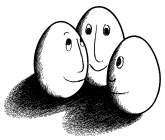
Diplomarbeit

Validierung und Regelrevision in einem wissensbasierten System zur Layoutplanung von Chemieanlagen

Andreas P. Schröder



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

Dortmund, 26. Mai 2006

Betreuer:

Prof. Dr. Katharina Morik
Dipl.-Inform. Hanna Köpcke

Erklärung

Hiermit erkläre ich, Andreas P. Schröder, die vorliegende Diplomarbeit mit dem Titel *Validierung und Regelrevision in einem wissensbasierten System zur Layoutplanung von Chemieanlagen* selbständig verfasst und keine anderen als die hier angegebenen Hilfsmittel verwendet, sowie Zitate kenntlich gemacht zu haben.

Dortmund, 26. Mai 2006

Danksagung

Ich möchte mich an dieser Stelle herzlich bei meinen Betreuern Katharina Morik und Hanna Köpcke für ihre Ratschläge, ihre Kommentare und ihre Unterstützung bei der Anfertigung dieser Arbeit bedanken.

Ebenso bedanken möchte ich mich bei den Mitarbeitern, Diplomanden, studentischen Hilfskräften und Fußballspielern des Lehrstuhls für künstliche Intelligenz für das motivierende Arbeitsumfeld und die Unterstützung bei Problemen jeglicher Art.

Einen besonderen Dank verdienen auch Frau Luhle und Herr Sander für das Korrekturlesen der Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In vielen Bereichen der Wirtschaft, unter anderem auch in der Industrie- und Anlagentechnik, werden seit langer Zeit schon wissensbasierte Systeme eingesetzt, um Ingenieure und Techniker bei ihrer Arbeit zu unterstützen und das vorhandene Know-How der Experten zu sichern, zu verbessern und vielen Mitarbeitern zugänglich zu machen. Insbesondere im Bereich der wissensbasierten Konstruktion (engl. knowledge-based engineering) werden solche Systeme mit Erfolg eingesetzt. Beispiele dafür sind etwa der Flugzeugvorentwurf [20], die Erstellung von flexiblen Verpackungen [42], oder – wie in der vorliegenden Arbeit beschrieben – bei der Layoutplanung von Chemieanlagen [23].

Mit dem Einsatz solcher Systeme soll es möglich werden, Wissen über einen begrenzten Anwendungsbereich in einem System abzulegen, so dass wiederkehrende Aufgaben besser, schneller und weniger fehleranfällig gelöst werden können. Ebenso soll das Wissen auf – im Rahmen des Anwendungsbereichs – neue Aufgaben angewendet werden und diese damit effizienter bearbeitet werden können.

Damit ergeben sich zwei wesentliche Vorteile für den Einsatz solcher Systeme in einem Unternehmen. Zum einen können Arbeitsabläufe beschleunigt und eine konstante Qualität der Bearbeitung erreicht werden. Zum anderen wird das Unternehmen durch die systematische Erfassung des Wissens unabhängig von den Kenntnissen und der Intuition weniger Experten, da das Wissen in explizierter Form vorliegt. Damit einhergehend wird eine systematische Dokumentation und Konservierung des Know-Hows erreicht.

Als Beispiel für einen solchen Anwendungsfall dient in dieser Arbeit die Umsetzung von Expertenwissen zur Layoutplanung von Chemieanlagen in eine entsprechende Wissensbasis. Das für die Erstellung der Wissensbasis notwendige Wissen lieferte die Dissertation "Rechnergestützte Optimierung der Layoutplanung von Chemieanlagen" von Peter Leuders [23]. Eine computergenerierte Ansicht der unter Verwendung dieser Wissensbasis erstellten Anlage "Trieste", einer chemietechnischen Anlage zur Gasvorbehandlung, ist in Abbildung 1.1 dargestellt. Die verwendete Wissensbasis wird für einen Teilaspekt der Konstruktion solcher Anlagen, nämlich für die korrekte Platzierung einzelner Bestandteile der Anlage unter Berücksichtigung von Anforderungen der verwendeten Bauteile an ihren Standort und für sicherheitsrelevante sowie konstruktionsbedingte Anforderungen, verwendet.

Die Wissensbasis unterstützt also nur die Layoutplanung, eine wesentliche Aufgabe im Gesamtverfahren zur Konstruktion einer solchen Anlage. Im ersten Schritt

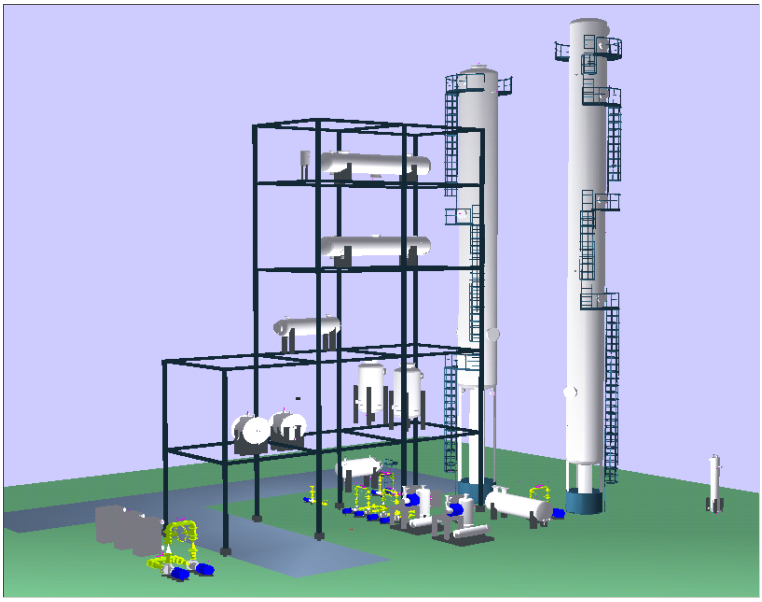


Abbildung 1.1.: Computergenerierte Ansicht der Chemieanlage “Trieste”

leitet das Expertensystem aus einer abstrakten Beschreibung der Anlage die bei der Generierung einer Aufstellung einzuhaltenden Anforderungen der einzelnen Komponenten ab. Ein von dem wissensbasierten System unabhängiger Algorithmus verarbeitet diese Anforderungen (engl. constraints) dann. Er berechnet eine Aufstellung, d.h. er weist jedem Bauelement einen Standort zu, so dass möglichst viele der Constraints dieses Bauelements erfüllt werden.

Nachdem das für ein Expertensystem zur Layoutplanung einer Chemieanlage notwendige Wissen bereits in der Dissertation von Leuders erarbeitet und bereitgestellt wurde, wird in dieser Arbeit untersucht, wie sich eine darauf aufbauende Wissensbasis auf Fehler überprüfen lässt. Es wird untersucht, ob das abgelegte Wissen korrekt ist und wie sich darin eventuell enthaltene Fehler aufdecken und korrigieren lassen.

1.1. Motivation

Die Erstellung einer Wissensbasis für ein Expertensystem verläuft oft in einem iterativen Prozess, in dem die Wissensbasis immer wieder modifiziert und erweitert wird. Dies hängt damit zusammen, dass neues Wissen ergänzt und bereits eingepflegtes Wissen geändert werden muss. Ebenso kann es vorkommen, dass bisher als korrekt angenommenes Wissen sich als falsch herausstellt und korrigiert werden muss. Dies kann etwa dann vorkommen, wenn bisher allgemein formuliertes Wissen in bestimmten Fällen, die bisher nicht berücksichtigt wurden, fehlerhaft arbeitet und deshalb soweit spezialisiert werden muss, dass es auf diese Fälle nicht mehr

angewendet wird. Es ist daher bei der Arbeit mit Expertensystemen von großer Bedeutung, nicht nur deren einmalige Erstellung, sondern auch die im Rahmen der Wartung und Erweiterung solcher Systeme immer wieder vorgenommenen Modifikationen rechtzeitig zu berücksichtigen. Dieser Punkt muss vom Ersteller einer Wissensbasis beachtet werden. Er sollte aber auch von Entwicklern von Software zur Erstellung wissensbasierter Systeme nicht vernachlässigt werden, um den Anwender bei seiner Arbeit bestmöglich zu unterstützen.

Bei der Erstellung, aber auch bei der weiteren Wartung der Wissensbasis für ein Expertensystem sind Kenntnisse aus zwei Bereichen notwendig: Zum einen das Wissen über den Anwendungsbereich, welches in der Wissensbasis abgelegt werden soll, zum anderen Kenntnisse über eine sinnvolle Modellierung des abzulegenden Wissen in einem wissensbasierten System. Eine wesentliche Erleichterung in dem Prozess der Erstellung und Wartung einer Wissensbasis wäre es daher, wenn der Anwender durch den Einsatz geeigneter Software bereits während einer Änderung der Wissensbasis auf mögliche Probleme und Fehler aufmerksam gemacht wird und diese somit verhindern kann.

In dieser Arbeit soll ein Teilaspekt der Unterstützung von Anwendungsexperten bei der Arbeit mit wissensbasierten Systemen betrachtet werden. Es wird davon ausgegangen, dass bereits eine funktionsfähige Wissensbasis vorliegt. Diese wird nun weitgehend automatisiert daraufhin untersucht, ob sie Schwachstellen oder sogar Fehler enthält, die für die Funktion des Expertensystems von Bedeutung sind.

1.2. Ziele

Es wird in dieser Arbeit untersucht, in wieweit es möglich ist, die Erstellung und Wartung einer Wissensbasis zur Layoutplanung von Chemieanlagen durch die automatisierte Untersuchung auf Entwurfsprobleme und -fehler zu unterstützen. Um dieses Ziel zu erreichen, wird zunächst eine Software-Umgebung namens SIMPLE[1] zur Arbeit mit Wissensbasen implementiert, welche die folgenden Aufgaben erfüllt:

- Darstellung der Wissensbasis in einer für den Anwender übersichtlichen Form sowie die Unterstützung einfacher Modifikationen der Wissensbasis (d. h. Hinzufügen und Löschen von Prädikaten und Fakten sowie die Modifikation von Regeln)
- Laden und Abspeichern von Wissensbasen in einem XML-Dialekt.
- Auswertung der Wissensbasis mittels zwei verschiedener Inferenz-Verfahren und Weitergabe der Ergebnisse an einen Algorithmus zur Berechnung der Aufstellung einer Chemieanlage

[1] **S**imple **I**nference **M**echanism and **P**rogrammable **L**ogic **E**nvironment

- Bereitstellung eines Rahmens zur Implementierung von Verfahren zur Untersuchung der Wissensbasis auf Anomalien und deren Präsentation dem Benutzer.

Auf Basis dieser Rahmenanwendung werden dann Verfahren zur Verifikation und Validierung der Wissensbasis implementiert und untersucht. Bei den Verfahren zur Verifikation handelt es sich um eines zur Auffindung von Redundanzen in der Regelbasis und um eines zur Widerspruchsentdeckung. Daneben wird ein Verfahren zur Validierung eingesetzt, welches die Ergebnisse eines Algorithmus zur Berechnung von Aufstellungen dazu verwendet, die Regeln anhand des Erfüllungsgrades der von ihnen abgeleiteten Constraints zu bewerten. Die Ergebnisse des Verfahrens zur Widerspruchsentdeckung werden dann mit diesem Verfahren verglichen.

Die implementierten Verfahren werden schließlich dahingehend bewertet, ob und wie gut sie in der Lage sind, für den Entwurfs- und Wartungsprozess der Wissensbasis relevante Ergebnisse zu liefern.

1.3. Gliederung der Arbeit

Im Folgenden wird eine kurze Übersicht über die Gliederung und den weiteren Inhalt dieser Arbeit gegeben.

In Kapitel 2 werden die für diese Arbeit relevanten Grundlagen beschrieben. Nach einer Übersicht über die Eigenschaften häufig anzutreffender Typen von Expertensystemen, den fallbasierten und regelbasierten Systemen, wird darauf eingegangen, wie regelbasierte Systeme entworfen werden, mit welchen Problemen der Entwurf verbunden ist und woraus diese Probleme resultieren. Daran anschließend folgen ein Abschnitt, der sich mit der Validierung und Verifizierung wissensbasierter Systeme befasst sowie ein Abschnitt mit Erläuterungen zur Wahl einer geeigneten Wissensrepräsentation für regelbasierte Systeme. Das Kapitel endet mit einer Übersicht über den Aufbau und die Funktionsweise der dieser Arbeit zugrunde liegenden Wissensbasis zur Layoutplanung von Chemieanlagen.

Kapitel 3 befasst sich mit den in regelbasierten Systemen auftretenden Anomalien. Zunächst wird eine Übersicht über bekannte Arten von Anomalien wie Redundanz, Ambivalenz, Zirkularität, Defizienz und inaktive Regeln gegeben. Es wird erläutert, mit welchen Schwierigkeiten die Entdeckung von Widersprüchen verbunden ist. Darauf folgen Abschnitte zu Truth Maintenance Systemen, also Systemen, die den Anwender bei der Revision der Wissensbasis unterstützen, und zu den Grundlagen der Theorierevision.

Nach den in den vorherigen Kapiteln erläuterten Grundlagen befasst sich Kapitel 4 mit den praxisorientierten Aspekten dieser Arbeit, d. h. mit der Erstellung des Frameworks SIMPLE und der Erläuterung der darin implementierten Verfahren. Es handelt sich dabei um die zur Auswertung der Wissensbasis verwendeten Inferenzmechanismen, die Verfahren zur Widerspruchs- und Redundanzentdeckung

und um die Berechnung der Regelperformanz. Es werden der Aufbau der Software sowie die eingesetzten Algorithmen beschrieben.

In Kapitel 5 werden schließlich die bei den Anwendung der Verfahren zur Anomalieentdeckung gesammelten Ergebnisse vorgestellt und bewertet. Daran anschließend findet sich ein Fazit und ein Ausblick auf mögliche Erweiterungen und Fortsetzungen der Arbeit.

Im Anhang A ist die in dieser Arbeit betrachtete Regelbasis zu finden.

2. Grundlagen

In dieser Arbeit wird die Entwicklung eines Systems beschrieben, welches Expertenwissen zur Aufstellungsplanung für Chemieanlagen verarbeiten kann. Insbesondere soll es das Wissen speichern und es auf eine Problemstellung, d. h. auf die Daten einer in der Entwicklung befindlichen Chemieanlage anwenden. Wesentliche Aufgabe ist aber auch, das gespeicherte Wissen zu überprüfen, zu warten und zu pflegen. Diese Aufgabenstellung erfüllen im Allgemeinen wissensbasierte Systeme. Es wird daher zunächst eine Übersicht über die verschiedenen Typen wissensbasierter Systeme sowie deren Grundlagen gegeben, um zu zeigen, warum ein regelbasiertes System diesen Anforderungen am besten entspricht.

2.1. Wissensbasierte Systeme und Expertensysteme

Wissensbasierte Systeme sind Systeme, die in der Lage sind, Informationen zu einem begrenzten Anwendungsbereich aufzunehmen und das gespeicherte Wissen dann zur Nutzung zur Verfügung zu stellen. Mit "Nutzung" ist dabei nicht nur gemeint, das Wissen auf neue Problemstellungen anzuwenden. Das Wissen sollte auch in einer für Anwender aufbereiteten, verständlichen Form dargestellt werden. Damit unterscheiden sich wissensbasierte Systeme zum Beispiel von neuronalen Netzen, die Wissen zwar speichern, aber nicht in einer für den Anwender verständlichen Form darstellen können. Mit dem Begriff "Anwender" ist hier und im Folgenden sowohl ein Experte gemeint, der Wissen in einem wissensbasierten System bearbeitet, aber auch ein Anwender, der das abgelegte Wissen zur Bearbeitung einer Aufgabenstellung nutzt.

Der Begriff "Expertensystem" wird oft als Synonym für wissensbasiertes System verwendet. In [13] geben Giarratano und Riley eine Definition an, welche das Verhalten eines Expertensystems in den Vordergrund stellt:

> "An expert system is a computer system that emulates the decision-making ability of a human expert in a restricted domain."

Die Definition von Edward Feigenbaum (zitiert nach [17]) gibt genauere Hinweise auf die in einem Expertensystem enthaltenen Bestandteile "knowledge" und "inference procedure" zur Erzielung des gewünschten Verhaltens:

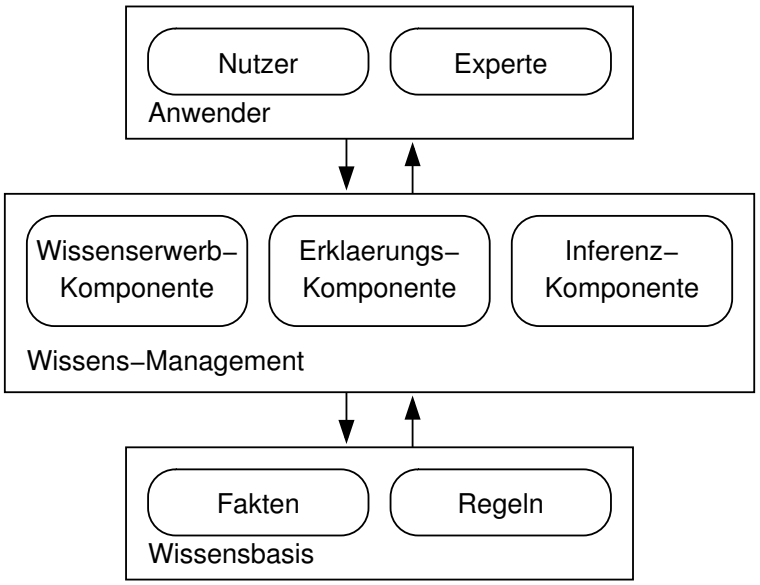


Abbildung 2.1.: Modularer Aufbau eines wissensbasierten Systems (nach [15])

> "An intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions."

2.1.1. Aufbau wissensbasierter Systeme

Wissensbasierte Systeme bestehen typischerweise aus mehreren Modulen, die auf einander aufbauen und erst durch ihre Interaktion die gewünschte Funktionalität zur Verfügung stellen. Ein typischer Aufbau ist in Abbildung 2.1 dargestellt. Die meisten wissensbasierten Systeme bestehen aus mehreren der folgenden Komponenten:

- Wissensbasis
- Inferenzkomponente
- Erklärungskomponente
- Wissenserwerbskomponente
- Benutzerschnittstelle

Die *Wissensbasis* enthält das dem System zur Verfügung stehende Wissen, welches auf neue, noch unbekannte Problemstellungen angewendet werden kann. Wesentliches Unterscheidungsmerkmal verschiedener Systeme ist die Form der Wissensrepräsentation, die Einfluss auf die Ausdrucksstärke aber auch auf den Aufwand zur Auswertung des Wissens hat.

Die *Inferenzkomponente* greift auf die Wissensbasis zu und wendet das Wissen auf Eingabedaten an, um so neues Wissen abzuleiten. Hierbei kann es sich, wie zum Beispiel beim Case Based Reasoning, um ein Verfahren handeln, welches das gestellte Problem mit bereits bekannten und gelösten Problemstellungen vergleicht und die ähnlichsten davon selektiert oder auch um komplexere Mechanismen, die in Form von Regelsystemen abgelegte Wissensbasen auswerten können. Die Auswahl einer geeigneten Inferenzkomponente ist abhängig von der Wahl der Wissensrepräsentation.

Die *Erklärungskomponente* liefert dem Anwender Informationen darüber, wie das System eine Lösung für das gestellte Problem gefunden hat bzw. warum genau eine bestimmte Lösung gefunden wurde. Diese Komponente muss nicht zwingend vorhanden sein. Sie erleichtert aber zum einen den Zugriff eines Anwenders auf das Wissen, da er mit ihrer Hilfe erkennen kann, welches Wissen dem System bekannt ist und wie dieses Wissen angewendet wird. Zum anderen wird die Wartung, Pflege und Überprüfung des gespeicherten Wissens vereinfacht, da leicht nachvollzogen werden kann, warum das System eventuell andere als die gewünschten Antworten auf eine Problemstellung liefert.

Die *Wissenserwerbskomponente* ermöglicht dem System, neues Wissen in die Wissensbasis aufzunehmen. Dies kann entweder manuell geschehen, indem der Anwender die Wissensbasis entsprechend seinen Vorstellungen verändert und ergänzt oder auch durch das selbständige, automatische Erlernen neuen Wissens, z. B. aus klassifizierten Beispielen oder auch durch die Extraktion von Wissen aus Texten.

Das Zusammenspiel der genannten Komponenten eines wissensbasierten Systems wird zusammengefasst und dem Benutzer durch eine geeignete *Benutzerschnittstelle* präsentiert, so dass dieser sich nicht mit den zugrunde liegenden Formalismen konfrontiert sieht, sondern das Gesamtsystem effizient zur Lösung der gestellten Aufgaben verwenden kann. Je nach Anwendungsfall können auch mehrere Benutzerschnittstellen existieren, die jeweils auf die Bedürfnisse einer Anwendergruppe zugeschnitten sind, also zum Beispiel für Anwender, die das gespeicherte Wissen nutzen wollen und Experten, die die Wissensbasis untersuchen und modifizieren möchten.

2.2. Typen

Wissensbasierte Systeme lassen sich anhand verschiedener Merkmale unterscheiden. Wesentlich ist vor allem die Art der Wissensrepräsentation, da hiervon die weiteren Eigenschaften und Fähigkeiten der Systeme abhängig sind. Häufig werden die Varianten der *fallbasierten* und *regelbasierten* Systeme verwendet, die im Folgenden jeweils kurz mit ihren Vor- und Nachteilen vorgestellt werden.

2.2.1. Fallbasierte Systeme

Zur Repräsentation des gespeicherten Wissens in fallbasiert arbeitenden Systemen (engl. Case Based Reasoning, CBR) wird eine Menge von Anwendungsfällen in Form von Problemstellungen zusammen mit jeweils geeigneten Lösungsvorschlägen gespeichert. Soll nun eine für das System neue Problemstellung bearbeitet werden, so wird diese mit den bereits bekannten Fällen verglichen und im einfachsten Fall die Lösung einer möglichst ähnlichen Problemstellung zurückgegeben. Eine weiterführende Übersicht über die Grundlagen fallbasierter Systeme ist in [3] zu finden.

Um den Aufwand der Verwaltung und Auswertung gering zu halten, werden oftmals nicht alle gelösten Probleme mit ihren Lösungen gespeichert, sondern nur jeweils prototypische Problemstellungen ausgewählt, die möglichst viele Problemstellungen abdecken. In manchen Anwendungsbereichen entsprechen die Problemstellungen nicht einmal realen Fällen, sondern werden künstlich erzeugt, um mit wenigen Beispielen möglichst viele Anfragen beantworten zu können. Unter Umständen werden die in Frage kommenden Lösungsvorschläge bevor sie dem Anwender präsentiert werden, basierend auf den Unterschieden zwischen dem zu lösenden Problem und den gespeicherten Fällen, modifiziert oder spezialisiert und so an das neu gestellte Problem angepasst.

Der Vorteil dieser Form wissensbasierter Systeme ist, dass keine aufwendige Konvertierung der Problemstellungen in eine interne Wissensrepräsentation notwendig ist; im Gegenteil können die aus dem Anwendungsbereich stammenden Beispiele oft fast unverändert übernommen werden, sofern nicht prototypische, in der Realität selten vorkommende Problemstellungen generiert werden sollen. Damit kann das System leicht von den Experten des Anwendungsbereichs eingesetzt und erweitert werden, ohne dass Spezialisten aus dem Bereich wissensbasierter Systeme für die Einpflegung des Wissens benötigt werden.

Außerdem ist die grundlegende Form der Lösungsfindung relativ einfach und performant durchzuführen, da nur ein Vergleich der gestellten Aufgabe mit den bereits gespeicherten Fällen vorgenommen wird und die Lösung des bzw. der ähnlichsten Fälle zurückgegeben wird.

Ein grundlegender Nachteil dieser Systeme besteht darin, dass das Wissen implizit in Form von Beispielen gespeichert wird. Dadurch ist der Zugriff auf das gespeicherte Wissen für den Benutzer schwierig; das System kann nur die zur Lösung herangezogenen, gespeicherten Fälle als Erklärung für ein bestimmtes Verhalten liefern. Ebenso ist die Wartung der Wissensbasis kompliziert. Es ist oft nicht klar, welche Teile der Wissensbasis angepasst werden müssen, um ein verändertes Verhalten des Systems zu erzielen bzw. einen Fehler in der Wissensbasis zu beheben. So wird der Prozess der fortwährenden Veränderung und Anpassung an geändertes Wissen (siehe auch Abschnitt 2.4.4 zu Sloppy Modelling) nur unzureichend unterstützt.

2.2.2. Regelbasierte Systeme

Zur Repräsentation des Wissens in regelbasierten Systemen wird eine Menge von Regeln und Fakten verwendet. Eine Regel entspricht der Form

$$\text{IF } p_1 \text{ AND } \ldots \text{ AND } p_n \text{ THEN } c$$

Bei den p_i handelt es sich um die Prämissen der Regel, d. h. um die Bedingungen, die erfüllt sein müssen, damit die Regel angewendet werden kann. c ist die Konklusion, also das Ergebnis der Regel, welches Gültigkeit erlangt, wenn alle Prämissen erfüllt sind.

Neben den Regeln kann die Wissensbasis Fakten, also Informationseinheiten enthalten, die im Gegensatz zu Regeln uneingeschränkt gültig sind. Sie lassen sich auch als Regeln ohne Prämissen auffassen. Die Fakten einer Wissensbasis können Hintergrundwissen ausdrücken, als Eingabedaten des Anwenders auftreten, die das System verarbeiten soll oder durch das System – genaugenommen durch dessen Inferenzkomponente – abgeleitet werden und dabei als Zwischenergebnisse der Berechnung oder Ausgabedaten dienen.

Auswertung der Regeln

Es gibt zwei Möglichkeiten der Anwendung der Regeln durch die Inferenzkomponente – einen Regelinterpreter. Zum einen die vorwärtsgerichtete oder datengetriebene Auswertung (engl. forward chaining inference). Dabei werden die Konklusionen derjenigen Regeln der Wissensbasis hinzugefügt, deren Prämissen alle erfüllt werden können. Dieser Prozess wird solange fortgesetzt, bis keine weitere Regel mehr angewendet werden kann. Es muss dabei sichergestellt sein, dass das Verfahren auch dann terminiert, wenn die Regelbasis Zyklen enthält.

Ebenso kann eine zielgetriebene oder rückwärtsgerichtete Auswertung (engl. backward chaining inference) durchgeführt werden. Hier wird an das System eine Anfrage nach der Erfüllbarkeit eines Fakts bzw. einer Menge von Fakten gestellt. Daraufhin werden diejenigen Regeln untersucht, die als Konklusion diesen Fakt enthalten und geprüft, ob alle Prämissen mindestens einer dieser Regeln erfüllbar sind. Dies geschieht, indem wiederum Anfragen an das System nach der Erfüllbarkeit der Prämissen gestellt werden. Eine Anfrage ist erfüllbar, wenn sie entweder von einer Regel abgeleitet werden kann, deren Prämissen gleichzeitig erfüllbar sind oder wenn sie als Fakt in der Wissensbasis bereits enthalten ist.

Ausdruckskraft und Berechnungskomplexität

Die regelbasierten Systeme können weiterhin klassifiziert werden anhand der Form des gespeicherten Wissens. Je nach Wahl der Wissensrepräsentation ergibt sich zum einen eine größere bzw. geringere Ausdruckskraft, d. h. es wird festgelegt,

welches Wissen dargestellt werden kann. Zum anderen entscheidet die Form der Wissensrepräsentation auch über den Aufwand bei der Verarbeitung der Regeln.

Im einfachsten Fall können die Regeln in Aussagenlogik repräsentiert werden. Sowohl bei Prämissen als auch Konklusionen der Regeln handelt es sich um Aussagen, die entweder wahr oder falsch sein können. Die Auswertung von Regeln dieser Form ist sehr einfach und effizient möglich. Der Umfang repräsentierbaren Wissens ist jedoch auch sehr beschränkt.

Im Gegensatz dazu kann man auch die gesamten Möglichkeiten der Prädikatenlogik erlauben. Damit erhält man eine wesentlich größere Ausdruckskraft gegenüber der Aussagenlogik, erkauft sich dies aber durch die wesentlich höhere Komplexität der Auswertung der Regeln.

Es ist deshalb notwendig, einen Kompromiss zwischen der benötigten Ausdruckskraft und der dafür in Kauf genommenen Komplexität der Auswertung zu finden. Viele Systeme setzen daher Regeln ein, die Wissen auf Basis von Hornlogik repräsentieren. Es handelt sich dabei um eine Wissensrepräsentation, die auf der funktionsfreien Prädikatenlogik erster Stufe basiert. Sie bietet einen guten Kompromiss zwischen einer für die meisten Anwendungsfälle ausreichenden Ausdruckskraft und einer effizient möglichen Auswertung der Wissensbasis.

