Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-01

Simulation Environment for Multi-Agent Worlds

- Benutzeranleitung

Stefan Bussmann

Januar 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Krupp-Atlas, Mannesmann-Kienzle, Philips, Sema Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Simulation Environment for Multi-Agent Worlds
- Benutzeranleitung

Stefan Bussmann

DFKI-D-92-01

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-9104).

§ 0. Inhaltsverzeichnis

1. Konzepte der Simulationsumgebung 1
   1.1 Einführung 1
   1.2 Darstellung der Agenten 2
   1.3 Simulation des Szenarios 3
   1.4 Kommunikation zwischen Agenten und Szenario 4
2. Benutzen der Simulationsumgebung 5
   2.1 Configure-Scenario 5
   2.2 Load-Rule-Sets 6
   2.3 Monitor-Agents 7
   2.4 Run-Scenario 8
   2.5 System 9
3. Beispiel "Tower of Hanoi" 9
   3.1 Ziel der Simulation 9
   3.2 Definieren der Regelsätze 10
   3.3 Programmieren der Simulationsprozeduren 11
   3.4 Handhaben der Menüs 12
4. Generelle Programmierhinweise 13
5. Funktionen zur Simulation des Szenarios 15
6. Syntax und Semantik der Regelsätze 16
   6.1 Syntax 16
   6.2 Semantik 17
   6.3 Eingabe der Regelsätze 18

§ 1. Konzepte der Simulationsumgebung

1.1 Einführung

Die Simulationsumgebung, im folgenden mit SEMAW (*Simulation Environment for Multi-Agent Worlds*) abgekürzt, ist entwickelt worden, um verschiedene Multi-Agenten Szenarien zu testen. Der Begriff "Multi-Agenten Szenario" wird in diesem Rahmen nicht festgelegt (eine Einschränkung von SEMAW ist nicht beabsichtigt - diese wird sich aus der Anwendbarkeit der Umgebung ergeben), jedoch besteht eine intuitive Vorstellung des Begriffes, die sich wie folgt erläutern läßt: Unter Agenten werden Objekte einer Welt verstanden, deren Verhalten man untersuchen und programmieren will. Es kann sich also um Menschen handeln, um leblose Objekte, z.B. selbststeuernde Roboter oder Computer, oder sogar um Objekte, denen man indirekt ein Verhalten zuweist, wie das bei dem Beispielszenario *Tower of Hanoi* der Fall ist, auf das ich später zurückkommen werde. Das Präfix

"Multi" soll zum Ausdruck bringen, daß in der Regel mehrere Agenten des Szenarios untersucht werden sollen und folglich auch simuliert werden müssen. Dem Begriff "Szenario" unterliegt zwar die intuitive Vorstellung eines Ausschnitts einer realistischen Welt, so wie wir als Menschen sie erfahren, jedoch besteht bezüglich der Umgebung keine Einschränkung, was auch dadurch zum Ausdruck kommt, daß die Umgebung keine Hilfsmittel zur Definition und Verwaltung eines Szenarios zur Verfügung stellt, sondern das Szenario vollständig programmiert werden muß.

Bei der Entwicklung von SEMAW wurden zwei Ziele verfolgt: Erstens sollte man in der Lage sein, daß Verhalten der Agenten deklarativ zu beschreiben (was von SEMAW nur teilweise erfüllt wird, wie man aus der Definition der Regelsätze erkennen kann - siehe § 6). Zudem gehört auch die Anforderung, das Verhalten der Agenten in verschiedene, selbständige Bereiche aufzuteilen, und diese getrennt zu simulieren, sprich den Agenten zu modularisieren. Zu diesem Zweck wurde das Konzept der Komponenten eingeführt, welche getrennt programmiert und überwacht werden können. Zweitens sollte die Modellierung der Agenten streng von der des Szenarios getrennt werden. Das bedeutet einmal, daß eine klar definierte Schnittstelle zur Welt existiert, damit die Agenten keine unübersichtlichen, trickreichen Veränderungen des Szenarios vornehmen, und desweiteren, daß die Agenten sich nicht gegenseitig verändern, sondern nur über die Welt miteinander kommunizieren. Natürlich könnte man einwenden, daß solche programmiertechnischen Tricks (insbesondere aus Effizienzgründen) sinnvoll sein können, was auch hier nicht bestritten werden soll, doch ist die Umgebung entwickelt worden, um das Verhalten der Agenten zu untersuchen, und deshalb soll so etwas vermieden werden.

