufskanal Endkunde gegeben. Um festzustellen, welche Merkmale für eine starke Empfehlung notwendig wären, kann Abduktion verwendet werden. Dazu werden alle Hypothesen, die Auswirkungen auf die Empfehlung haben, zusammengefasst. Diese sind in Quelltext 39 aufgestellt. Wird nun die gewünschte Beobachtung `strongRecommendation` definiert und DLV mit Abduktion aufgerufen so erhält man die Ausgabe in Quelltext 40.

```
strongRecommendation :- inPriceBorder, newUnit,
  os(windows_8), quality(asGoodAsNew).
strongRecommendation :- inPriceBorder, newUnit,
  os(windows_8), quality(almostNew).
strongRecommendation :- inPriceBorder, newUnit,
  os(windows_7), quality(almostNew), packaging(originalBox).
strongRecommendation :- inPriceBorder, avgUnit,
  os(windows_8), quality(almostNew), packaging(originalBox).

weakRecommendation :- inPriceBorder, avgUnit, os(windows_7),
    quality(almostNew).
weakRecommendation :- inPriceBorder, avgUnit, os(windows_7),
    quality(asGoodAsNew).
weakRecommendation :- outOfOtherPriceBorder, newUnit,
  os(windows_8), quality(almostNew).
weakRecommendation :- outOfOtherPriceBorder, newUnit,
  os(windows_8), quality(asGoodAsNew), packaging(neutralBox)
     .

noRecommendation :- not strongRecommendation, not
   weakRecommendation.
```

Quelltext 38: Auszug aus der PM - Empfehlung des Verkaufskanals

Dadurch wird sichtbar, dass entweder durch Erneuern das Gerätes oder durch Ändern des Betriebssystems eine starke Empfehlung möglich wäre. Somit ist eine Anwendung an der PM tatsächlich möglich. Die Tatsache, dass eine Änderung des Herstellungsdatums realistisch nicht möglich ist, sei hier vernachlässigt.

```
inPriceBorder.
outOfOtherPriceBorder.
inOtherPriceBorder.
oldUnit.
avgUnit.
newUnit.
os(windows_8).
os(windows_7).
os(windows_xp).
os(linux).
packaging(originalBox).
packaging(neutralBox).
quality(asGoodAsNew).
quality(almostNew).
quality(minorUse).
quality(moderateUse).
quality(strongTracesOfUse).
```

Quelltext 39: Abduktion der PM - Hypothesen

```
DLV [build BEN/Dec 21 2011   gcc 4.6.1]

Diagnosis: os(windows_8)
Diagnosis: newUnit
```

Quelltext 40: Abduktion der PM, Lösung

6.4 Einschätzung von Abduktion als Engineering-Technik

In diesem Kapitel wurde Abduktion in ASP vorstellt. Dabei wurde in Abschnitt 6.2 gezeigt, dass sich Abduktion vollständig in ASP abbilden lässt und somit keine Erweiterung ist. Dies wurde in Abschnitt 6.2.1 an der Implementation DLV im Detail vertieft. Als praktisches Beispiel wurde die PM in Abschnitt 6.3 verwendet, um eine Anwendung aufzuzeigen.

Da bereits gezeigt ist, dass sich Abduktion vollständig in ASP abbilden lässt, sind hier keine neuen wissenschaftlichen Erkenntnisse zu erwarten. Des weiteren ließe sich Abduktion zwar an der PM anwenden, die praktische Relevanz erscheint dem Autor aber gering. Dies zeigte sich bei der Vorbereitung der PM für die Abduktion. Die Fakten der Hypothese wurden durch Betrachtung aller Regeln ermittelt. Da es sich hierbei um *relativ einfache*[17] Regeln handelt ohne Rekursionen, war nach der Inspektion dieser für ein geschultes Auge klar, welche Fakten das gewünschte Ergebnis liefern werden. Es

[17]Bei den Begriffen *relativ einfache* handelt es sich um eine intuitive praktische Bezeichnung.

wird deswegen angenommen, dass erst mit einer sehr großen Anzahl und Komplexität an Regeln ein praktisch relevanter Nutzen aus Abduktion gezogen werden kann.

7 Abschlussbemerkung

Diese Arbeit hatte zum Ziel, ASP Engineering Ansätze mit einer praxisbezogenen Reflektion zu untersuchen. Dazu wurde zu Beginn eine historische Einführung in das Softwareengineering gegeben. Diese führte vor Augen, worin die Notwendigkeit von Softwareengineering liegt und welche Methoden bis heute zum Einsatz kommen.