2.3. Notation

In den folgenden Abschnitten dieser Arbeit werden mehrfach Beispiele für Fakten wie

`¬blau(ball)`

und Regeln wie

Rule1: `farbe(X, blau)` & `¬eckig(X)` → `ball(X)`

gegeben. Daher sollen an dieser Stelle einige Hinweise zu der verwendeten Syntax aufgeführt werden:

- Regeln bestehen aus einer oder mehreren Prämissen (hier `farbe(X, blau)` und `¬eckig(X)`), dem Folgerungspfeil und einer Konklusion (`ball(X)`).
- Variablenbezeichner beginnen mit einem Großbuchstaben, ansonsten handelt es sich um Konstanten.
- Die Prämissen einer Regel sind mit dem Zeichen "&" konjunktiv verknüpft.
- Die Regeln gelten als implizit allquantifiziert.

- Die Negation eines Literals wird durch das Zeichen ¬ gekennzeichnet.
- Regeln werden in SIMPLE durch einen Namen identifiziert. Sollte der Name einer Regel von Bedeutung sein, so wird er vor die Regeldefinition gestellt (hier "Rule1").
- Prädikate werden in der Form `Name/n` notiert, wobei n die Stelligkeit des Prädikats ist (z. B. `farbe/2`).

2.4. Erstellung und Bearbeitung von Wissensbasen

Bevor mit einem wissensbasierten System gearbeitet werden kann, muss es zunächst mit einer Basisausstattung an Wissen versehen werden. Dabei gibt es verschiedene Möglichkeiten, dieses Wissen zu erwerben. Zum einen den automatischen Erwerb von Wissen. Hierbei wird das Wissen meist aus einer gegebenen Menge von relevanten Beispielen extrahiert. Zum anderen kann das Wissen auch manuell entweder durch die Befragung eines Experten gewonnen werden oder durch den Experten selbst in die Wissensbasis eingegeben werden.

2.4.1. Automatischer Wissenserwerb

In manchen Anwendungsfällen ist es möglich, Wissen aus einer Sammlung von Beispielen zu extrahieren. Es existieren einige Verfahren, die aus einer Menge von im Voraus klassifizierten Beispielen Regeln ableiten können – als Beispiel für ein solches Verfahren sei hier das Verfahren FOIL [32] von Quinlan erwähnt.

Diese Form des Lernens wird auch "induktives Konzeptlernen" genannt. Ein Konzept C ist eine Teilmenge eines Universums U von Objekten. Gegeben ist eine Menge E von Beispielen, die entsprechend ihrer Zugehörigkeit zu dem zu erlernenden Konzept in die zwei Teilmengen der positiven Beispiele $E^+ = \{e|e \in C\}$ und der negativen (Gegen-)Beispiele $E^- = \{e|e \notin C\}$ eingeteilt sind. Es soll daraus eine intensionale Konzeptbeschreibung erlernt werden, d. h. von einer extensionalen, aufzählenden Beschreibung eines Konzepts abstrahiert werden, so dass die Beispiele und auch zur Zeit des Lernens noch unbekannte Objekte korrekt klassifiziert werden können. Das Problem "Induktives Konzeptlernen" ist nach [22] wie folgt definiert:

Definition 1 (Induktives Konzeptlernen) *Es sei eine Menge $E = E^+ \cup E^-$ von positiven und negativen Beispielen eines Konzepts C gegeben. Finde eine Hypothese H, ausgedrückt in einer Konzeptbeschreibungssprache L, so dass*

- *jedes positives Beispiel $e \in E^+$ von H abgedeckt wird und*

- *kein negatives Beispiel* $e \in E^-$ *von* H *abgedeckt wird.*

Die Konzeptbeschreibung H ist also eine Funktion, die für jedes Objekt aus U entscheidet, ob es dem Konzept C angehört oder nicht. Man kann das Erlernen einer Konzeptbeschreibung H auch als Suche im Raum aller Konzeptbeschreibungen auffassen. Die Größe und Art des Raumes der Konzeptbeschreibungen wird von der Konzeptbeschreibungssprache und von der Art des vorhandenen Hintergrundwissens beeinflusst.

2.4.2. Manueller Wissenserwerb

Oftmals wird eine Wissensbasis auch von Hand entworfen. Beim manuellen Wissenserwerb wird das Wissen von einem oder einigen Experten des Anwendungsbereichs akquiriert. Es muss dabei zwischen dem direkten und dem indirekten Erwerb des Wissens unterschieden werden.

Beim direkten Erwerb des Wissens ist der Experte selbst für die Eingabe des Wissens in das wissensbasierte System zuständig. Hierbei ist es eher unwahrscheinlich, dass Fehler bei der Eingabe des Wissens auftreten. Dafür kann es eher passieren, dass das Wissen in einer ungünstigen oder schlecht wartbaren Form eingegeben wird, da der Anwender meist nur Experte auf seinem Fachgebiet, nicht aber im Bereich der Wissensmodellierung (engl. knowledge engineering) ist. In diesem Fall ist es sehr hilfreich, wenn der Experte bei der Eingabe des Wissens umfassend durch das System unterstützt wird, etwa durch Konsistenzprüfungen, graphische Darstellungen der Wissensbasis, etc.

Beim indirekten manuellen Wissenserwerb wird das Wissen bei der Befragung eines Experten durch eine Wissensingenieur (engl. knowledge engineer, KE) akquiriert. Dieses Verfahren ist naturgemäß fehleranfälliger als die direkte Eingabe durch den Experten, da sowohl beim Interview des Experten als auch bei der Eingabe des Wissens durch den Wissensingenieur Fehler eingebracht werden können. Andererseits kann der Wissensingenieur dafür sorgen, dass das Wissen in einer wartbaren und sinnvoll strukturierten Form eingegeben wird und den Anwendungsexperten auf eventuelle Unklarheiten hinweisen.

Es gibt einige Systeme, die den manuellen Erwerb von Wissen unterstützen, indem sie dem Anwender Hinweise darauf geben, welches Wissen vom System benötigt wird, um eine bestimmte Aufgabe lösen zu können bzw. besser als bisher lösen zu können. Eines der ersten Systeme, welches den Wissenserwerb unterstützte, war das System TEIREISIAS [9], einem mit dem wissensbasierten System MYCIN zusammenarbeitendes Wissensakquisitionssystem. MYCIN ist ein System zur Erstellung medizinischer Diagnosen. Wenn MYCIN keine oder eine inkorrekte Diagnose stellt, so kann TEIREISIAS den Anwender bei der Auffindung und Korrektur der verantwortlichen Regel unterstützen. Es führt den Anwender durch die bei der Generierung der Diagnose angewendeten Regeln, um die für den Fehler verantwortliche Regel zu finden. Ebenso unterstützt TEIREISIAS das Hinzufügen neuer

Regeln, da es aufgrund ähnlicher Regeln Vorschläge für weitere Prämissen generiert. Ein Nachteil von TEIREISIAS ist jedoch, dass das System keine Informationen über die Aufgabe der Regeln (z. B. Datenabstraktion, Heuristiken, Ergebnisverfeinerung) nutzt [41].

In der weiteren Entwicklung von Wissensakquisitionssystemen wird immer mehr Wert auf die Einbeziehung der Rolle der Regeln gelegt, die diese bei der Generierung einer Lösung spielen. Wissensbasierte Systeme setzen unterschiedliche Lösungsstrategien für die Erfüllung ihrer Aufgabe ein, z. B. *propose-and-revise* oder *heuristic classification* [40]. Im Rahmen einer Lösungsstrategie haben die einzelnen Regeln jeweils unterschiedliche Aufgaben bei der Bestimmung einer Lösung, wie etwa Generieren einer Lösung oder Verfeinern einer bereits abgeleiteten Lösung.

Zunächst entstanden Systeme wie MORE [12] oder SALT [24]. Diese sind auf den Einsatz einer bestimmten Lösungsstrategie beschränkt und können nur im Rahmen dieser Strategie die Wissensakquisition unterstützen. Mit der Entwicklung von Systemen wie EXPECT [40] wird es möglich, dass die Wissensakquisition für eine vom Anwender wählbare vorgegebene Lösungsstrategie unterstützt wird.

2.4.3. Erstellung von Wissensbasen

Bei der Erstellung von Wissensbasen gibt es verschiedene Vorgehensweisen, die – vor allem beim manuellen Wissenserwerb – Methoden aus der Softwareentwicklung ähnlich und mehr oder weniger gut zur Arbeit mit wissensbasierten Systemen geeignet sind.

Es sollte beim Entwurf eines wissensbasierten Systems wie bei jedem Softwaresystem Wert auf die Punkte

- Strukturiertheit / Modularisierung,
- Übersichtlichkeit,
- Kompaktheit und
- Erweiterbarkeit

gelegt werden, damit das System jederzeit wartbar bleibt und seine Funktionsweise nachvollzogen werden kann.

Im Bereich der Softwaretechnik gibt es viele Vorgehensweisen, die die Entwicklung einer Anwendung in mehrere Phasen strukturieren und die aufeinander aufbauen. Diese Vorgehensweise lässt sich auch auf die Entwicklung wissensbasierter Systeme anwenden.

Der manuelle, also nicht durch automatische Lernverfahren unterstützte Wissenserwerb läuft – wendet man ein Verfahren ähnlich dem zur Entwicklung von Software an – klassisch in den folgenden drei Phasen ab:

- Erstellung einer Spezifikation

- Herstellung eines Entwurfsmodells
- Umsetzung in eine Implementierung

Zunächst wird die Aufgabe des Expertensystems möglichst genau spezifiziert, zum Beispiel durch die Befragung eines Experten. Dies ist vor allem deshalb wichtig, um später die Implementierung anhand der Spezifikation zu validieren (siehe auch Abschnitt 2.5 zur Validierung und Verifizierung).

Die bei der Spezifikation des Systems vom Experten gewonnenen Antworten können nun in ein Entwurfsmodell umgesetzt werden. Hierbei wird die Struktur der Wissensdarstellung festgelegt. Dabei ist darauf zu achten, dass das Wissen in einer Form vorliegt, die für den Benutzer leicht verständlich ist und die einfach wartbar und erweiterbar ist.

Erst wenn das Entwurfsmodell fertiggestellt ist, wird mit der Umsetzung bzw. Implementierung der Wissensbasis begonnen.

Bei dieser Vorgehensweise werden aber einige Punkte außer Acht gelassen, die für die Erstellung und Nutzung eines Expertensystems wesentlichen sind. Man setzt voraus, dass der Experte sein Wissen korrekt und vollständig zum Ausdruck bringen kann. Außerdem wird nicht berücksichtigt, dass sich das Wissen auch noch während der Entwicklung und Nutzung des Systems ändern kann, z. B. wenn Fehler festgestellt werden oder wenn neue Erkenntnisse in die Wissensbasis integriert werden sollen.

Andere Verfahren, die nicht unbedingt auf einem festgelegten Stufenplan basieren – zum Beispiel ähnlich dem *Rapid Prototyping* oder dem *Incremental Development* – erscheinen daher besser auf den Bereich wissensbasierter Systeme übertragbar zu sein. Zusammengefasst werden diese Methoden unter Begriffen wie *agile* oder *evolutionäre* Softwareentwicklung, wie in [45] erläutert wird.

Rapid Prototyping ist ein Vorgehen in der Softwareentwicklung, welches darauf basiert, schon in relativ frühen Phasen der Entwicklung funktionsfähige Prototypen zu entwickeln, die bestimmte funktionale Aspekte der Software demonstrieren, wie zum Beispiel einen Prototypen, der nur eine graphische Benutzeroberfläche darstellt ohne eine tiefergehende Funktion zu haben. Dadurch soll erreicht werden, dass relativ häufig ein Abgleich zwischen den Anforderungen des Auftraggebers und der bisherigen Umsetzung stattfindet und der Fortschritt der Entwicklung so ständig überwacht werden kann.

Das Entwicklungsmodell "Extreme Programming" versucht hingegen, die aus der klassischen Softwareentwicklung bekannten Entwicklungsphasen (Anforderungsanalyse, Systemdesign, Programmierung, Tests) in sehr kurzen zeitlichen Abständen zu wiederholen. Das Entwicklungsziel soll also in vielen kleinen approximativen Schritten erreicht werden, wodurch besonders flexibel auf Änderungen der Anforderungen reagiert werden können soll. Es wird großer Wert auf ständige Tests gelegt, die die korrekte Funktion des Systems gewährleisten sollen.

Diese Vorgehensweisen der Softwareentwicklung lassen sich auf die Entwicklung wissensbasierter Systeme dahingehend übertragen, dass zunächst ein funktionsfähiger Prototyp einer Wissensbasis entwickelt wird, der mit der Zeit um weitere Aspekte ergänzt und modifiziert wird. Ebenso wie in der Softwareentwicklung muss ständig überprüft werden, ob das bereits gespeicherte Wissen durch die Veränderungen und durch die Integration neuen Wissens nicht geändert wurde und die korrekte Funktion der Wissensbasis erhalten wurde. Dies entspricht den in der Softwareentwicklung oftmals eingesetzten Unit-Tests, die nach jedem Entwicklungsschritt automatisiert ausgeführt werden können und die Einhaltung der spezifizierten Eigenschaften der Software garantieren sollen.

Ein Nachteil dieser Vorgehensweise ist die schlechte Planbarkeit der Entwicklung. Gerade in großen Softwareprojekten kann das Erreichen eines bestimmten "Reifegrades" der Software zu einem festgelegten Zeitpunkt nicht garantiert werden. Dies ist für die Auftraggeber einer Softwareentwicklung ein wichtiger Punkt, so dass häufig ein bestimmtes "klassisches" Entwicklungsmodell für einen Auftrag gefordert wird (z. B. Das V-Modell für Aufträge von staatlichen Auftraggebern).

2.4.4. Sloppy Modelling

Betrachtet man den Wissenserwerb in einem wissensbasierten System als den Transfer von einer Repräsentationsform (der persönlichen Repräsentation des Wissens eines Experten) in eine andere (die Repräsentationsform des Expertensystems), so setzt man voraus, dass der Experte bereits eine vollständige Repräsentation des Wissens besitzt und diese explizit ausdrücken kann.

In [27, Abschnitt 1.2.3 f] wird hingegen beschrieben, dass das Wissen einem Experten vielfach nicht explizit bewusst ist. Gerade Experten entwickeln oftmals unbewusste Fähigkeiten, die sie selbst nicht ohne weiteres erklären können. Ein einfaches Beispiel sei hier die menschliche Alltagssprache. Obwohl es kaum jemandem schwer fällt, korrekte Sätze zu bilden, ist es für viele Menschen nicht einfach möglich, zu erklären, nach welchen Regeln ein korrekter Satz gebildet wird.

So wird der Experte erst durch die von einem Interviewer bzw. durch die von der Wissenserwerbskomponente des Systems gestellten Fragen dazu gebracht, eine explizite Repräsentation des Wissens zu bilden. Hierbei kommt es aber oft dazu, dass der Experte schlichte und naive Erklärungen für sein Verhalten findet, die nicht in jedem Fall korrekt sein müssen. Ebenso werden Spezialfälle und Ausnahmen oftmals schlicht vergessen.

Daher ist es in den meisten Fällen nicht möglich, eine Wissensbasis in einem Durchgang von Spezifikation, Entwurf und Implementierung zu erstellen. Statt dessen ist es notwendig, zunächst eine einfache Version der Wissensbasis zu erstellen, sie zu testen und nach und nach um zusätzliche Aspekte zu erweitern. Diese Vorgehensweise wird in [27] als *sloppy modelling* bezeichnet.

Ein wissensbasiertes System muss daher also die folgenden Anforderungen er-

füllen, um für eine am sloppy modelling ausgerichtete Entwicklung geeignet zu sein.

- Die Wissensbasis sollte zu jedem Zeitpunkt veränderbar und erweiterbar sein.
- Es sollte sowohl möglich sein, Regeln zu ändern als auch die zur Erstellung der Regeln verwendeten Elemente zu ändern, zum Beispiel Hinzufügen, Ändern und Löschen von Prädikaten, Änderung der Stelligkeit von Prädikaten, etc.
- Die Wissensbasis sollte überprüfbar sein, d. h. das System sollte auf Fehler und Inkonsistenzen in der Wissensbasis aufmerksam machen.
- Es ist vorteilhaft, wenn Auswirkungen von Änderungen an der Wissensbasis sichtbar gemacht werden können, um zu entscheiden, ob die Änderungen tatsächlich durchgeführt werden sollen.

Auch in anderen Bereichen der Softwareentwicklung halten Hilfsmittel zur Unterstützung von Verfahren ähnlich dem sloppy modelling Einzug. Ein Beispiel dafür sind die Möglichkeiten des *Code Refactoring* in Entwicklungsumgebungen wie Eclipse (siehe [1] und [2]) zur automatisierten Bearbeitung des Sourcecode, die über das einfache Suchen und Ersetzen hinausgehen. Sie transformieren den Sourcecode einer Anwendung anhand fester Regeln und helfen dabei, unerwartete und unerwünschte Auswirkungen zu vermeiden. Ansätze zur Anwendung des Refactorings gibt es aber auch im Bereich der wissensbasierten Systeme, wie zum Beispiel [4] zeigt.

Ein Beispiel für ein System, welches die an der Idee des sloppy modelling orientierte Arbeit unterstützt, ist MOBAL. Das Benutzerhandbuch zu MOBAL [38] nennt als Hauptanwendungen die inkrementelle Modellierung und die Analyse von Wissensbasen. Die Erstellung von Wissensbasen wird unterstützt durch Tools zum Wissenserwerb wie dem Rule Discovery Tool und dem Concept Learning Tool. Die Modifikation von Wissen wird durch das Knowledge Revision Tool unterstützt, welches in der Lage ist, inkorrekte Inferenzen aus einer Wissensbasis zu entfernen. Zur Analyse steht unter anderem das Sort Taxonomy Tool zur Verfügung, welches die in einer Wissensbasis auftretenden Objekte (Konstanten) in Sorten und Klassen von Sorten einteilt.

2.5. Validierung und Verifizierung

Die Überprüfung wissensbasierter Systeme wird oft unter dem Begriff *Validation and Verification* (V&V) zusammengefasst. Darunter versteht man allgemein die Evaluierung der Qualität von Software-Systemen. Es soll nun zunächst definiert

werden, was diese Begriffe ausdrücken und wo die Unterschiede liegen, um dann die in dieser Arbeit genutzten Verfahren entsprechend einzuordnen.

In [6] schafft Boehm eine einprägsame Unterscheidung der beiden Begriffe *Validierung* und *Verifizierung*:

Verification is building the system right.

Validation is building the right system.

Wie auch Preece in [30] erläutert, ist also mit Validierung der Vorgang gemeint, ein System daraufhin zu überprüfen, ob es den vor der Erstellung spezifizierten Anforderungen gerecht wird, ob es also die Wünsche des Auftraggebers korrekt und vollständig erfüllt. Es wird also bei der Validierung überprüft, ob das in dem wissensbasierten System gespeicherte Wissen dem Wissen entspricht, welches der Experte im Kopf trägt. Die Validierung erfolgt in den meisten Fällen anhand von Testdatensätzen, für die bereits bekannt ist, welche Ergebnisse das Softwaresystem dafür liefern soll.

Die Verifikation befasst sich hingegen mit der Fragestellung, ob das System korrekt aufgebaut wurde, d. h. ob die Wissensbasis keine Fehler im Sinne von Fehlern in der Software enthält. In regelbasierten Systemen könnte es sich hierbei zum Beispiel um logische Fehler handeln.

Die beiden Wege zur Überprüfung eines Systems auf Fehler sind nicht unabhängig von einander. Ein System, welches nicht korrekt aufgebaut wurde (bei dem also bei der Verifikation Fehler festgestellt werden), wird höchstwahrscheinlich auch nicht den spezifizierten Anforderungen genügen (welche bei der Validierung des Systems überprüft werden). Der grundsätzliche Unterschied ist darin begründet, dass bei der Verifikation das System für sich alleinstehend (seine innere Struktur) betrachtet wird und versucht wird, Fehler darin zu finden. Die Validierung betrachtet das System von außen, indem überprüft wird, ob das System das beabsichtigte Verhalten zeigt.

Es können also nun die Verfahren der V&V eingeteilt werden in solche, die zur Validierung dienen und solche, die zur Verifizierung dienen. Das in dieser Arbeit zunächst beschriebene Verfahren zur Entdeckung von Widersprüchen in regelbasierten Systemen gehört eindeutig zur Verifikation, da es sich hierbei um ein Verfahren handelt, welches auf Basis der zugrunde liegenden Logik die Regelbasis auf formale Fehler überprüft. Es wird nur die Regelbasis für sich betrachtet und das Verfahren ist weder auf Testdaten noch auf die Bewertung von Ausgaben des Systems angewiesen.

Das Verfahren ist aber auch eng mit der Validierung verbunden, da es zum sinnvollen Einsatz des Verfahrens notwendig ist, Widersprüche, die im Anwendungsbereich existieren, auf logische Widersprüche abzubilden. An diesem Punkt können die Ergebnisse der Validierung eingesetzt werden, um durch die Auswertung von Testdaten Hinweise darauf zu erlangen, welche Widersprüche des An-

wendungsbereichs von Bedeutung für das System sind (siehe auch Abschnitt 3.3 zum Widerspruch in regelbasierten Systemen).

Die Untersuchung der Regelbasis auf Redundanzen lässt sich ebenso als Verifikation auffassen. Auch hier werden keine externen Daten verwendet und nur die interne Struktur der Regelbasis zur Auffindung von Redundanzen genutzt.

Ein Beispiel für ein Verfahren, welches sich der Validierung, nicht aber der Verifizierung zuordnen lässt, ist die Überprüfung der Regelbasis durch Testfälle, deren Resultate bereits bekannt sind. Es wird dabei überprüft, ob das System das gewünschte Verhalten zeigt, nicht aber, ob das System korrekt aufgebaut ist, um dieses Verhalten zu erreichen. Daher ist die Untersuchung der Regelperformanz eindeutig der Validierung zuzuordnen. Es werden Testdatensätze benötigt, die in die Wissensbasis eingegeben werden. Die Ausgaben werden dann durch den Algorithmus zur Aufstellungsberechnung weiterverarbeitet und erst diese Ergebnisse werden dann genutzt, um Aussagen über ein eventuell fehlerhaftes Verhalten der Wissensbasis machen zu können.

2.6. Wissensrepräsentation

Wie bereits in Abschnitt 2.1.1 erläutert wurde, besteht ein wissensbasiertes System unter anderem aus einer Wissensbasis und einer Inferenzkomponente. Im Folgenden wird für regelbasierte Systeme beschrieben, auf Basis welcher theoretischen Grundlagen diese beiden Komponenten arbeiten.

Bevor man eine Wissensbasis erstellen kann, muss zunächst festgelegt werden, wie das in ihr enthaltene Wissen dargestellt werden soll. Die Aussagenlogik ist im Allgemeinen nicht dazu geeignet, das Wissen einer Regelbasis darzustellen, da mit ihrer Hilfe nur Aussagen über einzelne Objekte gemacht werden können, nicht aber über Klassen von Objekten. Es ist aber ein wesentliches Merkmal von wissensbasierten Systemen, das Wissen in abstrakter und allgemeiner Form und nicht auf einen einzelnen Anwendungsfall spezialisiert abzuspeichern.

Deshalb verwendet man in Wissensbasen meist die Prädikatenlogik zur Darstellung des Wissens. Sie kann als Erweiterung der Aussagenlogik betrachtet werden, wobei zusätzlich zur Verknüpfung von Aussagen mit *und* und *oder* auch der Geltungsbereich und die Eigenschaften und Relationen von Objekten betrachtet werden.

Eine Klasse von Objekten und die zwischen ihnen bestehenden Relationen werden dabei durch Prädikatsymbole, ihr Geltungsbereich durch Existenz- und Allquantoren beschrieben. Damit ist es dann zum Beispiel möglich, Aussagen der Form

$$\forall x : pred_1(x) \rightarrow pred_2(x)$$

zu machen, also eine Beziehung zwischen Klassen von Objekten herzustellen.

Solange nur Aussagen über Individuen getroffen werden, spricht man von Prädikatenlogik erster Stufe. Werden hingegen auch Aussagen über Prädikate getroffen, so spricht man von Prädikatenlogik höherer Stufe. Ein Beispiel dafür wäre zum Beispiel die Aussage

$$\exists pred : \forall x : pred_1(x) \rightarrow pred(x)$$

was Umgangssprachlich soviel bedeutet wie "Es gibt eine Klasse *pred*, der alle Individuen angehören, die auch der Klasse $pred_1$ angehören.".

Würde man jedoch die gesamten Möglichkeiten der Prädikatenlogik zulassen, würde die Inferenz, also die Schlussfolgerung von Wissen aus der vorliegenden Wissensbasis nicht mehr rekursiv berechenbar sein. Genau genommen würde die Inferenzkomponente dann einen automatischen Beweiser darstellen.

Aus diesem Grund muss in regelbasierten Systemen die Prädikatenlogik eingeschränkt werden. Üblicherweise wird die funktionsfreie Prädikatenlogik erster Stufe (engl. first order predicate logic, FOPL) verwendet. Das bedeutet, dass in den Regeln nur Literale, jedoch keine Funktionen verwendet werden dürfen.

2.6.1. Nicht-Monotones Schließen

Die bei der Konstruktion wissensbasierter Systeme oft verwendeten, monotonen Logiken haben den Nachteil, dass sämtliches Wissen in einer expliziten Form eingegeben werden muss. Es ist nicht möglich, ein Standardverhalten festzulegen, zu dem dann spezialisierte Ausnahmen definiert werden können. Jeder von der Wissensbasis abgedeckte Fall muss explizit eingegeben werden. Vielen Anwendern fällt es jedoch leichter, Wissen in Form von Default-Regeln und Ausnahmen von diesen Regeln zu formulieren.

Dies spiegelt sich bei der Konstruktion wissensbasierter Systeme in der Frage wider, ob eine monotone oder eine nicht-monotone Logik verwendet werden soll. Eine monotone Logik wird nach [37] wie folgt definiert.

Definition 2 (Monotone Logik) *Eine Wissensbasis B setzt eine monotone Logik ein, wenn für jeden Fakt F, der in B ableitbar ist, gilt, dass er auch von jeder Wissensbasis B_i ableitbar ist, für die $B \subseteq B_i$ gilt.*

In einer nicht-monotonen Wissensbasis ist es somit möglich, dass die Menge ableitbaren Wissens beim Hinzufügen weiterer Fakten zur Wissensbasis sich ändert oder sogar kleiner wird. In solchen Wissensbasen wird oft ein unknown-Operator verwendet, der, zu einer Regel hinzugefügt, dafür sorgt, dass die Regel nur solange angewendet werden kann, wie das im unknown-Operator angegebene Wissen nicht in der Wissensbasis bekannt bzw. ableitbar ist. Damit lassen sich Regeln formulieren, die angewendet werden, solange "nichts genaueres bekannt" ist.

Ein einfaches Anwendungsbeispiel für den Einsatz des unknown-Operators soll hier gegeben werden:

`vogel(V)` → `kann_fliegen(V)`
`pinguin(P)` → `vogel(P)`

Die hier angegebene Regelbasis kommt für die Eingabe `pinguin(tux)` fälschlich zum Ergebnis, dass der Pinguin `tux` fliegen kann. Um die Regelbasis zu korrigieren, kann man nun das bereits bestehende Wissen anpassen, und kommt so zur Regel

`vogel(V)` & ¬`pinguin(V)` → `kann_fliegen(V)`

Der Nachteil bei dieser Korrektur der Regelbasis ist, dass für jede Eingabe `vogel(V)` auch entweder `pinguin(P)` oder ¬`pinguin(P)` angegeben werden muss. Hier bietet der unknown-Operator einen entscheidenden Vorteil.

`vogel(V)` & unknown(¬`kann_fliegen(V)`) → `kann_fliegen(V)`

Wird die erste Regel mit Hilfe des unknown-Operators umformuliert, so erreicht man ein default-Verhalten. Solange nichts darüber bekannt ist, dass ein bestimmter Vogel nicht fliegen kann (Eingabe ¬`kann_fliegen(X)`), so wird angenommen, dass er fliegen kann.

Leider entsteht durch die Verwendung des unknown-Operators ein neues Problem. Es ist nun für die Ausgabe des Systems von Bedeutung, in welcher Reihenfolge die Regeln ausgewertet werden. Ein Beispiel hierfür sind die beiden Regeln

unknown(`A(x)`) → `B(x)`
unknown(`B(x)`) → `A(x)`

Wird zunächst die erste Regel ausgewertet, so wird `B(x)` abgeleitet und `A(x)` kann nicht mehr abgeleitet werden. Wird hingegen die zweite Regel zuerst ausgewertet, so wird `A(x)` abgeleitet und `B(x)` kann nicht mehr inferiert werden. Es gibt also je nach Reihenfolge der Regelauswertung verschiedene Ausgabemengen. Der Umgang mit diesem Verhalten wird unter dem Namen *Answer Set Programming* untersucht, welches hier aber nicht weiter betrachtet werden soll.

