

Seminar: A.I. Tools (WS06/07)

Jess, Teil 2

Jens Hauptert

(hauptert@xantippe.cs.uni-sb.de)

Saarbrücken, 21. April 2007

Betreuer:

Dr. Michael Kipp

Universität des Saarlandes
Fachbereich 6.2, Informatik
Lehrstuhl für Künstliche Intelligenz
Prof. Dr. Dr. h.c. mult. W. Wahlster

Abstract

Die vielen Vorteile und Möglichkeiten von regelbasierten Produktionensystemen lassen sich durch die Symbiose mit Java-Konzepten noch erweitern und vertiefen. JESS als ein in Java geschriebener Vertreter dieser Gruppe von Systemen, bietet nicht nur die Integration von und in Java und erlaubt sowohl die Nutzung von Java-Konzepten in JESS (JESS als Skriptsprache für Java) als auch regelbasierte Elemente für Java, sondern erreicht auch eine gute Performance durch die Verwendung des optimierten RETE-Algorithmus. Bei der Erstellung von Projekten lassen sich mit diesen Vorteilen durch die vorherige Planung und Modellierung von Systemen mit Flussdiagrammen schon mit wenigen Regeln funktionierende Expertensysteme erstellen.

Inhaltsverzeichnis

1	Einleitung	4
2	Jess und Java	5
3	Unter der Oberfläche	6
3.1	JESS-Vorgehensweise	6
3.2	Konfliktauflösung	7
3.3	Laufzeit	8
3.4	RETE-Algorithmus	8
3.5	Leistungsvergleich	12
4	PC-Reparaturassistent	14
4.1	Vom Flussdiagramm zur Regel	14
4.2	Rückwärtsverkettung	15
5	Zusammenfassung	16
	Literatur	18

1 Einleitung

Regelbasierte Systeme finden heute Anwendung in vielen Bereichen des täglichen Lebens. So gibt es zum Beispiel regelbasierte Systeme für Prognosen, Verfügbarkeitsvoraussagen, Distributionsplanung oder Bedarfs- und Lagersteuerung. Sie zeichnen sich insbesondere dadurch aus, dass der Programmablauf nicht prozedural vorgegeben werden muss. Die Regelbasis kann dabei relativ intuitiv erstellt werden und ist meist weniger abstrakt als prozeduraler Programmcode. Der große Vorteil solcher Systeme ist, dass sich diese mit dem vorhandenen Wissen und den gegebenen Regeln selbst zu einem festgelegten Ziel vorarbeiten.

Die Grundlagen und fundamentale Konzepte wurden bereits im ersten Beitrag zu JESS ausführlich erläutert. Dieser Teil zeigt ergänzend weiterführende Details zum Aufbau von JESS und zu Konzepten, die in diesem Zusammenhang oft zum Einsatz kommen.

Von anderen Systemen hebt sich JESS durch neue Fähigkeiten und verbesserte Performanz ab [FH03]. Dabei stellt sich als innovativstes Konzept die Verzahnung von JESS und Java heraus. Dadurch hat der Entwickler einerseits neue Möglichkeiten regelbasierte Konzepte in bestehende Java-Anwendungen zu integrieren. Durch die Verfügbarkeit der gesamten JESS-Funktionalität als Java-API lassen sich beliebige Anwendungen mit den Spezialitäten von JESS zu Expertensystemen erweitern und somit die optimale Geschwindigkeit bei solchen Berechnungen erreichen. Andererseits bietet sich JESS als Interpreter und Skriptsprache für Java an. Dadurch, dass JESS dem Entwickler erlaubt innerhalb von Regeln auf die vollständige Java-API zuzugreifen, lassen sich kleinere Codeblöcke problemlos im Vorhinein testen oder einzelne Algorithmen überprüfen, ohne ein einziges mal ein Java-Programm kompilieren zu müssen. Auch zur Erstellung von schnellen Prototypen mit oder ohne grafischer Oberfläche bietet sich JESS an [FH07b].

Durch die Verwendung eines optimierten RETE-Algorithmus wurde auch die Berechnung innerhalb der JESS-Verarbeitungslogik beschleunigt, so dass sich JESS, obwohl es nicht in einer nativen Programmiersprache geschrieben ist, durchaus mit anderen vergleichbaren Produkten messen kann. Teilweise werden sogar deutlich bessere Ergebnisse als bei bewährten Produkten erzielt.

Weiterhin zeichnet sich JESS dadurch aus, das Konzept der *Rückwärtsverkettung* nutzen zu können. Das aus diesem Grund mögliche zielgetriebene Beschaffen von Informationen oder Wissen erleichtert beispielsweise das Design von Dialogsystemen.

Bei der Erstellung der Regeln für solche Systeme können den Entwickler als Ergänzung zum JESS-System die Konzepte der *Flussdiagramme* unterstützen. Da auch Nicht-Fachleuten solche Diagramme erstellen und der Transfer von Diagrammen zu JESS-Regeln intuitiv und leicht durchgeführt werden kann, empfehlen sich diese Systeme umso mehr für den praktischen Einsatz.

Zur Darstellung aller erwähnten Punkte gliedert sich diese Arbeit in drei Teile und fasst in jedem Kapitel ein Aspekt genauer auf. Zuerst werden die Möglichkeiten der Verknüpfung von Java und JESS dargelegt. Anschließend werden im Kapitel „*Unter der Oberfläche*“ die Funktionsweisen, Strategien und Optimierungsmöglichkeiten aufgezeigt. Im dritten und letzten Bereich werden die Möglichkeiten von JESS an einem praktischen Beispiel gezeigt, wobei hier auch die *Rückwärtsverkettung* und das Konzept der Flussdiagramme erläutert wird.

