Verwaltung von Abhängigkeiten in kooperativen wissensbasierten Arbeitsabläufen[1]

Barbara Dellen, Frank Maurer & Jürgen Paulokat
Universität Kaiserslautern, AG Expertensysteme Prof. Richter
Postfach 3049, 67653 Kaiserslautern
e-Mail: {dellen, maurer, paulokat}@informatik.uni-kl.de

Abstract.

In diesem Papier beschreiben wir eine Methode zur Spezifikation und Operationalisierung von konzeptuellen Modellen kooperativer wissensbasierter Arbeitsabläufe. Diese erweitert bekannte Ansätze um den Begriff des Agenten und um alternative Aufgabenzerlegungen. Das Papier beschreibt schwerpunktmäßig Techniken, die unserem verteilten Interpreter zugrunde liegen. Dabei gehen wir insbesondere auf Methoden ein, die Abhängigkeiten zwischen Aufgaben behandeln und ein zielgerichtetes Backtracking effizient unterstützen.[2]

1 Einleitung und Überblick

Modellbasiertes Knowledge Engineering ist der aktuelle Stand bei der Entwicklung wissensbasierter Systeme. Der bekannteste Ansatz ist dabei KADS [18]. Dieser erlaubt, *konzeptuelle Modelle* von Expertensystemen auf einem sehr hohen Abstraktionsniveau zu beschreiben. Neben den ursprünglich vorhandenen informellen Beschreibungsmitteln wurden verschiedene formale und/oder operationale Sprachen für konzeptuelle Modelle entwickelt [6]. Keine dieser Sprachen erlaubt es, eine Teilaufgabe einem *Bearbeiter* oder *Agenten* zuzuordnen. Dies bedeutet, daß Arbeitsabläufe, an denen mehrere Agenten bzw. Systeme bei der Problemlösung kooperieren müssen, nicht „natürlich“ beschreibbar sind. Unsere Anwendungsgebiete (Erstellung von Bebauungsplänen und Software Engineering[3]) erfordern allerdings solche kooperativen Arbeitsabläufe. Von daher sind die bekannten Ansätze nicht ausreichend.

Unser Ziel ist nun, einen methodischen Rahmen und die dazugehörigen Werkzeuge zu entwickeln, die die Beschreibung und prototypische Implementierung von kooperativen wissensbasierten Anwendungen ermöglicht. Dabei ergeben sich in unserem Anwendungsgebiets folgende Anforderungen:

1 Die hier beschriebene Arbeit wurde teilweise von der VW-Stiftung unter Vertrag I/69 374 gefördert im Rahmen des interdisziplinären Projektes „Intelligenter Bebauungsplan“. Die Projektpartner sind die Arbeitsgruppen Prof. Richter (Informatik), Prof. Stich (ARUBI) und Prof. Streich (ARUBI) an der Universität Kaiserslautern.

2 Dieses Papier ist eine Überarbeitung und Erweiterung von [8], wobei insbesondere im dritten Abschnitt wesentliche Änderungen vorgenommen wurden.

3 Die Beschreibung der Anwendungsgebiete ist nicht Gegenstand dieser Arbeit (vgl. dazu [9]).

- Modellierung von kooperativen wissensbasierten Systemen
  Das konzeptuelle Modell muß die Kooperation der verschiedenen Agenten beschreiben. Deshalb muß der methodische Rahmen den Begriff des Agenten enthalten.
- Integration von menschlichen Bearbeitern in den Problemlöseprozeß
  Die vollständige Formalisierung jedes Problemlöseschritts ist in unserer Domäne nicht möglich: Viele (Teil-)Aufgaben werden auch in Zukunft von menschlichen Bearbeitern gelöst werden, wobei der Zugriff auf benötigtes Wissen (z.B. Gesetze, Verordnungen, Richtlinien, Normen etc.) vereinfacht werden soll. Die Ergebnisse dieser Problemlöseschritte werden anschließend dem wissensbasierten System übergeben und dann im weiteren Verlauf des Problemlöseprozesses benutzt.
- Selektion der Aufgabenzerlegung *während* der Problemlösung
  Unser Ansatz zielt auf die Unterstützung von Design-Aufgaben, wobei die Bebauungsplanung als Beispiel dient. Design-Probleme können in der Regel auf verschiedene Weisen gelöst werden. Die geeignete kann nur aufgrund des aktuell vorliegenden Problems bestimmt werden. Deshalb läßt unser konzeptuelles Modell alternative Aufgabenzerlegungen zu, von denen zur Laufzeit des Systems eine gewählt wird.[4]
- Zielgerichtetes Backtracking
  Die Selektion einer Aufgabenzerlegung stellt eine Entscheidung dar. Oft basiert sie auf zusätzlichen Annahmen, die sich im Nachhinein als falsch herausstellen können und dann zu einer Inkonsistenz im Problemlöseprozeß führen. Als Folge davon muß eine der Entscheidungen, die zu der Inkonsistenz geführt haben, zurückgenommen werden. Ein zielgerichtetes, daher effizientes Backtracking wird von den bisher implementierten Interpretern für konzeptuelle Modelle nicht unterstützt.

Im zweiten Abschnitt geben wir einen kurzen Überblick über die von uns zur Modellierung von kooperativen wissensbasierten Systemen benutzte Terminologie. Die Architektur unseres Interpreters und die Abhängigkeitsstrukturen für zielgerichtetes Backtracking bilden den Kern des Papiers und werden im dritten Abschnitt beschrieben. Abschnitt 4 umfaßt ein Beispiel. Der letzte Abschnitt faßt unsere Ergebnisse zusammen und diskutiert unseren Ansatz.

2 Modellierung von kooperativen wissensbasierten Systemen

Um kooperative wissensbasierte Systeme zu beschreiben benutzt unsere Methodik vier Grundbegriffe:[5] Aufgabe (task), Methode (method), Konzept (concept) and Agent (agent). Im Folgenden werden wir sie grob definieren. Eine detaillierte Beschreibung ist in [7] zu finden. In diesem Beitrag konzentrieren wir uns auf die Operationalisierung des Modells.

Aufgaben und Methoden: Eine Aufgabe wird durch ihr Ziel, ihre Eingaben und ihre Ausgaben beschrieben. Um eine Aufgabe zu bearbeiten wird eine Methode angewandt. Für

4 Die Selektion von Aufgabenzerlegungen zur Laufzeit kann als Beispiel für die Verwendung der (nicht ausgearbeiteten) Strategieebene der KADS-Methodik angesehen werden.

5 Die Begriffe werden im wesentlichen analog zu KADS und dem Components-of-Expertise-Ansatz [14] benutzt. Deshalb gehen wir nicht auf die Details unserer Sprache für die konzeptuelle Modellierung ein.

jede Aufgabe kann es eine endliche Menge alternativer Methoden geben. Methoden werden von Agenten ausgeführt.