In dem Rest des ersten Paragraphen soll näher auf die einzelnen Konzepte von SEMAW eingegangen werden. In Paragraph 2 wird die Benutzeroberfläche erklärt und einige Hinweise auf deren Handhabung gegeben. Paragraph 3 erläutert anhand des Beispiels *Tower of Hanoi* die prinzipielle Vorgehensweise bei Definition und Testen eines Szenarios. In Paragraphen 4 bis 6 wird auf die technischen Details der Programmierung, insbesondere generelle programmiertechnische Hinweise (Paragraph 4), Programmieren der Prozeduren zur Simulierung des Szenarios (Paragraph 5) und Programmieren der Agenten (Regelsätze - Paragraph 6), eingegangen.

1.2 Darstellung der Agenten

Ein Agent besteht aus einer Menge von Komponenten, wobei jede Komponente aus einem Regelsatz besteht, der das Programm symbolisiert. Bei der Definition des Agenten werden die Komponenten durch Namensvergabe erzeugt. Alle Komponenten sind gleich aufgebaut, sie bestehen aus einer Kopie desselben Regelinterpreters, können aber mit verschiedenen Regelsätzen geladen werden, wobei eine Komponente nur einen Regelsatz haben kann. Die Komponenten sind also unabhängig von einander und können nur über Nachrichten miteinander kommunizieren, worauf in 1.4 näher eingegangen wird.

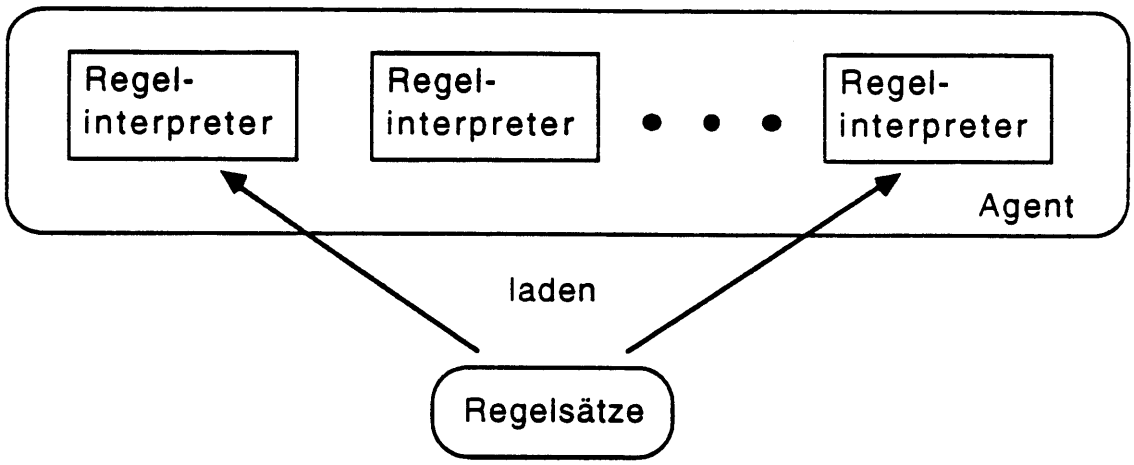


Bild 1: Agentenmodell

Regelsätze bestehen aus einem Namen, der zur Identifikation dient, einer Menge von Variablen, einer Menge von Regeln und einer Menge von Regelsätzen. Die Definition ist also rekursiv; ihre genaue Syntax und Semantik wird in Paragraph 6 genauer definiert. Variablen sind einfache (Lisp-) Symbole, die einen beliebigen Wert annehmen können. Lediglich ihre Zuweisung muß durch den Interpreter vorgenommen werden. Eine Regel besteht aus einer Bedingung und einer Folge von Befehlen. Die Bedingung und jeder Befehl sind Lisp-Ausdrücke, in denen die Variablen referenziert werden können. Dies ermöglicht es, die Berechnung von Werten auf das unterliegende Lisp-System, SEMAW ist in Common-Lisp implementiert, abzuwälzen und dadurch eine möglichst mächtige Ausdruckskraft der Regelsätze zu erreichen. Lediglich die Kontrolle des Programmflusses wird durch den Interpreter explizit durchgeführt.