2 Jess und Java

Die Interaktion zwischen JESS und Java ist „bidirektional“, dass heißt es lassen sich sowohl von JESS aus Java-Objekte und -Funktionen nutzen, als auch Java-Anwendungen entwickeln, die JESS-Funktionalität verwenden.

Java aus Jess nutzen Innerhalb eines JESS-Produktionensystems besteht vollständiger Zugriff auf Java-Funktionen und Konzepte. Dabei bildet JESS eine Skriptsprache für Java, wodurch JESS auch als Interpreter für Java-Anwendungen angesehen werden kann. Beispielsweise zeigt Listing 1 wie man durch den `bind`-Operator Java-Objekte an Elemente des JESS-Working Memory bindet oder mittels des `call`-Operators Java-Funktionen mit einer Argumentliste aufrufen kann. Dadurch lässt sich einerseits jede regelbasierte Anwendung mit eigenen Konzepten erweitern, wodurch der Funktionsumfang erweitert wird. Andererseits bietet JESS damit die Möglichkeit des Prototyping für Java. Auf dem JESS-Prompt lassen sich leicht kleinere Java-Codeblöcke testen oder sogar GUIs erstellen und evaluieren.

```
Jess> (bind ?prices (new java.util.HashMap))
<External-Address:java.util.HashMap>
Jess> (call ?prices put bread 0.99)
Jess> (call ?prices put peas 1.49)
Jess> (call ?prices put peas)

;;(call <WM-Eintrag> <Methode> <Argument1> <Argument2> ...)
```

Listing 1: Java mit JESS-Skripten

Java-Anwendungen mit Jess-Funktionalität Ebenso leicht lassen sich in die andere Richtung Konzepte eines Produktionensystems durch den Einsatz von JESS in Java-Anwendungen nutzen. Hierzu steht eine eigene API zur Verfügung, die alle Funktionen von JESS in einer Klasse `Rete` kapselt (siehe Listing 2). Durch die Methode `executeCommand()` lassen sich einzelne JESS-Anweisungen übergeben. Anschließend lässt sich das Produktionensystem mit `run()` starten und einzelne Element in der Wissensbasis mit `fetch()` auslesen. Somit bietet JESS auch die Möglichkeit, die Vorteile und Möglichkeiten von regelbasierten

Systemen in begrenztem Umfang in Java-Applikationen einzusetzen, beispielsweise nur an bestimmten Stellen, an denen „normaler“, oft unübersichtlicher, prozeduraler Code die Lesbarkeit einschränken würde und unter Umständen die spätere Wartbarkeit verschlechtert wird.

```

import jess.*;
...
Rete engine = new Rete();
engine.executeCommand("(assert fact (a 10) (b 20))");
engine.run();
Value val = engine.fetch("fact");

```

Listing 2: JESS-Komponenten in Java nutzen

Datentypenkonvertierung Zum Datenaustausch zwischen Konstrukten von JESS und Java sind feste Richtlinien definiert. Dabei ist zu beachten, dass einige JESS-Typen nicht in die intuitiv vermuteten Java-Typen konvertiert werden. Ein Beispiel sind die JESS-Schlüsselwörter `TRUE` und `FALSE`, die je nach Kontext zu den Java-Typen *String* oder *boolean* konvertiert werden. Es ist dabei nicht vorhersehbar, für welchen Typ sich JESS bei überladenen Methoden entscheidet. Eine genauere Übersicht liefert Tabelle 1.

Jess Typ	Java Typ
<code>RU.EXTERNAL_ADDRESS</code>	The wrapped object
<code>null</code>	null
<code>TRUE, FALSE</code>	String, boolean
<code>RU.INTEGER</code>	long, short, byte, ...
<code>RU.LIST</code>	Java Array

Tabelle 1: Datentypenkonvertierung

3 Unter der Oberfläche

Der nun folgende Abschnitt erläutert die Arbeitsweise von JESS und die verwendeten Strategien.

3.1 Jess-Vorgehensweise

Die grundlegende Vorgehensweise von JESS nach dem Aufruf von `(run)`, dass heißt nach dem die JESS-Engine gestartet wurde, gliedert sich in 5 Schritte. Zuerst werden alle Regeln gesucht, die zu den Fakten der Working Memory (WM) passen. Dies sind genau die Regeln, deren Prämisse sich mit dem WM unifizieren lassen. Sollte sich keine Regel finden, bricht JESS an dieser Stelle ab und beendet die Verarbeitung. Findet sich dagegen genau eine Regel wird diese

aktiviert und ausgeführt. Sind zwei oder mehr Regeln gefunden worden, stellt JESS eine Agenda zusammen, die alle möglichen Regeln enthält. Anschließend wird nach einer festgelegten Strategie eine Regel auf der Agenda ausgewählt, aktiviert und ausgeführt. Schließlich beginnt der Zyklus von vorne, in dem erneut alle Regeln gesucht werden, die sich mit dem unter Umständen veränderten WM unifizieren lassen.