Wir unterscheiden zwischen atomaren[6] und komplexen (zusammengesetzten) Methoden. Atomare Methode weisen Variablen aktuelle Werte zu. Eine komplexe Methode wird als Datenflußgraph beschrieben. Ein Datenflußgraph besteht aus Knoten, die (Unter-)Aufgaben und Variable definieren, und aus Kanten, die den Aufgaben ihre Ein- bzw. Ausgaben zuordnen. Jeder Variable ist ihr Typ zugeordnet, d.h. die Konzeptklasse (s.u.), die zur Laufzeit instanziiert und deren Instanz dann an die Variable gebunden wird. Jede Unteraufgabe kann mit Hilfe von Methoden weiter zerlegt werden. In diesem Sinn kann die Zerlegung der Gesamtaufgabe als UND-ODER-Baum verstanden werden. Bei diesem entsprechen Aufgaben den UND-Knoten, wohingegen Methoden ODER-Knoten sind. Die für die Lösung einer Aufgabe sinnvolle Methode wird erst zur Laufzeit bestimmt.

Konzepte: Datenstrukturen werden mit einem objekt-zentrierten Ansatz modelliert, wobei wir zwischen Klassen und Instanzen unterscheiden. Klassen definieren die Struktur der Instanzen durch eine Menge von Attributen (Slots). Jedem Attribut wird ein Typ und eine Kardinalität zugeordnet. Typen sind andere Konzeptklassen oder Basistypen wie SYMBOL, STRING, REAL, etc.[7] Konzeptklassen werden benutzt, um die Variablen der Datenflußgraphen zu typisieren.

Agenten: Aufgaben werden von Agenten bearbeitet. Im konzeptuellen Modell wird für jede Aufgabe definiert, welche Agenten fähig sind, sie zu bearbeiten. Wir unterscheiden dabei zwischen zwei verschiedenen Arten von Agenten: Menschen und Rechner. In Abhängigkeit von den zugeordneten Agenten werden Methoden in einer natürlichen oder formalen Sprache beschrieben.

Mit dem definierten Rahmen kann eine abstrakte Spezifikation eines kooperativen wissensbasierten Systems - ein konzeptuelles Modell - beschrieben werden. Die nächste Frage, die sich nun stellt, ist, wie diese (prototypisch) operationalisiert wird, d.h. wie implementiert man einen Interpreter für diese Spezifikation.

3 Operationalisierung von konzeptuellen Modellen

Im ersten Abschnitt beschreiben wir die Architektur unseres Interpreters für konzeptuelle Modelle. In Abschnitt 3.2 erläutern wir, wie der Interpreter die Aufgaben-Agenda verwaltet. Der letzte Abschnitt umfaßt die Beschreibung der Mechanismen zur Abhängigkeitsverwaltung, die das zielgerichtete Backtracking unterstützen.

3.1 Architektur des Interpreters

Der Interpreter hat eine Client-Server-Architektur. Den Server bezeichnen wir als Scheduler. Dieser speichert anstehende Aufgaben und die aktuell gültigen Variablenbindungen. Zur Verwaltung von Abhängigkeiten zwischen Aufgaben benutzen wir ein erweitertes TMS (vgl. Abschnitt 3.3). Aufgaben werden von Clients bearbeitet: Ein Client greift auf eine Aufgabe zu, bestimmt die anzuwendende Methode, führt diese aus und transferiert die dabei entstehenden Ergebnisse zurück zum Scheduler. Die Schnittstelle zwischen Server und Client ist in [12] genau beschrieben.

Um Aufgaben zu bearbeiten werden Methoden angewandt. Komplexe Methoden zerlegen Aufgaben in Unteraufgaben und definieren Variablen. Atomare (primitive) Methoden belegen Variable mit Werten. Die Auswahl einer Methode ist eine Entscheidung im Problemlöseprozeß, die später zurückgezogen werden kann.

Abbildung 1 erläutert die Arbeitsweise des Interpreters: Aufgabe A eines konzeptuellen Modells wird gestartet. *User-1* wählt *method-1* um Aufgabe A zu zerlegen. Deshalb werden die Aufgaben *A-1-1* und *A-1-2* in die Agenda der anstehenden Aufgaben aufgenommen. Anschließend akzeptiert *User-2* die Aufgabe *A-1-1* und *Computer-1* bearbeitet Aufgabe *A-1-2*. Der resultierende Zustand ist in Abbildung 1 dargestellt.

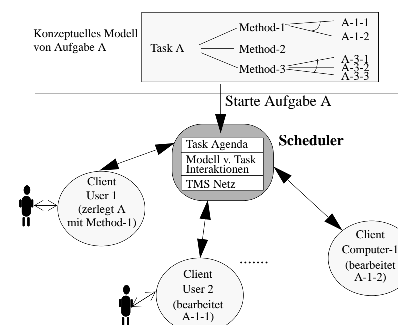

Abbildung 1: Architektur des Interpreters

3.2 Verwaltung von Aufgaben

In diesem Abschnitt beschreiben wir die Arbeitsweise des Schedulers mit Hilfe eines Zustandsübergangsdiagramms für Aufgaben (siehe Abbildung 2). Dieses zeigt die möglichen Zustände einer *einzelnen* Aufgabe während des Problemlöseprozesses.

Eine Aufgabe wird dem Scheduler übergeben und initialisiert (1). Nach der Initialisierung muß bestimmt werden, welcher Agent eine Aufgabe bearbeiten soll. Diese Information wird dem Scheduler beim Delegieren (2) übermittelt.

Wenn alle Eingaben vorhanden sind, wechselt eine Aufgabe in den Zustand „waiting“ (3). Die Eingaben sind verfügbar, wenn alle im Datenflußdiagramm vorangehenden Aufgaben durchgeführt worden und ihre Ausgabevariablen belegt sind.

Um eine Aufgabe zu bearbeiten, meldet sich ein Client bei dem Scheduler an und prüft, ob von ihm zu bearbeitende Aufgaben vorhanden sind. Von diesen akzeptiert er eine (4), die dann in den Zustand „in progress“ wechselt.

Der Client wendet nun eine Methode an und bearbeitet dadurch die Aufgabe. Ist die Methodenanwendung erfolgreich, so wechselt die Aufgabe in den Zustand „method applied“ (5). Handelt es sich um eine komplexe Methode, müssen die sich ergebenden Teilaufgaben weiter delegiert werden. Sind alle Unteraufgaben an die zuständigen Agenten

6 Atomare Methoden sind unser Analogon zu Inferenzschritten (inference steps) in KADS. Im Gegensatz zu KADS gibt es bei uns keine vordefinierte feinste Granularität von Methoden: Im Verlauf des Knowledge Engineering Prozesses kann eine zuerst atomare Methode weiter zerlegt werden.

7 Ausgehend von den Klassenspezifikation wird automatisch eine maskenorientierte Benutzungsschnittstelle erzeugt, die mit Hilfe eines graphische Interface-Builders überarbeitet werden kann.