2.7. Beschreibung der untersuchten Wissensbasis

Die in dieser Arbeit zur Untersuchung der implementierten Verfahren zur Anomalieentdeckung verwendete Wissensbasis beruht auf der Dissertation "Rechnergestützte Optimierung der Layoutplanung von Chemieanlagen" [23] von Herrn Leuders. Er stellt in seiner Arbeit eine Regelmenge auf, mit der versucht wird, das Wissen von Ingenieuren über die Layoutplanung von Chemieanlagen zu explizieren. Dieses Wissen umfasst die Anforderungen, die für einzelne Komponenten einer Anlage bei der Platzierung innerhalb der Anlage erfüllt sein müssen. Mit dem Begriff *Komponente* sind hier chemietechnische Bauelemente der Anlage wie z.B. Kolonnen, Behälter, Wärmetauscher, Pumpen, etc. gemeint.

Die Anforderungen einer Komponente lassen sich in zwei Gruppen einteilen. Die eine Gruppe umfasst Anforderungen der Komponente an ihren Aufstellungsort, unäre Anforderungen genannt. Dabei handelt es sich im Anforderungen wie *im Erdgeschoss*, *an einem Zugangsweg*, *auf dem Dach*, etc. Die andere Gruppe umfasst binäre Anforderungen, also Anforderungen der Komponente an ihre Nachbarschaft zu anderen Komponenten. Damit sind Anforderungen der Form *Komponente A steht in der Nähe von Komponente B* oder *Komponente A steht oberhalb von Komponente B* gemeint.

Die Anforderungen der Komponenten basieren auf dem Wissen aus verschiedenen Bereichen der Chemietechnik. Es handelt sich dabei um Anforderungen aus der Sicherheitstechnik, Anforderungen aus der Bauart (eine Komponente muss z. B. gewartet werden oder benötigt zur Montage bestimmte Zugangsvoraussetzungen, wie etwa einen Zugangsweg oder die Erreichbarkeit mit einem Kran) und der Betriebsweise einzelner Komponenten (eine Komponente muss etwa Wärme abführen können und benötigt daher ausreichende Lüftung). Daneben gibt es Anforderungen, die auf der Kombination mehrerer Komponenten basieren. Vielfach ist die relative Lage zweier Komponenten zueinander von Bedeutung, die durch Rohrleitungen miteinander verbunden sind. So ist es zum Beispiel zweckmäßig, einen Tank und eine Pumpe, die ein Medium aus dem Tank fördert, so anzuordnen, dass der Tank sich oberhalb der Pumpe befindet, damit das Medium mit Hilfe der Schwerkraft leichter gefördert werden kann. Ebenso besteht die Anforderung, Komponenten die durch Rohrleitungen aus einem besonders teuren Material verbunden sind, möglichst dicht nebeneinander zu positionieren, um so Kosten zu sparen.

Die in der Chemietechnik zunächst erstellten Regeln sind mit einer Gewichtung versehen worden, die beschreibt, mit welcher Priorität eine bestimmte Anforderung erfüllt werden muss. Die Gewichtung wird in dieser Arbeit, insbesondere von dem Verfahren zur Widerspruchsentdeckung, nicht berücksichtigt, da die Gewichtungen in vielen Fällen anscheinend dazu eingesetzt werden, sich widersprechende Anforderungen aufzulösen, anstatt das Wissen dahingehend zu erweitern, dass solche Widersprüche nicht auftreten können. Erst für ein Verfahren zur Berechnung einer Aufstellung können die Gewichtungen von Bedeutung sein. Sollte für eine Komponente nur ein Teil ihrer Anforderungen erfüllbar sein, kann das Platzierungsverfahren diejenigen mit der höchsten Gewichtung zunächst bevorzugen.

Die Regeln aus der Chemietechnik wurden in eine Wissensbasis für das System SIMPLE umgesetzt. Es ist das Ziel, ein System zu entwickeln, welches als Eingabe eine abstrakte Beschreibung einer Chemieanlage bekommt. Die Beschreibung besteht aus den verwendeten Komponenten und deren Verbindungen sowie der für die Anlage zur Verfügung stehende Aufstellungsfläche mit der vorgegebenen Infrastruktur, dem Stahlbau, Zugangswegen und Übergabepunkten. Daraus soll das System die Menge der Anforderungen der Komponenten ableiten, die bei der Generierung einer Aufstellung berücksichtigt werden müssen.

Auf Grundlage der Anlagenbeschreibung und der abgeleiteten Anforderungen kann nun ein geeigneter Platzierungsalgorithmus eine Aufstellung berechnen, die die abgeleiteten Anforderungen zumindest weitgehend berücksichtigt.[1] Ein Beispiel dafür ist der von Frau Köpcke [21] umgesetzte Forward Checking Algorithmus, der in dieser Arbeit auch für die Berechnung der Regelperformanz eingesetzt wird (siehe Abschnitt 4.4.3).

Neben der Einhaltung dieser Anforderungen ist es auch Ziel, eine möglichst kostengünstige Aufstellung zu finden. Daher berechnet der Platzierungsalgorithmus ein Kostenmaß, welches die Einhaltung der durch die Wissensbasis abgeleiteten Anforderungen, aber auch die Gesamt-Rohrlänge aller Verbindungen berücksichtigt und versucht, dieses Maß zu minimieren.

Ergänzend zu den Daten der Wissensbasis zur Layoutplanung stehen die Entwurfsdaten der Chemieanlage "Trieste", einer chemietechnischen Anlage zur Gasvorbehandlung, zur Verfügung. Diese werden verwendet, um die Funktionsweise der Wissensbasis an einem realen Beispiel zu untersuchen. Die Wissensbasis enthält insgesamt 62 Prädikate, mit deren Hilfe 119 Regeln gebildet werden. Diese Regeln greifen auf 43 in der Wissensbasis enthaltene Fakten zu. Mit der Eingabe der Entwurfsdaten der Chemieanlage "Trieste", bestehend aus insgesamt 718 Fakten, werden bei der Ausführung der datengetriebenen Inferenz 80 unäre und 48 binäre Anforderungen abgeleitet.

[1] Ein Algorithmus, der immer die optimale Aufstellung findet, ist nicht effizient umsetzbar, da das zugrundeliegende Platzierungsproblem NP-hart ist

3. Anomalien in regelbasierten Systemen

Bei der Verifizierung eines wissensbasierten Systems wird versucht, Fehler zu finden, die typischerweise in solchen Systemen auftreten und deren Art nicht vom Einsatzbereich abhängt, sondern von dem Aufbau und der Struktur des Systems (engl. domain independent anomalies).

Man kann hier nicht direkt von Fehlern sprechen. Es handelt sich zunächst eher um Anomalien, denn nicht jeder bei der Verifikation gefundene Fehler führt auch bei der Anwendung des Systems zu Problemen und muss daher zwingend beseitigt werden. Anomalien sind aber starke Hinweise darauf, dass das System Fehler enthält. Sie sollten daher genau daraufhin überprüft werden, ob sich der Fehler bei der Verwendung der Wissensbasis auswirkt. Es ist nicht nur wichtig, ob eine Anomalie bei der Anwendung der Wissensbasis auf eine Menge von Eingabedaten die Ausgabe beeinflusst. Es ist genauso wichtig, zu berücksichtigen, ob ein Fehler eventuell bei zukünftigen Änderungen im Rahmen der Pflege und Erweiterung der Wissensbasis Probleme verursachen könnte. So führt zum Beispiel eine redundante Regel zu keinen Fehlern bei der Ableitung von Wissen. Es ist jedoch möglich, dass sie bei zukünftigen Änderungen übersehen wird und dann einen Fehler verursachen könnte.

3.1. Typen von Anomalien

Bei den betrachteten regelbasierten Systemen treten häufig die in den folgenden Abschnitten beschriebenen Anomalien auf, wie auch Preece in [29] erläutert. Es wird an dieser Stelle nur eine Übersicht über die möglichen Anomalien und ihre Eigenschaften gegeben. Eine genauere Erläuterung der Verfahren zur Anomalie-Entdeckung, die auf die in dieser Arbeit betrachteten Wissensbasis angewendet werden, findet sich in Abschnitt 4.4.

3.1.1. Redundanz

Ein Element einer Wissensbasis ist genau dann redundant, wenn für jede erlaubte Eingabe die selbe Ausgabe erzielt wird und zwar unabhängig von der Einbeziehung des redundanten Elements in die Wissensbasis bzw. dem Ausschluss des betrachteten Elements aus der Wissensbasis.

Redundante Elemente einer Wissensbasis führen nicht direkt zu Fehlern. Sie sollten trotzdem vermieden werden. Zum einen können sie die Geschwindigkeit der Auswertung der Wissensbasis verringern. Außerdem kann es bei Änderungen an der Wissensbasis vorkommen, dass redundantes Wissen übersehen wird. Damit können Folgefehler wie zum Beispiel Inkonsistenzen entstehen, da etwa die zu einer zu ändernden Regel redundanten Regeln nicht mit angepasst werden. Auch wird es für den Benutzer schwerer, zu verstehen, warum ein bestimmter Fakt abgeleitet wurde.

Beispiel Gegeben sind die Regeln

`rund(X)` → `kann_rollen(X)`
`kann_rollen(X)` → `ball(X)`
`rund(X)` → `ball(X)`

In diesem Beispiel wird das von der dritten Regel repräsentierte Wissen schon von den ersten beiden Regeln abgedeckt. Die dritte Regel ist damit redundant und kann ohne die Beeinträchtigung der Funktion dieser Regelmenge entfernt werden.

3.1.2. Ambivalenz

Es liegt eine ambivalente Wissensbasis vor, wenn für eine erlaubte Eingabe eine Menge von nicht zulässigen Ausgaben inferiert werden kann. Bei den nicht zulässigen Ausgaben kann es sich um solche handeln, die aufgrund des Anwendungsbereichs nicht sinnvoll oder nicht erwünscht sind. Ebenso kann es sich um Ausgaben handeln, die logisch nicht sinnvoll sind, also zum Beispiel Ausgaben, die einen Fakt und seine Negation gleichzeitig enthalten.

Beispiel Eine erlaubte Eingabe sei {`rund(a)`, `klein(a)`}, gegeben seien die folgenden Regeln:

`rund(X)` → `ball(X)`
`rund(X)` & `klein(X)` → `murmel(X)`

Bei der Anwendung dieser Regeln auf die Eingabe ergibt sich die in diesem Beispiel nicht erlaubte bzw. nicht sinnvolle Ausgabe { `ball(a)`, `murmel(a)`}.

3.1.3. Zirkularität

Eine Wissensbasis ist zirkulär, wenn sie eine Teilmenge von Regeln enthält, deren Konklusion jeweils die Prämisse einer anderen Regel dieser Menge ist.

Beispiel Gegeben seien die Regeln

`rund(X)` → `ball(X)`
`ball(X)` → `rund(X)`

Sobald die Prämisse einer Regel erfüllt wird, erfüllt ihre Konklusion die Prämisse der anderen Regel, so dass es, sofern solche Fälle im Inferenz-Mechanismus nicht behandelt werden, zu einer Endlosschleife kommen kann. Diese Form von Anomalien muss genau daraufhin untersucht werden, ob es sich tatsächlich um zirkuläre Regeln handelt, oder vielleicht eher um eine Rekursion mit einem definierten Abbruchkriterium.

3.1.4. Defizienz

Eine Wissensbasis ist defizient, wenn es eine zulässige Eingabe gibt, für die eine Ausgabe inferiert werden sollte, diese aber nicht inferiert wird. Dies ist ein starker Hinweis darauf, dass das in der Wissensbasis abgelegte Wissen nicht vollständig ist. Es ist dann zu prüfen, ob die von der Wissensbasis nicht abgedeckten Eingaben für den Anwendungsbereich von Bedeutung sind, oder ob eventuell die Menge der erlaubten Eingaben weiter eingeschränkt werden muss.

Beispiel Für die folgenden Regeln wird für jede Eingabe als Ausgabe entweder `rollt(X)` oder ¬`rollt(X)` erwartet. Gegeben sind die Regeln

`rund(X)` → `rollt(X)`
¬`rund(X)` & `eckig(X)` → ¬`rollt(X)`

Wenn nun {¬`rund(a)`, ¬`eckig(a)`} oder {¬`eckig(a)`} erlaubte Eingaben sind, so wird für diese keine Ausgabe erzeugt.

Neben diesen von Preece genannten Anomalien ist auch noch denkbar, dass Teile der Wissensbasis zwar korrekt funktionieren, jedoch nicht zur Erzeugung der gewünschten Ausgabe beitragen.

3.1.5. Inaktive Regeln

Eine Regel R ist inaktiv, wenn mindestens eine der beiden folgenden Voraussetzungen erfüllt ist:

- R besitzt eine Prämisse, die weder durch die Anwendung anderer, nicht inaktiver Regeln, noch durch eine erlaubte Eingabe erfüllbar ist.
- Die Konklusion von R wird weder als Ausgabe verwendet, noch kann durch sie eine Prämisse einer anderen Regel erfüllt werden.

Eine Regel ist also inaktiv, wenn sie weder direkt noch indirekt durch eine erlaubte Eingabe aktiviert werden kann oder weder direkt noch indirekt zu einer relevanten Ausgabe beiträgt. Ebenso kann eine Regel inaktiv sein, wenn ihre Prämissen Bedingungen enthalten, die im Anwendungsbereich der Wissensbasis niemals gleichzeitig erfüllbar sind.

Beispiel Die folgende Regel enthält keinen logischen Fehler, ist jedoch im Anwendungsbereich nicht erfüllbar und somit in der Wissensbasis wirkungslos. Es muss hier von einem Anwendungsbereich mit nur einfarbigen Bällen ausgegangen werden.

`rund(X)` & `rot(X)` & `blau(X)` → `ball(X)`

Solche Regeln bedingen zwar kein inkorrektes Verhalten der Wissensbasis, sie erschweren jedoch deren Wartung und sind zu vermeiden, da sie nicht zur korrekten Funktion der Regelbasis beitragen und deren Wartbarkeit durch die unnötig größere Regelmenge einschränken.

3.2. Hilfen zur Anomalie-Entdeckung

Die Verfahren zur Entdeckung von Anomalien sind alle auf die syntaktische Analyse der Regelbasis angewiesen, da sie die Bedeutung (Semantik) des gespeicherten Wissens nicht erfassen können. Daher existieren einige Methoden, die Wissensbasis mit zusätzlichem Wissen über die Zusammenhänge der enthaltenen Fakten, Regeln und Prädikate anzureichern, sodass die Verifikation erleichtert wird.

3.2.1. Impermissible Sets

Es werden beim Entwurf einer Wissensbasis Mengen von Literalen erzeugt, die nicht gleichzeitig ableitbar und als Ausgabe des Systems erscheinen dürfen. In dem System Eva [8]) werden etwa standardmäßig für alle Prädikate $\mathtt{P}_i/\mathtt{n}$ die Mengen $\{\mathtt{P}_i(\mathtt{X}_1, \ldots, \mathtt{X}_n), \neg\mathtt{P}_i(\mathtt{X}_1, \ldots, \mathtt{X}_n)\}$ als nicht-zulässige Menge definiert. Weitere Mengen können vom Benutzer ergänzt werden. Systeme wie KB-Reducer [14], Covadis [35] und Cover [31] setzen Impermissible Sets dazu ein, Ambivalenzen in Regelbasen aufzudecken.

Die Definition solcher Mengen dient vor allem dazu, Ambivalenzen zu entdecken, die bei einer syntaktischen Überprüfung der Regelbasis alleine nicht zu entdecken wären.

3.2.2. Invarianten

Durch die Definition von Invarianten werden Bedingungen festgelegt, die in der Regelbasis zu jeder Zeit erfüllt sein müssen. Diese können zum Beispiel bei der

Entdeckung von Widersprüchen hilfreich sein. Invarianten sind eng verwandt mit Impermissible Sets, wobei Invarianten sich auf den gesamten Zustand der Wissensbasis beziehen, Impermissible Sets hingegen nur auf die Ausgabe. In dem System EVA [8] etwa können mit Hilfe des Metaprädikats *incompatible* Mengen von Literalen definiert werden, die nicht gleichzeitig ableitbar sein dürfen. Die Einhaltung der Invarianten kann jederzeit durch den "semantic checker" von EVA überprüft werden.

3.2.3. Prädikat-Relationen

Eine weitere Möglichkeit der Überprüfung einer Wissensbasis auf Anomalien wird durch die Verknüpfung von Prädikaten oder Literalen durch Relationen wie *is-a* oder *is-synonym* realisiert. Damit werden Verfahren zur Aufdeckung von Redundanzen in die Lage versetzt, auch Redundanzen aufzudecken, die allein durch die syntaktische Analyse der Wissensbasis nicht zu erkennen sind. So kann zum Beispiel erkannt werden, dass eine Regel allgemeiner als eine andere ist. Damit wird die speziellere von der allgemeinen Regel überdeckt und ist daher redundant. Das Wissen um solche Relationen kann entweder mit Hilfe von Metadaten bekanntgemacht werden oder auch in Form von Regeln in die Wissensbasis integriert werden. Ein Beispiel für den Einsatz solcher Relationen liefert das bereits erwähnte System EVA mit der Relation *incompatible* und der in EVA enthaltene "extended structure checker", der bei der Analyse von Redundanzen *is-a*- und *is-synonom*-Relationen berücksichtigt.

3.2.4. Datentypen

Bei der Definition von Prädikaten ist es möglich, nicht nur deren Stelligkeit, sondern auch den Datentyp der einzelnen Argumente anzugeben. Solche Datentypen können zum einen primitive Datentypen sein, wie sie in vielen Programmiersprachen verwendet werden (Boolean, Integer, Float, String, etc.), aber auch komplexere Datentypen wie Datumswerte oder auch zusammengesetzte Typen. Dies kann dabei hilfreich sein, Fehler bei der Erstellung von Regeln zu vermeiden, wie etwa beim Vergleich von zwei Argumenten, die aufgrund ihres unterschiedlichen Datentyps niemals gleich sein können. Die Definition der Datentypen wird von einigen Systemen auch dazu eingesetzt, die Defizienz von Wissensbasen aufzudecken. So kann zum Beispiel sichergestellt werden, dass bei einem Prädikat, welches ein Argument mit endlichem Wertebereich enthält, alle Werte, die das Argument annehmen kann, in den Prämissen von Regeln überprüft werden. So kann vermieden werden, dass eine Regelbasis für eine erlaubt Eingabe keine Ausgabe liefert.

Einen mit der Auszeichnung der Prädikate mit Datentypen verwandten Ansatz verwendet das System MOBAL [38]. Es stellt das Sort Taxonomy Tool zur Verfügung, welches die Objekte (Konstanten) einer Wissensbasis anhand ihres Auftre-

tens in Literalen in Sorten und Klassen von Sorten aufteilt. Es wird also versucht, anhand der Wissensbasis die Konstanten ihren Datentypen zuzuordnen. Damit erhält der Anwender ein Werkzeug, um zu überprüfen, ob Konstanten nur in den beabsichtigen Literalen verwendet werden.

3.3. Widerspruch in regelbasierten Systemen

Bei der Arbeit mit der Wissensbasis eines regelbasierten Systems stellt man oft fest, dass es durch geringe Modifikationen der Wissensbasis sehr schnell zu Widersprüchen kommen kann. Doch wann genau liegt ein Widerspruch vor? Es lassen sich zwei grundlegende Typen von Widersprüchen unterscheiden. Zum einen gibt es den logischen Widerspruch und zum anderen den Widerspruch innerhalb des Anwendungsgebietes des wissensbasierten Systems.

Ein logischer Widerspruch liegt immer dann vor, wenn aus einer Menge von Eingabe-Fakten unter Verwendung der Regelbasis sowohl ein Fakt als auch die Negation dieses Fakts abgeleitet werden können. Die Widersprüche aus dem Anwendungsbereich eines wissensbasierten Systems treten nicht auf Ebene der logischen Verarbeitung der Eingabe in Erscheinung, sondern erst durch die Interpretation der Daten bei der Verwendung in ihrem Anwendungsgebiet. Beispielhaft dafür ist ein Widerspruch, der bei der Arbeit mit der Regelbasis zur Layoutplanung von Chemieanlagen aufgetreten ist.

Bestimmte Bauteile der Chemieanlagen haben Anforderungen an ihren Standort, die dazu führen, dass sie nur im Erdgeschoss oder auch nur in der obersten Etage aufgestellt werden dürfen. Für manche Bauteile wurden nun die Fakten

```
anforderung_1(K, im_erdgeschoss)
```

und

```
anforderung_1(K, oberste_etage)
```

abgeleitet. Solange die Anlage nicht nur aus einer Etage besteht, ist zu erkennen, dass es sich hier um einen Widerspruch handelt.

Obwohl der Widerspruch für den Anwender leicht ersichtlich ist, kann man diesen maschinell nicht feststellen, solange man nicht das Wissen um diese Widersprüchlichkeit der Regelbasis hinzufügt. Dies kann relativ einfach durch die Eingabe von weiteren Regeln geschehen, in diesem Fall zum Beispiel durch die Regeln

```
anforderung_1(K, im_erdgeschoss) → ¬anforderung_1(K, oberste_etage)
```

und

```
anforderung_1(K, oberste_etage) → ¬anforderung_1(K, im_erdgeschoss)
```

Durch das Hinzufügen solcher Regeln wandelt man einen Widerspruch aus dem Anwendungsgebiet in einen logischen Widerspruch um, der dann mit maschinellen Mitteln bearbeitet werden kann.

In oben gegebenen Beispiel würden mit den zusätzlichen beiden Regeln die Fakten

```
¬anforderung_1(K, oberste_etage)
```

und

```
¬anforderung_1(K, im_erdgeschoss)
```

abgeleitet. Damit würde ein logischer Widerspruch entstehen, der mit maschinellen Verfahren zur Widerspruchsentdeckung aufzudecken ist.

Während dieses Vorgehen Widersprüche bearbeitbar macht, indem zusätzlichen Wissens in die Regelbasis eingefügt wird, welches zunächst nicht für die Anwendung der Regelbasis notwendig ist, setzen einige Systeme ein ähnliches Verfahren ein, welches aber mit Metawissen arbeitet.

Das System EVA setzt zum Beispiel die aus dem Bereich des objektorientierten Entwurfs bekannten Relationen *is-a* und *is-synonym* ein, um Redundanzen in der untersuchten Regelbasis besser aufdecken zu können (vgl. Abschnitt 3.2.3). Genauso könnte auch das Metawissen über den Widerspruch von Literalen in Form einer Relation *contradicts* bekanntgegeben werden. Ein Vorteil dieses Verfahrens ist, dass das Metawissen von den Regeln und Fakten zur Problemlösung getrennt ist und somit die Übersichtlichkeit einer solchen Wissensbasis erhalten bleibt.

Ein großes Problem stellt die Identifikation von relevanten Widersprüchen dar. Während einige Widersprüche wie das oben angegebene Beispiel offensichtlich sind, muss das nicht in jedem Fall so sein. So könnte man es in der Regelbasis zur Layoutplanung von Chemieanlagen auch als Widerspruch auffassen, wenn in einer Etage des Stahlbaus so viele Komponenten stehen sollen, dass der Stellplatz in dieser Etage nicht mehr ausreichend ist. Um einen solchen Widerspruch modellieren zu können, ist etwa das Wissen notwendig, welches auch in einem separaten Algorithmus zur Aufstellungsberechnung implementiert ist. Es ist also abzuwägen, bis zu welchem Grad Widersprüche für die Funktion der Wissensbasis wesentlich sind und ob sie mit vertretbarem Aufwand modellierbar sind. Ebenso schwierig ist die Identifizierung wesentlicher Widersprüche aus dem Anwendungsbereich. Zum Teil werden sie schon während dem Entwurf des Systems offensichtlich, zum Teil können sie aber auch erst bei der Validierung des Systems mit Hilfe von Testdaten entdeckt werden.

3.4. Theorierevision

Werden in einer Wissensbasis Anomalien entdeckt, so müssen diese ausgewertet werden, um festzustellen, ob sie für die Funktion der Wissensbasis relevant sind

und entfernt werden müssen. Zunächst sollte dazu festgestellt werden, ob eine Anomalie negative Auswirkungen auf die Funktionsweise des Expertensystems hat. Ist dies der Fall, so muss sie zwingend behoben werden. Aber auch wenn sich keine direkten negativen Auswirkungen ergeben, sollte überprüft werden, ob die Anomalie indirekt dazu führt, dass bei zukünftigen Änderungen der Wissensbasis die Entstehung von Fehlern begünstigt wird oder die Übersichtlichkeit der Wissensbasis leidet.

Wird bei der Evaluierung einer Anomalie entschieden, dass sie aus der Wissensbasis entfernt werden muss, so stellt sich die Frage, wie dies in einer Weise geschehen kann, die sicherstellt, dass der Teil der Wissensbasis, der nicht von der Anomalie betroffen ist, unverändert bleibt und die Anomalie durch eine möglichst geringe Änderung beseitigt werden kann. Es wird also eine minimale Modifikation der Wissensbasis gesucht, die die Anomalie behebt, aber die sonstige Funktion der Wissensbasis so wenig wie möglich beeinträchtigt.

Die Probleme, die mit dem Entfernen einer Anomalien verbunden sind, hängen von der Art der Anomalie ab. Handelt es sich um redundante Regeln, so kann die Anomalie zum Beispiel dadurch behoben werden, dass alle zu der Regel redundanten Regeln entfernt werden. Da es sich um eine Redundanz handelt, ändert sich durch das Entfernen einer dieser Regeln die Funktion der Wissensbasis nicht. In diesem Fall hat also die Entfernung der Anomalie keine Auswirkungen auf die korrekte Funktion der verbleibenden Wissensbasis.

Handelt es sich bei der Anomalie hingegen um einen Widerspruch, der entfernt werden soll, so ist die Frage nach einer möglichst geringen oder sogar minimalen Änderung der Wissensbasis wesentlich schwieriger zu beantworten. Wird eine Regel verändert oder entfernt, so kann dies Auswirkungen auf andere Bestandteile der Wissensbasis haben, zum Beispiel dann, wenn die Konklusion der entfernten Regel in anderen Regeln als Prämisse genutzt wird. Es könnten dann inaktive Regeln entstehen. Dadurch könnte die Wissensbasis defizient werden. Es würden also durch die naive Entfernung einer Regel neue Anomalien in die Wissensbasis eingefügt.

Im Falle eines Widerspruchs in der Wissensbasis ist es deshalb notwendig, die Anomalie zunächst genau zu untersuchen, um entscheiden zu können, wie sie am besten entfernt werden kann. Zunächst ist nicht klar, welche Ursache der Widerspruch hat. Es ist möglich, dass bei der Bearbeitung der Wissensbasis schlicht das Vorzeichen einer Prämisse oder Konklusion einer Regel vertauscht wurde. Es kann aber auch sein, dass der Widerspruch beabsichtigt ist, jedoch durch eine entsprechende Spezialisierung der an der Ableitung des Widerspruchs beteiligten Regeln sichergestellt werden soll, dass niemals beide Regeln gleichzeitig aktiviert werden können.

Ein Beispiel für den letztgenannten Fall sind die folgenden beiden Regeln, in denen sich die Prämissen `blau(X)` und `rot(X)` (unter der Voraussetzung einfarbiger Bälle) gegenseitig ausschließen. Die Regeln sind also soweit spezialisiert, dass sie

niemals gleichzeitig aktiviert werden können.

`ball(X)` & `blau(X)` → `besitzer(X, hans)`
`ball(X)` & `rot(X)` → ¬`besitzer(X, hans)`

Sofern sich die Prämissen der beiden Regeln derart unterscheiden, dass sie niemals gleichzeitig erfüllbar sind, so handelt es sich nicht um einen für die Funktion der Wissensbasis relevanten Widerspruch. Dies ist aber leider für ein Verfahren zur Widerspruchsentdeckung nicht klar, da das Wissen über den gegenseitigen Ausschluss der Prämissen `blau(X)` und `rot(X)` nicht vorhanden ist. Es ist möglich, dieses Wissen in Form von Metadaten über Prädikate einem Verfahren zur Widerspruchsentdeckung bekannt zu machen, so dass solche Regelmengen nicht mehr als Widerspruch erkannt werden.

Es gibt bereits einige Systeme zur Unterstützung der Theorierevision. Solche Systeme betrachten zum einen das bereits in der Wissensbasis enthaltene und von Experten oder aus Literatur erworbene Wissen und zum anderen empirische Daten. Das System FORTE [34] versucht etwa mit Methoden wie "propositional theory refinement", "first order induction" und "inverse resolution" eine minimale Revision der Wissensbasis zu finden, die auf den gegebenen Beispielen korrekt arbeitet. Weitere Systeme, die die Theorierevision unterstützen, sind EITHER [28], CLINT [33] oder WHY [36].

Ein weiteres Beispiel für ein System, welches in der Lage ist, minimale Änderungen einer Wissensbasis interaktiv mit Unterstützung des Anwenders durchzuführen, ist das System MOBAL [27]. Das darin integrierte Knowledge Revision Tool (KRT) unterstützt den Anwender bei der Revision einer Theorie. Dazu wählt er eine Menge von Fakten aus, die aus der Wissensbasis ausgeschlossen, d.h. die in der Wissensbasis nicht mehr abgeleitet werden sollen. Das KRT berechnet zu diesem Fakt dann mehrere Varianten für eine minimale Revision der Wissensbasis (engl. minimal base revision), die verhindert, dass der widerspruchverursachende Fakt weiterhin abgeleitet wird.