Im Anschluss wurde in Kapitel 2 eine Einführung in Answer Set Programming gegeben. Hierbei wurde am Beispiel Prolog auf die Probleme mit bestehenden logischen Systemen eingegangen und herausgestellt, wo der große Vorteil im stabilen Modell und damit ASP liegt. Es wurde weiterhin DLV, die in dieser Arbeit verwendete Implementation von ASP, im Detail vorgestellt. Dazu wurden Aufruf und Syntax der Sprache gezeigt. Vertiefend wurden in Abschnitt 2.2.7 unzulässige Formulierungen aufgezeigt. Dieser Abschnitt präsentiert damit auch indirekt die Schwächen von ASP im Vergleich zu Prolog. In Prolog sind Regeln möglich, die in ASP nicht lösbar und damit verboten sind.

Da in dieser Arbeit Wert darauf gelegt wurde, Techniken nicht nur unter wissenschaftlichen Gesichtspunkten zu analysieren, sondern auch an Hand einer praktischen Implementation, wurde in Kapitel 3 die Firma GG-Net und ihre Implementation der Preis-Maschine in DLV vorgestellt.

Als erste Technik wurde in Kapitel 4 das naive Ausprobieren zur Verifikation von Answer Set Programmen untersucht. Dazu wurde ein Verfahren entworfen und beschrieben, das naives Ausprobieren als Verifikationstechnik realisiert. Dieses Verfahren wurde klassifiziert und in einem Experiment auf praktische Anwendbarkeit untersucht. Dabei kann als Erkenntnis festgehalten werden, dass naives Ausprobieren unter geeigneten Bedingungen eine zielführende Verifikationstechnik ist. Es wurde gezeigt, dass die Herausforderung in der geeigneten Reduktion der Eingabemerkmale liegt, aber wenn möglich auf Grund ihres einfachen Charakters schnell verstanden ist und geringe Risiken einer fehlerhaften Implementation mit sich bringt.

In Kapitel 5 wurden Abstrakte Interpretation und Datenflussanalyse mit dem Fokus untersucht, ob die Lösung aus Kapitel 4 in ihrer Komplexität reduziert werden kann. Dazu wurde basierend auf der klassischen aktiven Variablenanalyse die aktive Merkmal Analyse für ASP entwickelt und diese abschließend an einem Auszug der PM verwendet. Hierbei konnte gezeigt werden, dass mit Hilfe von Abstrakter Interpretation die Menge von Eingabemerkmalen in einem Answer Set Programm reduziert werden kann und somit eine Lösung gefunden wurde, die der Motivation zu Grunde lag.

Als letzte Technik wurde Abduktion in Kapitel 6 vorgestellt. Diese kann verwendet werden, um automatisch Fakten zu finden, die für eine gewünschte Lösung benötigt werden. Es wurde gezeigt, dass sich Abduktion vollständig in ASP abbilden lässt und DLV bereits ein auf Abduktion spezialisiertes Frontend besitzt. Diese wurde auch am praktischen Beispiel der PM verwendet. Die gewonnen Erkenntnisse waren hier ernüchternd und weisen nur auf eine sinnvolle Anwendung in einer Umgebung mit sehr vielen Regeln

hin.

Zusammenfassend wurden in dieser Arbeit Engineering Techniken für ASP gezeigt. Dabei wurde ihr Anwendungsfeld umrissen sowie geeignete und ungeeignete Einsatzszenarien skizziert. Es wurde im Detail herausgearbeitet, unter welchen Bedingungen eine Anwendung einer Technik von Vorteil ist und wo sich keine sowie nachteilige Effekte zeigen. Es kann abschließend festgestellt werden, dass dabei keine der Techniken immer von Vorteil ist.

Akronyme

SopoNr Sonderpostennummer
PM Preis-Maschine
SOPO Sonderposten
ASP Answer Set Programming
ALP Abductive Logic Programming

Literatur

[1] Kent Beck. *Extreme programming explained : embrace change*. Addison-Wesley, us ed edition, October 2001.

[2] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages II*, pages 331–367. ACM, New York, NY, USA, 1996.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[5] Marc Denecker and Antonis Kakas. Abduction in Logic Programming. In Antonis Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, chapter 16, pages 99–134. Springer, Berlin, July 2002.

[6] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[7] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2 edition, August 2004.

[8] Noboru Iwayama and Ken Satoh. Computing abduction by using tms with top-down expectation. *J. Log. Program.*, 44(1-3):179–206, 2000.

[9] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1 edition, February 1999.

[10] K. W. Kennedy. Node listings applied to data flow analysis. In *Proceedings of the*

2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '75, pages 10–21, New York, NY, USA, 1975. ACM.

[11] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[12] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[13] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. September 1998.

[14] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[15] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3):241–273, November 1999.

[16] Taiichi Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press.

[17] Ken Schwaber. SCRUM Development Process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995.

[18] Allen van Gelder, Kenneth Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.