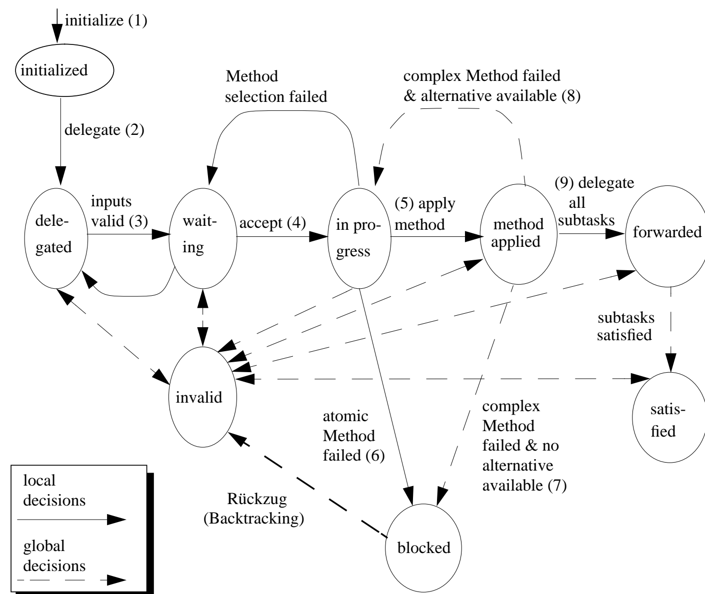

Abbildung 2: Zustandsdiagramm für Aufgaben (vgl. [12])

delegiert worden, wechselt die Aufgabe in den Zustand „forwarded“ (9). Bei atomaren Methoden entfällt die Delegierungsphase und die Aufgabe wandert über den Zustand „forwarded“ direkt in den Zustand „satisfied“.

Wenn alle Teilaufgaben bearbeitet worden sind und ihre Ergebnisse vorliegen, wechselt die übergeordnete Aufgabe in den Zustand „satisfied“.

Wenn die Anwendung einer Methode fehlschlägt und eine Alternative vorhanden ist, dann wechselt die Aufgabe wieder in den Zustand „in progress“ (8). Ansonsten geht die Aufgabe in den Zustand „blocked“ über (6) (7).[8]

Die mit durchgezogenen Linien gekennzeichneten Übergänge können durch lokale Entscheidungen ausgelöst werden. Im Gegensatz dazu erfolgen die Übergänge mit gestrichelten Linien (in Abbildung 2) aufgrund globaler Entscheidungen. Wenn z.B. der Scheduler aufgrund eines Backtracking-Schrittes entscheidet, daß eine Aufgabe mit Hilfe einer anderen Methode zerlegt wird, dann kann die Arbeit an den Teilaufgaben der bisher gültigen Methode eingestellt werden, da deren Ergebnisse nicht mehr benötigt werden.

Das Notwendigkeit von Backtracking-Schritten zeigt der „blocked“-Zustand: Eine Aufgabe ist blockiert, wenn keine Methode mehr anwendbar ist, um sie zu bearbeiten. Auf der andern Seite ist die Bearbeitung der Aufgabe notwendig, um das Gesamtproblem zu lösen. Das bedeutet, daß der Problemlöseprozeß in eine Sackgasse geraten ist. Diese kann nur verlassen werden, wenn der Zustand des gesamten Problemlöseprozesses in Betracht gezogen wird. Eine der gültigen Entscheidungen muß zurückgezogen und durch eine andere Alternative ersetzt werden. Alle Entscheidungen und Variablenbelegungen, die auf der nun ungültigen, zurückgezogenen Methode basieren, sind ebenfalls als ungültig zu markieren. Diese Berechnungen, die den Problemlöseprozeß wieder in einen konsistenten Zustand versetzen, werden von einem Mechanismus zur Verwaltung von Abhängigkeiten unterstützt. Das dafür notwendige Abhängigkeitsnetzwerk wird beim Durchlaufen der Zustandübergänge aufgebaut und in den nächsten Abschnitten näher beschrieben.

3.3 Abhängigkeitverwaltung durch den Scheduler

Die im Laufe des Lösungsprozesses anfallenden Aufgaben werden durch die Anwendung von Methoden gelöst. Ein Agent muß sich während des Lösungsprozesses für eine der alternativen Methoden entscheiden. Die durch die ausgewählten Methoden hergeleiteten Teilaufgaben und Konzeptinstanzen stellen die aktuell gültige Lösung, die gültigen Entscheidungen den Lösungsweg dar. Der Scheduler verwaltet den aktuellen Stand des Problemlöseprozesses und die entstehenden Abhängigkeiten. Zu seinen Aufgaben gehört die Verwaltung

- der Zustände, die eine Aufgabe annehmen kann,
- der Beziehungen, die sich aus der Zerlegung von Aufgaben in Teilaufgaben ergeben,
- der durch die Anwendung von komplexen oder atomaren Methoden entstehenden Entscheidungen,
- der Abhängigkeiten, die zwischen der Entscheidung für eine atomare Methode und der damit verbundenen Variablenbelegung bestehen[9] (insbesondere müssen der Datenfluß und die damit verbundenen zeitlichen Abhängigkeiten zwischen den Aufgaben gesteuert werden),
- der Delegierung von Aufgaben an ihre Bearbeiter.

Der Scheduler unterstützt den Rückzug von Entscheidungen, die zu einer inkonsistenten Lösung führen. Der Rückzug von Entscheidungen hat in der Regel globale Auswirkungen auf den Lösungsprozeß:

- Die Anwendung einer komplexen Methode auf eine Aufgabe führt zu der Zerlegung in neue Teilaufgaben. Der Rückzug einer solchen Entscheidung führt dazu, daß die aus dieser Zerlegung resultierenden Aufgaben und Lösungen ihre Gültigkeit verlieren. Dieser Mechanismus pflanzt sich in dem gesamten Lösungsbaum fort.
- Da durch atomare Methoden Variablen belegt werden und im weiteren Lösungsverlauf Entscheidungen auf diesen Belegungen aufbauen, kann der Rückzug von atomaren Methoden weitreichende Auswirkungen haben. Nach dem Rückzug muß jede auf der Gültigkeit einer nun ungültigen Belegung beruhende Lösung zurückgenommen und überdacht werden.

---

8 Die vorliegende Implementierung geht davon aus, daß es keine alternative Methoden zur Bearbeitung von atomaren Aufgaben gibt.

9 Eine Variable wird ausschließlich durch atomare Methoden mit Werten belegt. Jede (mögliche) Variablenbelegung ist mit einer Entscheidung verknüpft.

- Sind alle einer Aufgabe zugeordneten Methoden nicht (mehr) anwendbar, kann die Aufgabe nicht mehr bearbeitet werden. Die resultierende Blockade kann durch chronologisches Backtracking behoben werden. Effizienter und sinnvoller ist es jedoch, ursachengesteuertes Backtracking durchzuführen.

Für Entscheidungen, die von der Gültigkeit anderer Entscheidungen abhängen, werden daher Rückzugsbedingungen formuliert. Damit kann das System die Ursachen einer Inkonsistenz ermitteln und zur Auflösung von Blockaden durch abhängigkeitsgerichtetes Backtracking beitragen.

Im folgenden werden die Strukturen beschrieben, die die angesprochenen Abhängigkeiten abbilden, den Rückzug von Entscheidungen unterstützen und abhängigkeitsgerichtetes Backtracking ermöglichen [5]. Als Basismodell dient uns das allgemeine Planungs- und Designmodell REDUX [10] [11], welches die Dekomposition von Aufgaben, den Entscheidungsrückzug und abhängigkeitsgerichtetes Backtracking ermöglicht.

Abhängigkeiten entsprechen logischen Implikationen. Um die Menge aller Abhängigkeiten zu verwalten setzen wir ein TMS [3] ein. Die Änderungen der logischen Werte von Formeln werden durch den Markierungsalgorithmus des TMS effizient propagiert. Unser System bildet alle Aufgaben, Entscheidungen und deren Abhängigkeiten auf TMS-Strukturen ab.