Im folgenden Abschnitt wird die theoretische Formalisierung der Anforderungen an eine Operation zum Ausschluss von Fakten aus einer Wissensbasis beschrieben.

3.4.1. Grundlagen der Theorierevision

Soll eine Wissensbasis – z. B. zur Beseitigung von Anomalien – modifiziert werden, so gibt es drei Operatoren, die dafür auf die Wissensbasis angewendet werden können. Im folgenden wird eine Wissensbasis B als gegeben vorausgesetzt.

Fakt-Entfernung Ein Fakt F soll aus B nicht mehr abgeleitet werden, d. h. es wird eine Funktion $\hat{-}$ mit

$$\hat{-} : B, F \mapsto B'$$

benötigt, die B auf B' abbildet, so dass gilt

$$B' \not\vdash F$$

Konsistentes Hinzufügen eines Faktes Es soll sichergestellt werden, dass ein Fakt F aus B abgeleitet wird und die Wissensbasis konsistent bleibt, d. h. es wird eine Funktion $\hat{+}$ mit

$$\hat{+} : B, F \mapsto B'$$

gesucht, so dass gilt

$$B' \vdash F \text{ und } B' \text{ ist konsistent}$$

Auflösung der Inkonsistenz Es soll sichergestellt werden, dass die Wissensbasis B entweder einen Fakt F oder sein Gegenteil $\neg F$ ableitet, d. h. es wird eine Funktion $\hat{\pm}$ mit

$$\hat{\pm} : B, F \mapsto B'$$

gesucht, so dass gilt

$$B' \vdash F \oplus B' \vdash \neg F$$

wobei das $\oplus$ als "exklusive oder" gelesen werden muss. B soll also so modifiziert werden, dass entweder F oder $\neg F$ abgeleitet wird, aber nicht beide zugleich.

Wie in [43] gezeigt wird, stehen diese drei Funktionen in einem engen Zusammenhang. Die beiden Funktionen $\hat{+}$ und $\hat{\pm}$ können alleine durch die Funktion $\hat{-}$ dargestellt werden. Die Operation $\hat{+}$ (Konsistentes Hinzufügen eines Faktes) kann umgesetzt werden, indem zunächst die Negation des hinzuzufügenden Faktes F mittels $\hat{-}$ entfernt wird und die Wissensbasis dann um F erweitert wird:

$$B \hat{+} F = \left(B \hat{-} \neg F\right) \cup F$$

Weiterhin ist es möglich, die Funktion $\hat{\pm}$ (Auflösung der Inkonsistenz) mit Hilfe der zweiten Funktionen darzustellen. Dazu wird eine Auswahlfunktion $\phi : F \mapsto \phi(F) \in \{F, \neg F\}$ benötigt, die angibt, ob bei der Auflösung der Inkonsistenz entweder F oder $\neg F$ beibehalten werden soll. Dann lässt sich $\hat{\pm}$ ausdrücken als

$$B \hat{\pm} F = B \hat{+} \phi(F)$$

Dass heißt, um eine Theorie zu revidieren, ist es erforderlich, einen Revisionsoperator $\hat{-}$ zu finden, der den Anforderungen des Anwenders genügt. Die wichtigste Anforderung des Anwenders ist dabei sicherlich, die Wissensbasis so wenig wie möglich zu verändern. Diese Anforderung hat Wrobel in [43] als *Minimal Base Revision Operator* formalisiert. Daneben werden die Gärdenfors-Postulate [16] als Basis angesehen, die von jedem Revisionsoperator erfüllt werden sollten ([43]):

- $B \hat{-} F$ ist eine abgeschlossene Theorie
- $B \hat{-} F \subseteq B$ (Inklusion)

- Wenn $F \notin B$, dann $B \hat{-} F = B$
- Wenn $F \notin Cn(\emptyset)$, dann $F \notin B \hat{-} F$ (Erfolg)
- Wenn $Cn(F) = Cn(G)$, dann $B \hat{-} F = B \hat{-} G$ (Erhalt)
- $B \subseteq Cn(B \hat{-} F \cup \{F\})$ (Wiederherstellung)

3.5. Truth Maintenance

Bei einem wissensbasierten System muss damit gerechnet werden, dass sich das darin abgelegte Wissen jederzeit ändern kann. Gründe für Änderungen können nach [37] zum Beispiel sein:

- Es werden Regeln oder Fakten eingegeben, welche aus verschiedenen, sich widersprechenden Quellen stammen.
- Es werden neue Regeln oder Fakten eingegeben, die älteren, bereits gespeicherten Regeln oder Fakten widersprechen.
- Bereits eingegebene Regeln oder Fakten werden ungültig, da sich die von ihnen modellierte Umwelt geändert hat oder da der Ersteller sie nicht mehr für notwendig erachtet.

Erweiterungen eines wissensbasierten Systems, die den Benutzer bei der Revision der Wissensbasis unterstützen können, werden *Truth Maintenance System* genannt. Um auszudrücken, dass kein wissensbasiertes System die Wahrheit (engl. Truth) seiner Ausgabe garantieren kann, nennt Shapiro in [37] hierfür auch den Begriff des *Belief Revision System*. Ein solches System soll einige der folgende Aufgaben erfüllen können:

- Entdeckung von Widersprüchen
- Identifizierung von Fakten und Regeln, die einen Widerspruch verursachen
- Entfernung von widerspruchverursachenden Elementen aus der Wissensbasis, um den Widerspruch zu beheben
- Entfernung derjenigen Ableitungen, die durch die Entfernung eines widerspruchverursachenden Elements ungültig gewordenen und nicht mehr ableitbar sind
- Verhinderung der erneuten Eingabe eines widerspruchverursachenden Elements

Damit diese Anforderungen erfüllt werden können, muss jedes Truth Maintenance System mindestens die folgenden Operationen beherrschen:

- Die von einem Fakt oder einer Regel abgeleiteten Elemente müssen aufgefunden werden können.
- Die Elemente, die zur Ableitung eines Fakts führen, müssen aufgefunden werden können.
- Ein Element der Wissensbasis und aller darauf basierenden Ableitungen müssen aus der Wissensbasis entfernt werden können.

Shapiro nennt in [37] zwei Arten von Truth Maintenance Systemen, die die oben genannten Anforderungen erfüllen. Die Justification Based Truth Maintanance Systems (JTMS) und die Assumption Based Truth Maintenance Systems (ATMS). Die JTMS wurden erstmals 1979 in von Doyle [10] vorgestellt, während die ATMS 1986 in [18] von de Kleer beschrieben wurden. Die Systeme unterscheiden sich hauptsächlich in der Art und Weise, wie sie feststellen, welches Wissen von einem Element der Wissensbasis abgeleitet wurde bzw. welches Wissen der Wissensbasis genutzt wurde, um ein Element abzuleiten.

3.5.1. Justification Based Truth Maintenance Systems

Um Ableitungen zurückverfolgen zu können, wird bei Justification Based Truth Maintenance Systems ein Abhängigkeitsgraph (engl. Truth Maintenance Network, TMN) aufgebaut. Dieser besteht aus Knoten, die ein Literal repräsentieren, und Knoten, die eine Rechtfertigung (engl. Justification) dieses Literals darstellen. Rechtfertigungen sind Knoten der Form

$$J = \langle I | O \rightarrow n \rangle$$

wobei I die Menge der "in"-Knoten ist. Es handelt sich dabei um die Menge der Literale, die in der Wissensbasis ableitbar sein müssen, damit die Folgerung n ableitbar wird. Die Menge O der "out"-Knoten enthält hingegen diejenigen Literale, die in der Wissensbasis nicht ableitbar sein dürfen, damit n abgeleitet werden kann. n ist das Literal, welches Aufgrund der Rechtfertigung J ableitbar wird.

Der Abhängigkeitsgraph enthält gerichtete Kanten. Es verlaufen Kanten von Literalknoten zu einem Rechtfertigungsknoten und von einem Rechtfertigungsknoten zu einem Literal. Die Kanten, die von Literalen zu einem Rechtfertigungsknoten verlaufen, bedeuten, dass die Literale alle zur selben Zeit ableitbar sein müssen (bei in-Knoten) bzw. nicht ableitbar sein dürfen (bei out-Knoten), damit die Rechtfertigung gültig wird. Die Kanten von einem Rechtfertigungsknoten zu einem Literalknoten geben an, dass das Literal ableitbar wird, sofern die Rechtfertigung gültig wird. In-Kanten werden durch ein Plus-Zeichen, out-Kanten durch ein Minus-Zeichen an der entsprechenden Kante gekennzeichnet.

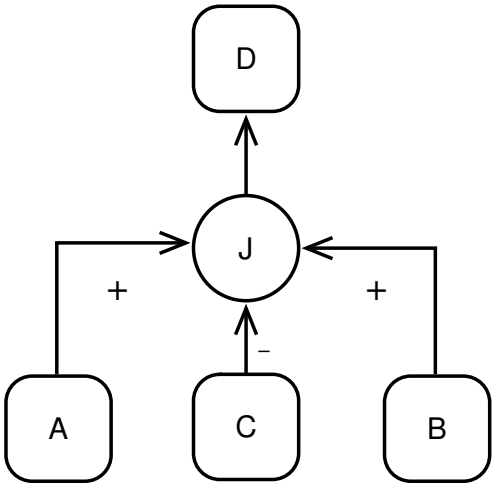


Abbildung 3.1.: Beispiel eines JTMS-Abhängigkeitsgraphen

Das Beispiel eines Abhängigkeitsgraph für eine Wissensbasis, die die Regel

$$A \wedge B \wedge unknown(C) \rightarrow D$$

und die Rechtfertigung

$$J = \langle \{A, B\} | \{C\} \rightarrow D \rangle$$

enthält, ist in Abbildung 3.1 angegeben.

Ist die Menge O aller Rechtfertigungsknotens leer, so handelt es sich um eine klassische, monotone Regel. Ist O hingegen nicht leer, so handelt es sich um eine nicht-monotone Regel, denn die Rechtfertigung wird nur gültig, solange bestimmte Fakten in der Wissensbasis nicht ableitbar sind. Sind sowohl I als auch O leer, so handelt es sich um einen Fakt, denn die Rechtfertigung ist dann unabhängig vom Inhalt der Wissensbasis immer gültig.

3.5.2. Assumption Based Truth Maintenance Systems

In einem ATMS werden wie bei einem JTMS die Abhängigkeiten einer Ableitung von ihren Voraussetzungen aufgezeichnet. Im Gegensatz zu den JTMS wird hier aber an jeder Ableitung eine Liste der Umgebungen (engl. Environments) verwaltet, in denen diese Ableitung gültig ist. Eine Umgebung ist eine Datenstruktur, die einen gültigen Zustand der Wissensbasis beschreibt. Ein Beispiel eines solchen Systems ist SNeBR von Martins und Shapiro [25].

Ein Nachteil der ATMS ist, dass an jedem Knoten eine Liste aller Umgebungen verwaltet wird, in denen dieser Knoten gültig ist. Für n Voraussetzungen gibt es aber 2^n Mengen, die eventuell verwaltet werden müssen. Damit ist dieser Ansatz zunächst nur für exemplarisch kleine Systeme geeignet. Ein Vorteil dieser Systeme ist aber, dass leicht von einer Umgebung zu einer anderen gewechselt werden kann,

da ja immer alle möglichen Umgebungen an einem Knoten vorgehalten werden. Bei JTMS ist es aufwendiger, beim Wechsel von einer Umgebung zu einer anderen die Label aller relevanten Kanten im TMN neu zu berechnen.

3.5.3. Gegenüberstellung mit Simple

Die hier beschriebenen Arten von Truth Maintenance Systems dienen dazu, bei der Arbeit mit einer Wissensbasis ständig zu überwachen, ob Widersprüche entstehen und diese zu verhindern und den Benutzer bei der Beseitigung der Widersprüche zu unterstützen.

Im Gegensatz dazu ist die in dieser Arbeit vorgestellte Software SIMPLE so konzipiert, dass eine Überprüfung der Wissensbasis durch den Benutzer angestoßen wird. Es wird dennoch ein vereinfachtes Truth Maintenance Network verwaltet, welches die von einer Regel abgeleiteten Fakten und die in einer Regel verwendeten Prädikate verwaltet. Dieses Abhängigkeitsnetzwerk kann dazu genutzt werden, bei der Feststellung eines Widerspruchs dem Benutzer die Entscheidung zu erleichtern, wie dieser am besten behoben werden kann. So ist es möglich, jederzeit festzustellen, von welchen Regeln ein Fakt abgeleitet wurde bzw. welche Fakten von einer Regel abgeleitet wurden. Aufgrund der Verwaltung der in den Regeln jeweils verwendeten Prädikaten kann nach einer Änderung der Wissensbasis auch einfach bestimmt werden, welche Regeln neu ausgewertet werden müssen und welche Konsequenzen sich aus der Änderung ergeben.

4. Simple - Ein Framework zur Wartung und Pflege regelbasierter Systeme

Die im folgenden behandelte Software namens SIMPLE[1] ist ein im Rahmen dieser Arbeit erstelltes Framework für die Arbeit mit regelbasierten Systemen. Der Schwerpunkt liegt dabei auf Funktionen, die die Erstellung und Pflege dieser Systeme unterstützen. Insbesondere wurden Verfahren zur Anomalieentdeckung integriert.

Es gibt eine enge Kopplung mit dem von Frau Köpcke implementierten Forward Checking Algorithmus [21] zur Layoutplanung von Chemieanlagen, damit an diesem Anwendungsbeispiel die Funktion der Software gezeigt und untersucht werden kann.

4.1. Aufbau von Simple

Die in der Programmiersprache Java implementierte Software besteht im wesentlichen aus zwei Komponenten. Die eine Komponente ist eine Klassenbibliothek im Java-Namensraum `edu.udo.cs.ai.inference`, die die zur Arbeit mit regelbasierten Systemen wesentliche Funktionalität

- datengetriebene Inferenz und
- zielgetriebene Inferenz

sowie zur Verifikation und Validierung der Wissensbasis Verfahren zur

- Widerspruchsentdeckung,
- Redundanz-Überprüfung und
- Berechnung der Regelperformanz

enthält. Die Klassenbibliothek wird ergänzt durch eine grafischen Benutzeroberfläche im Java-Namensraum `edu.udo.cs.ai.inference.gui` welche die oben genannten Verfahren dem Benutzer zugänglich macht. Diese besteht aus den folgenden, zum Teil als Plugin realisierten Komponenten:

[1] Der Sourcecode der Software ist unter `http://www.a-netz.de/simple.de.php` verfügbar

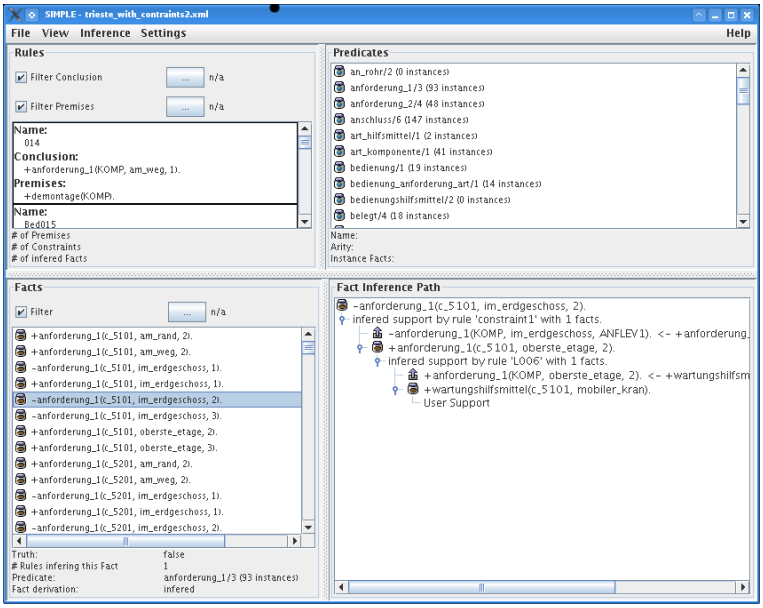


Abbildung 4.1.: GUI mit der Übersicht über die Elemente einer Regelbasis (Logik-Browser)

Logik-Browser Eine Oberfläche, die den Inhalt der Wissensbasis sowie die für die Ableitung eines Fakts verantwortlichen Regeln kompakt darstellt (Abbildung 4.1)

Prädikat-Editor Modul zum Hinzufügen und Löschen von Prädikaten

Fakt-Editor Modul zum Hinzufügen, Ändern und Löschen von Fakten

Regel-Editor Modul zum Hinzufügen, Bearbeiten und Löschen von Regeln (Abbildung 4.2)

Widerspruchsentdeckung Bedienoberfläche des Verfahrens zur Widerspruchentdeckung (Abbildung 4.3)

Redundanz-Untersuchung Darstellung der Ergebnisse des Verfahrens zur Redundanzentdeckung

Logik-Anfrage Beantwortung von Anfragen zur Ableitbarkeit eines Literals mit Hilfe der zielgetriebene Inferenz

CSP-Kontrolle Oberfläche zur Bedienung des Verfahrens zur Berechnung von Aufstellungen mit Hilfe des CSP-Algorithmus von Frau Köpcke.

Regel-Visualisierung Darstellung der Abhängigkeiten zwischen Regeln in Form eines Graphen.

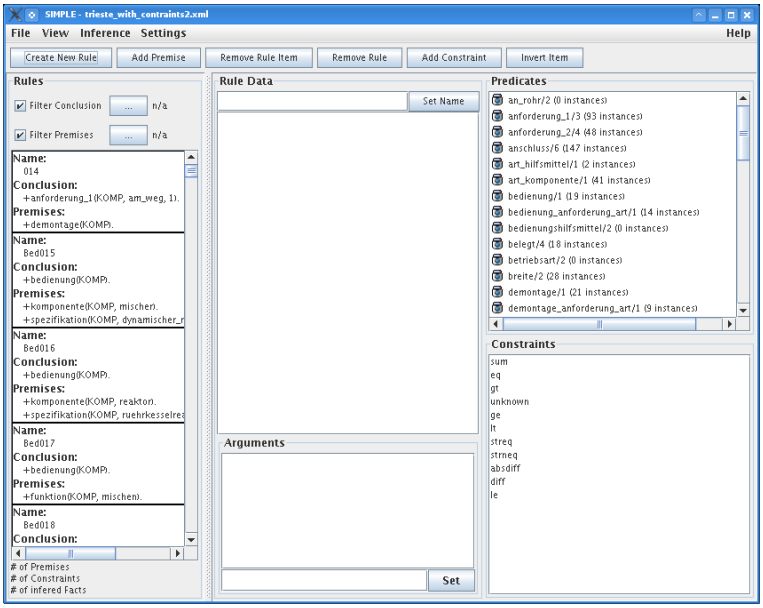


Abbildung 4.2.: Editor zum Bearbeiten der Regeln

Die Trennung der Software in eine Bibliothek von logikbezogenen Funktionen und die Benutzeroberfläche wurde durchgeführt, um die Implementierung der im Folgenden beschriebenen Verfahren (siehe Abschnitte 4.3 und 4.3) auch in anderen Softwareprojekten zu ermöglichen. Sowohl der Quellcode der Benutzeroberfläche als auch der der Bibliothek enthalten Kommentare im Javadoc-Format, die weitere Hinweise zur Verwendung und Funktionsweise geben und so das Verständnis des Quellcodes erleichtern.

Das Framework ist modular aufgebaut und daraufhin ausgelegt, dass es durch weitere Komponenten ergänzt werden kann. Dazu steht ein Plugin-System zur Verfügung, welches es ermöglicht, dass SIMPLE um weitere Komponenten ergänzt wird, die die zugrundeliegende Infrastruktur zur Arbeit mit den Logikdaten nutzen und gegenseitig Dienste zur Verfügung stellen können.

4.1.1. Datenformat

Die Arbeitsdaten – d. h. die Wissensbasis bestehend aus Prädikaten, Regeln und Fakten – können entweder über eine Programmierschnittstelle (engl.: Application Programming Interface, API) an SIMPLE übergeben werden oder aus Dateien im XML-Format eingelesen werden. Eine Definition des verwendeten XML-Formats in Form einer XML Schema Definition nach [39] findet sich im Anhang C. Ein Beispiel für eine einfache Wissensbasis ist in Abbildung 4.4 dargestellt.

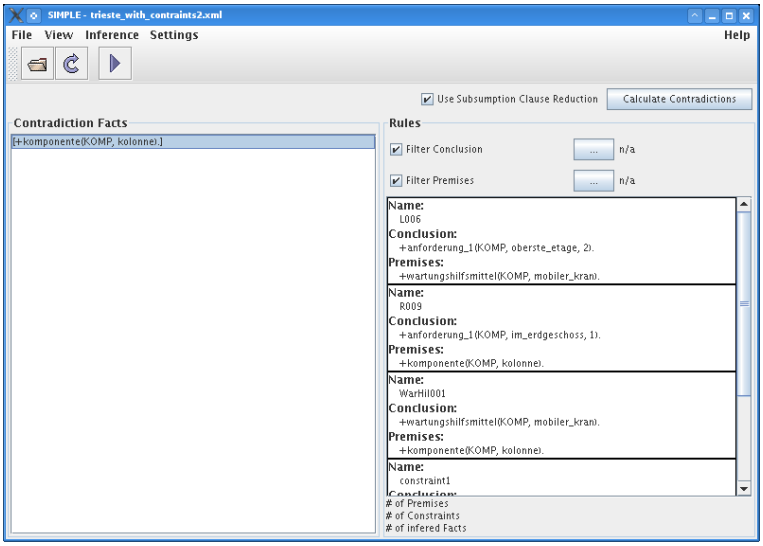


Abbildung 4.3.: Modul zur Widerspruchsentdeckung

4.1.2. Organisation der Logikdaten

Die in einer Wissensbasis enthaltenen Daten werden von SIMPLE in einem Objekt vom Typ `LogicSet` verwaltet. Das LogicSet kann mit Daten gefüllt werden, indem mit Hilfe der Klasse `LogicSetLoader` eine geeignete XML-Datei geladen, oder ein leeres LogicSet erzeugt wird. Diesem können dann mit Hilfe der Methoden

- `createPredicate(...)`,
- `createFact(...)` und
- `createRule(...)`

die logischen Elemente Prädikat, Fakt und Regel hinzugefügt werden. Die Funktionalität und Datenhaltung dieser logischer Elemente wird durch die drei Klassen

- `Predicate` ,
- `Fact` und
- `Rule`

realisiert, deren Vererbungshierarchie in Abbildung 4.5 dargestellt ist. Es ist zu beachten, dass zunächst Prädikat-Objekte erstellt und dem `LogicSet`-Objekt hinzugefügt werden müssen, bevor die darauf basierenden Fakten und Regeln erstellt werden können.

Die den Logik-Objekten zugrunde liegende Basisklasse `LogicItem` unterstützt einen "Tagging"-Mechanismus, der dazu verwendet werden kann, jedes Logik-Objekt mit Metadaten auszuzeichnen. Tags sind beliebige Zeichenketten, von denen

```
<LogicSet>
  <Predicate name="a" arity="2" />
  <Predicate name="b" arity="1" />
  <Predicate name="c" arity="1" />
  <Rule name="regel1">
    <Conclusion predicate="c">
      <Var value="X" />
    </Conclusion>
    <Premise predicate="a">
      <Var value="X" />
      <Var value="Y" />
    </Premise>
    <Premise predicate="b">
      <Var value="X" />
    </Premise>
    <Premise predicate="b">
      <Var value="Y" />
    </Premise>
  </Rule>
  <Fact predicate="a" />
    <Var value="q" />
    <Var value="r" />
  </Fact>
  <Fact predicate="b" />
    <Var value="s" />
  </Fact>
</LogicSet>
```

Abbildung 4.4.: XML-Darstellung einer einfachen Wissensbasis

beliebig viele an ein Logik-Objekt angehängt werden können. Für die Interpretation der Tags sind die einzelnen Module von SIMPLE zuständig. Der Zugriff darauf erfolgt mit Hilfe der Methoden `addTag(...)`, `hasTag(...)` und `removeTag(...)`. Diese Tags werden auch in den XML-Dateien bei der Speicherung einer Wissensbasis persistent abgelegt. Das Verfahren zur Widerspruchsentdeckung nutzt den Tagging-Mechanismus zum Beispiel zur Kennzeichnung von Prädikaten, die den Eingabe- bzw. Ausgabedaten zuzuordnen sind.

Einer alternativer Mechanismus zur Auszeichnung von Logik-Objekten in Form von "Bags" steht ebenfalls zur Verfügung. Hiermit können beliebige Java-Objekte an ein von `LogicItem` abgeleitetes Objekt unter Angabe eines Zugriffsschlüssels (einer Zeichenkette) angehängt werden. Dieser Mechanismus wird etwa von der Regelperformanz-Auswertung genutzt. Die in einem Bag abgelegten Daten werden jedoch nicht persistent gespeichert.

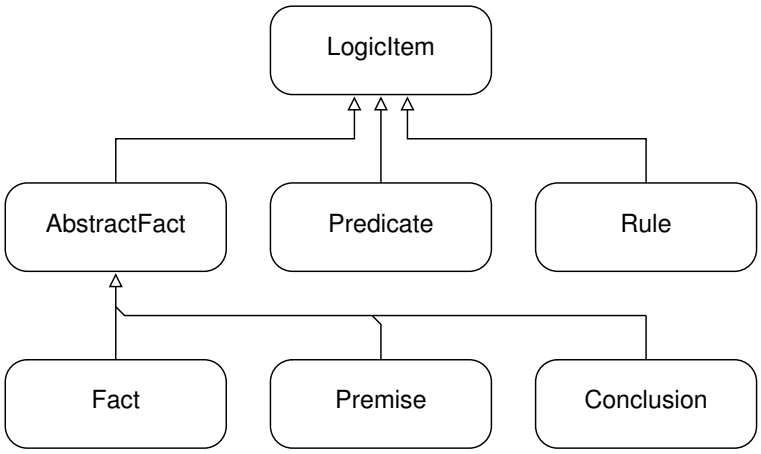


Abbildung 4.5.: Vererbungshierarchie der wesentlichen Logik-Datentypen

Sowohl die beiden implementierten Inferenz-Verfahren als auch die Verfahren zur Entdeckung von Anomalien arbeiten jeweils auf einem `LogicSet`-Objekt und sind damit so gekapselt, dass sie auch unabhängig von der Benutzeroberfläche in anderen Projekten eingesetzt werden können.

4.2. Wissensrepräsentation

In diesem Abschnitt wird die Art der Wissensrepräsentation in SIMPLE beschrieben werden. Wie schon in Abschnitt 2.6 erläutert, ist die Wahl einer angemessenen Wissensrepräsentation von großer Bedeutung sowohl für die Funktion als auch für die Akzeptanz eines Expertensystems. Es wird damit einerseits eine Entscheidungen über die Ausdruckskraft und die Berechnungskomplexität getroffen. Andererseits hat die Wahl der Repräsentation aber auch Einfluss darauf, wie verständlich das Verhalten des Systems für den Anwender und wie aufwendig und verständlich die Formulierung und Eingabe des zu speichernden Wissens ist.

Die in SIMPLE gewählte Repräsentation orientiert sich in wesentlichen Punkten an derjenigen des Inferenzmechanismus IM-2 (siehe [11]) von MOBAL, einem System zum Wissenserwerb und maschinellem Lernen ([27]). Wird im folgenden von einer Wissensbasis gesprochen, ohne genauere Angaben zur Repräsentation zu machen, so ist implizit die Repräsentation entsprechend SIMPLE gemeint.

Die verwendete Notation für Fakten, Regeln und Prädikate entspricht weitgehend der Darstellung dieser Elemente in dieser Arbeit. Eine Ausnahme stellt das Negationszeichen für Literale dar. Literale werden in SIMPLE durch das Voranstellen eines Minus-Zeichens vor das Prädikatsymbol als negiert gekennzeichnet. Die Persistenz-Funktion zur Speicherung einer Wissensbasis über die Programmnutzungsdauer hinaus verwendet einem XML-Dialekt, der sich von der hier verwendeten Notation unterscheidet. Anhand des Beispiels in Abbildung 4.4 und der

XML-Schema-Definition in Anhang C sollte jedoch deutlich werden, wie die XML-Daten zu interpretieren sind.

Ebenso wie in MOBAL wird Wissen in SIMPLE durch Fakten und Regeln repräsentiert. Ein Fakt ist ein Prädikatsymbol zusammen mit einer der Stelligkeit des Prädikats entsprechenden Anzahl von Konstanten, z. B.

```
form(ball, rund)
```

Variablenbezeichner sind dadurch gekennzeichnet, dass sie mit einem Großbuchstaben beginnen, ansonsten handelt es sich um Konstanten. Fakten können auch negiert werden, wie in

```
¬form(ball, eckig)
```

Die Negation wird hier nicht als "negation by failure" interpretiert. "negation by failure" beschreibt das Prinzip, dass die Negation eines Faktes $\neg F$ dann als wahr angenommen wird, wenn der Fakt F anhand der Wissensbasis nicht bewiesen werden kann. Im Gegenteil wird in SIMPLE ein Fakt (auch ein negierter) nur dann als wahr angesehen, wenn er in der Wissensbasis explizit gespeichert ist oder darin abgeleitet werden kann. Damit können zum Beispiel die Fälle auftreten, dass ein Fakt F abgeleitet werden kann, über $\neg F$ aber nichts bekannt ist oder, dass sowohl F als auch $\neg F$ abgeleitet werden können.