3.2 Konfliktauflösung

Eine kritische Stelle in diesem Ablauf ist die Auswahl einer bestimmten Regel. Diese Stelle wird als Konfliktauflösung bezeichnet, da alle Regeln die auf der Agenda gelistet werden in Konflikt zueinander stehen. Um diesen Konflikt zu beheben, werden sogenannte *Konfliktauflösungsstrategien* verwendet. JESS bietet für diesen Fall zwei unterschiedliche Strategien an, einerseits die „Breiten“- und andererseits die „Tiefen“-Strategie.

Die „Breiten“-Strategie erinnert von ihrem Namen her an die Breitensuche, bei der zuerst alle direkten Nachfolger besucht werden. In JESS führt dieser Ansatz dazu, dass Regeln genau in der Reihenfolge ihrer Aktivierung feuern.

Die „Tiefen“-Strategie ist das Standardverfahren und arbeitet etwas komplizierter. Dazu wird, wie bei allen anderen Strategien auch, zur Bestimmung der nächsten zufeuernden Regel zunächst jede Regel mit einem sogenannten *sali-ence*-Wert versehen. Dieser Wert gibt an, wann die Regel aktiviert wurde. Dies geschieht indem JESS einen Zähler verwendet, der bei jeder Regelausführung inkrementiert wird. Sobald nun eine Regel aktiviert wird, wird der salience-Wert dieser Regel auf den Wert des Zählers gesetzt. Das heißt, je später eine Regel aktiviert wurde, umso größer ist ihr salience-Wert. Letztlich feuert bei der „Tiefen“-Strategie dann immer die Regel, die den höchsten salience-Wert besitzt, das heißt praktisch immer die Regel, die als letztes aktiviert wurde. Um in diese Auswahl manuell eingreifen zu können, ist es möglich für jede Regel den salience-Wert manuell verschieben zu können. Das bedeutet mit der Codezeile

```
(declare (salience +/- <Wert>))
```

 (1)

kann der salience-Wert um den Wert <Wert> verschoben werden. Somit kann die Priorität einer Regel beeinflusst werden, wodurch man zum Beispiel eine Regel abstimmen kann indem man ihren salience-Wert stark verkleinert.

Zusätzlich zu den beiden in JESS vorhandenen Konfliktauflösungsstrategien ist es möglich, über eine Implementierung der in JESS vorhandenen Java-API `jess.Strategy` und dem festlegen durch

```
(set-strategy <classname>)
```

 (2)

eine eigene Strategie zu erstellen. Dies ist allerdings nur in Ausnahmen nötig.

3.3 Laufzeit

Betrachtet man sich die in Kapitel 3.1 erläuterte Vorgehensweise, so beschreibt diese nur einen primitiven Algorithmus. Denn dabei wird schlicht eine Liste aller Regeln vorgehalten und in jedem Durchgang alle Prämissen jeder Regel mit dem Working Memory verglichen um eine passende Kombination zu finden. Dadurch ergibt sich eine Laufzeit von

$$O(R * F^p) \quad (3)$$

wobei R die Anzahl der Regeln, F die Anzahl der Fakten im Working Memory und p die durchschnittliche Anzahl der Vergleiche („if's“) pro Regel darstellt. Dies bedeutet, dass die Laufzeit exponentiell mit der Anzahl der Vergleiche anwächst. Beobachten kann man dabei die Schwachstelle dieses primitiven Algorithmuses, nämlich die Tatsache, dass die Mehrzahl der Tests bei jeder Iteration das gleiche Ergebnis haben, da sich das Working Memory bei jedem Durchgang immer nur minimal verändert, so dass die Mehrheit der Prämissentests aller Regeln zum selben Ergebnis evaluieren. Zusätzlich lässt sich feststellen, dass viele Tests mehrfach pro Zyklus durchgeführt werden, da unter Umständen viele Prämissen von verschiedenen Regeln zu großen Teilen übereinstimmen. Durch diese beiden Schwächen wird das System bei großen Regelsystemen sehr langsam, da die Verarbeitung äußerst ineffizient erfolgt.

Um diesen Missstand zu beseitigen, müssen zuerst Informationen über regelbasierte Systeme zusammengetragen werden. Dabei fällt auf, dass die Anzahl der Regeln üblicherweise konstant bleibt, dass heißt es werden nur vereinzelt neue Regeln hinzugefügt oder entfernt, aber selten die gesamte Regelmenge ausgetauscht. Das Working Memory hingegen ändert sich praktisch in jedem Zyklus. Allerdings kann man auch hier feststellen, dass diese Änderungen meist nur wenige Elemente betreffen. Aus diesen Erkenntnissen lässt sich die Regelverarbeitung optimieren und sehr effizient gestalten.

3.4 RETE-Algorithmus

Idee Die Grundidee besteht darin, bereits bekannte Testergebnisse so weit wie möglich wieder zu verwenden. Erneute Tests sollen sich dabei nur auf tatsächlich geänderte Fakten beschränken. Aus diesem Ansatz ist der *RETE*-Algorithmus entstanden. Der Name leitet sich dabei vom lateinischen „*rete*“ = Netz ab. Entwickelt wurde der Algorithmus 1979 von *Charles Forgy* an der Carnegie Mellon University [For82]. Dieses System erlaubt eine äußerst effiziente Regelverarbeitung, welche im Folgenden näher beschrieben wird. Der RETE-Algorithmus ist weit verbreitet und wird in vielen regelbasierten Systemen eingesetzt; unter anderem in OPS5¹, CLIPS², ART³, und JESS.

