Wissensbasierte Systeme

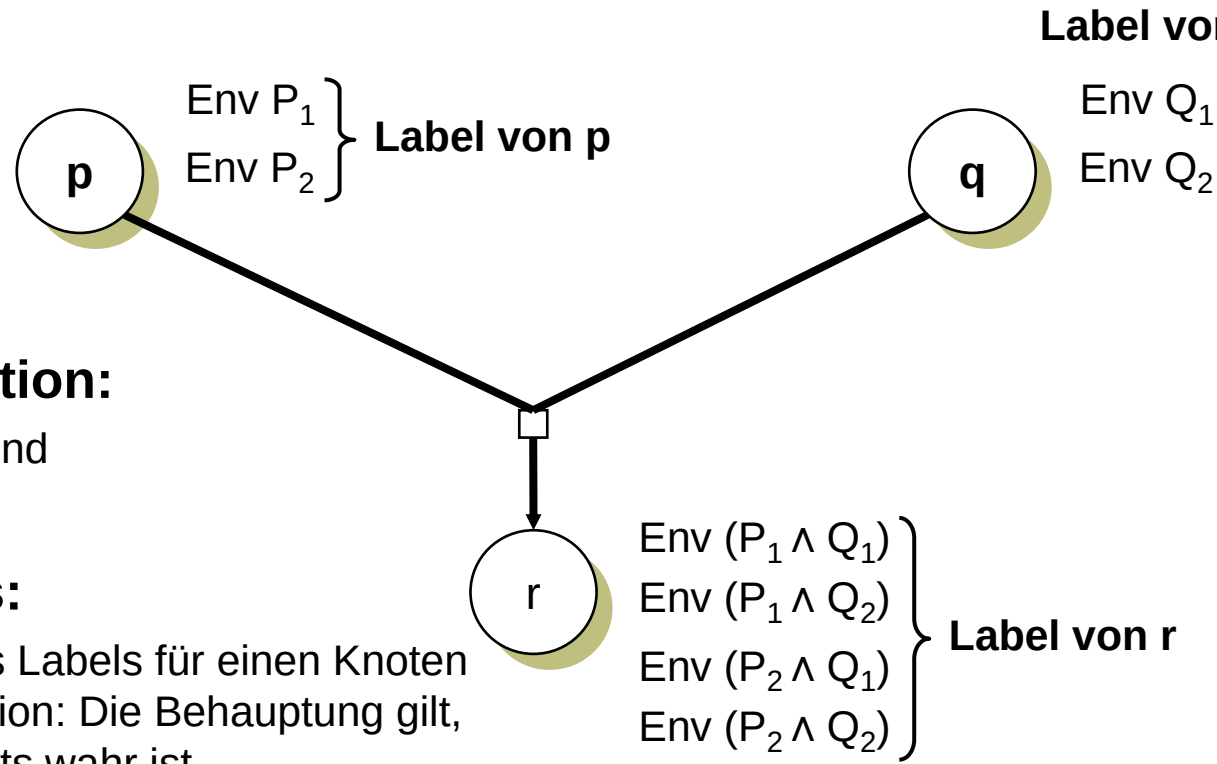
Vorlesung 8 vom 08.12.2004

Sebastian Iwanowski
FH Wedel

Wissensbasierte Systeme

1. Motivation
2. Prinzipien und Anwendungen
3. Logische Grundlagen
4. Suchstrategien
5. Modellbasierte Diagnose
   - Kandidatengenerierung
   - ➔ Konfliktgenerierung
   - Zusammenspiel von Wertpropagierung, Konfliktgenerierung, Kandidatengenerierung
   - Weitere Funktionalitäten von MDS
   - Komponentenmodellierung
6. Andere Diagnosemethoden
7. Weitere Wissensrepräsentationsformen
8. Bewertung wissensbasierter Systeme

Labelaktualisierung in einem ATMS



Bedeutung der Justification:

- r gilt, wenn p und q wahr sind (Konjunktion)

Bedeutung eines Labels:

- Mehrere Environments des Labels für einen Knoten entsprechen einer Disjunktion: Die Behauptung gilt, wenn eine der Environments wahr ist

Eliminierung überflüssiger Environments:

- Widersprüchliche Environments können weggelassen werden.
- Damit können auch alle Environments weggelassen werden, die Konflikte enthalten.
- Environments, die aus anderen Environments desselben Labels folgen, können weggelassen werden.