Der Interpreter arbeitet in Zyklen. Am Anfang eines jeden Zyklus bestimmt er die Menge der Regeln, deren Bedingungsteile zu wahr (in Common-Lisp zum Symbol **t**) ausgewertet werden. Die zu den feuernden Regeln gehörenden Befehle werden in der Reihenfolge, wie sie aufgeschrieben wurden, zu einer Liste verkettet und ausgeführt. Dabei kann der Interpreter nach jeder Ausführung eines Befehls unterbrochen werden; insbesondere also inmitten der Abarbeitung einer Regel. Sind alle Befehle ausgeführt, beginnt der Zyklus von vorne. Feuern keine Regeln mehr, stoppt der Interpreter und die Komponente wird inaktiv.

Der Interpreter einer Komponente ist genau dann aktiv, wenn er Befehle ausführen soll. Er wird aktiv gesetzt, wenn die Komponente mit einem Regelsatz geladen wird, oder eine Nachricht erhält (siehe 1.4). Kann der Interpreter keine feuernde Regel mehr finden, also keinen Befehl ausführen, so wird er inaktiv.

Die Kontrolle, wann welche Komponente Rechenzeit erhält, kann in drei verschiedenen Modi durchgeführt werden: **sequential**, **agent-parallel** oder **component-parallel**.

Sequential bedeutet, daß das System einen Agenten sucht, der (mindestens) eine aktive Komponente (also eine Komponente, die rechnen will) hat, und den Interpreter dieser Komponente unbegrenzt arbeiten läßt, bis dieser von selbst stoppt. Danach wird eine aktive Komponente im selben Agenten gesucht, dann ein neuer Agent, der eine aktive Komponente hat, bis keine mehr vorhanden sind, und das System stoppt.

Bei den parallelen Modi kann eine Komponente nur solange Befehle ausführen, bis eine vom Benutzer festgelegte Grenze erreicht wird. Der Interpreter muß die Kontrolle abgeben, und sein Zähler wird zu null gesetzt. Der Vorteil paralleler Modi ist, daß man dadurch Parallelität simulieren kann, die gröber als die Ausführung eines Befehls ist. Eine Komponente im parallelen Modus muß damit rechnen, nach jedem Befehl unterbrochen zu werden. **Agent-parallel** bewirkt, daß ein Agent die Kontrolle behält, bis keine Komponente mehr aktiv ist. Die Komponenten des Agenten rechnen parallel, die Menge der Agenten aber sequentiell. Im Modus **component-parallel** rechnen alle Komponenten (sämtlicher Agenten) parallel.

Das Szenario kann gestartet werden, indem einmal die Interpreter direkt angestoßen werden. In diesem Fall sucht das System eine aktive Komponente (im entsprechenden Modus) bis keine mehr gefunden werden, und das System hält. Oder eine Benutzer-definierte Prozedur wird aufgerufen, die den Kontrollfluß übernimmt und die Interpreter selbst anstößt.

Während eines Szenario-Laufs können Agenten und ihre Kommunikationsbemühungen überwacht werden, indem *monitor-flags* gesetzt werden. Einmal kann bestimmt werden von welchen Komponenten (welcher Agenten) **send**-Befehle ausgegeben werden; also wann eine Komponente an wen eine Nachricht geschickt hat. Desweiteren kann der aktuelle Regelsatz, in dem sich der Interpreter gerade befindet, ausgegeben werden. Diese Information kann durch die Ausgabe der ausgeführten Befehle und deren Argumente erweitert werden.