3.3.1 Abhängigkeiten zwischen Methoden und Entscheidungen

Jede Anwendung einer Methode stellt eine Entscheidung dar. Ist eine Methode zurückgewiesen, ist die damit verbundene Entscheidung für die Methode ungültig. Eine Entscheidung wird durch das Prädikat *decision* dargestellt. Dieses Prädikat ist gültig, wenn die Methode den gültigen Lösungsweg für die Aufgabe darstellt. Das Prädikat *rejected-decision* drückt den Rückzug einer Entscheidung aus. Die Ursachen für einen Rückzug werden in Form von Bedingungen formuliert, die die Gültigkeit des Prädikats implizieren. Die Prädikate *decision* und *rejected-decision* stellen keine Negation voneinander dar. Eine Entscheidung kann auch dann noch zurückgezogen sein, wenn die Bedingungen für ihren Rückzug nicht mehr vorliegen. Auf diese Weise verhindert man, daß Lösungswege, auf die man nach dem Rückzug einer Entscheidung ausgewichen ist, verworfen werden, sobald die Rückzugsbedingungen des alten Lösungswegs ihre Gültigkeit wieder verlieren [10] [11].[10]

3.3.2 Modellierung der Zustandsübergänge

Die in 3.2 definierten Zustände werden durch die Prädikate *delegated, waiting, in-progress, method-applied* und *forwarded* dargestellt. Um die Zustandsübergänge modellieren zu können, werden darüber hinaus die Zustände *executable*, *unreduced* und *reduced* definiert. Eine Aufgabe befindet sich im Zustand *executable*, wenn

- alle der Aufgabe zugeordneten Eingabevariablen instanziiert sind und
- die Aufgabe an einen oder mehrere Agenten delegiert worden ist.

Der Zustand *unreduced* stammt aus der REDUX-Begriffswelt und beschreibt den Zustand einer Aufgabe vor einer Methodenanwendung. Existiert zu einer Aufgabe eine gültige Entscheidung, dann befindet sie sich im Zustand *reduced*. Die beiden Zustände stellen jedoch nicht die Negation voneinander dar: über das Prädikat *unreduced* werden zusätzlich die Mechanismen für die Erkennung und Behandlung von Blockaden und die Gültigkeit der Aufgabe gesteuert. Nur eine gültige Aufgabe darf sich im Zustand *unreduced* befinden.

Mit der Delegierung (ausgedrückt durch das Prädikat *valid-delegation)* wandert die Aufgabe *T* in den Zustand *delegated*. Um sicherzustellen, daß die Aufgabe beim Wechsel in einen anderen Zustand den Zustand *delegated* verläßt, wird dessen Gültigkeit von den Zuständen *waiting*, *unreduced* und *in-progress* abhängig gemacht.

$$\textit{valid-delegation(T)} \wedge \textit{unreduced(T)} \wedge \neg \textit{waiting(T)} \wedge \neg \textit{in-progress(T)} \Rightarrow \textit{delegated(T)} \qquad (GL\ 1)$$

Wenn

- alle der Aufgabe zugeordneten Eingabevariablen instanziiert sind und die Aufgabe an einen oder mehrere Agenten delegiert worden ist (Prädikat *executable* gültig) und
- die Aufgabe gültig und noch nicht bearbeitet (*unreduced*) ist,

befindet sich die Aufgabe im Zustand *waiting* und ist damit zur Bearbeitung freigegeben.

$$\textit{unreduced(T)} \wedge \textit{executable(T)} \wedge \neg \textit{accepted(T)} \Rightarrow \textit{waiting(T)} \qquad (GL\ 2)$$

Das Prädikat *accepted* ist ungültig, solange noch kein Agent die Aufgabe bearbeiten will.

Wenn ein Agent die Aufgabe *T* akzeptiert, wird das Prädikat *in-progress* gültig.

$$\textit{unreduced(T)} \wedge \textit{executable(T)} \wedge \textit{accepted(T)} \Rightarrow \textit{in-progress(T)} \qquad (GL\ 3)$$

Die Anwendung einer Methode *m* bewirkt, daß die dazugehörige Entscheidung *decision(m)* gültig wird. Wird eine Methode angewandt, dann wechselt die bearbeitete Aufgabe in den Zustand *reduced*. Die folgende Formel beschreibt diesen Zusammenhang an einer Aufgabe *T*, der die Methoden $m_1, \ldots, m_j$ zugeordnet sind.[11]

$$\exists! i(\textit{decision}(m_i)) \Rightarrow \textit{reduced(T)} \quad i \in \{1..j\} \qquad (GL\ 4)$$

10 Petrie unterscheidet zwischen dem Rückzug und der Notwendigkeit eines Rückzugs einer Entscheidung und behebt dadurch folgendes Problem von herkömmlichen TMS: Für die Fahrt von Kaiserslautern nach Hamburg wird als optimale Entscheidung das Auto ausgewählt. Dann stellt das System fest, daß kein Benzin mehr vorhanden ist, macht die erste Entscheidung daraufhin notwendigerweise ungültig, zieht die Entscheidung für das Auto zurück und wählt den Zug als Alternative. Anschließend wird eine Fahrkarte gekauft. Stellt das System im weiteren Verlauf des Problemlöseprozesses fest, daß wieder Benzin vorhanden ist, so würde ein Standard-TMS automatisch wieder die Entscheidung für das Auto gültig machen, obwohl bereits eine Fahrkarte für den Zug vorhanden ist und aufgrund der Entscheidung für den Zug bereits viele weitere Entscheidungen getroffen worden sein können.

11 Das System stellt durch übergeordnete Mechanismen sicher, daß keine zwei Methoden gleichzeitig zur Bearbeitung einer Aufgabe angewendet werden. Das bedeutet, daß es nur genau eine ($\exists!$i) gültige Entscheidung für eine der alternativen Methoden geben kann. Die in (GL 4) angegebene Formel ließe sich in der Aussagenlogik auch als disjunktive Verknüpfung der einzelnen Entscheidungen *decision*(m_1) bis *decision*(m_j) darstellen. Der besseren Lesbarkeit wegen haben wir die Darstellung mit dem Quantor gewählt.