Anwendung eines ATMS durch den Problemlöser

Eingabe vom Problemlöser:

- Annahmeknoten
- “Normale” Knoten
- Justifications zwischen den Knoten

Ausgabe an den Problemlöser:

- Menge der minimalen Konflikte

Das ATMS macht automatisch:

Das sind sehr viele Operationen !

- Erzeugung der Labels für die Annahmeknoten
- Aktualisierung der Labels aller Conclusions, von denen sich der Label eines Antecedents geändert hat
- Eliminierung aller überflüssigen Environments

Anwendung eines ATMS durch den Problemlöser

Eingabe vom Problemlöser:

- Annahmeknoten
- “Normale” Knoten
- Justifications zwischen den Knoten

Ausgabe an den Problemlöser:

- Menge der minimalen Konflikte

Zusammenspiel mit dem Kandidatengenerierer:

- Bilde alle Annahmeknoten für die Fokusdiagnosen
- Berechne alle Werte, die sich aus den Annahmen der Fokusdiagnosen ergeben, und bilde die entsprechenden Behauptungsknoten und Justifications
- Entnimm dem ATMS die neuen Konflikte

Optimierung 1: Focusing ATMS

Eigenschaft, die verbessert werden soll:

- Das ATMS berechnet ***alle*** minimalen Konflikte, die zu einer beliebigen Kombination von Annahmen gelten.
- Die meisten davon sind für den Problemlöser gar nicht von Interesse.

Daher zusätzliche Eingabe vom Problemlöser:

- Diejenigen Environments, die von Interesse sind (Fokusenvironments)

Ausgabe:

- Menge der minimalen Konflikte, die in einem Fokusenvironment enthalten sind

Geänderte Funktionsweise des ATMS:

- Environments werden nur gebildet, wenn sie Teilmengen eines Fokusenvironments sind
- Bei Eingabe eines neuen Fokusenvironments werden automatisch alle Labels aktualisiert

Optimierung 2: Lazy ATMS

Eigenschaft, die verbessert werden soll:

- Das ATMS muss seine Labels häufig aktualisieren
- Viele Aktualisierungen werden nach außen gar nicht sichtbar, bevor sie von neuem überschrieben werden.

Geänderte Funktionsweise des ATMS:

- Labels werden erst berechnet, wenn nach ihnen explizit gefragt wird (in der Regel wird vor allem nach dem Label des Widerspruchsknotens gefragt)

In MDS (DaimlerChrysler) realisierte Variante:

- ATMS, das gleichzeitig fokussiert und lazy arbeitet

Beschreibung in der Dissertation von Mugur Tatar

Wissensbasierte Systeme

1. Motivation
2. Prinzipien und Anwendungen
3. Logische Grundlagen
4. Suchstrategien
5. Modellbasierte Diagnose
   - Kandidatengenerierung
   - Konfliktgenerierung
   - ➔ Zusammenspiel von Wertpropagierung, Konfliktgenerierung, Kandidatengenerierung
   - Weitere Funktionalitäten von MDS
   - Komponentenmodellierung
6. Andere Diagnosemethoden
7. Weitere Wissensrepräsentationsformen
8. Bewertung wissensbasierter Systeme

Wertpropagierung und ATMS

Was versteht man unter Propagierung im MDS-Kontext ?

- Propagierung ist die Weiterleitung von Informationen über ein Netzwerk aus Kanten und Knoten.

Trennung von Wertpropagierung und ATMS:

- Das **ATMS** ist verantwortlich für die Propagierung der Environments in einem gegebenen Netzwerk von Wertabhängigkeiten.
- Das Netzwerk von Wertabhängigkeiten wird in einem Rule Propagator (**RP**) hergestellt, der die Justifications aus den Regeln für die Verhaltensmodi der Komponenten zusammensetzt.
- Der RP ist genauso faul wie sein ATMS:
  - Werte werden nur weiterpropagiert, wenn sie von Environments unterstützt werden, die im gegenwärtigen Fokus stehen.
  - Das ATMS teilt dem RP unaufgefordert mit, welche Werte neu von Fokusenvironments unterstützt werden und initiiert die **(Propagation) Tasks**

Wertpropagierung und ATMS

Was bringt die Trennung von Wertpropagierung und ATMS ?