¹Official Production System, Charles L. Forgy (siehe <http://www.cs.gordon.edu/local/courses/cs323/OPS5/ops5.html> (Stand:19.02.07))

²C Language Integrated Production System, NASA-Johnson Space Center (siehe <http://www.ghg.net/clips/CLIPS.html> (Stand:19.02.07))

³Automated Reasoning Tool, Inference Corporation

Knotentypen Der RETE-Algorithmus verwendet dabei intern ein Netz von untereinander verbundenen Knoten. Diese Knoten repräsentieren dabei einen oder mehrere Tests einer Regel. Dabei haben die Mehrzahl der Knoten genau einen Eingang und genau einen Ausgang. Diese werden als *Einerknoten* bezeichnet und repräsentieren Bedingungen, die nur ein Element des Working Memory betreffen, beispielsweise der Test, ob ein Element sich im Working Memory befindet. Ein Teil der Knoten allerdings besitzt zwei Eingänge und wird daher als *Zweierknoten* bezeichnet. Diese beschreiben dabei Beziehungen zwischen Einzelbedingungen, beispielsweise, ob zwei Elemente im Working Memory in bestimmten Slots die gleichen Werte besitzen. Eine letzte Gruppe stellen die *Terminalknoten* dar, die jede Regel abschließen (siehe Abbildung 3).

Die Ausgänge aller Knoten können dabei beliebig verzweigt und als Eingang für andere Knoten zu verwendet werden. Um die Bedingung, die der Knoten repräsentiert darzustellen, wird diese als Text in den Knoten geschrieben. Zum Beispiel bedeutet = x ?, dass das Fact x im Working Memory enthalten ist. Bei Zweierknoten lassen sich die beiden Eingänge in der Bedingung mit LEFT oder RIGHT ansprechen. So stellt beispielsweise die Regel LEFT.a == RIGHT.a den Fall dar, dass das Fact am linken und am rechten Eingang einen Slot a besitzen und dass der Wert dieses Slots bei beiden Eingängen des Zweierknotens identisch sein muss (siehe Abbildung 4).

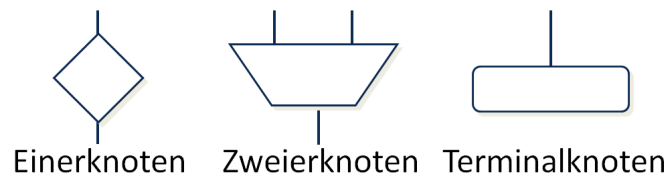


Abbildung 3: RETE-Knotentypen

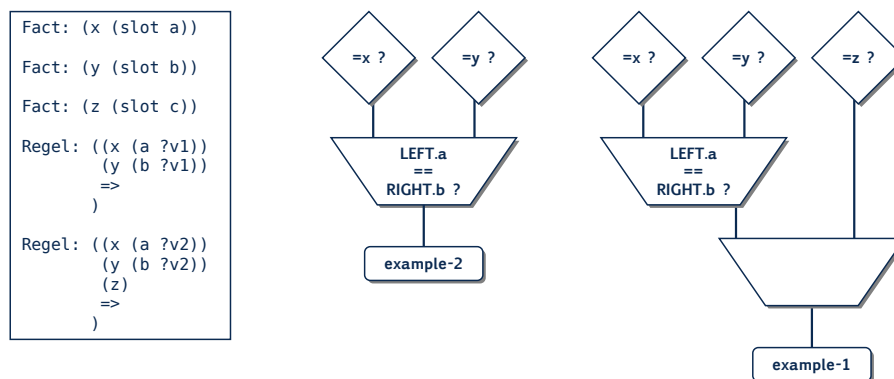


Abbildung 4: RETE-Übersicht

Optimierung Bei der Erstellung von Regelsystemen kommt es häufig vor, dass Teile der Prämissen vieler Regeln identisch sind. Wird aus solchen Systemen ein RETE-Netz erstellt, sind große Teile des Baumes identisch und kommen sehr häufig vor. Da die entsprechenden Tests unnötigerweise mehrfach durchgeführt werden, wird bei JESS eine optimierte RETE-Version verwendet. So zeigt Abbildung 5, einen unoptimierten Beispielbaum, der entsteht, wenn zuerst die Regel `example-1` und anschließend `example-2` eingefügt wird. Bei diesem Beispiel kann man die Redundanz sehr gut erkennen, denn alle Tests die zur Evaluierung der Regel `example-2` notwendig sind, werden schon bei der Regel `example-1` durchgeführt. Die Optimierung dieses Netzes erfolgt in zwei Stufen. Zuerst werden die Einerknoten mit den Bedingungen `=x ?` und `=y ?` die zur Regel `example-2` gehören entfernt und die Ergebnisse der identischen Knoten von Regel `example-1` verwendet (siehe Abbildung 6 auf der nächsten Seite). Anschließend wird auch der Zweierknoten, der den Test `LEFT.a == RIGHT.b` durchführt bei Regel `example-2` entfernt und durch das Ergebnis des äquivalenten Tests von Regel `example-1` ersetzt (siehe Abbildung 7 auf der nächsten Seite). Das neue optimierte RETE-Netz zeichnet sich nun dadurch aus, dass für die Regel `example-2` kein einziger neuer Test mehr notwendig ist, da das Ergebniss von Regel `example-1` übernommen werden konnten. Einzig der Terminalknoten bleibt noch als Repräsentant der Regel übrig.