1.3 Simulation des Szenarios

Der Ansatz, der für dieses System gewählt wurde, sollte keine Einschränkungen bezüglich des zu simulierenden Szenarios beinhalten. Aus diesem Grunde wird das Szenario durch beliebige Lisp-Funktionen simuliert, ohne jegliche explizite Information über die Diskurs-Welt zu repräsentieren. Das System erfordert eine eindeutige Schnittstelle der Funktionen, und nur über die darf zwischen Agenten und Szenario kommuniziert werden. Diese Maßnahme soll sicherstellen, daß die Interaktion der Agenten mit dem Szenario vollständig überwacht werden kann, und keine "heimlichen" Veränderungen sowohl des Szenarios, als auch der Agenten durchgeführt wird. Die Funktionen haben also nur die Schnittstelle (die später genau definiert wird) zur Verfügung zu stellen; inwieweit diese realisiert wird, ist ohne Belang (für das System).

Die Simulationsfunktionen müssen folgende Funktionalitäten bereitstellen:

1. Konfigurieren des Szenarios: Die notwendigen Festlegungen müssen getroffen werden, z.B. wieviele Agenten vorhanden sind - dies ist eine Größe, die oft variiert. Diese Funktion muß auch die Liste erstellen, anhand der das System erkennen kann, wieviele Agenten es erzeugen muß.
2. Rücksetzen des Szenarios: Da ein laufendes Szenario in seinen Startzustand zurücksetzbar sein muß.

3. Ausgeben von Bildschirm-Information: Dies dient zur Darstellung des Szenario-Zustandes.

4. Kommunikationsschnittstelle: Eine Funktion muß die Nachrichten des Agenten an die Welt

verarbeiten, so z.B. Nachrichten weiterleiten, die an andere Agenten gerichtet sind. Dabei kann es nur Nachrichten an den Agenten an sich schicken, nicht aber an einzelne Komponente, weil sie diese nicht "sieht" und nicht ansprechen kann. Das bedeutet insbesondere, daß auch Agenten andere Agenten höchstens als ganze Agenten sehen, sofern sie diese Agenten im Szenario überhaupt als solche erkennen.

1.4 Kommunikation zwischen Agenten und Szenario

Da die einzelnen Komponenten eines Agenten nicht die Möglichkeit haben, auf gemeinsame Datenstrukturen zuzugreifen, benötigen sie einen Mechanismus, durch den sie miteinander kommunizieren können. Dies wird durch das Nachrichten-Konzept bewerkstelligt. Eine Komponente ist in der Lage jeder anderen Komponente desselben Agenten, sofern sie den Namen dieser Komponente weiß, eine Nachricht zu schicken. Vom System aus gesehen handelt es sich lediglich um eine Liste von Lisp-Werten, die der adressierten Komponente in die (vordefinierte) Variable **message** als Listenelement eingefügt wird. Technisch läuft es so ab, daß der Nachrichteninhalt in die Variable der Komponente geschrieben wird, der Interpreter dieser aktiv wird (falls er dies noch nicht ist), und die sendende Komponente weiterrechnet. Die adressierte Komponente verarbeitet die Nachricht also erst, wenn sie wieder Rechenzeit erhält.

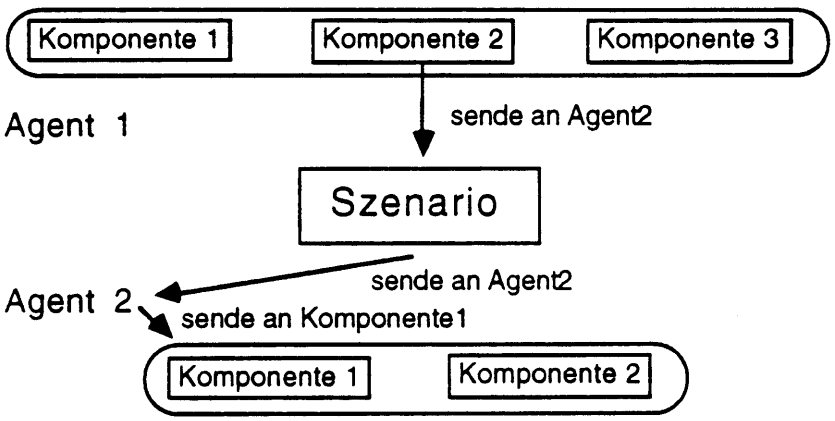


Bild 2: Kommunikation zwischen Agenten

Derselbe Mechanismus wird verwendet, um mit dem Szenario zu kommunizieren. Eine Komponente schickt eine Liste von Lisp-Werten an die Welt (**world**), welche von einer Funktion des Szenarios (**send-world**) verarbeitet wird, und erhält die Antwort in der (vordefinierten) Variable **send** abgelegt. Die Antwort, die möglicherweise **nil** sein kann, wird direkt erzeugt, bevor die