1. Antwort: Bessere Softwarearchitektur durch Modularisierung

- **Werte** entstehen meistens aus Beobachtungen (Messungen) und gezielten Eingaben. Diese sind spärlich, daher gibt es **nicht viele** resultierende Werte.

- **Environments** entstehen aus Annahmen über Verhaltensmodi. Von diesen gibt es **sehr viele** (selbst bei Einfachfehlern mindestens so viele wie Komponenten).

- Daher werden Fokusenvironments häufiger revidiert, als neue Werte berechnet werden. Diese Revision kann dann als ein ATMS-internes Problem behandelt werden.

Anm.: Die Aufteilung in ein RP- und ATMS-Modul fördert enge Modulbindung und lose Modulkopplung*)*

Wertpropagierung und ATMS

Was bringt die Trennung von Wertpropagierung und ATMS ?

2. Antwort: Einsatz des ATMS für erweiterte Aufgaben

- Es können auch andere Annahmen als Verhaltensmodi für Komponenten untersucht werden:

  Beispiele:

  - Annahmen über Werteingaben (control inputs)

    (für die Berechnung sinnvoller Testsituationen)

  - Annahmen über Komponentenzustände (bei dynamischen Komponenten)

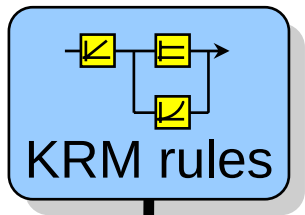
    (für dynamische Komponenten, deren Zustand unbekannt ist)

  - Annahmen über beliebige andere Werte

    (könnte für Beobachtungspunkte interessant sein)

Wertpropagierung und ATMS

Wissensbasis
(Komponentenmodellierung
plus Systemzusammenhang)



KRM:
Knowledge **R**epresentation **M**anager

RP System

RP Inference
- schedule tasks
- execute tasks (fire rules)
- make and store nodes, justifications, tasks

Add node,
justification,
task

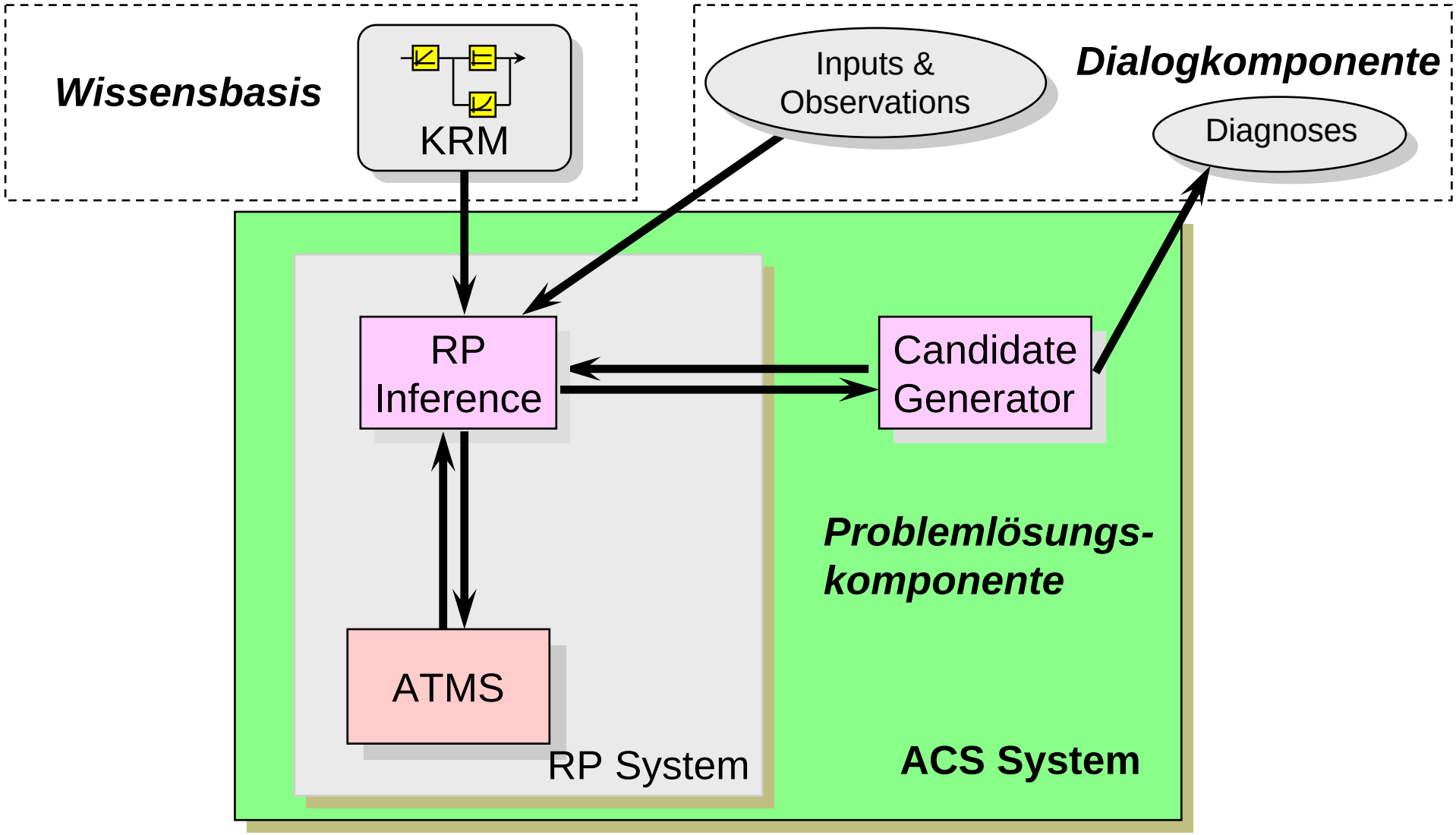
Activate task
(if in focus)