Inferentielle Zusammenhänge zwischen Fakten werden in Form funktionsfreier[2] Klauseln als Regeln notiert:

```
rund(B) & zweck(B,zum_spielen) → ball(B)
```

Die in einer Regel vorkommenden Variablen gelten als allquantifiziert. Sowohl die Prämissen als auch die Konklusion einer Regel dürfen negiert sein, daher handelt es sich nicht zwingend um Hornklauseln. Hornklauseln sind Klauseln mit einem positiven Literal, dem Klauselkopf und beliebig vielen negierten Literalen, dem Klauselkörper. Da das "negation by failure"-Prinzip nicht angewendet wird, können negierte Prämissen nur durch explizit in der Wissensbasis ableitbare negierte Fakten erfüllt werden. Ebenso führt eine negierte Konklusion einer Regel dazu, dass ein negierter Fakt in der Wissensbasis ableitbar wird, sobald die Prämissen der entsprechenden Regel erfüllbar sind.

Im Gegensatz zu MOBAL wird in SIMPLE keine Repräsentation von Wissen höherer Ordnung (Metaprädikate, -regeln und -fakten sowie Metametaprädikate und -fakten) umgesetzt.

[2] In Abschnitt 4.2.1 werden Möglichkeiten der Ergänzung von Regeln in SIMPLE um Operatoren und Beschränkungen erläutert. Werden diese verwendet, so sind die Regeln nicht mehr funktionsfrei. In den Verfahren zur Anomalie-Entdeckung wird von dieser Möglichkeit jedoch kein Gebrauch gemacht und daher im Folgenden von funktionsfreien Regeln ausgegangen.

Es wurde genau diese Form der Wissensrepräsentation gewählt, da sie zwei entscheidende Eigenschaften hat: Zum einen ist es performant möglich, eine auf dieser Repräsentation basierende Wissensbasis auszuwerten und zu analysieren. Andererseits ist eine große Ausdruckskraft vorhanden, so dass alle für die hier untersuchte Anwendung in der Chemietechnik wesentlichen Aspekte in einer für den Anwender einfachen Form beschrieben werden können. Ein wesentliches Merkmal ist die mögliche Koexistenz sich widersprechender Fakten in der Wissensbasis. Dies ist eine wichtige Eigenschaft für den Entwurf einer Wissensbasis. Die Wissensbasis bleibt selbst beim Vorhandensein von Widersprüchen auswertbar, da ein Literal und seine Negation in keiner direkten Beziehung zu einander stehen. Erst das Verfahren zur Widerspruchsentdeckung ändert diese Interpretation des gespeicherten Wissens, da es genau diese Widersprüche bearbeitet und dazu eine Beziehung zwischen einem Fakt und seiner Negation herstellt. So ist es möglich, während des Entwurfs der Wissensbasis Widersprüche zuzulassen und diese später mit Hilfe der Widerspruchsentdeckung zu bearbeiten und gegebenenfalls zu entfernen. Während dieses Prozesses ist es auch beim Vorhandensein von Widersprüchen ständig möglich, die Wissensbasis auszuwerten und weiter mit ihr zu arbeiten.

4.2.1. Regel-Erweiterungen

Die Möglichkeiten zur Formulierung von Regeln wurden in SIMPLE gegenüber den Möglichkeiten von MOBAL um zwei Aspekte ergänzt. Es handelt sich um Operatoren und Beschränkungen (engl. constraints). MOBAL unterstützt die Verwendung von Support-Sets, die teilweise den Beschränkungen von SIMPLE entsprechen.

Operatoren

Bei den Operatoren handelt es sich um eine Menge von Literalen, die an die Menge der Prämissen einer Regel angehängt werden können. Sie dienen dazu, bei der Anwendung einer Regel die in den Prämissen gebundenen Variablen in bestimmter Weise zu konvertieren, bevor die Konklusion der Regel abgeleitet wird. Dazu ein Beispiel. Es sei die folgende Regelbasis gegeben:

```
anzahl(ball, 3)
anzahl(bauklotz, 5)
anzahl(ball, B) & anzahl(bauklotz,K) & add(N, B, K) → anzahl(spielzeug, N)
```

Bei dem Literal `add(N, B, K)` handelt es sich um den Additionsoperator, welcher die erste Variable N an die Summe der letzten beiden Variablen, hier B und K bindet. Es wurden folgende Operatoren in SIMPLE implementiert:

`add(N,K,L)` Bindet die Variable N an die Summe der Variablen K und L. Es muss sichergestellt sein, dass es sich bei K und L entweder um numerische

Konstanten oder um Variablen handelt, die nur an numerische Konstanten gebunden werden.

`count(N, P, TV, A`$_1$`, ..., A`$_n$`)` Bindet die Variable N an die Anzahl der Fakten, die mit dem Literal $L =$`P(A`$_1$`, ..., A`$_n$`)` unifizierbar sind. TV kann die Werte *true* oder $false$ haben und entscheidet darüber, ob die Anzahl der Literale L oder $\neg L$ gezählt wird. Die Anzahl der Argumente n muss der Stelligkeit des Prädikats P entsprechen.

Beschränkungen

Angelehnt an die in MOBAL umgesetzten Stützmengen (engl. support sets), welche die Bindungsmöglichkeiten der in den Prämissen vorkommenden Variablen an bestimmte Konstanten beschränken, wurde in SIMPLE mit den Beschränkungen (engl. constraints) ein ähnlicher Mechanismus implementiert.

Beschränkungen können ebenso wie Operatoren an die Menge der Prämissen einer Regel angehängt werden und dienen dazu, die Bindung von Variablen an bestimmte Konstanten zu verhindern. Auch hierzu zunächst ein Beispiel:

```
murmel(m)
rund(m)
rund(b)
rund(B) & strneq(B,m) → ball(B)
```

In diesem Beispiel wird der Gültigkeitsbereich der Regel darauf beschränkt, dass die an die Variable B gebundene Konstante ungleich 'm' sein muss, damit die Regel angewendet werden darf.

Bisher sind in SIMPLE die im Folgenden aufgeführten Beschränkungen implementiert. Zunächst werden diejenigen Beschränkungen aufgeführt, die sich auf numerische Werte (sowohl numerische Konstanten als auch Variablen, die an numerische Konstanten gebunden werden) anwenden lassen. Ihnen gemeinsam ist, dass eine Anwendung auf nicht-numerische Werte nicht möglich ist und zu einem Fehler führt. In der derzeitigen Implementierung können nur ganzzahlige Werte verarbeitet werden.

`sum(S, A`$_1$`, ..., A`$_n$`)` Es wird überprüft, ob gilt

$$S = \sum_{i=1}^{n} A_i$$

`absdiff(D,A`$_1$`,...,A`$_n$`)` Die betragsmäßige Differenz der Argumente A_1 bis A_n muss gleich dem ersten Argument D sein, d. h. es muss gelten

$$D = |A_1 - A_2 - \ldots - A_n|$$

`diff(D, A1,...,An)` Die Differenz der Argumente A_1 bis A_n muss gleich dem ersten Argument sein, d. h.

$$D = |A_1 - A_2 - \ldots - A_n$$

Die folgenden Beschränkungen dienen dem Vergleich von zwei numerischen Werten und der Feststellung, ob es sich um einen numerisch interpretierbaren Wert handelt (`numeric`).

`eq(A, B)` Die beiden Werte A und B müssen numerisch interpretierbar sein und den selben Wert darstellen.

`ge(A, B)` Es muss $A \geq B$ gelten.

`gt(A, B)` Es muss $A > B$ gelten.

`le(A, B)` Es muss $A \leq B$ gelten.

`lt(A, B)` Es muss $A < B$ gelten.

`numeric(A1, ..., An)` Die Werte für A_1 bis A_n müssen numerisch interpretierbar sein.

Die nächsten beiden Beschränkungen dienen dem Vergleich von Zeichenfolgen.

`streq(A1,...,An)` Es muss gelten:

$$\forall i \in \{1, \ldots, n-1\} : A_i = A_{i+1}$$

`strneq(A1,...,An)` Es muss gelten:

$$\forall i \in \{1, \ldots, n-1\} : A_i \neq A_{i+1}$$

Eine Sonderrolle spielt die `unknown`-Beschränkung, da mit ihrer Verwendung die Inferenzprozedur nicht mehr monoton ist (siehe Abschnitt 2.6.1). Die Verwendung des `count`-Operators in Verbindung mit Beschränkungen kann ebenfalls zum Verlust der Monotonie-Eigenschaft führen. Eine Beschränkung der Form `unknown(P, TV, A1,...,An)` ist genau dann erfüllt, wenn in der Wissensbasis kein Fakt ableitbar ist, der von dem Literal L subsumiert wird.

$$L = \begin{cases} P(A_1, \ldots, A_n) \text{ falls } TV = true \\ \neg P(A_1, \ldots, A_n) \text{ falls } TV = false \end{cases}$$

Ein Literal L subsumiert einen Fakt F genau dann, wenn eine Substition σ existiert, so dass gilt

$$L\sigma = F$$

Anomalieentdeckung mit Operatoren und Beschränkungen

Es wurde nicht versucht, Operatoren und Beschränkungen in die Widerspruchsentdeckung von SIMPLE zu integrieren. Auch wenn das Auftreten dieser Elemente in einer Wissensbasis bei der Widerspruchsentdeckung oder Redundanzüberprüfung zu keinem Programmfehler führt, so muss man davon ausgehen, dass die dabei erzielten Ergebnisse nicht korrekt sind.

Die in einer Regel enthaltenen Beschränkungen werden zur Zeit von dem Verfahren zur Widerspruchsentdeckung nicht betrachtet. Damit werden eventuell mehr Widersprüche entdeckt, als tatsächlich vorhanden sind, wie das weiter unten folgende Beispiel von zwei Regeln zeigt, deren Konklusionen sich zwar widersprechen, deren gleichzeitige Aktivierung aber durch die Beschränkungen verhindert wird.

Das Verfahren zur Widerspruchsentdeckung basiert auf der Möglichkeit, für ein Element einer Regel dessen Negation zu bestimmen. Für ein Literal ist das kein Problem, da der Wahrheitswert eines Literals dazu einfach invertiert wird. Für Beschränkungen ist dies wesentlich schwieriger, da es von der Art der Beschränkung abhängt, wie die Negation aussieht. Würden in einer Wissensbasis zum Beispiel die beiden Regeln

`A(X)` & `ge(X, 3)` $\rightarrow$ `D(X)`
`A(X)` & `lt(X, 3)` $\rightarrow \neg$`D(X)`

verwendet, so würde das Verfahren zur Widerspruchsentdeckung zu dem Ergebnis kommen, dass die Eingabe der Literale `A(X)`, `ge(X,3)` und `lt(X,3)` zur Ableitung eines Widerspruchs führen würde. Die Beschränkungen werden also bei der Widerspruchsentdeckung als Literale aufgefasst. Das Verfahren wäre nicht in der Lage, zu erkennen, dass sich `ge(X, 3)` und `lt(X, 3)` gegenseitig ausschließen und daher kein Widerspruch abgeleitet werden kann.

Diesem Beispiel liegt das Problem zugrunde, zu entscheiden, ob eine Menge von Beschränkungen gemeinsam erfüllbar ist oder nicht. Ist sie es nicht, so liegt kein Widerspruch vor. Auch wenn es denkbar ist, dies für bestimmte Mengen von Beschränkungen zu entscheiden, so ist es im allgemeinen Fall für beliebige Beschränkungen nicht entscheidbar. Darüber hinaus ist es fraglich, ob es dann möglich wäre, dem Benutzer in geeigneter Weise darzustellen, welche Eingaben tatsächlich zu einem Widerspruch führen. Schon bei einer widerspruchverursachende Klausel, die die Beschränkungen `sum(2, A, B)` und `sum(2, B, C)` enthält, ist für den Benutzer nur schwer zu erkennen, für welche Eingaben für A, B und C nun tatsächlich ein Widerspruch abgeleitet wird.

Auch das Verfahren zur Aufdeckung von Redundanzen (siehe Abschnitt 4.4.1) ist derzeit nicht in der Lage, Beschränkungen und Operatoren korrekt zu behandeln. Das Verfahren basiert darauf, eine Regel auf Redundanz zu überprüfen, indem man alle nötigen Voraussetzungen dafür schafft, dass die Regel angewendet werden kann und dann diese Regel entfernt. Ändert sich die Menge der abgeleiteten Fakten dabei nicht, so ist die Regel redundant. Es ist für dieses Verfahren also notwendig,

der Wissensbasis (temporär) diejenigen Fakten hinzuzufügen, die es ermöglichen, dass die Regel aktiviert wird. Für normale Prämissen ist dies leicht möglich, indem für jede ein Fakt mit dem selben Prädikatsymbol und durch Konstanten ersetzte Variablen erzeugt wird (Anwendung der Skolem-Substitution).

Werden in der Regel nun Beschränkungen eingesetzt, so ist nicht mehr einfach, eine Substitution der vorkommenden Variablen zu finden, die die Regel aktiviert, denn dazu benötigte das System Kenntnisse über die Bedeutung der verwendeten Beschränkungen. Es müsste auch sichergestellt werden, dass die bei der Skolem-Substitution verwendeten Konstanten den von den Beschränkungen oder Operatoren erwarteten Datentypen (z.b. numerische Konstanten) entsprechen.

Ähnlich verhält es sich mit den Operatoren. Auch hier muss das Verfahren Kenntnisse über die Bedeutung eines Operators haben, um nach der "richtigen" Konklusion fragen zu können, die die Berechnungen in den Operatoren korrekt berücksichtigt. Es reicht in diesem Fall nicht aus, die auf den Prämissen berechnete Skolem-Substitution auf die Konklusion anzuwenden, da durch einen Operator ein Zusammenhang zwischen mehreren Variablen hergestellt wird, der von der Skolem-Substitution nicht berücksichtigt werden kann.

4.2.2. Inferenz

Die Verfahren zur Inferenz (siehe Abschnitt 4.3) in SIMPLE entsprechen in ihrer Funktion dem im Folgenden definierten Ableitungsoperator $\vdash$ (angelehnt an [43, S. 59ff]). Dazu wird zunächst die Definition des 1-Schritt-Ableitungsoperators $\vdash_1$ benötigt:

Definition 3 (Ein-Schritt-Ableitungsoperator $\vdash_1$) *Es sei B eine Wissensbasis und R eine Regel in B mit $R = L_1 \& \ldots \& L_n \rightarrow L_{n+1}$. Es seien $F_1, \ldots, F_n$ Fakten in B. Wenn eine Substitution σ existiert, so dass*

$$\forall i \in \{1, \ldots, n\} : L_i\sigma = F_i$$

gilt, dann gilt auch

$$B \vdash_1 L_{n+1}\sigma$$

Damit ist es nun möglich, die syntaktische Ableitbarkeit eines Faktes zu definieren.

Definition 4 (Syntaktische Ableitbarkeit) *Es sei B^* der Abschluss von B unter $\vdash_1$. Ein Fakt F ist syntaktisch ableitbar in einer Wissensbasis B, d. h. es gilt*

$$B \vdash F$$

genau dann, wenn $F \in B^*$.

4.2.3. Fakt-Korrektheit der Repräsentation

Im Idealfall würden die Inferenzregeln für eine Repräsentation den Operator $\vdash$ so definieren, dass alle Aussagen, die logisch ableitbar sind, auch abgeleitet werden können (Vollständigkeit, engl. completeness) und keine, die nicht logisch ableitbar sind, abgeleitet werden (Korrektheit, engl. soundness).

Ebenso wie in MOBAL sind die Inferenzregeln von SIMPLE zwar korrekt, aber nicht vollständig. Angenommen, eine Wissensbasis enthält die beiden Regeln

`spielzeug(B)` $\rightarrow$ `zweck(B,zum_spielen)`
`rund(B)` & `zweck(B,zum_spielen)` $\rightarrow$ `ball(B)`

so ist auch die Regel

`rund(B)` & `spielzeug(B)` $\rightarrow$ `ball(B)`

logisch ableitbar. Da der von SIMPLE verwendete Inferenzmechanismus keine Regeln ableiten kann, wird diese logisch implizierte Regel nicht abgleitet. Ebenso wird in der Wissensbasis, bestehend aus dem Fakt

$\neg$`ball(b)`

und der Regel

`rund(B)` $\rightarrow$ `ball(B)`

nicht der logisch korrekte Schluss

$\neg$`rund(b)`

gezogen.

Diese Einschränkungen des Inferenzverfahrens sind aber nicht von Bedeutung für deren Anwendbarkeit für praktische Problemstellungen.

Informell kann man sagen, dass die hier verwendete Repräsentation von positiven und negierten Fakten einer Repräsentation entspricht, in der nur positive Literale erlaubt sind und für jedes Prädikatensymbol p ein weiteres Symbol not_p zur Repräsentation negierter Literale eingeführt wird. Dies entspricht dem Verhalten der im SIMPLE verwendeten Repräsentation, in der der Inferenzmechanismus zunächst keine Beziehung zwischen zwei Literalen L und $\neg L$ herstellt. Der Wahrheitswert einer Aussage S ist also völlig unabhängig von dem Vorhandensein der Aussage $\neg S$. Die Herstellung einer Beziehung geschieht allein bei der Anwendung des Verfahrens zur Widerspruchsentdeckung (siehe Abschnitt 4.4.2).

Auch das in SIMPLE implementierte Inferenzverfahren zur zielgetriebenen Inferenz ist daraufhin ausgelegt, mit Inkonsistenzen in der Wissensbasis umzugehen, obwohl es eine Variante der Resolution einsetzt. Die klassische Resolution, welche zum Beweis der Ableitbarkeit einer Anfrage das Gegenteil annimmt und versucht, einen Widerspruch zu erzeugen, würde hier nämlich nicht funktionieren, da in einer

Wissensbasis, die bereits einen Widerspruch enthält, so jede Anfrage positiv beantwortet werden könnte. In SIMPLE wird jedoch eine Variante der SLD-Resolution eingesetzt, die mit widersprüchlichem Wissen umgehen kann.

Eine andere Sichtweise des Verhaltens der verwendeten Repräsentation stellt Wrobel in [43, Abschnitt 3.3.7] vor. Er zeigt, dass sich die verwendete Repräsentation abbilden lässt auf die von Blair und Subrahmanian in [5] vorgestellte "Generally Horn" Semantik:

Definition 5 (Generally Horn) *Es sei* $T_{\sqsubseteq} = \{unknown, false, true, both\}$ *die Menge der Wahrheitswerte. Ein generelle Horn-Klausel ist eine Aussage der Form*

$$L_1 : T_1 \& \ldots \& L_n : T_n \rightarrow L_{n+1} : T_{n+1}$$

mit $n > 0$*, den positiven Literalen* L_i *und* $T_i \in T_{\sqsubseteq}$*.*

Wrobel zeigt, dass sich die verwendete Repräsentation auf die in generellen Hornprogrammen (engl. Generally Horn Programs, GHPS) verwendete abbilden lässt und, dass die Inferenz für die Teilmenge der Repräsentation, in der nur positive Literale in Fakten und Regeln erlaubt sind, vollständig und korrekt ist.

Der Vorteil der in SIMPLE verwendeten Repräsentation von Wissen ist in zwei Punkten zu sehen. Zum einen ist der Inferenzmechanismus dafür ausgelegt, auch auf einer inkonsistenten Wissensbasis zu arbeiten, wie sie leicht bei der Erstellung und Erweiterung von Wissensbasen auftreten kann (siehe auch Abschnitt 2.4 zum Wissenserwerb in Expertensystemen). Außerdem ist es für den Benutzer einfach überschaubar, welche Anfragen an die Wissensbasis korrekt beantwortet werden können (Anfragen nach Fakten) und welche nicht (alle anderen Anfragen).

4.3. Verfahren zur Inferenz

Im folgenden werden die beiden zur Ableitung neuen Wissens aus der Wissenbasis zusammen mit einer Menge von Eingabefakten eingesetzten Verfahren vorgestellt. Es handelt sich dabei um die datengetriebene Inferenz und die zielgetriebene Inferenz durch SLD-Resolution.

4.3.1. Datengetriebene Inferenz

Die datengetriebene Inferenz ist eines der beiden hier verwendeten Verfahren zur Ableitung neuen Wissens aus der Regelbasis zusammen mit Eingabefakten. Die Daten werden hierbei unabhängig von Interessen und Fragestellungen des Benutzers ausgewertet, d. h. es werden solange sämtliche Regeln angewendet, bis keine weiteren Fakten mehr abgeleitet werden können.

Um die Inferenz effizient durchführen zu können, wird bereits beim Laden einer Wissensbasis ein Abhängigkeitsgraph aufgebaut, in welchem bei jedem Prädikat-Objekt die Menge der Regeln verwaltet wird, welche erneut ausgewertet werden

```
depth := 0;
currentRules := allRules;
while (currentRules ≠ ∅) & (depth < maxDepth)
begin
  foreach R ∈ currentRules
  begin
    infFacts := applyRule(R);

    doFactChanges([in] infFacts,
          [out] addFacts,
          [out] delFacts,
          [out] changedPreds);
  end;

  addFacts(addFacts);
  removeFacts(delFacts);

  currentRules := getAffectedRules(changedPreds);
  depth := depth + 1;
end;
```

Abbildung 4.6.: Algorithmus zur datengetriebenen Inferenz in Pseudocode-Darstellung

müssen, sofern sich das Prädikat ändert. Ein Prädikat wird als geändert betrachtet, wenn neue Fakten basierend auf diesem Prädikat instantiiert werden oder bestehende Fakten gelöscht werden. Dies bietet den Vorteil, dass in jedem bis auf den allerersten Inferenzschritt nicht alle Regeln ausgewertet werden müssen, sondern nur solche, in deren Prämissen als geändert markierte Prädikate verwendet werden.

Der Inferenz-Algorithmus (siehe Abbildung 4.6) durchläuft in einer Schleife solange die aufgeführten Schritte, bis sich kein Prädikat mehr ändert. Außerdem wird die äußere Schleife unterbrochen, wenn eine vom Benutzer vorgegebene, maximale Inferenztiefe überschritten wurde. Dies kann zum Beispiel dann passieren, wenn die Regelbasis zirkulär ist (siehe dazu Abschnitt 3.1.3 zu zirkulären Regelbasen).

Bei jeder Regelanwendung werden in der Methode `applyRule()` die von der Regel derzeit ableitbaren Fakten berechnet. Dann werden in `doFactChanges()` diejenigen Fakten berechnet, die noch in der Wissensbasis vorhanden sind, nun aber nicht mehr abgeleitet werden und somit gelöscht werden können (gespeichert in der Menge `delFacts`) und diejenigen Fakten, die bisher nicht in der Wissensbasis vorhanden sind, nun aber von der aktuellen Regel abgeleitet werden können (gespeichert in der Menge `addFacts`). Außerdem wird die Menge der geänderten Prädikate `changedPreds` aktualisiert.

Nach der Anwendung aller Regeln wird mit Hilfe der in `addFacts` und `delFacts`

akkumulierten Änderungen die Wissensbasis aktualisiert. Unter Verwendung der in `changedPreds` gespeicherten, geänderten Prädikate wird die Menge der Regeln `currentRules` berechnet, die im nächsten Schleifendurchlauf der äußeren while-Schleife ausgewertet werden müssen.

Während der Durchführung der Vorwärts-Inferenz wird außerdem in Instanzen der Klasse `FactSupportItem` bei jedem Fakt-Objekt gespeichert, von welchen Regeln es abgeleitet wurde bzw. ob es vom Benutzer eingegeben wurde. Damit lässt sich für den Benutzer übersichtlich darstellen, aufgrund welcher Regeln ein Fakt abgeleitet werden konnte (siehe Abbildung 4.9). Dies dient der einfacheren Pflege und Übersicht über eine Wissensbasis. Ebenso werden an jeder Regel Referenzen auf die von ihr abgeleiteten Fakten gespeichert. Somit werden die grundlegenden Funktionen eines Truth Maintenance Systems – die Bestimmung der von einer Regel abgeleiteten Fakten und der Regeln, die für die Ableitung eines Faktes sorgen – realisiert. Da das Verfahren zur Vorwärtsinferenz auch die "geänderten" Prädikate verwaltet, kann nach dem Hinzufügen oder dem Entfernen eines Elements der neue Zustand der Wissensbasis performant berechnet werden.

Ein Vorteil dieser Form der Inferenz ist, dass mit diesem Verfahren der nichtmonotone Operator *unknown* in der Regelbasis verwendet werden kann. Dieser Operator kann in die Liste der Prämissen von Regeln aufgenommen werden. Mit ihm lassen sich Bedingungen formulieren, die dafür sorgen, dass eine Regel nur dann erfüllbar wird, wenn bestimmtes Wissen *nicht* vorhanden ist (siehe dazu auch Abschnitt 2.6.1). Die folgende Regel ist zum Beispiel nur dann erfüllbar, wenn in der Wissensbasis kein Fakt `kolonne(KOMP)` vorhanden ist.

```
komponente(KOMP,ART) & unknown(komponente, true,KOMP,kolonne)
→ keine_kolonne(KOMP)
```

Bei der Auswertung der Regeln wird nach der Reihenfolge des Auftretens in den Eingabedateien bzw. nach dem Zeitpunkt des Hinzufügens zur Wissensbasis vorgegangen. Es wird nur eine der bei der Verwendung der `unknown`-Beschränkung potentiell mehreren Ausgabemengen gefunden und nicht nach weiteren Mengen gesucht.

4.3.2. Zielgetriebene Inferenz

Die zielgetriebene Inferenz ist dazu geeignet, Anfragen des Anwenders über die Ableitbarkeit (Beweisbarkeit) eines Faktes bzw. einer Menge von Fakten zu beantworten. Im Gegensatz zur datengetriebenen Inferenz wird dabei nicht die gesamte Wissensbasis ausgewertet und alle ableitbaren Fakten erzeugt, sondern nur die zur Beantwortung der Anfrage notwendigen Fakten bewiesen. Damit kann die zielgetriebene Inferenz in vielen Fällen schneller zu einem für den Anwender nützlichen Ergebnis führen.

Die Anfrage eines Benutzers kann aus einem Fakt bestehen, also einem Literal, welches vollständig mit Konstanten besetzt ist. In diesem Fall wird versucht, den

```
function queryFact(Literal query)
begin
  Set<Substitution> allSubsts;
  Set<Rule> rules;

  allSubsts += findDirectMatchingFacts(query);
  rules = getMatchingRules(query);

  foreach(Rule R ∈ rules)
  begin
    // Beschränkt die Variablen aus dem Anfrage-Literal
    // auf die Variablen aus der Konklusion der Regel
    forwardSubst = getUnifikator(R.Conclusion, query);

    Set<Substitution> subSubsts;
    subSubsts = searchSubstitution(depth, R, 0, forwardSubst);

    foreach (Substitution subSubst ∈ subSubsts)
    begin
      if(checkRuleConstraints(depth, R, subSubst))
        allSubsts += applyOperators(subSubst, rule);
    end;
  end;

  return allSubsts;
end;
```

Abbildung 4.7.: Algorithmus zur zielgetriebenen Inferenz in Pseudocode-Darstellung

Fakt zu beweisen oder die Nicht-Beweisbarkeit festzustellen. Wird hingegen ein Literal als Anfrage eingegeben, bei dem manche oder auch alle Argumente mit Variablen besetzt sind, so werden alle Fakten gesucht, die ableitbar sind und die spezieller als das Anfrageliteral sind, d. h. diejenigen Fakten, die von der Anfrage subsumiert werden.

Als Verfahren zur Beantwortung von Anfragen wird die SLD-Resolution eingesetzt. Dabei wird versucht, entweder vom Benutzer eingegebene Fakten oder Konklusionen von Regeln zu finden, die die Anfrage erfüllen könnten. Wird die Konklusion einer Regel gefunden, die die Anfrage erfüllen kann, so werden die Prämissen dieser Regel als neue Anfragen betrachtet, die gemeinsam erfüllt werden müssen, um die erste Anfrage zu beantworten. Dies führt zu einem rekursiv arbeitenden Verfahren, wie es in Abbildung 4.7 dargestellt ist. Es sei dabei B die Wissensbasis.

Die Methode `findDirectMatchingFacts` durchsucht zunächst die Wissensbasis nach Fakten, die die Anfrage *query* direkt beantworten können. Die Überprüfung der Fakten wird in zwei Schritten vorgenommen. Zunächst werden alle Fakten aus der Wissensbasis gesucht, die an allen mit Konstanten besetzten Argumentstellen übereinstimmen. In diesem Schritt werden also auch noch Fakten ausgewählt, die nicht zur Beantwortung der Anfrage geeignet sind; so wird zum Beispiel für die Anfrage `q(B,B)` die nicht korrekte Antwort `q(a,b)` gefunden.