§ 2. Benutzen der Simulationsumgebung

SEMAW ist eine interaktive Simulationsumgebung. Der Benutzer hat die Möglichkeit durch Menüs, den Verlauf der Simulation zu steuern. Er kann Szenarien konfigurieren, die Agenten mit entsprechenden Regelsätzen laden, und das aktuelle Szenario beliebig oft ablaufen lassen, wobei er einen Lauf auf verschiedene Weise überwachen kann.

Wenn alle Systemdateien geladen wurden, gelangen Sie durch die Tastenkombination *Select-R* in das Hauptmenü von *SEMAW*. Beschrieben wird in dieser Anleitung die Version 1.1. Dort ist der Bildschirm wie folgt aufgeteilt: Unter dem Programmnamen sehen Sie eine Menüzeile; beim ersten Aufruf erscheint dort das Hauptmenü, welches folgende Menüpunkte enthält:

Configure-Scenario Load-Rule-Sets Monitor-Agents Run-Scenario System

Der übrige Teil des Bildschirms ist in drei Fenster aufgeteilt, welche die Namen *Dialogue*, *System Display* bzw. *Scenario* in der linken, unteren Ecke tragen. Das Fenster *Scenario* dient nur zur Ausgabe des Szenarios. Sein Erscheinungsbild ist abhängig von dem simulierten Szenario, ist aber gedacht als visuelle Information über den aktuellen Stand im Szenario. Das Fenster *System Display* dient zur Ausgabe von textueller Information sowohl des Systems als auch des Szenarios. Vom System werden hier Modusinforamtionen, wie z.B. ob der Debug-Modus eingeschaltet ist, und vom Szenario können zum Beispiel statistische Informationen ausgegeben werden. Das Fenster *Dialogue* ist das aktuelle Lisp-Fenster, in dem zum einen auch Text ausgegeben wird, das aber vor allem zur Interaktion mit dem Benutzer (sofern dies nicht über Menüs geschieht) und dem (Lisp-) System dient.

Das Hauptmenü besteht aus zwei Untermenüs (*Configure-Scenario* und *Load-Rule-Sets*), sowie drei Pop-Up-Menüs (*Monitor-Agents*, *Run-Scenario* und *System*). Die Auswahl von *Configure-Scenario* bringt Sie in ein Untermenü, in dem Sie das Szenario konfigurieren und die zugehörigen Agenten erzeugen können. *Load-Rule-Sets* ruft ein Untermenü auf, in dem Agenten mit Regelsätzen geladen werden können, aber auch der *Running-Modus* des Systems festgelegt wird. Das Pop-Up-Menü *Monitor-Agents* dient zum Ausgeben von Debug-Informationen während eines Szenario-Laufes. Mit *Run-Scenario* kann das Szenario gestartet und zurückgesetzt werden. Die Menüoption *System* bietet verschiedene Systemdienste, wie das Verlassen von *SEMAW*, an. Die Menüs werden nun detailliert besprochen.

2.1 *Configure-Scenario*

Nach Aufrufen dieser Menüoption wird die Hauptmenüzeile durch folgendes Menü ersetzt:

Initialize-System Configure-Scenario Create-Agents Show-Configuration Return

Initialize-System

Die Konfiguration des Szenarios, sowie alle vorhandenen Agenten werden gelöscht. Damit ist das System wieder im Anfangszustand. Dieser Befehl hat keinen Einfluß auf die im System geladenen Regelsätze.