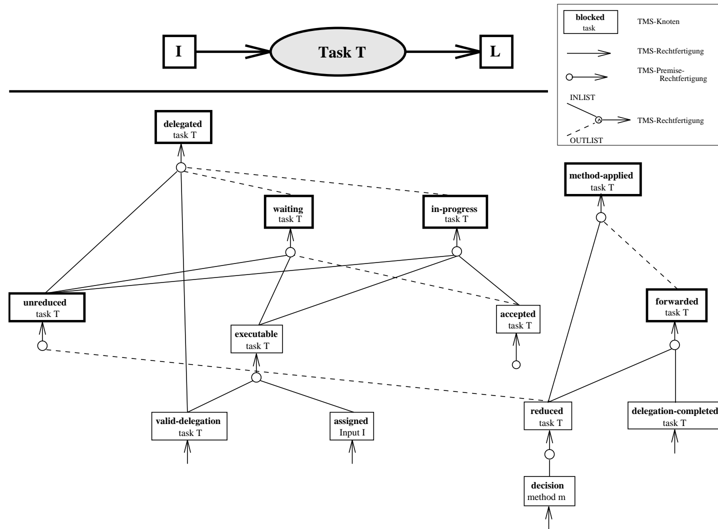

Abbildung 3: Zustandsübergänge auf TMS-Ebene

Das Prädikat *method-applied* ist gültig, wenn eine komplexe Aufgabe reduziert (reduced) und ihre Teilaufgaben noch nicht vollstöndig an die zuständigen Agenten delegiert worden sind.

$$\textit{reduced(T)} \wedge \neg \textit{forwarded(T)} \Rightarrow \textit{method-applied(T)} \qquad \textit{(GL 5)}$$

Der Zustand *forwarded* wird erreicht, wenn eine gültige Entscheidung vorliegt und die sich aus dieser Entscheidung ergebenden Teilaufgaben vollständig delegiert sind (Prädikat *delegation-completed* ist gültig).

$$\textit{reduced(T)} \wedge \textit{delegation-completed(T)} \Rightarrow \textit{forwarded(T)} \qquad \textit{(GL 6)}$$

Hat die Entscheidung zu einer Variablenbelegung geführt, wird der Zustand *forwarded* unmittelbar erreicht. Die Zustände *satisfied* und *blocked* stellen Standardabhängigkeiten aus REDUX dar.

In Abbildung 3 ist das resultierende TMS-Netz für eine Aufgabe *T* zu sehen. Mit Hilfe dieses Netzwerks kann die Gültigkeit von Prädikaten effizient aktualisiert werden.

3.3.3 Datenflußabhängigkeiten

Komplexe Methoden definieren Datenflüsse zwischen ihren Teilaufgaben. Wird eine Methode angewendet, werden die Eingabe- und Ausgabevariablen der Aufgabe auf die der Teilaufgaben abgebildet. Abbildung 4 zeigt Aufgabe *T*, die bei Anliegen einer Eingabe *I* die Ausgabe *O* erzeugt. Die Aufgabe *T* wird mit Methode m_1 in die Aufgaben T_1 und T_2 zerlegt. *I* dient dann Aufgabe T_1 als Eingabe. Die Ausgabe von T_1 wird in der neu

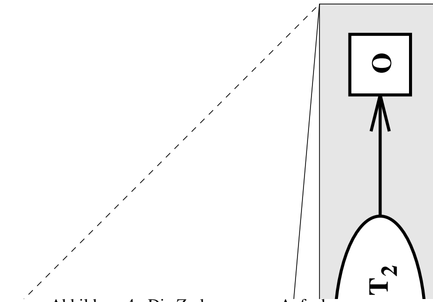

Abbildung 4: Die Zerlegung von Aufgaben

eingeführten lokalen Variable *L* abgelegt, die der Aufgabe T_2 als Eingabe dient. Das Ergebnis von T_2 wird in *O* gespeichert.

Atomare Methoden stellen formale oder informelle[12] Prozeduren für die Berechnung von Werten für Variable dar und erzeugen keine neuen Teilziele. Jeder mögliche Wert entspricht einer Variablenbelegung (*assignment(V=value)*). In Abbildung 5 ist dieser Zusammenhang für die Aufgabe T_1 aus Abbildung 4 dargestellt. Liegt eine Eingabe $I=i_1$ an *T* an, führt die Anwendung der atomaren Methode m_{11} zu der Generierung einer Konzeptinstanz l_1, die der Ausgabevariablen *L* zugewiesen wird (ausgedrückt durch das Prä-

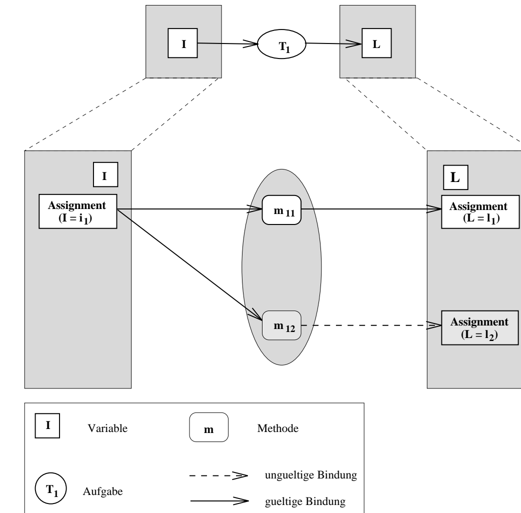

Abbildung 5: Die Belegung von Variablen

12 Informelle Prozeduren stellen textuelle Beschreibungen von Vorgehensweisen dar, die Menschen bei der Problemlösung anleiten.

dikat *assignment($L=l_1$))*. Für deren Berechnung werden die Daten, die durch die Instanz i_1 zur Verfügung gestellt werden, benutzt. Mit der Anwendung der Methode m_{12} wird eine andere Konzeptinstanz l_2 für L erzeugt. Da Methode m_{12} nicht ausgewählt ist, ist die Zuweisung von l_2 (*assignment($L=l_2$)*) nicht gültig (ausgedrückt durch die gestrichelten Linien), bleibt aber erhalten. Durch erneutes Anwenden der Methode m_{12}, wird auch die mit ihr verbundene Belegung wieder gültig, ohne daß die Prozedur, die die Daten erzeugt, erneut evaluiert werden muß. Dies ist insbesondere dann von Vorteil, wenn Menschen mit der Problemlösung betraut sind: Der Zwang, eine Aufgabe, die bereits gelöst wurde, erneut zu bearbeiten, kann zu Akzeptanzproblemen führen.

Die Datenflußabhängigkeiten werden nun beispielhaft für eine Aufgabe T formal beschrieben. Der Aufgabe T sind die atomaren Methoden m_1 und m_2 zugeordnet. Ihre Eingabevariablen sind I_1 bis I_u. Die Ausgabevariablen werden O_1 bis O_v genannt.

Um die Belegung von Ein-/Ausgabevariablen formulieren zu können, wird das Prädikat *assigned* definiert. Dieses ist gültig, wenn einer Variable V_i eine gültige Belegung v_{ij} zugeordnet ist.[13]

$$\exists! j\ (assignment(V_i=v_{ij})\ ) \Rightarrow assigned(V_i) \qquad (GL\ 7)$$

Die Beziehung zwischen der Ausführbarkeit einer Aufgabe und einer gültigen Belegung ihrer Inputvariablen wird durch das Prädikat *executable* beschrieben (vgl. Modellierung der Zustandsübergänge in Abschnitt 3.3.2).

$$assigned(I_1) \wedge assigned(I_2) \wedge \ldots \wedge assigned(I_u) \wedge valid\text{-}delegation(T) \Rightarrow executable(T) \qquad (GL\ 8)$$

Mit der Entscheidung für Methode m_j werden die Ausgabevariablen O_1 bis O_v belegt.

$$decision(m_j) \Rightarrow assignment(O_1=o_{1j}) \wedge assignment(O_2=o_{2j}) \wedge \ldots \wedge assignment(O_v=o_{vj}) \qquad (GL\ 9)$$