Im zweiten Schritt wird dann für jeden gefundenen Fakt untersucht, ob dieser von dem Anfrageliteral *query* subsumiert wird, d. h. es wird eine Subsitution gesucht, die *query* mit dem gefundenen Fakt unifiziert. Es werden nur die Fakten zur Beantwortung der Anfrage herangezogen, für die eine solche Substitution existiert. Es ist leicht nachvollziehbar, dass für das oben genannte Beispiel eine solche Substitution nicht existiert.

Nun müssen noch alle Regeln daraufhin überprüft werden, ob sie geeignet sind, die Anfrage zu beantworten. Dazu werden in der Methode `getMatchingRules` alle passenden Regeln gesucht. Zunächst wird dazu versucht, die Konklusion jeder Regel mit der Anfrage zu unifizieren. Wird ein Unifikator σ gefunden, so ist die Regel ein Kandidat für die Beantwortung der Anfrage. Die Regel kann die Anfrage natürlich aber nur dann erfüllen, wenn sie selbst erfüllbar ist. Dies wird mit erneuten Anfragen getestet, die mit Hilfe der SLD-Resolution beantwortet werden. Dazu wird zunächst die Substitution σ auf die Prämissen der Regel angewendet und dann versucht, diese gemeinsam zu erfüllen. Dabei wird in einem Backtracking-Verfahren versucht, eine Prämisse nach der anderen zu erfüllen und die Substitution jeweils um die bei der Erfüllung einer Prämisse gebundenen Variablen ergänzt. Sind alle Prämissen gemeinsam erfüllbar, so kann die Regel die Anfrage erfolgreich beantworten.

Im Gegensatz zur datengetriebenen Inferenz werden hier beim Beweis einer Anfrage keine Informationen darüber gesammelt, auf welchem Weg die Anfrage beweisbar ist, also von welchen Regeln und Fakten zur Ableitung der Anfrage führen. Der Grund dafür ist, dass die zielgetrieben Inferenz zum Beispiel bei der Redundanzüberprüfung der Wissensbasis recht häufig eingesetzt wird und daher möglichst performant ablaufen soll.

4.4. Verfahren zur Anomalie-Entdeckung

Die im folgenden vorgestellten Verfahren zur Redundanz-Überprüfung, zur Widerspruchsentdeckung und zur Berechnung der Regelperformanz dienen dazu, eine Regelbasis auf Anomalien hin zu überprüfen, die auf Fehler in der Wissensbasis hindeuten. Das ebenfalls vorgestellte Verfahren zur Auswertung der Regelperformanz wird nicht zur Suche nach Anomalien verwendet, liefert aber ebenfalls Hinweise auf fehlerhafte Regeln.

4.4.1. Redundanz-Überprüfung

Die Redundanz-Überprüfung ist ein Verfahren zur Aufdeckung von Anomalien in der Wissensbasis (siehe Abschnitt 3.1). Die Suche nach redundanten Regeln lässt sich vergleichsweise einfach und effizient durchführen, wie an dem in Abbildung 4.8 dargestellten Algorithmus zu erkennen ist. Das Verfahren basiert auf der in [29] beschriebenen Methode. Dabei wird für jede Regel der Wissensbasis mit Hilfe dieses Verfahrens überprüft, ob die Regel von anderen Regeln überdeckt wird, also redundant ist und damit nicht zur Funktion der Regelbasis beiträgt.

Um festzustellen, ob eine Regel R_i redundant ist, wird diese aus der Regelbasis entfernt. Dann wird für diese Regel eine Skolem-Substitution σ_i berechnet, d. h. eine Substitution, die alle in den Prämissen und der Konklusion der Regel vorkommenden Variablen durch eindeutige Konstanten ersetzt, die an keiner anderen Stelle der Wissensbasis verwendet werden. Nun wird σ_i auf die Prämissen $P_{i,j}$ der Regel angewendet und diese $\sigma_i P_{i,j}$ dann als Fakten (sie enthalten nur noch Konstanten) der Wissensbasis hinzugefügt. Die Wissensbasis enthält nun also garantiert diejenigen Fakten, die sämtliche Prämissen der entfernten Regel R_i erfüllen würden.

Nun wird die Substitution σ_i auf die Konklusion K_i der Regel angewendet und mit Hilfe der zielgetriebenen Inferenz geprüft, ob $\sigma_i K_i$ in der Wissensbasis ableitbar ist. Ist dies der Fall, so ist die Regel R_i redundant, da die Konklusion der Regel trotz Entfernung von R_i noch immer beweisbar ist. Es kann auch keine Ausnahme sein, dass die Konklusion nur mit genau diesen Skolem-Konstanten, nicht aber mit anderen Belegungen der Variablen beweisbar ist, da die für den Redundanz-Test verwendeten Skolem-Konstanten an keiner anderen Stelle der Wissensbasis verwendet werden. Da es sich um funktionsfreie Regeln handelt, kann auch die "Bedeutung" einer Konstante keinen Einfluss auf das Ergebnis der Regelüberprüfung haben.

Ist eine redundante Regel R_i gefunden, so kann diese dauerhaft aus der Regelbasis entfernt werden, da es ein oder mehrere andere Regeln gibt, die die gleiche Information repräsentieren und R_i somit nicht zur Funktion der Wissensbasis beiträgt.

Bei der Überprüfung der Chemietechnik-Wissensbasis wurden mehrere redundante Regeln entdeckt. Die Regelbasis kann somit vereinfacht werden, da diese Regeln nicht zur Funktion beitragen und entfernt werden können.

Ein Beispiel für eine solche redundante Regel ist

`medieneigenschaft(KOMP, feststoffhaltig)` → `wartung(KOMP)`

Das Wissen dieser Regel wird schon von den beiden Regeln

`medieneigenschaft(KOMP, feststoffhaltig)`
→ `wartungshilfsmittel(KOMP, gabelhubwagen)`

und

```
foreach(Rule R_i ∈ B)
begin
  B_i = B \ R_i;
  σ_i = getSkolemSubstitution(R_i);
  skolemFacts = ∅;
  foreach(Premise P_i ∈ R_i.Premises)
  begin
    skolemFacts += σ_i P_i;
  end;

  B_i += skolemFacts;

  redundant = queryFact(B_i, σ_i(R_i.Conclusion));

  if(redundant)
  begin
    redundantRules += R_i;
  end;
end
```

Abbildung 4.8.: Algorithmus zur Redundanz-Überprüfung

`wartungshilfsmittel(KOMP, HM)` → `wartung(KOMP)`

abgedeckt. Die Darstellung des Ableitungsbaums dieser Regeln für die Komponente p_5201 in der Benutzeroberfläche von SIMPLE ist in Abbildung 4.9 zu sehen.

Das Verfahren ist so angelegt, dass es nur diejenigen Regeln findet, die eine Abkürzung eines längeren Ableitungspfades darstellen. Ein Ableitungspfad ist ein Weg in einem gerichteten und-oder-Graphen. Dessen Knoten umfassen die Prämissen und Konklusionen der Regeln sowie "und"-Knoten. Die Kannten verlaufen von den Prämissenknoten einer Regel zu einem "und"-Knoten". Von diesem verläuft eine Kante zur Konklusion der Regel. Ein Beispiel für einen solchen Graphen für die Regelbasis

`a(X)` & `b(X)` → `c(X)`
`c(X)` → `d(X)`
`a(X)` → `d(X)`

ist in Abbildung 4.10 dargestellt. Existiert nun ein Weg in dem Graphen, der eine Abkürzung eines längeren Weges darstellt, so ist die von diesem kürzeren Weg repräsentierte Regel redundant. Sind zwei Ableitungswege gleich lang, beginnen bei denselben Prämissen, enden bei derselben Konklusion und umfassen nur eine Regel, so werden beide Regeln als redundant erkannt. So würden etwa die beiden Regeln

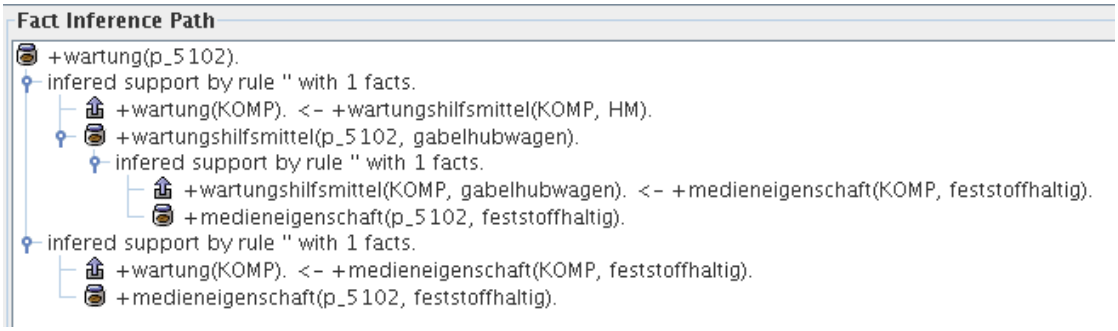


Abbildung 4.9.: Ableitungsbaum mit redundanten Regeln für die Komponente p_5201

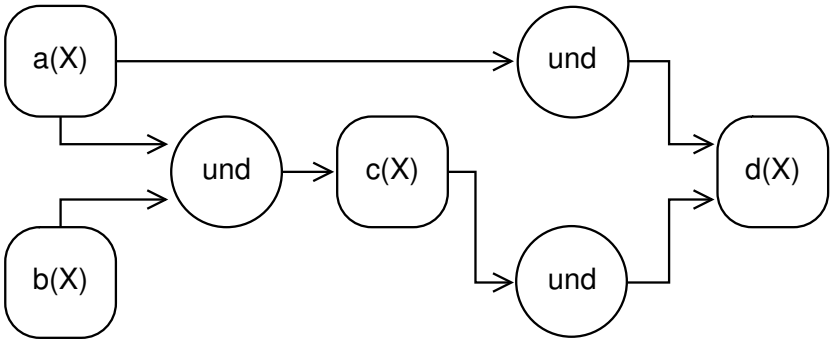


Abbildung 4.10.: und-oder-Regelgraph für eine einfach Regelbasis

`a(X)` → `b(X)`

und

`a(Y)` → `b(Y)`

als jeweils zur anderen redundant erkannt.

4.4.2. Widerspruchsentdeckung

Werden von einer Regelbasis widersprüchliche Ergebnisse abgeleitet, so ist dies ein starker Hinweis darauf, dass das in der Wissensbasis repäsentierte Wissen fehlerhaft ist oder bei der Umsetzung des Wissens in eine Regelbasis Fehler gemacht wurden. Es ist dann im Einzelfall zu klären, ob diese Fehler die gewünschte Funktion beeinträchtigen und somit behoben werden müssen, oder ob sie bei der beabsichtigten Anwendung keine Rolle spielen und daher toleriert werden können.

Die Entdeckung von Widersprüchen steht in engem Zusammenhang mit der Überprüfung einer Regelbasis auf ambivalente Inhalte (siehe Abschnitt 3.1.2). Man spricht von einer ambivalenten Regelbasis, sofern für eine zulässige Eingabe eine

unzulässige Ausgabe erreicht werden kann (siehe auch [29]). Bei solchen unzulässigen Ausgaben kann es sich um logische Widersprüche handeln, d. h. es werden ein Fakt und seine logische Negation gleichzeitig abgeleitet.

Es kann sich aber auch um nicht zulässige Ausgaben in Form von Widersprüchen aus dem Anwendungsbereich handeln. Ein Beispiel dafür wäre der in der Regelbasis zur Layoutplanung von Chemieanlagen aufgetretene Fehler, dass für eine Komponente zwei sich widersprechende Anforderungen abgeleitet wird. Einerseits soll die Komponente im Erdgeschoss, andererseits aber in der obersten Etage des Stahlbaus stehen. Es werden also für eine Komponente k die beiden Fakten

```
anforderung_1(k,oberste_etage)
anforderung_1(k,erdgeschoss)
```

abgeleitet, die Aufgrund des mehrgeschossigen Aufbaus der Chemieanlage nicht gleichzeitig erfüllbar sind. Um solche Widersprüche für einen Algorithmus zur Widerspruchsentdeckung bearbeitbar zu machen, muss in der Regelbasis entsprechendes Wissen über diese Widersprüchlichkeit enthalten sein. In diesem Fall könnte zum Beispiel die Regelbasis um die Regeln

`anforderung_1(KOMP,oberste_etage)` $\rightarrow \neg$`anforderung_1(KOMP,erdgeschoss)`

und

`anforderung_1(KOMP,erdgeschoss)` $\rightarrow \neg$`anforderung_1(KOMP,oberste_etage)`

ergänzt werden.

Wie in Abschnitt 3.3 beschrieben, ist es so möglich, einen Widerspruch aus dem Anwendungsbereich durch zusätzliches Wissen in einen logischen Widerspruch zu konvertieren. Damit wird es möglich, solche Widersprüche mit maschinellen Verfahren zu entdecken und zu bearbeiten. Sind in der Regelbasis logische Widersprüche enthalten, so spricht man auch von der Inkonsistenz der Regelbasis.

Zur Entdeckung logischer Widersprüche wird das von Wu und Su in [44] vorgestellte Verfahren verwendet. Dieses Verfahren liefert Antworten auf die folgenden Fragestellungen:

- Ist die Regelbasis inkonsistent?
- Welche Regeln sind für die Inkonsistenz verantwortlich?
- Unter welchen Umständen bzw. bei welchen Eingaben werden widersprüchliche Fakten abgeleitet?

Es ist an dieser Stelle wichtig, genau festzulegen, was mit dem Begriff *Inkonsistenz* gemeint ist, oder genauer, was der Unterschied zwischen dem in der formalen Logik und dem in Bezug auf regelbasierte Systeme verwendeten Begriff ist. In der formalen Logik wird eine Menge von Aussagen S als konsistent betrachtet, wenn

es *mindestens eine* Möglichkeit gibt, diese so zu interpretieren, dass es zu keinem Widerspruch kommt. Daher ist in der formalen Logik zum Beispiel die Menge der Aussagen

r_1: P $\rightarrow$ Q

r_2: P $\wedge$ R $\rightarrow \neg$Q

konsistent, da, falls P falsch ist, beide Aussagen wahr werden. Damit ist $\{\neg P\}$ ein Modell für S. Offensichtlich ist eine Regelbasis, die diese beiden Aussagen enthält, nicht hilfreich, da das System bei der Eingabe {P, R} die widersprüchliche Ausgabe $\{Q, \neg Q\}$ liefern würde. Dieses Beispiel zeigt, dass der in der Bedeutung der formalen Logik verwendete Begriff *Inkonsistenz* für den Bereich der regelbasierten Systeme modifiziert, genau genommen verschärft werden muss.

Daher definieren Ping Wu und Stanley Su in [44] die Inkonsistenz einer Regelbasis wie folgt:

Definition 6 (Inkonsistenz einer Wissensbasis) *Eine aus einer Menge von Regeln und Fakten bestehende Wissensbasis ist genau dann inkonsistent in Bezug auf eine Menge von Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$, wenn bei der Eingabe von $\{\alpha_1, \ldots, \alpha_n\}$ ein Widerspruch, d. h. ein Fakt und seine Negation abgeleitet werden können. Eine Wissensbasis ist konsistent, genau dann, wenn sie nicht inkonsistent ist.*

Das von Wu und Su vorgestellte Verfahren setzt die Level Saturation Resolution (LSR) für eine Klauselmenge S ein, um eine Regelbasis auf Inkonsistenz zu überprüfen. Die LSR wird dabei so definiert, dass mit dem folgenden Verfahren solange Resolventen aus der Menge S generiert werden, bis entweder eine leere Klausel abgeleitet wird oder nur noch bereits bekannte Klauseln abgeleitet werden. Formale Definition der LSR:

$$S_0 = S$$

$$S_n = \{\text{Resolventen von } C_1 \text{ und } C_2 | C_1 \in (S_0 \cup \ldots \cup S_{n-1})$$

$$\text{und } C_2 \in S_{n-1}\}, n = 1, 2, 3, \ldots$$

Es werden also in jedem Schritt n alle Resolventen aus $(S_0 \cup \ldots \cup S_{n-1})$ und S_{n-1} der Menge S_n hinzugefügt, bevor das nächste "Level" S_{n+1} gebildet wird. Damit ist sichergestellt, dass zunächst alle Zwischenschritte durch Resolution abgeleitet werden, bevor eventuell eine leere Klausel $\square$ resolviert wird.

Ein Nachteil dieses Verfahrens ist, dass zunächst viele irrelevante und redundante Klauseln abgeleitet werden. So kann zum Beispiel eine Tautologie generiert werden.

Da eine Tautologie immer wahr ist, kann sie aus einer Menge unerfüllbarer Klauseln entfernt werden, ohne dass sich die Unerfüllbarkeit der Klauselmenge ändert.

Ebenso sollten keine Klauseln mehrfach generiert werden und auch keine Klauseln, die von anderen Klauseln subsumiert werden. Die Klauselsubsumption ist dabei wie folgt definiert:

Definition 7 (Klauselsubsumption) *Eine Klausel C subsumiert eine Klausel D genau dann, wenn eine Substitution σ existiert, so dass $C\sigma \subseteq D$. D wird subsumierte Klausel genannt.*

Um den Aufwand bei der Resolution so gering wie möglich zu halten, werden sowohl Tautologien als auch subsumierte Klauseln während der Resolution entfernt. Wie in [19] und [7] gezeigt wurde, führt das Verfahren auch bei Anwendung dieser Lösch-Strategie korrekt zur Ableitung der leeren Klausel, falls die Eingabeklauselmenge S unerfüllbar ist. Wird die leere Klausel nicht abgeleitet, so ändert sich die Menge der von diesem Verfahren entdeckten widerspruchsverursachenden Eingaben ebenfalls nicht.

Nun ist aber noch nicht klar, welcher Zusammenhang zwischen der Unerfüllbarkeit der Klauselmenge S und der Inkonsistenz der zugrundeliegenden Regelbasis zusammenhängt. Um diesen Zusammenhang zu zeigen, werden die folgenden beiden Theoreme benötigt.

Theorem 1 *Ist C ein Resolvent der beiden Klauseln C_1 und C_2, so ist C eine logische Konsequenz von C_1 und C_2.*

Theorem 2 (Vollständigkeit der Resolution) *Eine Menge von Klauseln S ist unerfüllbar genau dann, wenn aus dieser Menge die leere Klausel $\square$ geschlussfolgert werden kann.*

Zu zeigen ist nun das folgende Theorem, welches etwas über den Zusammenhang zwischen der Unerfüllbarkeit einer Klauselmenge und der Inkonsistenz der Regelbasis aussagt.

Theorem 3 *Eine Regelbasis, bestehend aus einer Menge von Regeln R, ist inkonsistent in Bezug auf eine Menge von Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ genau dann, wenn R zusammen mit den Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ eine unerfüllbare Klauselmenge darstellt.*

Beweis: Teil 1: Sei B eine Regelbasis, bestehend aus Regeln $R = \{r_1, \ldots, r_m\}$, die inkonsistent in Bezug auf die Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ sind. Dann lassen sich aus $R \cup \{\alpha_1, \ldots, \alpha_n\}$ mindestens zwei Literale β und $\neg\beta$ ableiten. Mit Hilfe der Resolution wird aus β und $\neg\beta$ die leere Klausel $\square$ abgeleitet. Damit ist nach Theorem 2 die Klauselmenge $R \cup \{\alpha_1, \ldots, \alpha_n\}$ unerfüllbar.

Teil 2: Wenn Die Klauselmenge $R \cup \{\alpha_1, \ldots, \alpha_n\}$ unerfüllbar ist, so wird entsprechend Theorem 2 die leere Klausel $\square$ resolviert. Die leere Klausel kann aber nur aus zwei Literalen β und $\neg\beta$ resolviert werden. Das heißt, β und $\neg\beta$ können aus den Regeln R zusammen mit den Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ abgeleitet werden, woraus folgt, dass die Regeln R inkonsistent in Bezug auf die Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ sind.

Damit ist der Zusammenhang zwischen einer unerfüllbaren Klauselmenge und einer inkonsistenten Regelbasis klar. In regelbasierten Systemen ist es oftmals so, dass die Regelbasis aus formallogischer Sicht erfüllbar ist, in Bezug auf eine Menge von Eingabefakten jedoch inkonsistent ist und Widersprüche abgeleitet werden können. Außerdem kann man aus der Erfüllbarkeit jeder von zwei Klauselmengen (einer Regelbasis und einer Menge von Eingabefakten) nicht darauf schließen, dass die Vereinigung der beiden Mengen ebenfalls erfüllbar ist (vgl. das Beispiel der zwei Regeln r_1 und r_2 am Anfang dieses Abschnitts).

Die Überprüfung einer Regelbasis R auf Inkonsistenz in Bezug auf eine Menge von Eingabefakten $\{\alpha_1, \ldots, \alpha_n\}$ kann damit zurückgeführt werden auf das Problem, die Unerfüllbarkeit von $R \cup \{\alpha_1, \ldots, \alpha_n\}$ zu zeigen. Ein Problem bei dieser Vorgehensweise ist, dass die Menge $\{\alpha_1, \ldots, \alpha_n\}$ zunächst unbekannt ist. Ein Ansatz, alle für die Regelbasis relevanten Mengen von Eingabefakten zu konstruieren, z. B. indem sämtliche Teilmengen der Prämissen aller Regeln gebildet werden, scheidet aufgrund des hohen Berechnungsaufwands aus.

Die Vorgehensweise ist daher hier eine andere. Zunächst wird auf der Regelbasis in Klauselform die Level Saturation Resolution (LSR) solange durchgeführt, bis keine weiteren Klauseln mehr abgeleitet werden können. Während der LSR werden die durchgeführten Resolutionen aufgezeichnet, so dass später festgestellt werden kann, aus welchen Regeln eine Klausel resolviert wurde.

Wird schon während der LSR eine leere Klausel abgeleitet, so ist die Regelbasis in sich widersprüchlich und es kann mit Hilfe der aufgezeichneten Ableitungen festgestellt werden, welche Regeln an der Ableitung des Widerspruchs beteiligt sind.

Es soll nun die Vorgehensweise bei der Entdeckung von Widersprüchen vorgestellt werden. Es sei $R = \{r_1, \ldots, r_n\}$ die Menge der Regeln und Fakten der Wissensbasis in Klauselform. Die Menge K enthält diejenigen Klauseln, die später zur Berechnung der widerspruchverursachenden Fakten verwendet wird. Der hier verwendete Begriff der Unitklausel bezeichnet Klauseln, die genau ein Literal enthalten.

1. Durchführung der LSR auf der Menge R, bis keine weiteren Ableitungen mehr gefunden werden können.

2. Wird eine leere Klausel gefunden, so wird dies ausgegeben und die Widerspruchsentdeckung beendet. Die Regelbasis ist in sich widersprüchlich.

3. Füge der Menge K alle Unitklauseln hinzu, deren Literal in den Prämissen einer Regel enthalten ist.

4. Füge der Menge K alle Klauseln hinzu, die aus mehr als einem Literal bestehen und

   - deren Literale alle in den Prämissen von Regeln enthalten sind und
   - die nicht von einer anderen Klausel aus K subsumiert werden.

5. Konstruiere für jede Klausel aus K eine resolutions-komplementäre Formel, d. h. eine Formel f_i, die mit der Klausel aus K mittels Resolution die leere Klausel ergibt. Die Menge $\{f_i\}$ ist die Menge der potentiell einen Widerspruch verursachenden Eingaben.

Erhält man als Ergebnis eine leere Menge K und es wurde während der LSR keine leere Klausel generiert, so ist die Menge der Regeln R konsistent.

Jede Formel f_i repräsentiert eine Menge von Eingabefakten, die zur Ableitung eines Widerspruchs führen. Durch die Zurückverfolgung der aufgezeichneten Resolutionsschritte der mit f_i korrespondierenden Klausel $k_i \in K$ lässt sich feststellen, welche Regeln an der Ableitung von k_i beteiligt sind und bei Eingabe der Literale aus f_i einen Widerspruch generieren.

Damit können mit diesem Verfahren die drei zu Anfang dieses Abschnitts gesetzten Ziele *Entdeckung von Widersprüchen*, *Identifizierung der widerspruchserzeugenden Eingabefakten* und *Identifizierung der an der Ableitung eines Widerspruchs beteiligten Regeln* erfüllt werden.

Ein Nachteil der hier angewendeten LSR ist, dass zunächst eine große Menge von Klauseln gebildet wird, bevor das Verfahren entweder zu dem Ergebnis kommt, dass die Regelbasis keinen Widerspruch enthält oder ein solcher identifiziert wird. Obwohl es andere, effizientere Resolutions-Verfahren (z. B. SLD-Resolution) gibt, ist die LSR das hier am besten geeignetste Verfahren für die Widerspruchsentdeckung. Der Vorteil liegt in der aus der Verwendung von SLR resultierenden Vollständigkeit der Widerspruchsentdeckung.

Definition 8 (Vollständigkeit) *Ein Algorithmus zur Widerspruchsentdeckung wird dann als vollständig bezeichnet, wenn er alle Kombinationen von Eingabefakten identifizieren kann, die zur Ableitung eines Widerspruchs führen. Er muss für jede dieser Kombinationen die Menge der Regeln identifizieren können, die an der Ableitung des Widerspruchs beteiligt sind.*

Das hier vorgestellte Verfahren zur Widerspruchsentdeckung ist nach dieser Definition vollständig. Die LSR garantiert, dass jedes mögliche Paar von Klauseln der Regelbasis und bereits abgeleiteten Klauseln genutzt wird, um neue Ableitungen zu bilden. Daher werden allen möglichen Ableitungen auch generiert, woraus folgt, dass in der Menge K sämtliche Klauseln enthalten sind, die zusammen mit den

entsprechenden Eingabefakten zur leeren Klausel resolviert werden können. Ebenso ist klar, dass die Regelbasis widerspruchsfrei sein muss, wenn die Menge K beim Halt des Verfahrens leer ist und nicht vorher die leere Klausel resolviert wurde.

Es ist zu klären, ob die eingesetzte LSR unter allen Umständen terminiert. Aus dem Bereich des *Mechanical Theorem Proving* ist bekannte, dass ein Theorem-Beweiser nicht erfolgreich terminiert, wenn die zugrunde liegende Theorie kein Theorem ist, also die gegebene Menge der Klauseln nicht erfüllbar ist. Im Gegensatz dazu kann bei dem hier eingesetzten Verfahren garantiert werden, dass es in endlicher Zeit terminiert:

Beweis: Da die Menge der Regeln endlich ist, ist auch die Menge der darin vorkommenden Literale endlich. Es sei N die Menge der in den Regeln vorkommenden Atome (F und $\neg F$ zählen als ein Atom). Durch die Entfernung von Tautologien und subsumierten Klauseln kann die Anzahl der Atome in einem Resolventen nicht größer als N werden. In jedem Resolutionsschritt werden die Literale entfernt, auf denen die Resolution ausgeführt wird. Daher nimmt die Anzahl der Literale immer weiter ab, bis entweder die leere Klausel resolviert wird oder keine weiteren Resolventen mehr generiert werden können, die nicht bereits von anderen Resolventen subsumiert werden. In jedem Fall terminiert das Verfahren an dieser Stelle.

Werden andere, eventuell effizientere Resolutionsverfahren verwendet, so kann laut [44] die Vollständigkeit des Verfahrens nicht mehr garantiert werden. Es existiert dann also ein Trade-off zwischen der Performanz des Verfahrens und dessen Vollständigkeit.

4.4.3. Berechnung der Regelperformanz

Im Gegensatz zu den Verfahren zur Widerspruchs- und Redundanzentdeckung soll nun ein Verfahren zur Validierung vorgestellt werden. Es macht sich die Ergebnisse eines Algorithmus zur Berechnung einer Aufstellung zu nutze, um die Regeln der Wissensbasis zu bewerten. Im Gegensatz zu den bisher vorgestellten Verfahren handelt es sich hierbei um eine Validierung der Wissensbasis anhand realer Eingabedaten. Ein solches Verfahren kann nur in dem Rahmen Aussagen über die Wissensbasis liefern, wie es die zur Validierung eingesetzten Testdaten ermöglichen.

Das Verfahren basiert darauf, dass zunächst für eine Testeingabe die Anforderungen der einzelnen Komponenten abgeleitet werden und darauf dann der Algorithmus zur Berechnung einer Aufstellung ausgeführt wird. Bei dem hier verwendeten Algorithmus handelt es sich um den in [21] vorgestellten, auf dem *Constraint Satisfaction Problem* basierenden Algorithmus. Dieser Algorithmus ist in der Lage, nach der Berechnung einer Aufstellung für jeden Constraint dessen Erfüllungsgrad auszugeben.

Für jede Regel, die direkt einen Constraint ableitet, wird nun die Menge der von ihr abgeleiteten Constraints bestimmt und für jeden dieser Constraints sein

Erfüllungsgrad. Diese Ergebnisse werden dann zu einem Performanzwert für jede Regel zusammengefasst.