Set focus

ATMS

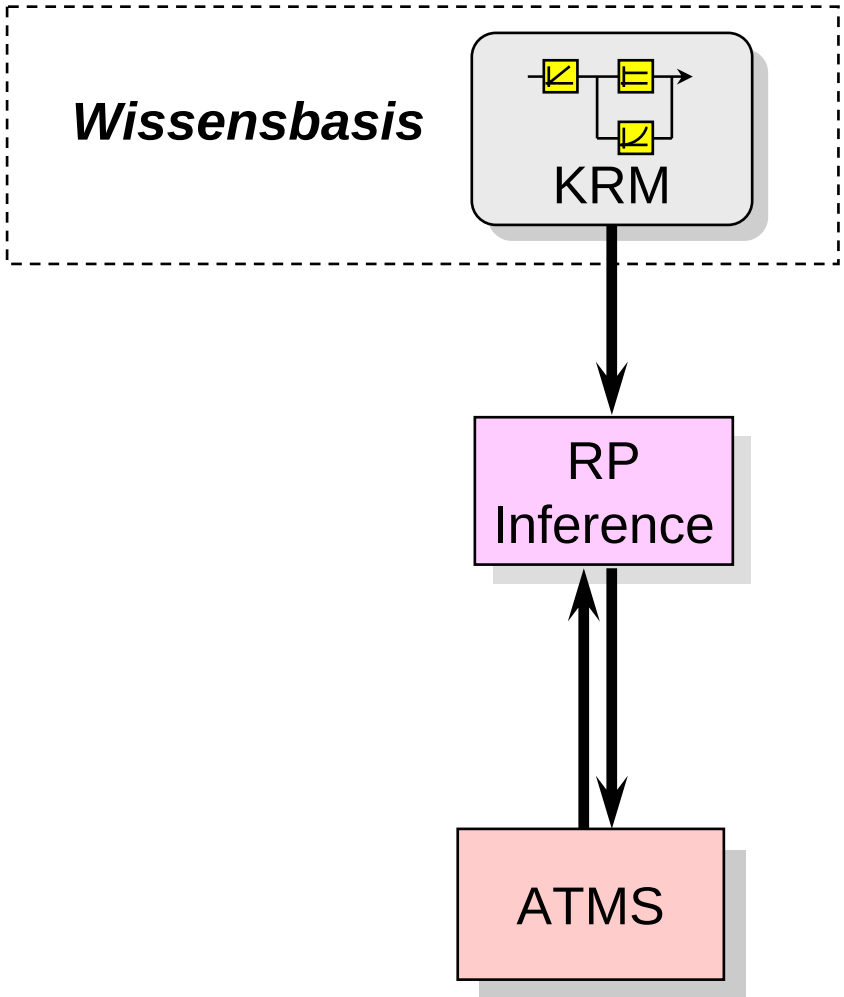
Report
conflicts

Zusammenspiel Kandidatengenerierer, RP und ATMS



ACS: **A**ssumption-based **C**onstraint **S**olver

Anforderung an die Wissensbasis



Was muss die Wissensbasis an die Inferenzkomponente liefern ?