Entscheidungen werden im Kontext der aktuell gültigen Eingabeinformation getroffen. Wenn sich die Belegungen der Eingabevariablen ändern, müssen alle auf den bisher gültigen Werten beruhenden Entscheidungen zurückgezogen werden. In dem Beispiel in Abbildung 6 geht die lokale Variable L als Eingabe in die Aufgabe T_2 ein. Jede Entscheidung die in T_2 getroffen wird, beruht daher auf der Belegung von L und muß zurückgenommen werden, sobald die Belegung nicht mehr gültig ist. Ist beispielsweise Entscheidung m_3 von T_2 aufgrund der Belegung l_1 von L getroffen worden, impliziert die Ungültigkeit des Prädikats *assignment($L=l_1$)* den Rückzug von m_3.

$$\neg assignment(L=l_1) \Rightarrow rejected\text{-}decision(m_3) \qquad (GL\ 10)$$

Der Markierungsalgorithmus des TMS stellt die Konsistenz der Lösung nach einer Änderung einer Variablenbelegung wieder her und bewirkt somit das dynamische Verhalten des Schedulers. Wird z.B. in Abbildung 6 die Entscheidung *decision(m_{11})* ungültig, dann propagiert das TMS dies weiter und *assignment($L=l_1$)* verliert ebenfalls seine Gültigkeit. Dies hat zur Folge, daß

13 Das System stellt sicher, daß nicht zwei unterschiedliche Belegungen einer Variable gleichzeitig gültig sind. Die Anmerkungen aus Fußnote 11 bzgl. des Quantors gelten hier analog.

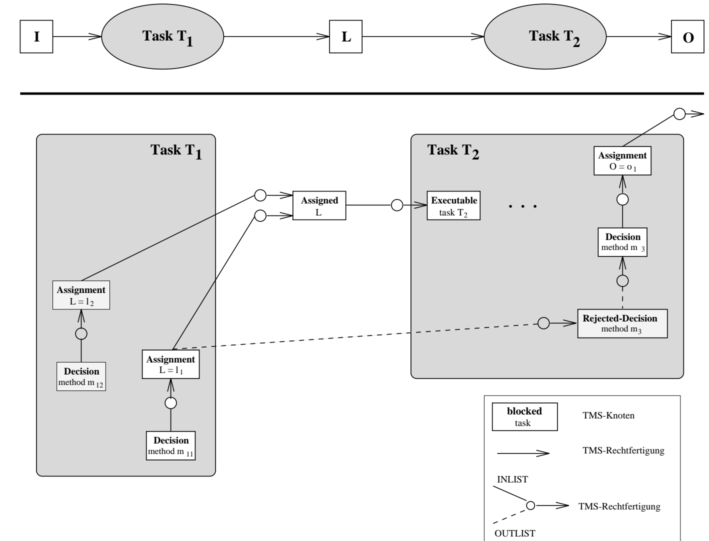

Abbildung 6: Datenflußabhängigkeiten auf TMS-Ebene

- der TMS-Knoten *assigned(L)* und alle davon abhängigen ungültig werden und
- der Knoten *rejected-decision(m_3)* gültig und infolgedessen der Knoten *decision(m_3)* ungültig wird. Anschließend verliert die von *decision(m_3)* abhängige Zuweisung *assignment($O=o_1$)* ihre Gültigkeit.

3.3.4 Die Delegierung von Aufgaben

Eine initialisierte Aufgabe wird durch die Delegierung ihren potentiellen Bearbeitern zugeordnet. Erst dann kann sich einer der Benutzer für die Aufgabe verantwortlich erklären und sie bearbeiten. Zu jeder Aufgabe ist daher das Prädikat *valid-delegation* definiert, das gültig wird, wenn die Aufgabe erfolgreich an einen Agenten delegiert worden ist. Dieses Prädikat erlaubt es, den Delegierungsvorgang von Aufgaben abzubilden (siehe auch Abschnitt 3.3.2).

Nach der Zerlegung einer Aufgabe, müssen die sich ergebenden Teilaufgaben delegiert werden. Erst dann geht die Aufgabe in den Zustand *forwarded* über. Um den erfolgreichen Abschluß der Delegierung aller Teilaufgaben feststellen zu können ist das Prädikat *delegation-completed* eingeführt worden, das gültig ist, wenn alle sich aus einer Methode m ergebenden Teilaufgaben T_1 bis T_n der Aufgabe T delegiert worden sind.

$$valid\text{-}delegation(T_1) \wedge valid\text{-}delegation(T_2) \wedge \ldots \wedge valid\text{-}delegation(T_n) \wedge decision(m) \Rightarrow delegation\text{-}completed(T) \qquad (GL\ 11)$$

Abbildung 7 zeigt das resultierende TMS-Netz.

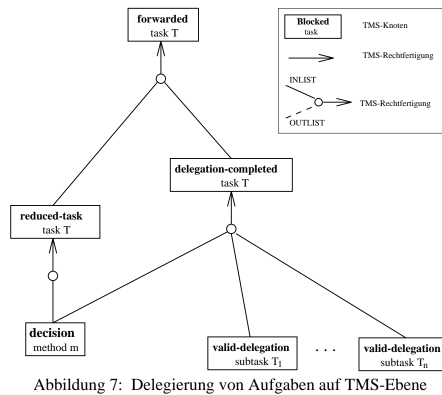

Abbildung 7: Delegierung von Aufgaben auf TMS-Ebene

4 Beispiel

Die beschriebenen Mechanismen werden im folgenden an einem Beispiel erläutert. Ausgangsbasis ist ein konzeptuelles Modell (Abbildung 8) zur Erstellung eines Entwurfsdokuments für ein Softwareprodukt *Terminkalender*. Die Ovale kennzeichnen die Aufgaben, in die der Entwurf zerfällt. Die Pfeile, die von den Ovalen wegführen, enden an den Methoden, die zur Lösung der Aufgaben zur Verfügung stehen. Die von den Methoden ausgehenden Pfeile zeigen auf die Teilaufgaben, in die die übergeordnete Aufgabe bei ihrer Anwendung zerlegt wird. Der Entwurfsprozeß gliedert sich in die beiden Aufgaben *Komponentenanforderungen entwickeln* und *Systemarchitektur entwickeln*. Für die Entwicklung der Systemarchitektur steht ein *funktionaler* bzw. *objektorientierter* Ansatz zur Auswahl. Jeder Ansatz führt zu einer eigenen Aufgabenstruktur. Die Komponentenanforderungen können nach *Mills*, *Hoare* oder *Guttag* spezifiziert werden. Für jede Alternative steht eine eigene komplexe Methode zur Verfügung. Die Aufgaben, in die die Aufgabe *Komponentenanforderungen entwickeln* und *Systemarchitektur entwikkeln* zerfällt, sind nicht weiter zerlegbar. Sie können nur durch die Anwendung atomarer Methoden gelöst werden. Die atomaren Methoden sind in dem Bild nicht dargestellt.