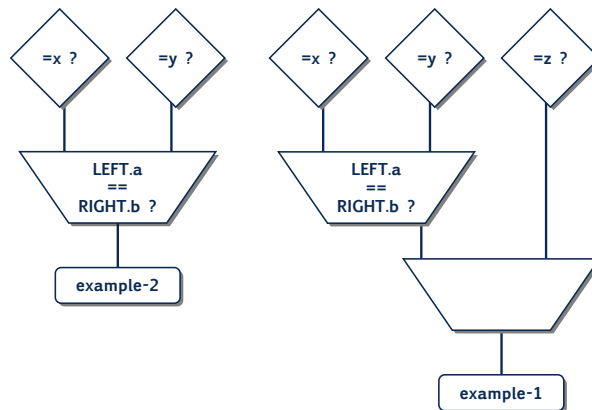


Abbildung 5: RETE-Algorithmus, Teil 1

Rückmeldung Bei der Eingabe einer neuen Regel erhält der Benutzer eine Rückmeldung von JESS die Aufschluss darüber gibt, wie die Regel in das RETE-Netz eingefügt wurde. Eine solche Ausgabe sieht beispielsweise wie folgt aus:

$$+ 1 + 1 + 2 = 1 = 1 = 2 + t \quad (4)$$

wobei jeder Ausdruck folgende Form hat:

$$[[[+ | =][1|2]]* | + t] \quad (5)$$

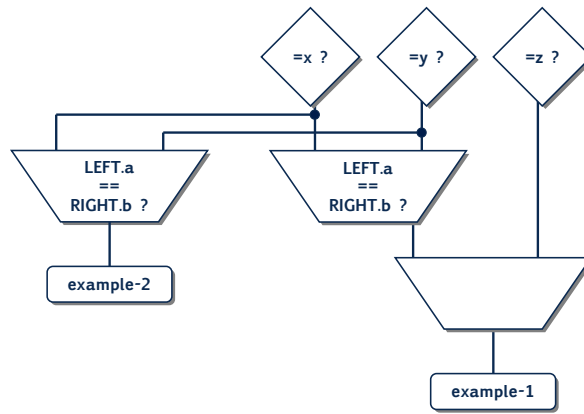


Abbildung 6: RETE-Algorithmus, Teil2

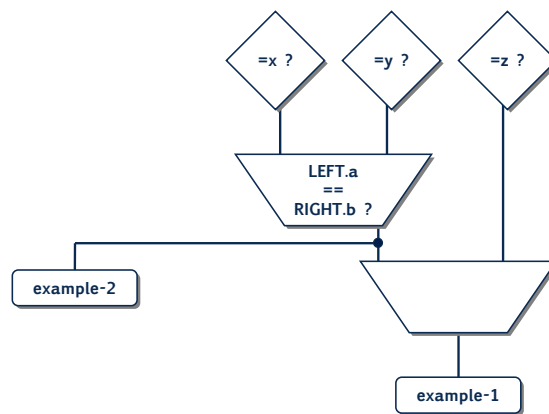


Abbildung 7: RETE-Algorithmus, Teil3

Diese Zahlen geben Aufschluss darüber, ob für eine Regel neue Knoten erzeugt wurden oder bestehende übernommen werden konnten. So gibt der Ausdruck +1 an, dass ein neuer Einerknoten erstellt wurde. Analog dazu bedeutet +2, dass ein neuer Zweierknoten erstellt wurde. Falls Einer- oder Zweierknoten übernommen werden konnten, wird dies mit = 1 oder = 2 signalisiert. Abgeschlossen wird jede Ausgabe mit +t da der Terminalknoten immer eingefügt wird um eine Regel abzuschließen. Dieser Terminalknoten stellt somit im Netz das Ergebnis dieser Regel dar. Einige Beispielnetze und deren Ausgabe zeigt Abbildung 8.

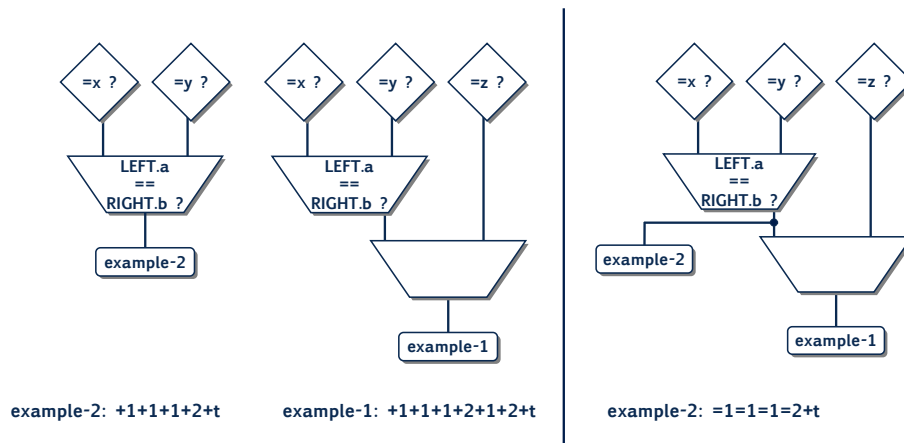


Abbildung 8: RETE-Rückmeldung in JESS

Visualisierung Um sich das erstellte RETE-Netz anzuschauen, bietet JESS den Befehl (`view`) an. Dieser öffnet ein eigenes Fenster, in dem das Netz in Form von farbigen Quadraten präsentiert wird (siehe Abbildung 9). Dabei ist die Darstellung des Netzes eng mit der RETE-Darstellung verwandt. So stellen rote Quadrate Einerknoten und grüne Quadrate Zweierknoten dar. Die jede Regel abschließenden Terminalknoten werde als cyan-farbene Quadrate angezeigt. Die einzige Besonderheit in der JESS-Repräsentation ist der *Rechts-nach-Links-Adapter*. Dieser wird intern verwendet, um den ersten Test der linken Seite einer Regel mit dem Netzwerk zu verknüpfen.