Formal lässt sich dieses Vorgehen wie folgt beschreiben: Es sei R die Menge der Regeln, die einen Constraint ableiten, C die Menge der Constraints und C_j die Menge der von Regel $r_j \in R$ abgeleiteten Constraints. Die Funktion $p_c : C \mapsto [0,1]$ liefert für jeden Constraint $c_i \in C$ dessen Erfüllungsgrad bei der Anwendung des Platzierungsalgorithmus. Dann lässt sich der Performanzwert $p_r : R \mapsto [0,1]$ einer Regel $r_j \in R$ ausdrücken als

$$p_r(r_j) = \frac{1}{|C_j|} \sum_{c_i \in C_j} p(c_i)$$

Der Performanzwert einer Regel wird also berechnet als Mittelwert des Erfüllungsgrades der von dieser Regel abgeleiteten Constraints. Regeln, die keine Constraints ableiten, bleiben von diesem Verfahren unberücksichtigt.

5. Bewertung der Verfahren und Fazit

In diesem Kapitel werden die Ergebnisse der Anwendung der beschriebenen Verfahren zur Entdeckung von Anomalien auf die Wissensbasis zur Layoutplanung von Chemieanlagen aufgeführt. Diese Ergebnisse werden anschließend bewertet und darauf folgend ein Ausblick auf mögliche weitere Aufgabenstellungen in diesem Bereich gegeben.

5.1. Redundanzentdeckung

Bei der Überprüfung der Wissensbasis wurden die in Tabelle 5.1 aufgeführten Redundanzen entdeckt. Es können damit vier der insgesamt 119 Regeln aus der Wissensbasis entfernt werden, ohne die Funktion zu beeinträchtigen.

5.1.1. Bewertung

Es zeigt sich, dass sich die Redundanzüberprüfung gut dazu einsetzen lässt, die redundanten Regeln einer Wissensbasis zu entfernen. Es ist dabei nicht notwendig, dass der Anwender besondere Kenntnisse über die Erstellung von Wissensbasen hat, da die Funktionalität der Wissensbasis durch die Entfernung von Redundanzen nicht geändert wird. Es ist aber dennoch sinnvoll, dem Anwender die Entscheidung zu überlassen, ob eine Regel tatsächlich entfernt wird, damit etwa Regeln, die aufgrund geplanter Modifikationen zukünftig relevant für die korrekte Funktionalität werden, nicht automatisch entfernt werden.

Redundante Regel	Funktionalität bereitgestellt durch
komponente(K,mischer) & spezifikation(K,dynamischer_mischer) → bedienung(K)	bedienung_anforderung_art(mischer), komponente(K,ART) & bedienung_anforderung_art(ART) → bedienung(K)
komponente(K,mischer) & spezifikation(K,dynamischer_mischer) → wartung(K)	wartung_anfordegung_art(mischer), komponente(K,ART) & wartung_anforderung_art(ART) → wartung(K)
wartungshilfsmittel(K,gabelhubwagen) & max_breite_kleiner(K,5000) & max_laenge_kleiner(K,3500) → anforderung_1(K,am_weg)	wartungshilfsmittel(K,gabelhubwagen) → wartung(K), wartung(K) → anforderung_1(K,am_weg)
medieneigenschaft(K,feststoffhaltig) → wartung(K)	medieneigenschaft(K,feststoffhaltig) → wartungshilfsmittel(K,gabelhubwagen), wartungshilfsmittel(K,gabelhubwagen) → wartung(K)

Tabelle 5.1.: Ergebnisse der Redundanz-Überprüfung

5.2. Widerspruchsentdeckung

Um nicht nur rein logische Widersprüche entdecken zu können, sondern auch solche, die mit dem Anwendungsbereich der Wissensbasis zusammenhängen, wurde die Wissensbasis um Regeln erweitert, die dazu führen, dass ein logischer Widerspruch abgeleitet wird, sobald ein entsprechender Widerspruch im Anwendungsbereich vorliegt. Ein Beispiel für einen Widerspruch aus dem Anwendungsbereich der betrachteten Wissensbasis ist die Anforderung, dass eine Komponente der Anlage sowohl im Erdgeschoss als auch in der obersten Etage des Stahlbaus platziert werden soll. Ein entsprechender logischer Widerspruch wird dann durch die beiden Regeln

`anforderung_1(K,im_erdgeschoss)` $\rightarrow \neg$`anforderung_1(K,oberste_etage)`

und

`anforderung_1(K,oberste_etage)` $\rightarrow \neg$`anforderung_1(K,im_erdgeschoss)`

abgeleitet.

Es ist daher eine Herausforderung für den Anwender, beim Einsatz der Widerspruchsentdeckung die aus dem Anwendungsbereich der Wissensbasis resultierenden Widersprüche zu identifizieren und geeignete Regeln aufzustellen, die diese in logische Widersprüche wandeln. Dieser Schritt kann einerseits bereits beim Entwurf der Wissensbasis geschehen, sofern es sich um den Experten des Anwendungsbereichs bekannte und relevant erscheinende Widersprüche handelt. Alternativ können solche Widersprüche durch die Validierung der Wissensbasis mit Hilfe von Testdaten aufgedeckt werden, wie es in dieser Arbeit mit der Untersuchung der Regelperformanz geschieht.

Eine weitere Schwierigkeit bei der Modellierung anwendungsbezogener Widersprüche besteht darin, einen Kompromiss zwischen dem Aufwand zur Modellierung, dem daraus resultierenden Nutzen und der Komplexität der Wissensbasis zu finden. Für den betrachteten Anwendungsfall wäre es zum Beispiel denkbar, Widersprüche zu modellieren, die daraus resultieren, dass eine Etage aufgrund vieler "neben"-Anforderungen überbelegt wird. Es stünde also nicht genug Platz zur Verfügung, um alle Komponenten, die nebeneinander platziert werden sollen, in einer Etage unterzubringen. Eventuell wäre die Aufdeckung eines solchen Widerspruchs für die Planer der Anlage sehr hilfreich. Es erscheint aber sinnvoller, diese Fragestellung von einem für diese Aufgabe spezialisierten Verfahren, also einem Platzierungsalgorithmus, beantworten zu lassen.

Bei der Anwendung der Widerspruchsentdeckung auf die Wissensbasis zum Layout von Chemieanlagen wurden zunächst 47 Widersprüche entdeckt. Bei der Untersuchung der Widersprüche stellte sich aber heraus, dass ein Großteil für die Funktion der Wissensbasis nicht von Bedeutung ist. In der Wissensbasis wird das Prädikat `komponente(K,ART)` dazu verwendet, den Typ einer Komponente

(z. B. Pumpe, Behälter, Kolonne, o. ä.) festzulegen. Für jede Komponente existiert also genau ein Fakt, welcher den Typ der Komponente beschreibt. Bei der Widerspruchsentdeckung wurden aber viele Widersprüche abgeleitet, die mehrere `komponente(K,ART)`-Literale für eine Komponente enthielten, also z. B. Eingaben wie

`komponente(K,pumpe)` & `komponente(K,tank)`

Es ist also notwendig, Widersprüche herauszufiltern, die sich widersprechende Literale enthalten, um zu einem für den Anwender nützliche Ergebnis zu kommen. Dies wurde für das Prädikat `komponente`/2 testweise durchgeführt, da dieses in den für die betrachtete Wissensbasis gefundenen Widersprüchen am häufigsten vorhanden ist. Für die Filterung beliebiger Wissensbasen wäre es notwendig, eine allgemein anwendbare Form der Repräsentation von sich gegenseitig ausschließenden Literalen zu finden.

Weiterhin werden Widersprüche gefunden, die durch die Eingabe von Fakten ausgelöst werden, deren Prädikate nicht als Benutzereingabe vorgesehen sind. Ein Beispiel wäre etwa die Eingabe

`wartungshilfsmittel(K,mobiler_kran)` & `anforderung_1(K,im_erdgeschoss)`

Während Literale des Prädikats `wartungshilfsmittel`/2 für die Eingabe durch den Anwender vorgesehen sind, ist dies für `anforderung_1`/2 nicht der Fall. Das Verfahren zur Widerspruchsentdeckung wurde daher um einen Filter erweitert, der Widersprüche verwirft, die durch Eingaben ausgelöst werden, welche Literale enthalten, die nicht zur Eingabe durch den Anwender vorgesehen sind. Zu diesem Zweck wurde der Metadaten-Tag "noinput" eingeführt (siehe Abschnitt 4.1.2 zum Tagging von logischen Elementen), mit dem Prädikate in der Wissensbasis gekennzeichnet werden können, deren Literale nicht durch den Benutzer eingegeben werden sollen. Eine Übersicht über die Resultate der Filterung der entdeckten Widersprüche gibt Tabelle 5.2 (die nach Anwendung der Filterung entdeckten Widersprüche sind im Anhang B aufgeführt). Wie man erkennen kann, wird die Anzahl der entdeckten Widersprüche durch die Anwendung der beiden Filtermethoden signifikant reduziert, so dass der Anwender nur noch wenige, für ihn relevante Widersprüche präsentiert bekommt.

Aber auch unter den nach Anwendung der Filter noch entdeckten Widersprüchen befinden sich einige, die für den Anwendungsbereich der Wissensbasis nicht von Bedeutung sein müssen. Ein Beispiel dafür ist der von den folgenden Literalen induzierte Widerspruch:

`komponente(K,leitstand)` & `wartungshilfsmittel(K, mobiler_kran)`

Dieser führt aufgrund einer Eingabe wie

`komponente(k,leitstand)`

Filterung	**Anzahl der Widersprüche**
ohne Filterung	47
Entfernung sich widersprechender Prädikate	36
Entfernung von Widersprüchen mit Nicht-Eingabe-Literalen	30
Entfernung von sich widersprechenden Prädikaten und Nicht-Eingabe-Literalen	19

Tabelle 5.2.: Abhängigkeit der Anzahl der entdeckten Widersprüche von der Ausfilterung sich widersprechender Literale und Nicht-Eingabe-Prädikate

zur Ableitung der Anforderung

```
anforderung_1(k,im_erdgeschoss)
```

und aufgrund der Eingabe

```
wartungshilfsmittel(k,mobiler_kran)
```

zur Ableitung von

```
anforderung_1(k,oberste_etage)
```

Ein solcher Widerspruch muss nun von Experten des Anwendungsbereichs evaluiert werden. Es muss geklärt werden, ob ein Leitstand als Wartungshilfsmittel einen mobilen Kran benötigen kann. Wenn nicht, ist der Widerspruch für die Funktion der Wissensbasis nicht relevant. Andererseits wäre es auch möglich, die an der Ableitung des Widerspruchs beteiligten Regeln dahingehend zu erweitern, dass kein Widerspruch mehr abgeleitet werden kann.

Ebenso werden aber auch Widersprüche entdeckt, die tatsächlich einen Fehler in der Wissensbasis darstellen. Ein solcher entsteht z. B. bei der Eingabe

```
komponente(l,kolonne)
```

Verantwortlich für die Ableitung dieses Widerspruchs sind die Regeln

```
komponente(K,kolonne) → wartungshilfsmittel(K,mobiler_kran)
wartungshilfsmittel(K,mobiler_kran) → anforderung_1(K,oberste_etage)
```

und

```
komponente(K,kolonne) → anforderung_1(K,im_erdgeschoss)
```

Hier liegt also eindeutig ein Fehler bei der Modellierung der Regelbasis vor, da niemals die beiden Anforderungen gleichzeitig erfüllt werden können. Es ist hier also notwendig, die Regeln entsprechend zu korrigieren, um die korrekte Funktion der Regelbasis sicherzustellen.

An dieser Stelle zeigt sich aber auch, dass es zur Korrektur eines Widerspruchs nicht immer genügt, die Regeln weiter zu spezialisieren, so dass sie nicht mehr gleichzeitig ableitbar sind. Der Grund für die Ableitung dieses Widerspruchs ist darin zu sehen, dass das Wissen über den Anwendungsbereich nicht ausreichend detailliert modelliert wurde. Eine Kolonne ist in diesem Fall ein so großes Bauteil, dass sie nicht im Stahlbau integriert, sondern nur daneben ebenerdig aufgestellt werden kann. Es ist aber nur für Komponenten, die im Stahlbau stehen, notwendig, dass sie, sofern sie zur Wartung einen mobilen Kran benötigen, in der obersten Etage platziert werden. In diesem Fall ist der Widerspruch also ein Hinweis auf eine unzureichende Modellierung des Anwendungsgebietes, da es zur Zeit nicht möglich ist, in der Wissensbasis Informationen über den Standort einer Komponente außerhalb des Stahlbaus abzulegen.

5.3. Regelperformanz

Die Untersuchung der Regelperformanz ist naturgemäß stark abhängig von der Art der Testdaten. In diesem Fall stand die Regelbasis zur Layoutplanung von Chemieanlagen in Kombination mit der Entwurfsbeschreibung der Anlage "Trieste", einer chemietechnischen Anlage zur Gasvorbehandlung, zur Verfügung. Eine fundierte Aussage über die Widerspruchsfreiheit einzelner Regeln kann nur anhand einer größeren Datenmenge gemacht werden. Leider ist der verwendete Algorithmus zur Berechnung von Aufstellungen nicht in der Lage, mehrere alternative Lösungen zu generieren. Ansonsten wäre es so möglich, auch mit einer kleinen Testdatenmenge einzelne Regeln bzw. die von ihnen abgeleiteten Constraints in verschiedenen Kombinationen auszuwerten und damit zu einer von der Beschaffenheit der Anlagenbeschreibung weniger abhängigeren Aussage über die Performanz einzelner Regeln zu kommen. Stehen mehrere Anlagenbeschreibungen zur Verfügung, so könnte man auch die Performanzwerte der Regeln für mehrere Anlagen zusammenfassen und so zu einem genaueren Ergebnis kommen.

Bei der Interpretation der Performanz einer Regel muss außerdem berücksichtigt werden, auf dem Erfüllungsgrad wievieler Constraints sie basiert. Erst bei Regeln, die sehr viele Constraints ableiten, ist eine geringe Regelperformanz ein Hinweis darauf, dass die Regel selbst für die schlechte Performanz verantwortlich ist und nicht die Konstellation der Testdaten oder die Leistung des Algorithmus zur Aufstellungsberechnung, die unter Umständen die Erfüllung bestimmter Constraints nicht erlauben.

Aufgrund der geringen Testdatenmenge lässt sich hier nur feststellen, dass die Regelperformanz für eine Regel, die eine große Anzahl von anderen Regeln widersprechenden Constraints ableitet, sehr gering ist. Eine Übersicht über die bei der Berechnung der Regelperformanz erzielten Ergebnisse gibt Tabelle 5.3. Es ist zu erkennen, dass Regeln, die widersprüchliche Fakten ableiten, durchweg eine schlechte Performanz aufweisen (etwa die Regeln L001 und L100). Bei der Aus-

wertung der Regelperformanz konnte außerdem ein weiterer Widerspruch aus dem Anwendungsbereich identifiziert werden. Viele Komponenten haben die Anforderung, dass sie an einem Zugangsweg der Anlage stehen. Dies widerspricht der Anforderung von Komponenten, in der obersten Etage des Stahlbaus aufgestellt zu werden. Deutlich wird das an der schlechten Performanz der Regeln L004 und 014, die beide die Anforderung "am_weg" ableiten.

Nachdem Regeln zur Entdeckung dieses Widerspruchs der Regelbasis hinzugefügt wurden (Regeln constraint_3_1, constraint_3_2 und constraint_3_3), konnte das Verfahren zur Widerspruchsentdeckung 59 weitere Widersprüche identifizieren. Viele dieser neu entdeckten Widersprüche enthielten aber wieder sich gegenseitig ausschließende Literale, so dass angenommen werden kann, dass sich die Anzahl der Widersprüche nach Anwendung eines geeigneten Filters, ähnlich zu dem für sich gegenseitig ausschließende `komponente`-Literale, erheblich reduziert.

Die Untersuchung der Regelperformanz stellt also eine hilfreiche Ergänzung des Verfahrens zur Widerspruchsentdeckung dar. Es lassen sich einerseits die Relevanz bereits entdeckter Widersprüche überprüfen, aber andererseits auch Hinweise auf bisher nicht betrachtete Widersprüche aus dem Anwendungsbereich finden.

5.4. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Wissensbasis zur Layoutplanung von Chemietechnikanlagen auf Anomalien untersucht. Dazu wurde zunächst das Framework SIMPLE implementiert, in dessen Rahmen dann die Verfahren zur Widerspruchsentdeckung, zur Redundanzüberprüfung und zur Untersuchung der Regelperformanz implementiert wurden.

Das Verfahren zur Redundanzüberprüfung lieferte Ergebnisse, die direkt zur Verbesserung der Wissensbasis beitragen können. Die redundanten Regeln lassen sich ohne weitere Auswirkungen aus der Wissensbasis entfernen. Die Anwendung des Verfahrens trägt somit zur Übersichtlichkeit und Wartbarkeit der Wissensbasis bei. Es bedeutet keinen wesentlichen Aufwand für den Benutzer, dass Verfahren einzusetzen.

Auch das Verfahren zur Widerspruchsentdeckung konnte mit Erfolg auf die Wissensbasis angewendet werden. Hier zeigte sich jedoch, dass die Wissensbasis zunächst um das Wissen über auf dem Anwendungsbereich resultierende Widersprüche und sich gegenseitig ausschließende Literale erweitert werden muss, damit das Ergebnis der Widerspruchsentdeckung für den Anwender nutzbar wird. Wird dies berücksichtigt, so werden Widersprüche entdeckt, die tatsächlich auf relevante Fehler in der Wissensbasis hindeuten. Es ist aber auch dann immer noch die Aufgabe des Anwenders, die Widersprüche zu evaluieren und festzustellen, ob diese für die Arbeit der Wissensbasis von Bedeutung sind oder ignoriert werden können.

An diesem Punkt könnte sich eine Arbeit anschließen, die sich zum einen damit beschäftigt, wie Widersprüche aus dem Anwendungsbereich entdeckt und in logi-

Name der Regel	Anzahl Constraints	Regel-Performanz
R008	1	1.0
R009	2	1.0
R012	2	1.0
R007	1	1.0
L023	4	0.884
L024	5	0.884
L008	14	0.883
R010	1	0.874
L002	4	0.873
L003	4	0.873
L007	2	0.846
R011	6	0.833
L018	4	0.819
L023rev	6	0.816
L120	2	0.797
L016	4	0.722
L100	7	0.5
L004	19	0.479
014	19	0.479
L015	9	0.414
L005	4	0.249
L031	4	0.249
L042	3	0.0
R001	1	0.0
R002	1	0.0
L001	2	0.0
L006	2	0.0
L009	2	0.0
L181	1	0.0

Tabelle 5.3.: Ergebnisse der Regelperformanz-Auswertung

sche Widersprüche überführt werden können. Außerdem ist es für die allgemeine Verwendung der Widerspruchsentdeckung notwendig, die Beziehung von Literalen zueinander zu beschreiben, insbesondere deren gegenseitigen Ausschluss.

Die Untersuchung der Regelperformanz zeigte, dass diese erst dann sinnvoll einsetzbar ist, wenn einerseits mehrere alternative Aufstellungen für eine abgeleitete Menge von Constraints berechnet werden können und andererseits mehr Testdaten vorhanden sind, sodass man im längerfristigen Einsatz der Wissensbasis Informationen darüber sammeln kann, die Constraints welcher Regeln immer wieder schlecht erfüllt werden können. Leider liefert die Untersuchung der Regelperformanz selbst dann keine Aussagen darüber, warum die Constraints der betroffenen Regeln nicht erfüllt werden können. An dieser Stelle wäre es auch denkbar, einen alternativen Algorithmus zur Berechnung von Aufstellungen zu verwenden, der in der Lage ist, mehrere alternative Aufstellungen zu generieren und die Regelperformanzwerte dann für mehrere Aufstellungen zu akkumulieren. Ein geeignetes Verfahren wäre zum Beispiel der in [26] von Mierswa vorgestellte genetische Algorithmus.

Es wäre interessant, das Verhalten und die Verwertbarkeit der Ergebnisse der Untersuchung der Regelperformanz bei einer größeren Menge von Testdaten genauer zu untersuchen und nach Verfahren zu suchen, die nicht nur die Regeln in die Untersuchung einbeziehen, die direkt Constraints ableiten, sondern auch diejenigen, die weiter vorne im Ableitungspfad eines Constraints liegen.

Während der Beschäftigung mit der in dieser Arbeit betrachteten Wissensbasis zeigte sich an vielen Stellen, dass sich Methoden der Softwareentwicklung gut auf den Entwurf von regelbasierten Systemen übertragen lassen. Prinzipien wie Modularisierung, Strukturierung und Orthogonalität lassen sich auch für regelbasierte Systeme umsetzen. Auch Methoden der testgetriebenen Softwareentwicklung wie Unit-Tests sowie ausgereifte Refactoring-Mechanismen lassen sich in solchen Systemen gut einsetzen und würden die Qualität solcher Systeme weiter verbessern. Die Untersuchung der Übertragbarkeit aktueller Methoden der Softwareentwicklung auf regelbasierte Systeme wäre daher sicher auch ein Aufmerksamkeit verdienendes Thema.

Mit dem Einsatz der in SIMPLE implementierten Verfahren zur Anomalieentdeckung konnte die Wissensbasis zur Layoutplanung von Chemieanlagen in einigen Punkten verbessert werden. Es werden aber auch Hinweise auf Fehler gefunden, die von einem Anwendungsexperten evaluiert werden müssen, die aber auf Verbesserungsmöglichkeiten im Entwurfsprozess der Wissensbasis hinweisen. Insgesamt sind die Verfahren damit in der Lage, den Entwicklungsprozess einer Wissensbasis zu unterstützen, zu beschleunigen und insgesamt zu einer besser strukturierten Wissensbasis beizutragen.

Anhang A.

Inhalt der Wissensbasis

Die folgende Liste umfasst die Regeln zur Ableitung von Constraints für die Aufstellung von Komponenten chemietechnischer Anlagen. Diese Regeln zusammen mit den im folgenden Abschnitt aufgeführten Fakten stellt die gesamte untersuchte Wissensbasis dar.

A.1. Regeln

014: `demontage(KOMP)` → `anforderung_1(KOMP, am_weg, 1)`

Bed015: `komponente(KOMP, mischer)`
& `spezifikation(KOMP, dynamischer_mischer)` → `bedienung(KOMP)`

Bed016: `komponente(KOMP, reaktor)`
& `spezifikation(KOMP, ruehrkesselreaktor)` → `bedienung(KOMP)`

Bed017: `funktion(KOMP, mischen)` → `bedienung(KOMP)`

Bed018: `betriebsart(KOMP, batch)` → `bedienung(KOMP)`

Bed019: `betriebsart(KOMP, semi_batch)` → `bedienung(KOMP)`

Bed020: `eigenschaft(KOMP, mit_heizschlange)` → `bedienung(KOMP)`

BedHil001: `betriebsart(KOMP, batch)`
→ `bedienungshilfsmittel(KOMP, gabelhubwagen)`

BedHil002: `betriebsart(KOMP, semi_batch)`
→ `bedienungshilfsmittel(KOMP, gabelhubwagen)`

BedHil003: `spezifikation(KOMP, festbett)`
→ `bedienungshilfsmittel(KOMP, gabelhubwagen)`

BedHil004: `spezifikation(KOMP, manuelle_befuellung)`
→ `bedienungshilfsmittel(KOMP, gabelhubwagen)`

CtStep1_001: `demontagehilfsmittel(KOMP, HM)` → `demontage(KOMP)`

CtStep1_002: `wartungshilfsmittel(KOMP, HM)` → `wartung(KOMP)`

CtStep1_003: `komponente(KOMP, TYP)` → `art_komponente(KOMP)`

CtStep1_008: `demontagehilfsmittel(KOMP, HM)` → `art_hilfsmittel(HM)`

CtStep1_009: `wartungshilfsmittel(KOMP, HM)` → `art_hilfsmittel(HM)`

CtStep1_010: `bedienungshilfsmittel(KOMP, HM)` → `art_hilfsmittel(HM)`

Dem001: `medieneigenschaft(KOMP, verschmutzend)` & `volumen(KOMP, VOL)`
& `volumen_lt_20(KOMP)` → `demontage(KOMP)`

DemHil001: `komponente(KOMP, pumpe)`
→ `demontagehilfsmittel(KOMP, gabelhubwagen)`

DemHil002: `komponente(KOMP, rohrbuendel_wat)`
→ `demontagehilfsmittel(KOMP, mobiler_kran)`

DemHil003: `volumen_gt_15(KOMP)` & `gewicht_gt_15000(KOMP)`
→ `demontagehilfsmittel(KOMP, mobiler_kran)`

L001: `demontage(KOMP)` & `demontagehilfsmittel(KOMP, mobiler_kran)`
& `max_laenge_gt_5000(KOMP)` → `anforderung_1(KOMP, oberste_etage, 3)`

L002: `verbindung(CON, KOMP1, KOMP2)` & `komponente(KOMP1, TYP1)`
& `komponente(KOMP2, TYP2)`
& `rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)`
& `rohrleitung_dia_gt_250(CON)` → `anforderung_2(KOMP1, KOMP2, nahe, 2)`

L003: `verbindung(CON, KOMP1, KOMP2)` & `komponente(KOMP1, TYP1)`
& `komponente(KOMP2, TYP2)`
& `rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)`
& `rohrleitung_dia_gt_350(CON)` → `anforderung_2(KOMP1, KOMP2, nahe, 1)`

L004: `bedienung(KOMP)` → `anforderung_1(KOMP, am_weg, 3)`

L005: `wartungshilfsmittel(KOMP, gabelhubwagen)` & `min_laenge(KOMP, MIN_L)`
& `max_laenge(KOMP, MAX_L)` & `min_laenge_lt_5000(KOMP)`
& `max_laenge_lt_3500(KOMP)` → `anforderung_1(KOMP, am_weg, 2)`

L006: `wartungshilfsmittel(KOMP, mobiler_kran)`
→ `anforderung_1(KOMP, oberste_etage, 2)`

L007: `wartung(KOMP)` & `wartungshilfsmittel(KOMP, mobiler_kran)`
→ `anforderung_1(KOMP, am_rand, 2)`

L008: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP1, TYP1)
& komponente(KOMP2, TYP2)
& rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_150(CON) → anforderung_2(KOMP1, KOMP2, nahe, 3)

L009: wartungshilfsmittel(KOMP, mobiler_kran) & min_laenge_gt_5000(KOMP)
→ anforderung_1(KOMP, oberste_etage, 3)

L013: bedienungshilfsmittel(KOMP, gabelhubwagen)
→ anforderung_1(KOMP, am_weg, 1)

L015: wartung(KOMP) → anforderung_1(KOMP, am_weg, 2)

L016: komponente(KOMP1, TYP1) & komponente(KOMP2, TYP2)
& verbindung(VNAME, KOMP1, KOMP2) & funktion(KOMP2, vakuumerzeugung)
→ anforderung_2(KOMP1, KOMP2, nahe, 1)

L017: komponente(KOMP1, silo) & komponente(KOMP2, silo)
→ anforderung_2(KOMP1, KOMP2, ueber, 2)

L018: demontage(KOMP) & demontagehilfsmittel(KOMP, mobiler_kran)
& laenge(KOMP, LAENGE) & breite(KOMP, BREITE) & laenge_lt_5000(KOMP)
& breite_lt_5000(KOMP) → anforderung_1(KOMP, am_rand, 3)

L019: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_50(CON) → anforderung_2(KOMP1, KOMP2, neben, 3)

L020: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP1, TYP1)
& komponente(KOMP2, TYP2)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_150(CON) → anforderung_2(KOMP1, KOMP2, nahe, 2)

L021: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP1, TYP1)
& komponente(KOMP2, TYP2)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_50(CON) → anforderung_2(KOMP1, KOMP2, nahe, 3)

L022: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP1, TYP1)
& komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_350(CON) → anforderung_2(KOMP1, KOMP2, nahe, 2)

L023: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)
& komponente(KOMP1, KOMP1TYP) & rohrleitung_dia_gt_250(CON)
→ anforderung_2(KOMP1, KOMP2, neben, 2)

L023rev: verbindung(CON, KOMP2, KOMP1) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)
& komponente(KOMP1, KOMP1TYP) & rohrleitung_dia_gt_250(CON)
→ anforderung_2(KOMP2, KOMP1, neben, 2)

L024: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, carbon_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_150(CON) → anforderung_2(KOMP1, KOMP2, neben, 3)

L025: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_200(CON) → anforderung_2(KOMP1, KOMP2, neben, 1)

L026: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP2, uebergabepunkt)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_150(CON) → anforderung_2(KOMP1, KOMP2, neben, 2)

L027: verbindung(CON, KOMP1, KOMP2) & komponente(KOMP1, TYP1)
& komponente(KOMP2, TYP2)
& rohrleitung(CON, DIA, stainless_steel, AGG, ANFANG, ENDE)
& rohrleitung_dia_gt_200(CON) → anforderung_2(KOMP1, KOMP2, nahe, 1)

L028: gewicht_gt_10000(KOMP) → anforderung_1(KOMP, im_erdgeschoss, 3)

L029: wartungshilfsmittel(KOMP, rohrbuendelzugmaschine)
→ anforderung_1(KOMP, am_rand, 2)