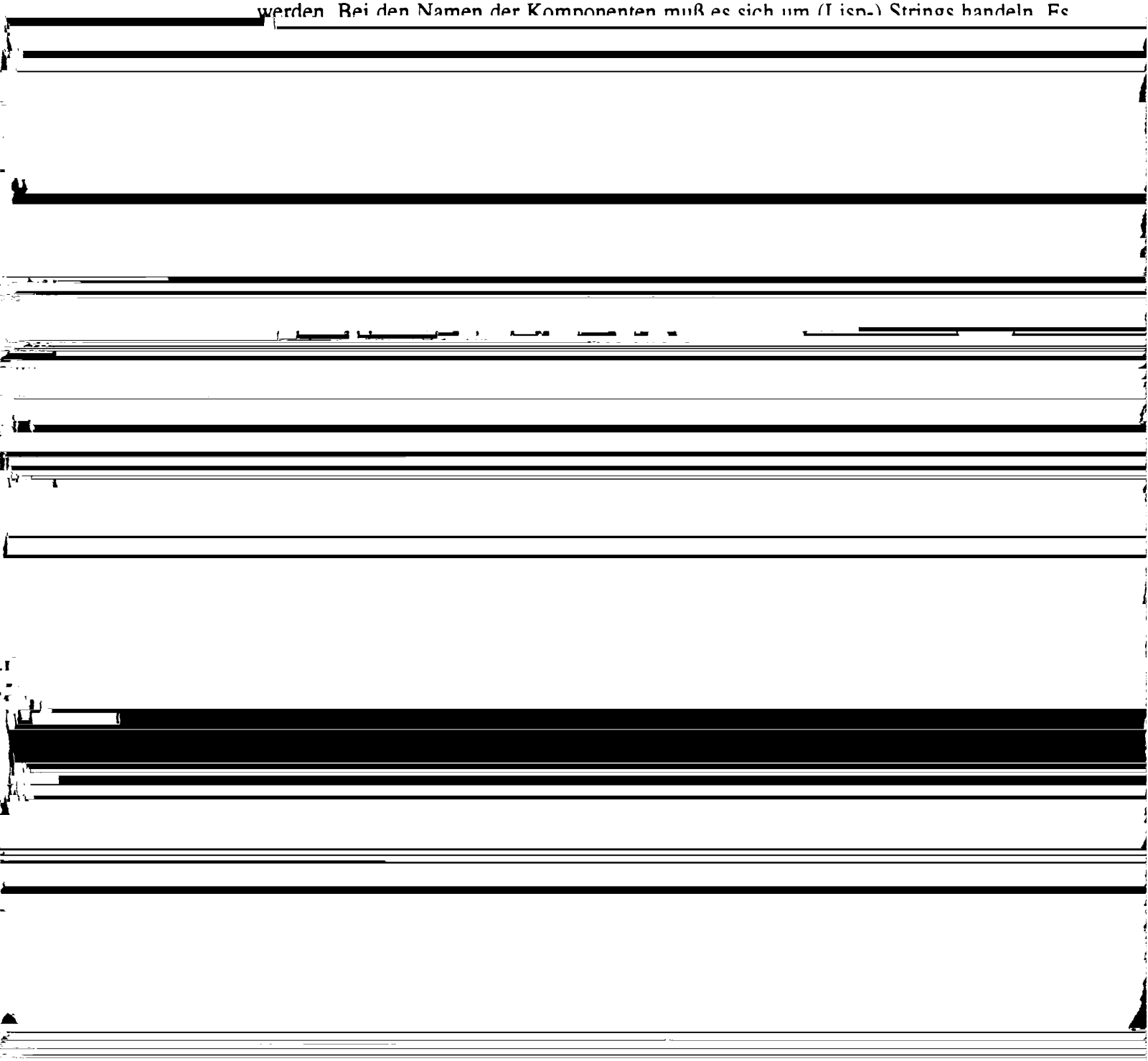
Configure-Scenario

Das System wird automatisch initialisiert und das Szenario wird konfiguriert. Dazu wird die Funktion **scenario:configure-scenario** aufgerufen (siehe § 5). Die Agenten

werden nicht automatisch erzeugt, die Option *Create-Agents* muß also ausgeführt werden, bevor mit anderen Dingen fortgefahren werden kann.