3.5 Leistungsvergleich

Hintergrund Um die Praxistauglichkeit von JESS besser einschätzen zu können, wird dessen Leistungsfähigkeit mit dem schon länger im Einsatz befindlichen *CLIPS* verglichen. Während JESS ein Java-basiertes System ist, das 1995 entwickelt wurde und dessen Ziel es war, regelbasierte Systeme mit Java zu verknüpfen, war die Motivation der Entwicklung des C-basierten *CLIPS*, welches bereits 10 Jahre früher entstand (1985), der Ersatz der damaligen kommerziellen und kostenintensiven Werkzeuge. Interessant in diesem Zusammenhang ist auch die Tatsache, dass beide Hersteller, aus dem nahen Umfeld der Rüstungsindustrie und des Militärs stammen. So wurde *CLIPS* von einem NASA-Team

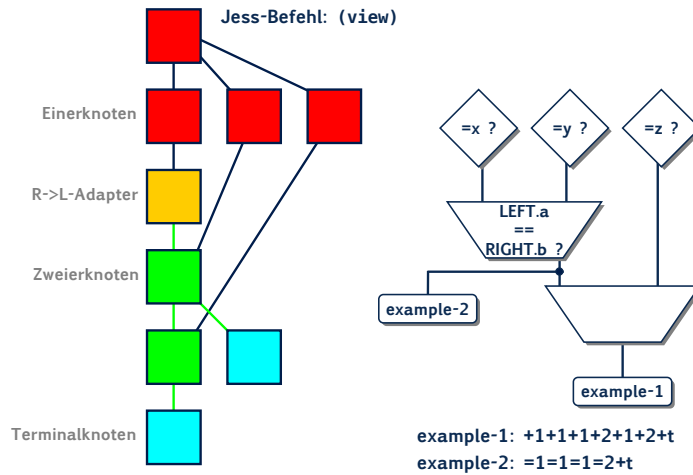


Abbildung 9: Internes JESS-RETE-Netz

entwickelt während JESS ein Produkt der Sandia National Labs ist. Diese Firma ist für die Konstruktion aller nicht nuklearen Teile von Nuklearwaffen zuständig.

Benchmark Die Idee des Benchmarks ist ein Paradebeispiel für regelbasierte Systeme. Die Aufgabe des sogenannten *Manners Benchmark* besteht darin, Gäste auf einer Geburtstagsfeier an einen runden Tisch so zu verteilen, dass möglichst häufig Personen mit gleichem Geschlecht und gleichen Interessen nebeneinander sitzen. Abbildung 10 zeigt dabei, die Dauer der Berechnung bei vorgegebener Anzahl von Gästen:

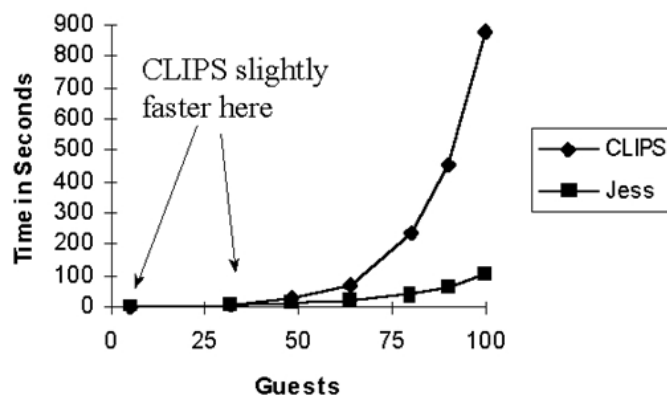


Abbildung 10: Manners Benchmark

Wie man deutlich erkennen kann, ist die Leistung beider System bis circa 50 Gäste noch annähernd gleich. Allerdings kann sich JESS ab 75 Gästen deutlich

von CLIPS absetzen und erreicht laut Hersteller bei ungefähr 200 Gästen etwa die 20-fache Leistung von CLIPS. Dies entspricht in etwa 80.000 Regeln pro Sekunde oder 100.000 Fakten pro Sekunde (Pentium III 800 MHz). Somit ist die Leistung von JESS trotz der Einschränkungen der Java-VM deutlich höher als ältere Systeme [FH07a].

4 PC-Reparaturassistent

Anhand eines PC-Reparaturassistenten als regelbasiertes Expertensystem zur Fehlerdiagnose, sollen die Möglichkeiten von Flussdiagrammen und das Konzept der Rückwärtsverkettung näher erläutert werden.