L031: demontagehilfsmittel(KOMP, gabelhubwagen) & min_laenge(KOMP, MIN_L)
& max_laenge(KOMP, MAX_L) & min_laenge_lt_3500(KOMP)
& max_laenge_lt_3500(KOMP) → anforderung_1(KOMP, am_weg, 2)

L040: komponente(KOMP1, behaelter) & spezifikation(KOMP1, waschbehaelter)
& komponente(KOMP2, silo) → anforderung_2(KOMP1, KOMP2, ueber, 3)

L041: komponente(KOMP, behaelter) & spezifikation(KOMP1, waschbehaelter)
& komponente(KOMP, dekanter) → anforderung_2(KOMP1, KOMP2, ueber, 3)

L042: komponente(KOMP1, behaelter) & komponente(KOMP2, pumpe)
& verbindung(VERB, KOMP1, KOMP2) → anforderung_2(KOMP1, KOMP2, ueber, 2)

L043: komponente(KOMP1, behaelter)
& spezifikation(KOMP1, suspensionsbehaelter)
& komponente(KOMP2, zentrifuge) → anforderung_2(KOMP1, KOMP2, ueber, 3)

L044: komponente(KOMP1, behaelter)
& spezifikation(KOMP1, feuchtgutbehaelter) & komponente(KOMP2, trockner)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

L050: komponente(KOMP1, brecher) & komponente(KOMP2, muehle)
→ anforderung_2(KOMP1, KOMP2, ueber, 2)

L051: komponente(KOMP1, brecher) & komponente(KOMP2, brecher)
→ anforderung_2(KOMP1, KOMP2, ueber, 2)

L061: komponente(KOMP1, silo) & komponente(KOMP2, granulator)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

L080: komponente(KOMP, leitstand) → anforderung_1(KOMP, im_erdgeschoss, 2)

L081: komponente(KOMP, leitstand) → anforderung_1(KOMP, am_rand, 2)

L090: komponente(KOMP1, muehle) & komponente(KOMP2, muehle)
→ anforderung_2(KOMP1, KOMP2, ueber, 2)

L100: komponente(KOMP, pumpe) & ¬spezifikation(KOMP, dosierpumpe)
→ anforderung_1(KOMP, im_erdgeschoss, 1)

L110: komponente(KOMP, reaktor) & spezifikation(KOMP, wirbelschichtreaktor)
→ anforderung_1(KOMP, im_erdgeschoss, 2)

L120: komponente(KOMP, rohrbuendel_wat)
& spezifikation(KOMP, kolonnenverdampfer)
→ anforderung_1(KOMP, am_rand, 2)

L140: komponente(KOMP, silo) → anforderung_1(KOMP, am_rand, 3)

L150: volumen_gt_50(KOMP) & ¬komponente(KOMP, tank)
→ anforderung_1(KOMP, am_rand, 3)

L151: komponente(KOMP, tank) → anforderung_1(KOMP, oberste_etage, 1)

L160: komponente(KOMP1, trockner) & komponente(KOMP2, uebergabepunkt)
→ anforderung_2(KOMP1, KOMP2, neben, 2)

L170: komponente(KOMP, verdichter) & spezifikation(KOMP, turboverdichter)
→ anforderung_1(KOMP, im_erdgeschoss, 2)

L171: komponente(KOMP, verdichter) & spezifikation(KOMP, kolbenverdichter)
→ anforderung_1(KOMP, im_erdgeschoss, 1)

L172: komponente(KOMP, verdichter) & ¬spezifikation(KOMP, turboverdichter)
→ anforderung_1(KOMP, im_erdgeschoss, 2)

L180: komponente(KOMP1, waermetauscher) & komponente(KOMP2, pumpe)
→ anforderung_2(KOMP1, KOMP2, ueber, 3)

L181: komponente(KOMP, rohrbuendel_wat)
& spezifikation(KOMP, kopfkondensator)
→ anforderung_1(KOMP, oberste_etage, 3)

L182: komponente(KOMP, waermetauscher) & spezifikation(KOMP, luftkuehler)
→ anforderung_1(KOMP, oberste_etage, 1)

L183: komponente(KOMP1, waermetauscher) & funktion(KOMP1, kopfkondensation)
& komponente(KOMP2, kolonne) → anforderung_2(KOMP1, KOMP2, nahe, 2)

L184: komponente(KOMP1, waermetauscher)
& spezifikation(KOMP1, kopfkondensator) & komponente(KOMP2, behaelter)
& spezifikation(KOMP2, ruecklaufbehaelter)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

L191: komponente(KOMP1, zentrifuge) & komponente(KOMP2, behaelter)
& spezifikation(KOMP2, filtratbehaelter)
→ anforderung_2(KOMP1, KOMP2, ueber, 3)

L192: komponente(KOMP1, zentrifuge) & komponente(KOMP2, behaelter)
& spezifikation(KOMP2, feuchtgutbehaelter)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

L200: komponente(KOMP1, zyklon) & komponente(KOMP2, uebergabepunkt)
→ anforderung_2(KOMP1, KOMP2, neben, 2)

L201: komponente(KOMP1, zyklon) & komponente(KOMP2, silo)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

R001: komponente(KOMP1, rohrbuendel_wat)
& spezifikation(KOMP1, kopfkondensator)
& komponente(KOMP2, rohrbuendel_wat) & spezifikation(KOMP2, kondensator)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

R002: komponente(KOMP1, rohrbuendel_wat)
& spezifikation(KOMP1, kondensator) & komponente(KOMP2, behaelter)
& spezifikation(KOMP2, destillatvorlage)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

R003: komponente(KOMP1, rohrbuendel_wat)
& funktion(KOMP1, kolonnenverdampfung) & komponente(KOMP2, kolonne)
& spezifikation(KOMP2, vakuumkolonne)
→ anforderung_2(KOMP1, KOMP2, nahe, 1)

R005: komponente(KOMP1, kolonne) & spezifikation(KOMP1, vakuumkolonne)
& komponente(KOMP2, kolonne) → anforderung_2(KOMP1, KOMP2, nahe, 1)

R007: komponente(KOMP, behaelter) & spezifikation(KOMP, dusche)
→ anforderung_1(KOMP, oberste_etage, 1)

R008: komponente(KOMP, blackbox) & spezifikation(KOMP, vakuumpumpe)
→ anforderung_1(KOMP, im_erdgeschoss, 1)

R009: komponente(KOMP, kolonne) → anforderung_1(KOMP, im_erdgeschoss, 1)

R010: komponente(KOMP1, rohrbuendel_wat)
& spezifikation(KOMP1, kondensator) & komponente(KOMP2, behaelter)
& spezifikation(KOMP2, destillatvorlage)
→ anforderung_2(KOMP1, KOMP2, nahe, 2)

R011: komponente(KOMP, platten_wat) & spezifikation(KOMP, umlaufkuehler)
→ anforderung_1(KOMP, im_erdgeschoss, 1)

R012: komponente(KOMP, behaelter)
& spezifikation(KOMP, ruecklaufbehaelter)
→ anforderung_1(KOMP, im_erdgeschoss, 1)

R013: komponente(KOMP1, blackbox) & spezifikation(KOMP1, kopfkondensator)
& komponente(KOMP2, rohrbuendel_wat)
& spezifikation(KOMP2, kopfkondensator)
→ anforderung_2(KOMP1, KOMP2, ueber, 1)

War001: betriebsart(KOMP, batch) → wartung(KOMP)

War002: betriebsart(KOMP, semi-batch) → wartung(KOMP)

War003: spezifikation(KOMP, festbett) → wartung(KOMP)

War004: spezifikation(KOMP, mit_filter) → wartung(KOMP)

War005: funktion(KOMP, mischen) → wartung(KOMP)

War006: funktion(KOMP, kristallisation) → wartung(KOMP)

War007: funktion(KOMP, reaktion) → wartung(KOMP)

War008: medieneigenschaft(KOMP, verschmutzend) → wartung(KOMP)

War009: medieneigenschaft(KOMP, stark_verschmutzend) → wartung(KOMP)

War010: medieneigenschaft(KOMP, feststoffhaltig) → wartung(KOMP)

War027: komponente(KOMP, mischer)
& spezifikation(KOMP, dynamischer_mischer) → wartung(KOMP)

War028: komponente(KOMP, zyklon) & spezifikation(KOMP, hydrozyklon)
→ wartung(KOMP)

WarHil001: komponente(KOMP, kolonne)
→ wartungshilfsmittel(KOMP, mobiler_kran)

WarHil002: medieneigenschaft(KOMP, feststoffhaltig)
→ wartungshilfsmittel(KOMP, gabelhubwagen)

WarHil003: `komponente(KOMP, brecher)`
→ `wartungshilfsmittel(KOMP, gabelhubwagen)`

WarHil004: `komponente(KOMP, muehle)`
→ `wartungshilfsmittel(KOMP, gabelhubwagen)`

WarHil005: `komponente(KOMP, pumpe)`
→ `wartungshilfsmittel(KOMP, gabelhubwagen)`

bedienung_by_component_type: `bedienung_anforderung_art(ART)`
& `komponente(KOMP, ART)` → `bedienung(KOMP)`

constraint1_1: `anforderung_1(KOMP, oberste_etage, ANFLEV1)`
→ ¬`anforderung_1(KOMP, im_erdgeschoss, 1)`

constraint1_2: `anforderung_1(KOMP, oberste_etage, ANFLEV1)`
→ ¬`anforderung_1(KOMP, im_erdgeschoss, 2)`

constraint1_3: `anforderung_1(KOMP, oberste_etage, ANFLEV1)`
→ ¬`anforderung_1(KOMP, im_erdgeschoss, 3)`

constraint2_1: `anforderung_1(KOMP, im_erdgeschoss, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 1)`

constraint2_2: `anforderung_1(KOMP, im_erdgeschoss, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 2)`

constraint2_3: `anforderung_1(KOMP, im_erdgeschoss, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 3)`

constraint3_1: `anforderung_1(KOMP, am_weg, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 1)`

constraint3_2: `anforderung_1(KOMP, am_weg, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 2)`

constraint3_3: `anforderung_1(KOMP, am_weg, LEV)`
→ ¬`anforderung_1(KOMP, oberste_etage, 3)`

demontage_by_component_type: `demontage_anforderung_art(ART)`
& `komponente(KOMP, ART)` → `demontage(KOMP)`

demontage_by_medium: `demontage_anforderung_medieneigenschaft(EIG)`
& `medieneigenschaft(KOMP, EIG)` → `demontage(KOMP)`

wartung_by_component_type: `komponente(KOMP, ART)`
& `wartung_anforderung_art(ART)` → `wartung(KOMP)`

A.2. Fakten

```
art_hilfsmittel(gabelhubwagen)
art_hilfsmittel(mobiler_kran)
bedienung_anforderung_art(brecher)
bedienung_anforderung_art(dekanter)
bedienung_anforderung_art(geblaese)
bedienung_anforderung_art(granulator)
bedienung_anforderung_art(mischer)
bedienung_anforderung_art(muehle)
bedienung_anforderung_art(platten_wat)
bedienung_anforderung_art(pumpe)
bedienung_anforderung_art(rohrbuendel_wat)
bedienung_anforderung_art(siebmaschine)
bedienung_anforderung_art(ventilator)
bedienung_anforderung_art(verdichter)
bedienung_anforderung_art(wat)
bedienung_anforderung_art(zentrifuge)
demontage_anforderung_art(dekanter)
demontage_anforderung_art(filter)
demontage_anforderung_art(geblaese)
demontage_anforderung_art(platten_wat)
demontage_anforderung_art(pumpe)
demontage_anforderung_art(trockner)
demontage_anforderung_art(ventilator)
demontage_anforderung_art(verdichter)
demontage_anforderung_art(zentrifuge)
demontage_anforderung_medieneigenschaft(feststoffhaltig)
demontage_anforderung_medieneigenschaft(stark_verschmutzend)
wartung_anforderung_art(brecher)
wartung_anforderung_art(dekanter)
wartung_anforderung_art(geblaese)
wartung_anforderung_art(granulator)
wartung_anforderung_art(kristallisator)
wartung_anforderung_art(maschine)
wartung_anforderung_art(mischer)
wartung_anforderung_art(muehle)
wartung_anforderung_art(pumpe)
wartung_anforderung_art(reaktor)
wartung_anforderung_art(sichter)
wartung_anforderung_art(siebmaschine)
wartung_anforderung_art(silo)
wartung_anforderung_art(trockner)
wartung_anforderung_art(verdichter)
wartung_anforderung_art(zentrifuge)
```

Anhang B.

Entdeckte Widersprüche

An dieser Stelle werden die bei der Untersuchung der Wissensbasis zur Layoutplanung von Chemieanlagen entdeckten Widersprüche aufgeführt, bereinigt um solche mit sich widersprechenden Literalen und Nicht-Eingabe-Literalen.

Widerspruchverursachende Eingabe	**Betroffene Regeln**

`demontage(KOMP)`
`gewicht_gt_10000(KOMP)`
`gewicht_gt_15000(KOMP)`
`max_laenge_gt_5000(KOMP)`
`volumen_gt_15(KOMP)`
	L028, DemHil003, constraint1_3, L001

`demontage(KOMP)`
`gewicht_gt_10000(KOMP)`
`komponente(KOMP, rohrbuendel_wat)`
`max_laenge_gt_5000(KOMP)`
	L028, constraint1_3, L001, DemHil002

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, behaelter)`
`max_laenge_gt_5000(KOMP)`
`spezifikation(KOMP,ruecklaufbehaelter)`
`volumen_gt_15(KOMP)`
	R012, DemHil003, constraint1_1, L001

Tabelle B.1.: Ergebnisse der Widerspruchsentdeckung - Teil 1

Widerspruchverursachende Eingabe	**Betroffene Regeln**

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, blackbox)`
`max_laenge_gt_5000(KOMP)`
`spezifikation(KOMP, vakuumpumpe)`
`volumen_gt_15(KOMP)`
	DemHil003, R008, constraint1_1, L001

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, leitstand)`
`max_laenge_gt_5000(KOMP)`
`volumen_gt_15(KOMP)`
	L080, DemHil003, constraint1_2, L001

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, platten_wat)`
`max_laenge_gt_5000(KOMP)`
`spezifikation(KOMP, umlaufkuehler)`
`volumen_gt_15(KOMP)`
	R011, DemHil003, constraint1_1, L001

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, pumpe)`
`max_laenge_gt_5000(KOMP)`
`¬spezifikation(KOMP,dosierpumpe)`
`volumen_gt_15(KOMP)`
	DemHil003, constraint1_1, L100, L001

`demontage(KOMP)`
`gewicht_gt_15000(KOMP)`
`komponente(KOMP, reaktor)`
`max_laenge_gt_5000(KOMP)`
`spezifikation(KOMP,`
`wirbelschichtreaktor)`
`volumen_gt_15(KOMP)`
	DemHil003, constraint1_2, L001, L110

`demontagehilfsmittel(KOMP, mobiler_kran)`
`gewicht_gt_10000(KOMP)`
`max_laenge_gt_5000(KOMP)`
	L028, CtStep1_001, constraint1_3, L001

Tabelle B.2.: Ergebnisse der Widerspruchsentdeckung - Teil 2

Widerspruchverursachende Eingabe	Betroffene Regeln
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, behaelter) max_laenge_gt_5000(KOMP) spezifikation(KOMP,ruecklaufbehaelter)	CtStep1_001, R012, constraint1_1, L001
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, blackbox) max_laenge_gt_5000(KOMP) spezifikation(KOMP,vakuumpumpe)	CtStep1_001, R008, constraint1_1, L001
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, leitstand) max_laenge_gt_5000(KOMP)	CtStep1_001, L080, constraint1_2, L001
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, platten_wat) max_laenge_gt_5000(KOMP) spezifikation(KOMP,umlaufkuehler)	CtStep1_001, R011, constraint1_1, L001
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, pumpe) max_laenge_gt_5000(KOMP) ¬spezifikation(KOMP,dosierpumpe)	CtStep1_001, constraint1_1, L100, L001
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, reaktor) max_laenge_gt_5000(KOMP) spezifikation(KOMP,wirbelschichtreaktor)	CtStep1_001, constraint1_2, L001, L110
demontagehilfsmittel(KOMP, mobiler_kran) komponente(KOMP, verdichter) max_laenge_gt_5000(KOMP)	CtStep1_001, L172, constraint1_2, L001, L170

Tabelle B.3.: Ergebnisse der Widerspruchsentdeckung - Teil 3

Widerspruchverursachende Eingabe	**Betroffene Regeln**

`gewicht_gt_15000(KOMP)`
`komponente(KOMP,verdichter)`
`max_laenge_gt_5000(KOMP)`
`volumen_gt_15(KOMP)`
	CtStep1_001, L172, DemHil003, L001, constraint1_2, L170

`komponente(KOMP, kolonne)`	R009, constraint1_1, WarHil001, L006
`komponente(KOMP, verdichter)` `wartungshilfsmittel(KOMP, mobiler_kran)`	L172, constraint2_2, L006, L170

Tabelle B.4.: Ergebnisse der Widerspruchsentdeckung - Teil 4

Anhang C.

Datenformat der Logik-Daten

Das Framework SIMPLE kann die Definition einer Wissensbasis aus Dateien im XML-Format einlesen. Diese Daten müssen festgelegten Regeln entsprechen. Sie werden dazu beim Einlesen mit Hilfe der folgenden XML-Schema-Definition nach [39] validiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Arg">
    <xs:complexType>
      <xs:attribute
          name="predicate"
          type="xs:string"
          use="required"/>
      <xs:attribute
          default="0"
          name="declpos"
          type="xs:string"
          use="optional"/>
      <xs:attribute
          default="0"
          name="usepos"
          type="xs:string"
          use="optional"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Conclusion">
    <xs:complexType>
      <xs:sequence>
        <xs:element
            maxOccurs="unbounded"
            minOccurs="0"
            ref="Var"/>
      </xs:sequence>
      <xs:attribute
```

```
            name="predicate"
            type="xs:string"
            use="required"/>
        <xs:attribute
            default="true"
            name="truth"
            use="optional">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="true"/>
              <xs:enumeration value="false"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name="Constraint">
      <xs:complexType>
        <xs:sequence>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Var"/>
        </xs:sequence>
        <xs:attribute
            name="type"
            type="xs:string"
            use="required"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="Fact">
      <xs:complexType>
        <xs:sequence>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Var"/>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Tag"/>
        </xs:sequence>
        <xs:attribute
```

```
            name="predicate"
            type="xs:string"
            use="required"/>
        <xs:attribute
            default="true"
            name="truth"
            use="optional">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="true"/>
              <xs:enumeration value="false"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name="Include">
      <xs:complexType>
        <xs:attribute
            name="file"
            type="xs:string"
            use="required"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="LogicSet">
      <xs:complexType>
        <xs:sequence>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Include"/>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Predicate"/>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Fact"/>
          <xs:element
              maxOccurs="unbounded"
              minOccurs="0"
              ref="Rule"/>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Operator">
    <xs:complexType>
      <xs:sequence>
        <xs:element
            maxOccurs="unbounded"
            minOccurs="0"
            ref="Var"/>
      </xs:sequence>
      <xs:attribute
          name="type"
          type="xs:string"
          use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Predicate">
    <xs:complexType>
      <xs:sequence>
        <xs:element
            maxOccurs="unbounded"
            minOccurs="0"
            ref="Arg"/>
        <xs:element
            maxOccurs="unbounded"
            minOccurs="0"
            ref="Tag"/>
      </xs:sequence>
      <xs:attribute
          name="name"
          type="xs:string"
          use="required"/>
      <xs:attribute
          name="arity"
          type="xs:string"
          use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Premise">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element
             maxOccurs="unbounded"
             minOccurs="0"
             ref="Var"/>
        <xs:element
             maxOccurs="unbounded"
             minOccurs="0"
             ref="Tag"/>
      </xs:sequence>
      <xs:attribute
          name="predicate"
          type="xs:string"
          use="required"/>
      <xs:attribute
          default="true"
          name="truth"
          use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="true"/>
            <xs:enumeration value="false"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name="Rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element
             maxOccurs="1"
             minOccurs="1"
             ref="Conclusion"/>
        <xs:element
             maxOccurs="unbounded"
             minOccurs="0"
             ref="Premise"/>
        <xs:element
             maxOccurs="unbounded"
             minOccurs="0"
             ref="Constraint"/>
        <xs:element
             maxOccurs="unbounded"
             minOccurs="0"
```

```
                ref="Operator"/>
        </xs:sequence>
        <xs:attribute
              name="name"
              type="xs:string"
              use="optional"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="Var">
      <xs:complexType>
        <xs:attribute
              name="value"
              type="xs:string"
              use="required"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="Tag">
      <xs:complexType>
        <xs:attribute
              name="name"
              type="xs:string"
              use="required"/>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Literaturverzeichnis

[1] *Eclipse Framework*. http://www.eclipse.org

[2] *Java Development Tools for Eclipse*. http://www.eclipse.org/jdt

[3] Aamodt, Agnar ; Plaza, Enric: Case-based reasoning: foundational issues, methodological variations, and system approaches. In: *AI Commun.* 7 (1994), March, Nr. 1, 39–59. http://portal.acm.org/citation.cfm?id=196108.196115. – ISSN 0921–7126

[4] Baumeister, Joachim ; Puppe, Frank ; Seipel, Dietmar: Refactoring Methods for Knowledge Bases. In: *Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004, Whittlebury Hall, UK, October 5-8, 2004, Proceedings* Bd. 3257, Springer, 2004. – ISBN 3–540–23340–7, S. 157–171

[5] Blair, H. A. ; Subrahmanian, V. S.: Paraconsistent logic programming. In: *Theor. Comput. Sci.* 68 (1989), Nr. 2, S. 135–154. http://dx.doi.org/http://dx.doi.org/10.1016/0304-3975(89)90126-6. – DOI http://dx.doi.org/10.1016/0304–3975(89)90126–6. – ISSN 0304–3975

[6] Boehm, Barry W.: Verifying and Validating Software Requirements and Design Specifications. In: *IEEE Software* 1 (1984), Nr. 1, S. 75–88

[7] Chan, C.L. ; Lee, R.C.: *Symbolic Logic and Mechanical Theorem Prooving*. New York, USA : Academic Press, Inc., 1973

[8] Chang, C. L. ; Combs, J. B. ; Stachowitz, R. A.: A report on the Expert Systems Validation Associate (EVA). In: *Expert Systems with Applications* 1 (3) (1990), S. 217–230

[9] Davis, R.: *Application of meta-level Knowledge to the construction, maintenance, and use of large knowledge bases*, Stanford University, Diss., 1976

[10] Doyle, Jon: A Truth Maintenance System. In: *Artificial Intelligence* (1979), Nr. 12, S. 231–272

[11] Emde, Werner: *Informatik-Fachberichte*. Bd. 281: *Modellbildung, Wissensrevision und Wissensrepräsentation im Maschinellen Lernen*. Springer, 1991. – ISBN 3–540–54523–9

[12] Eshelman, L.: MOLE: A knowledge-acquisition system for cover-and-differentiate systems. In: *Automating Knowledge Acquisition for Expert Systems* (1988)

[13] Giarratano, Joseph C. ; Riley, Gary: *Expert Systems.* Boston, MA, USA : PWS Publishing Co., 1998. – ISBN 0534950531

[14] Ginsberg, A.: Knowledge-base reduction: A new approach to checking knowledge bases for inconsistency and redundancy. In: *Proc. 7th National Conference on Artificial Intelligence (AAAI 88)* 2 (1988), S. 585–589

[15] Gottlob, Georg ; Frühwirt, Thomas ; Horn, Werner: *Expertensysteme.* Springer Verlag Wien, 1990

[16] Gärdenfors, Peter: *Knowledge in Flux – Modelling the Dynamics of Epistemic States.* Cambridge, MA : MIT Press, 1988

[17] Harmon, Paul ; King, David: *Expert systems: artificial intelligence in business.* New York, NY, USA : John Wiley & Sons, Inc., 1985. – ISBN 0–471–80824–5

[18] Kleer, Johan D.: An assumption-based TMS. In: *Artificial Intelligence* (1986), Nr. 28, S. 127–162

[19] Kowalski, R.: *Studies in the Completeness and Efficiency of Theorem-proving by Resolution*, Univ. of Edinburgh at Edinburgh, Scotland, Diss., 1970

[20] Kranz, Dr.-Ing. O. ; Kokorniak, Michael: Einfache Neuformulierung von Entwurfszielen. In: *Digital Engineering Magazin* (2005), Nr. 4, S. 29–31

[21] Köpcke, Hanna ; Schröder, Andreas: Constraint Programming versus Logik: Vergleich zweier Ansätze zur Aufstellungsplanung von Chemieanlagen. (2004), Nr. CI-183/04

[22] Lavrač, Nada ; Džeroski, Sašo: *Inductive Logic Programming – Techniques and Applications.* Hertfortshire : Ellis Horwood, 1994

[23] Leuders, Peter: *Rechnergestützte Optimierung der Layoutplanung von Chemieanlagen*, University of Dortmund, Dortmund, Germany, Dissertation, 2002

[24] Marcus, Sandra: SALT: A Knowledge Acquisition Tool for Propose-and-Revise Systems. In: *Automating Knowledge Acquisition for expert systems* (1988), S. 81–121

[25] Martins, J. P. ; Shapiro, S. C.: A model for believ revision. In: *Artificial Intelligence* 35 (1988), S. 25–79

[26] Mierswa, Ingo: Incorporating Fuzzy Knowledge into Fitness: Multiobjective Evolutionary 3D Design of Process Plants. In: *Proc. of the Genetic and Evolutionary Computation Conference GECCO 2005*, 2005

[27] Morik, Katherina ; Kietz, Jörg-Uwe ; Emde, Werner ; Wrobel, Stephan: *Knowledge Acquisition and Machine Learning.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993. – ISBN 0125062303

[28] Ourston, Dirk ; Mooney, Raymond J.: Changing the rules: A comprehensive approach to theory refinement. In: *Proceedings of the Eighth National Conference on Artificial Intelligence*. Boston, MA : Morgan Kaufman, 1990, S. 815–820

[29] Preece, Alun: Validation of Knowledge-Based Systems: The State-of-the-Art in North America. In: *Journal of Communication and Cognition - Artificial Intelligence* 11 (1994), S. 381 – 413

[30] Preece, Alun: Evaluating Verification and Validation Methods in Knowledge Engineering. In: *Micro-Level Knowledge Management*, Morgan-Kaufman, 2001, S. 123–145

[31] Preece, Alun D.: Verification of rule-based expert systems in wide domains. In: *Research and Development in Expert Systems VI: Proc. Expert Systems 89* (1989), S. 66–77

[32] Quinlan, J. R.: Learning Logical Definitions from Relations. In: *Mach. Learn.* 5 (1990), Nr. 3, S. 239–266. `http://dx.doi.org/10.1023/A:1022699322624`. – DOI 10.1023/A:1022699322624. – ISSN 0885–6125

[33] Raedt, L. D.: *Interactive Theory Revision. An Inductive Logic Programming*. Academic Press, 1992

[34] Richards, Bradley L. ; Mooney, Raymond J.: Automated Refinement of First-Order Horn-Clause Domain Theories. Boston : Kluwer Academic Pulishers, 1995, S. 95–131

[35] Rousset, M.-C.: On the consistence of knowledge bases: the COVADIS system. In: *Proc. European Conference on AI (ECAI 88)* (1988), S. 79–84

[36] Saitta, Lorenza ; Botta, Marco ; Neri, Filippo: Multistrategy learning and theory revision. In: *Machine Learning* 11 (2/3) (1993), S. 153–172

[37] Shapiro, Stuart C.: Belief Revision and Truth Maintenance Systems: An Overview and a Proposal. Version: 1998. `http://citeseer.ist.psu.edu/shapiro98belief.html` (98-10). – Forschungsbericht. – Elektronische Ressource

[38] Sommer, Edgar ; Emde, Werner ; Kietz, Jörg-Uwe ; Morik, Katharina ; Wrobel, Stefan: *Mobal 4.1b9 User Guide*. 1996

[39] Sperberg-McQueen, C. M. ; Thompson, Henry: *XML Schema*. `http://www.w3.org/XML/Schema`. Version: 2006

[40] Swartout, Bill ; Gil, Yolanda: EXPECT: Flexible Representations for Flexible Acquisition. In: *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'95)* (1995)

[41] Swartout, Bill ; Gil, Yolanda: *Flexible Knowledge Acquisition Through Explicit Representation of Knowledge Roles*. 1996

[42] Wendenburg, Michael: Bohren mit Wenn und Dann. In: *Digital Engineering Magazin* (2005), Nr. 5, S. 48–49

[43] Wrobel, Stefan: *Concept Formation and Knowledge Revision*. Kluwer, 1994

[44] Wu, Ping ; Su, Stanley Y. W.: Rule validation based on logical deduction. In: *CIKM '93: Proceedings of the second international conference on Information and knowledge management*. New York, NY, USA : ACM Press, 1993. – ISBN 0–89791–626–3, S. 164–173

[45] Zeller, Prof. Dr.-Ing. A. ; Snelting, Gregor: *Skript zur Vorlesung Softwaretechnik I*. Universität des Saarlandes, Lehrstuhl für Softwaretechnik, 2001/2002

Index