- Regeln für die Wertzusammenhänge in den einzelnen Verhaltensmodi (*Komponentenmodellierung*)
- Kenntnis über die Wertdomänen: Wann gelten zwei Werte als widersprüchlich ?

MDS löst diese Anforderungen durch das Anbieten einer Constraint-Sprache für die Komponentenmodellierung

MDS Constraint Language

Value domains:

- finite / non-finite;
- numeric (e.g. integers, floating point, ...);
- numeric intervals (e.g. [-1, 1]);
- boolean;
- symbolic (e.g. #open, #close);
- special structures (e.g. for electric);

⇨ open to extensions (domain properties must be specified at Smalltalk level);

MDS Constraint Language

Constraint types:

- **MDS Constraint:**
  - set of propagation rules;
- **MDS Propagation Rule:**
  - has a procedural interpretation (computes values given other values);
  - defined (at low level) using Smalltalk;
- **Generic Rule Set:**
  - feature to build higher-level constructs;

MDS Constraint Language

Examples

1) Constraint: “a = b”

- MDS encoding using propagation rules:

  { R1: ==> a := [b]. “computes **a** from **b**”

  R2: ==> b := [a]. “computes **b** from **a**” };

- MDS encoding using the generic rule-set EQ:

  { R: ==> EQ (a , b). };

2) Constraint: “a = b + c”

- MDS encoding using propagation rules:

  { R1: ==> a := [b + c]. “computes **a** from **b** and **c**”

  R2: ==> b := [a - c]. “computes **b** from **a** and **c**”

  R3: ==> c := [a - b]. “computes **c** from **a** and **b**” };

- MDS encoding using the generic rule-set SUM:

  { R: ==> SUM (a, b, c). };

MDS Constraint Language

Examples

3) Constraint: (ass=OK ==> out = in1 + in2) &

(ass=broken ==> out = 1)

- MDS encoding using propagation rules :

```
{ R1:   [ ass == #ok ]      ==> out := [ in1 + in2 ].
  R2:   [ ass == #ok ]      ==> in1 := [ out - in2 ].
  R3:   [ ass == #ok ]      ==> in2 := [ out - in1 ].
  R4:   [ ass == #broken ]  ==> out := [ 1 ].          };
```

- MDS encoding using generic rule sets:

```
{ R1:   [ ass == #ok ]      ==> SUM ( out, in1, in2 ).
  R2:   [ ass == #broken ]  ==> EQ ( out, [ 1 ] ). };
```

MDS Constraint Language

Syntax (excerpt)

```
<constraintDef>      ::= <constraintName>':'  '{' <ruleList> '}' ';'
<ruleList>           ::= <propagRule> [ <propagRule> ]*
<propagRule>         ::= <ruleName>':' [ <ifPredicateList> ] '==>' <conseqList> '.'
<ifPredicateList>    ::= <ifPredicate> [ ',' <ifPredicate> ]*
<ifPredicate>        ::= '[' <SmalltalkCode> ']'
<conseqList>         ::= <conseq> [ ';' <conseq> ]*
<conseq>             ::= <assignment> |  '#contradiction' | <ruleSetCall>
<assignment>         ::= <variableName> ':=' <valueBlock>
<valueBlock>         ::= '[' <SmalltalkCode> ']'
<ruleSetCall>        ::= <ruleSetName> '(' <callParameterList> ')'
<callParameterList> ::= { <variableName> | '['<constantExpr>']'  }  [ ',' <callParameterList> ]*
```

MDS Constraint Language

Semantics (excerpt)

Rule R: <ifPredicateList> ==> <concl1>; <concl2>; ...

- equivalent with several rules each having one conclusion and the same <ifPredicateList>

Basic rule:

- has only one conclusion, namely a variable assignment or the symbol #contradiction.

A basic rule can *fire* if:

- all the constraint variables mentioned in the <ifPredicateList> and in the conclusion, except for the assignment variable, have associated values;
- the input values satisfy all the test predicates from the <ifPredicateList>;

Beim nächsten Mal:

Zusätzliche Funktionalitäten von MDS Komponentenmodellierung