4.1 Vom Flussdiagramm zur Regel

Bei der Erstellung von Regeln für Produktionensysteme stellen Flussdiagramme oft eine große Hilfestellung dar. Durch die Darstellung als Diagramm lassen sich auf einfache Art und Weise Regeln generieren, da die Diagramme bei einem festgelegten Startpunkt beginnen und für verschiedene Fälle das Diagramm bis zu einem Zielpunkt durchlaufen. Dabei werden Entscheidungsknoten passiert, die einen Zustand überprüfen, der in der Regel üblicherweise wahr oder falsch ist. Somit kann jeder Durchlauf vom Start bis zum Ziel als eigene Regel angesehen werden und jeden der dabei passiertten Entscheidungsknoten als Prämisse der Regel verwendet werden (siehe Abbildung 11). Abschließend müssen nur noch die durchzuführenden Aktionen auf der rechten Seite der Regel definiert werden. Somit bieten Flussdiagramme dem Ersteller von Regeln ein wichtiges Hilfsmittel, da Flussdiagramme die Problematik auf für Personen verständlich machen, die nicht mit regelbasierten Systemen vertraut sind.

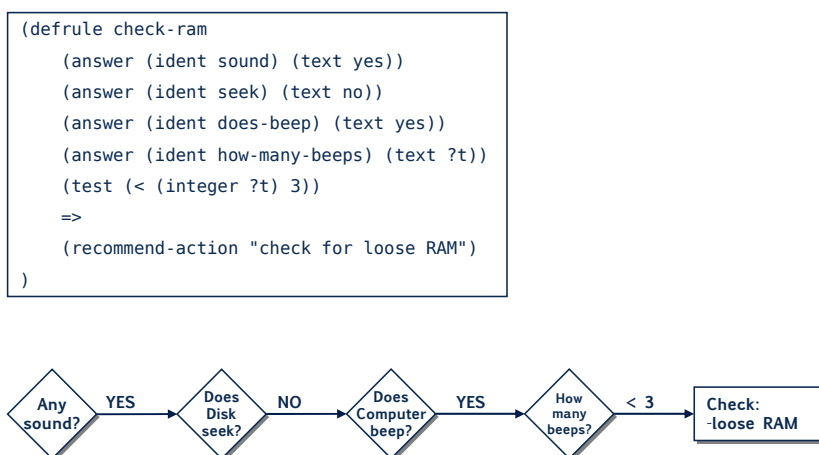


Abbildung 11: Zusammenhang zwischen Flussdiagramm und JESS-Regel

4.2 Rückwärtsverkettung

Zur Verarbeitung von Regeln kam bisher ausschließlich die Vorwärtsverkettung zum Einsatz. Dabei wird wie in Kapitel 3.1 beschrieben, in jedem Durchgang eine Regel gesucht, deren Prämisse zu den Fakten des Working Memory passt. Als Alternative zu dieser Methode bieten regelbasierte Systeme zusätzlich noch die *Rückwärtsverkettung*. Bei dieser Vorgehensweise werden bestimmte Fakten als Ziel (*goal*) definiert. Anschließend sucht der Algorithmus eine Regel, deren Konklusion das Ziel generiert. JESS jedoch beherrscht keine echte Rückwärtsverkettung, sondern nur eine interne Simulation über Vorwärtsverkettung. Daraus ergibt sich der Umstand, dass man in JESS nicht einfach mit einem Befehl die Rückwärtsverkettung aktivieren kann und anschließend arbeitet JESS in umgekehrter Richtung. Vielmehr sind in JESS die bereits erwähnten *goals* notwendig um die Rückwärtsverkettung nutzen zu können. Dazu wird das Ziel mit folgender Zeile festgelegt:

```
(do-backward-chaining <fact>) (6)
```

Ergänzend werden noch Hilfsregeln angelegt (siehe Abbildung 12), die die Aufgabe haben das fehlende Element im WM zu generieren. Diese Regeln setzen dabei über ihre Prämissen das Vorhandensein von Fakten der Form (**need-*<fact>***) voraus und generieren bei ihrer Ausführung das noch fehlende Element (**<fact>**) (siehe Listing 12).

```
(defrule trigger123
  (need-<fact>)
  =>
  (assert <fact>)
)
```

Listing 12: JESS-Hilfsregel zur Rückwärtsverkettung

JESS fügt daher in diesem Fall automatisch das Element (**need-*<fact>***) in die Wissensbasis ein um die Rückwärtsverkettung zu starten und das fehlende Element (**<fact>**) zu generieren. Daher ergibt sich der in Abbildung 13 skizzierte Ablauf, falls das Fakt **answer** als Ziel deklariert wurde. Steht eine Regel zur Verfügung, die als Prämisse **answer** benötigt und sich **answer** nicht im Working Memory befindet, wird automatisch das Fakt **need-answer** generiert (Schritt 1). Anschließend kann die Regel **supply-answers**, die spezielle für die Rückwärtsverkettung hinzugefügt wurde feuern (Schritt 2). Die Besonderheit dieser Regel ist, dass diese als Prämisse **need-answer** benötigt und als Konklusion **answer** generiert. Eine solche Regel bietet sich beispielsweise für den Fall an, dass ein bestimmtes Fakt noch fehlt um eine Regel ausführen zu können. Diese Regel ist unter Umständen in der Lage das fehlende Fakt zu generieren. Im letzten Schritt 3 steht nun das Fakt **answer** im Working Memory, so dass die ursprüngliche Regel **power-supply-broken** feuern kann.