Damit das Projekt *Terminkalender* anlaufen kann, muß ein neuer Scheduler gestartet werden und die initiale Aufgabe sowie die Agenten, an die sie delegiert wird, bestimmt werden. Daraufhin können sich die für das Projekt verantwortlichen Agenten einlogen und die an sie delegierten und freigegebenen Aufgaben bearbeiten. Der erste Schritt im Lösungsprozeß ist die Bearbeitung der Aufgabe *Entwurf*. Der verantwortliche Agent akzeptiert die initiale Aufgabe und zerlegt sie mit Hilfe der einzigen zur Verfügung stehenden Methode *Architektur/Anforderungen* in die Teilaufgaben *Komponentenanforderungen entwickeln* und *Systemarchitektur entwickeln*. Der Clientprozeß teilt dem Scheduler die Reduzierung der Aufgabe durch die Methode *Architektur/Anforderungen* mit. Als nächstes werden die beiden Teilaufgaben weiter delegiert. Die Methode *Architektur/Anforderungen* definiert den in Abbildung 9 gezeigten Datenfluß zwischen den beiden Teilaufgaben *Komponentenanforderungen entwickeln* und S*ystemarchitektur entwickeln*. Die Aufgabe *Systemarchitektur entwickeln* kann mit Hilfe des als Eingabe zur Verfügung stehenden *Anforderungsdokumentes Terminkalender* bearbeitet werden. Das Ergebnis ist das *Systementwurfsdokument SED*, welches die Grundlage für die Bearbeitung der Aufgabe *Komponentenanforderungen entwickeln* darstellt.

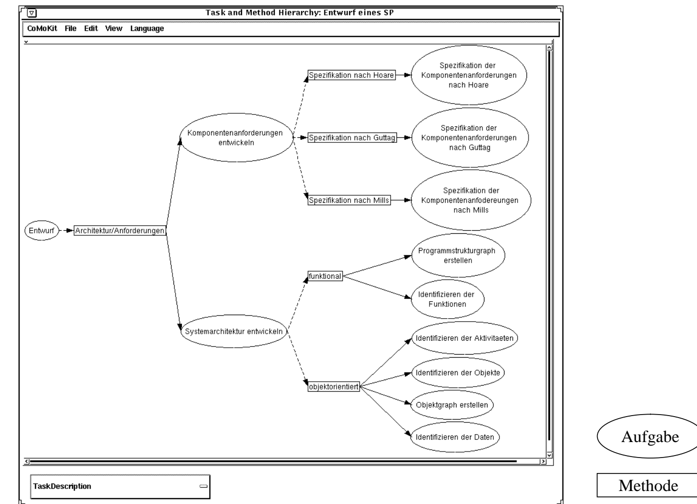

Abbildung 8: Aufgaben und Methoden

Die Aufgabe *Systemarchitektur entwickeln* kann entweder über einen *objektorientierten* (Methode *objektorientiert*) oder f*unktionalen* (Methode *funktional*) Ansatz gelöst werden. Beide Alternativen führen zu einer spezifischen Aufgabenzerlegung. Der verantwortliche Agent entscheidet sich in diesem Beispiel für den funktionalen Ansatz, zerlegt die Aufgabe damit in die Teile *Identifizieren der Funktionen* und *Programmstruktur-*

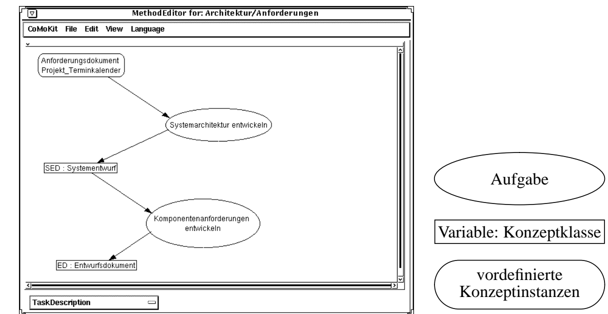

Abbildung 9: Datenfluß der Methode *Architektur/Anforderungen*

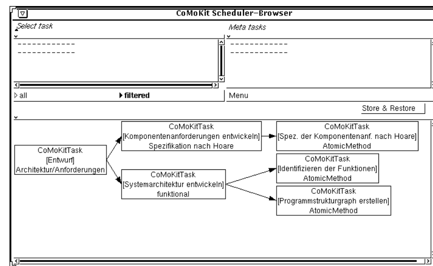

Abbildung 10: Zustand des Lösungsprozesses *vor* dem Rückzug der Entscheidung

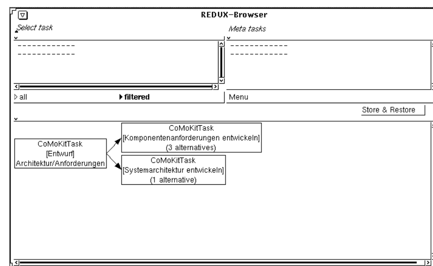

Abbildung 11: Zustand des Lösungsprozesses *nach* dem Rückzug der Entscheidung

graph erstellen. Anschließend delegiert er die Teilaufgaben. Diese werden nicht weiter zerlegt. Sie erzeugen Daten.

Die Benutzer bearbeiten die ihnen zugewiesenen Aufgaben in der vorgegebenen Reihenfolge solange, bis das Problem gelöst ist. Alle Aufgaben befinden sich damit im Zustand *satisfied*. In Abbildung 10 ist der eingeschlagene Lösungsweg zu sehen. Die Kästchen stellen die bearbeiteten und zum aktuellen Lösungsprozeß gehörenden Aufgaben dar. Anhand dieses Fensters lassen sich die Auswirkungen einer Umplanung auf den Lösungsprozeß nachvollziehen. Wird beispielsweise der funktionale Ansatz durch den Rückzug der Methode *funktional* verworfen, ergibt sich die in Abbildung 11 gezeigte Situation. Diejenigen Aufgaben und Entscheidungen, die von dem funktionalen Ansatz abhängig sind, sind zurückgezogen worden. Das gleiche gilt für die durch sie erzeugten Lösungen. Da die Entscheidung für die *Spezifikation nach Hoare* nach Vorliegen des *Systementwurfs* (vgl. Abbildung 9) gefallen war, wird sie ebenfalls zurückgezogen. Die Effekte des Rückzugs werden an die Entwickler weiterpropagiert, so daß der für die erfolgreiche Bearbeitung der Aufgabe *Systemarchitektur entwickeln* verantwortliche Agent erneut vor der Aufgabe steht, einen Lösungsansatz auszuwählen. Da der funktionale Ansatz nicht mehr zur Auswahl steht, muß er sich diesmal für den Ansatz *objektorientiert* entscheiden

5 Zusammenfassung und Diskussion

Ziel unserer Projekte zum Themenbereich „Kooperatives Arbeiten/Entwerfen“ ist die Entwicklung von Werkzeugen, die die Planung und Steuerung komplexer Arbeitsabläufe unterstützen und dadurch das Projektmanagement erleichtern.

In diesem Papier haben wir Methoden beschrieben, die die Modellierung kooperativer wissensbasierter Arbeitsabläufe erlauben. Die erstellten Modelle dienen als Eingabe eines Interpreters, der die bei der Abwicklung des Arbeitsablaufs entstehenden Daten verwaltet.

Desweiteren speichert der Interpreter Abhängigkeiten zwischen einzelnen Informationen. Dadurch ist es möglich, die von der Änderung einzelner Daten betroffenen Mitarbeiter (bzw. Systeme) zu informieren. Diese können auf die Änderungen reagieren und die von ihnen erzeugten Informationen entsprechend anpassen.[14]

Die entwickelten Techniken vereinfachen die Koordination der Arbeit mehrerer Agenten. Sie führen zu einem geringeren Abstimmungsaufwand und einer schnelleren Projektabwicklung.