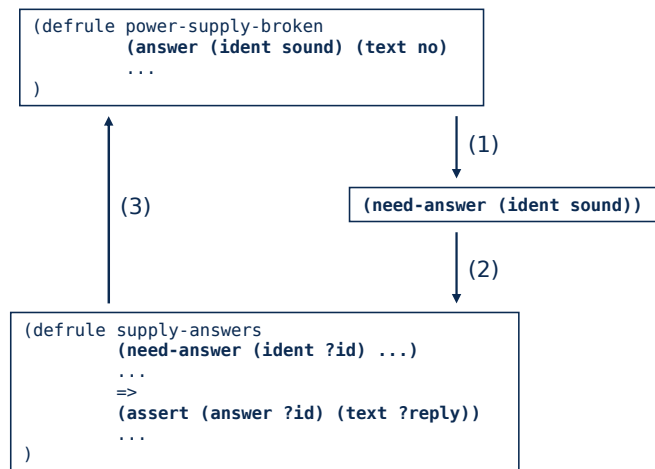


Abbildung 13: Beispiel einer Rückwärtsverkettung

5 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass JESS als ein Vertreter der regelbasierten Systeme viele innovative Konzepte bietet um sich von den Mitbewerbern abzuheben.

So ermöglicht die starke Verzahnung zwischen JESS und Java einen fließenden Übergang zwischen beiden Systemen. Auf der einen Seite lassen sich regelbasierte Konzepte in JESS mit Java-Objekt wie zum Beispiel einer GUI anreichern oder Java-Prototypen mit JESS erstellen, wodurch sich JESS als eine Skriptsprache für Java empfiehlt. Auf der anderen Seite erlaubt es die JESS-API in jedem beliebigen Java-Programm auf die Vorteile und Möglichkeiten von Produktionssystemen zurückgreifen zu können.

Betrachtet man sich dann den Aufbau und die Arbeitsweise von JESS genauer, stellt man fest, dass eine naive Implementierung nicht die Leistung erbringt, die für ein solches System erforderlich ist. Deshalb basiert JESS auf dem RETE-Algorithmus wodurch eine effiziente Verarbeitung von Regeln möglich wird. Die Benchmarkergebnisse zeigen, dass JESS nicht nur zu etablierten Systemen wie CLIPS aufgeschlossen, sondern diese in vielen Bereichen bereits überholt hat. Auch die Möglichkeiten zur Erstellung von Anwendungen, empfehlen den Einsatz von JESS. So bietet die Rückwärtsverkettung die automatische Ausführung von Aktionen bei Bedarf. Damit lassen sich einfache Dialogsysteme aufbauen, die Benutzer gezielt nach Informationen fragen, zum Beispiel in Form eines Interviews. Ein zusätzlicher Vorteil dieses Konzeptes ist die Konzentration auf notwendige Regeln. Da das System große Teile der Informationsbeschaffung übernimmt, kann sich der Entwickler auf die eigentliche Verarbeitung konzentrieren. Wie das Beispiel des PC-Reparaturassistenten zeigt, lassen sich bereits mit wenigen Regeln einfache Expertensysteme erstellen, was die Leistungsfähigkeit dieser Systeme beweist.

Resümierend lässt sich somit festhalten, dass regelbasierte Systeme viele Vorteile bieten. So lassen sich aus im Vorhinein erstellten Flussdiagrammen sehr leicht Regeln ableiten. Wodurch der eigentliche „Programmieraufwand“ sehr gering ist. Dieser Aufwand wird auch reduziert, weil der Kontrollfluss des Systems nicht festgelegt werden kann und muss. Dies schränkt den Entwickler zwar in einigen Fällen ein, erleichtert ansonsten aber die Arbeitsleistung und vermindert die Fehlerhäufigkeit. Diese Vorteile belegen die seit Jahren im Einsatz stehenden und weit verbreiteten Expertensysteme, die auf regelbasierten Konzepten basieren.

JESS im Besonderen vereint die gesamte Java-Welt mit der regelbasierter Systeme. Dadurch sind alle Konzepte und Vorteile der einen Seite auch in der anderen verfügbar. Somit bietet JESS eine Synthese von prozeduralem und deklarativem Programmieren. Dabei wirkt sich einzig nachteilig die fehlende Möglichkeit aus, objektorientierte Konzepte in JESS zu verwenden, wie es beispielsweise in *COOL*, einer objektorientierten Erweiterung von CLIPS möglich ist. Wiederum für JESS spricht die gute Leistung bei der Regelverarbeitung, da der optimierte RETE-Algorithmus auch große und komplexe Systeme mit ausreichender Performanz ermöglicht. Somit erlaubt JESS die Nutzung des Besten aus der prozeduralen und regelbasierten Welt.

Literatur

- [FH03] Friedman-Hill. *JESS In Action - Java Rule-based Systems*. Manning Publications Co., 2003. ISBN 1-93011-089-8.
- [FH07a] Ernest Friedman-Hill. *JESS, Implementing A High-Performance Symbolic Reasoning Engine In Java*, 21. Februar 2007. <http://aaaproduct.gsfc.nasa.gov/TEAS/Jess/JessUMBC/index.htm> (Stand: 20.04.2007).
- [FH07b] Ernest Friedman-Hill. *JESS, the Rule Engine for the Java Platform*, 19. Februar 2007. <http://herzberg.ca.sandia.gov> (Stand: 20.04.2007).
- [For82] Charles L. Forgy. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*, in: *Artificial Intelligence*, 1982.