Das hier beschriebene Modell erweitert REDUX um Abhängigkeiten zur Beschreibung des Datenflusses, der Zustandsübergänge und kooperative Aspekte. Unser Ansatz benutzt ein TMS zur effizienten Berechnung der logischen Zustände einer Menge von Formeln. Der Vorteil von REDUX und unseren Erweiterungen ist, daß wir dem TMS eine spezielle, für unsere Ziele sinnvolle Struktur aufgeprägt haben. Dadurch ist es möglich, „semantische“ Abhängigkeiten auf einen syntaktisch arbeitenden Formalismus (TMS) abzubilden.

Der vorgestellte Ansatz erweitert andere Knowledge Engineering Methoden um den (operationalen) Begriff des „Agenten“ und um multiple Aufgabenzerlegungen. Die hier beschriebenen Techniken können als Ergänzung zu den bekannten formalen/operationalen KADS-Sprachen gesehen werden, die der Spezifikation einzelner wissensbasierter Agenten dienen.

Unser Ansatz unterstützt einen „sanften“ Entwicklungsprozeß von kooperativen wissensbasierten Systemen, da am Anfang keine Aufgabe automatisiert werden muß. Trotzdem unterstützt unser System bereits den Problemlöseprozeß, indem es die Arbeit der einzelnen Bearbeiter koordiniert. Anschließend können die Entwickler entscheiden, welche Teilaufgabe automatisch bearbeitet werden soll (was zur Folge hat, daß der entsprechende Programmcode zu entwickeln ist).

Die Modellierung von verteilten wissensbasierten Systemen wird von dem vollständig implementierten Knowledge-Engineering-Werkzeug CoMo-Kit unterstützt. Der Server ist inklusive der beschriebenen Mechanismen zur Verwaltung von Abhängigkeiten implementiert. Die Clients kommunizieren mit dem Server über eine festgelegt Schnittstelle (ca. 30 Funktionen). Bereits realisiert ist die Client-Server-Kommunikation über UNIX-Sockets. Momentan arbeiten wir an der Verteilung der Aufgaben über ein lokales Netz mit Hilfe des objektorientierten Datenbanksystems GemStone.

14 Beispielsweise könnte in unserer Anwendung „Bebauungsplanung“ der für die Planung der Kanalisation zuständige Mitarbeiter auf das Verschieben einer Straße reagieren, das von einem Verkehrswegeplaner ausgelöst worden ist.

Durch die Anwendung von CoMo-Kit in den Bereichen „Bebauungsplanung“ und „Software Engineering“[15] haben sich zusätzliche Anforderungen an die Modellierungswerkzeuge und den Scheduler ergeben:

- **Methodendefinition noch während des Lösungsprozesses:** Komplexe Aufgaben, wie z.B. die Entwicklung eines großen Softwarepaketes, lassen sich vorab nicht vollständig beschreiben und in Teilaufgaben zerlegen, da sich neue Aufgaben oft erst nach Abschluß der ersten Projektphasen ergeben. Deshalb ist es notwendig, die Definition neuer Methoden noch während der Projektabwicklung zu ermöglichen.
- **Zerlegung von Objekten:** Analog zur Zerlegung von Aufgaben in Unteraufgaben, die mit Hilfe von Methoden beschrieben wird, muß es möglich sein, Objekte in ihre Bestandteile zu zerlegen. So ist es bei der Softwareentwicklung üblich, daß ein Programmpaket (= übergeordnetes Objekt) in mehrere Module (= Bestandteile) aufgeteilt wird, die von verschiedenen Programmierern implementiert werden.

Die Lösung dieser Probleme ist ein Ziel unserer aktuellen Arbeiten.

6 Literatur

[1] Angele, J.; Fensel, D.; Landes, D.; Studer, R: KARL: An Executable Language for the Conceptual Model. In: Proceedings of the Knowledge Acquisition for Knowledge-Based Systems, Workshop KAW’91, October 6-11, Banff, 1991.

[2] Breuker, J.; Wielinga, B.; van Someren, M.; de Hoog, R.; Schreiber, G.; de Greef, P.; Bredeweg, B.; Wielemaker, J.; and Billault, J.-P.: Model-Driven Knowledge Acquisition: Interpretation Models. Esprit Project P1098, University of Amsterdam (The Netherlands), 1987.

[3] Doyle, J.: A Truth Maintenance System, Artificial Intelligence, 12:231-272, 1979.

[4] de Greef, H. P., Breuker, J. A.: Analyzing System-User Cooperation in KADS, In [13].

[5] Dellen, B.: Verwaltung von Abhängigkeiten bei der Operationalisierung konzeptueller Modelle, Diplomarbeit Universität Kaiserslautern, 1995

[6] Fensel, D., van Harmelen, F.:A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, The Knowledge Engineering Review, vol 8, no 2, 1994.

[7] Maurer, F.: Hypermediabasiertes Knowledge Engineering für verteilte wissensbasierte Systeme, Dissertation Universität Kaiserslautern, 1993, auch: DISKI 48, infix-Verlag, ISBN 3-929037-48-3.

[8] Maurer, F., Paulokat, J.: Operationalizing Conceptual Models Based on a Model of Dependencies, in: A. Cohn (Ed.): ECAI 94. 11th European Conference on Artificial Intelligence, 1994, John Wiley & Sons, Ltd.

[9] Maurer, F., Pews, G.: Ein Knowledge Engineering Ansatz für kooperatives Design am Beispiel der Bebauungsplanung, Themenheft Knowledge Engineering KI 1/95, interdata Verlag, 1995

[10] Petrie, Ch.: Context Maintenance, in: Proceedings of AAAI-91, Menlo Park, California, 1991, MIT Press.

[11] Petrie, Ch.: Planning and Replanning with Reason Maintenance, Dissertation, University of Texas, Austin, 1991.

[12] Schmitz, W.: Multiple Aufgabenzerlegung von konzeptuellen Modellen, Diplomarbeit an der Universität Kaiserslautern, 1994

[13] Schreiber, G. (Ed.): Special Issue: The KADS Approach to Knowledge Engineering, Knowledge Acquisition, Vol. 4 No. 1, March 1992, Academic Press.

[14] Steels, L.: Components of Expertise, AI Magazine, 11 (2 (Summer)), 1990.

[15] van Harmelen, F., Balder, J.: $(ML)^2$: A Formal Language for KADS Models of Expertise, 1992, in [13].

[16] Wetter, Th.: First-order Logic Foundation of the KADS Conceptual Model, 1990, in [17].

[17] Wielenga, B., Boose, J., Gaines, B., Schreiber, G., van Someren, M. (ed.): Current Trends in Knowledge Acquisition, IOS Press, Amsterdam, May 1990.

[18] Wielinga, B.J.; Schreiber, A.Th.; Breuker, J.A.: KADS: A Modelling Approach to Knowledge Engineering, 1992, in [13].

15 Das Projekt „Wissensbasierte Planung und Steuerung von Softwareentwicklungsprozessen“ wird gefördert von der DFG im Rahmen des Sonderforschungsbereiches-1496 „Entwicklung großer Systeme mit generischen Methoden“.