Create-Agents

Diese Option erzeugt interaktiv die zum Szenario gehörigen Agenten. Die Anzahl der Agenten ist durch die Konfiguration des Szenarios vorbestimmt. Es besteht die Möglichkeit alle Agenten gleich zu gestalten, in dem Fall muß nur ein Agent exemplarisch konfiguriert werden, oder jeder Agent muß einzeln konfiguriert werden. Die Konfiguration eines Agenten geschieht durch Angabe seiner Komponenten, welche aus einer Liste von Namen für die Komponenten besteht, die dann automatisch erzeugt werden. Bei den Namen der Komponenten muß es sich um (Lisp-) Strings handeln. Es



ihre Namen angegeben werden. Das ist wichtig für die Zustellung von Nachrichten, da eine Nachricht an einen Agenten immer der ersten Komponente übergeben wird.

Show Configuration

Load-Rule-Sets

Mit dieser Funktion können die Agenten mit Regelsätzen geladen werden. Dazu muß bestimmt werden, welche Komponenten welcher Agenten mit welchem Regelsatz geladen wird. Zuerst können Sie auswählen, ob alle, einige oder nur ein Agent geladen werden soll. In diesem Menü (*load all agents*, *load several agents*, *load one agent*, *show rule sets*) können Sie sich auch mit der letzten Option einen Regelsatz anschauen. Sie erhalten sofort das Menü zum Auswählen eines Regelsatzes und können sich den Regelsatz auf dem

Nachrichten zu überwachen. Die Auswahl der Komponenten, die überwacht werden sollen, ist dem Verfahren beim Laden von Regelsätzen ähnlich.

View Running (Toggle)

Mit dieser Option wird die Ausgabe, welche Komponente welches Agenten gerade rechnet, ein- bzw. ausgeschaltet wird.

View Agents

Hier muß zuerst eine Komponente einer Menge von Agenten spezifiziert werden. Für diese wird dann eine Liste aller Regelsätze, d.h. der Wurzel und aller Söhne, aufgestellt. Man hat hier die Möglichkeit, daß einmal nur ausgegeben wird, wenn ein bestimmter Regelsatz ausgeführt wird, oder daß für diesen Regelsatz auch die ausgeführten Befehl (samt der Parameterbelegungen) zusätzlich aufgelistet werden. Weiterhin kann man zu beiden *rekursiv* angeben, so daß die Auswahl auch für alle Söhne gilt.

View Sending

Mit dieser Option wählt man eine Menge von Komponenten einer Menge von Agenten aus, für die jegliche Kommunikation ausgegeben wird. Es wird ausgegeben, wann eine Komponente eine Nachricht verschickt, und wann sie eine erhält.

Clear Agent

Für eine Menge von Komponenten (einer Menge von Agenten) wird die Interpretation nicht mehr überwacht. Hierbei wird die Überwachung einer Komponente generell ausgeschaltet, also nicht nur bezüglich einzelner Regelsätze.

Clear Sending

Für die spezifizierte Komponente (einer Menge von Agenten) wird keine Information mehr ausgegeben.

Clear All

Es wird keine Information mehr ausgegeben. Das gilt für alle Agenten.

2.4 *Run-Scenario*

Run-Scenario

Diese Menüoption ruft die vom Benutzer programmierte Funktion **scenario:run-scenario** auf.

Reset-Scenario

Diese Funktion hat denselben Effekt, wie die Menüoption *Reset-System* von *Load-Rule-Sets* (siehe 2.2). Das Szenario und die Agenten werden in den Startzustand zurückversetzt.

Run-Agents

Diese Funktion stößt den Interpreter an, wodurch die Agenten beginnen zu rechnen.

2.5 *System*

Dieses Menü bietet Funktionen, die mehr der generellen Handhabung der Simulationsumgebung (als Programm) dienen.

Toggle Moore

Im Interaktionsfenster kann die Option gesetzt werden, daß der Benutzer gefragt wird, wann der Bildschirm gescrollt werden soll, sobald er vollgeschrieben ist. Diese Funktion wird von verschiedenen Befehlen automatisch gesetzt und wieder gelöscht. Sie wird jedoch nicht von allen gesetzt, bei denen es sinnvoll erscheinen könnte, so z.B. bei der Ausgabe von Monitor-Informationen. Ferner bleibt die Option gesetzt, falls Befehle, die diese setzen, unterbrochen werden. Von daher besteht hier die Möglichkeit, diese Option ein- und auszuschalten.

Quit

Mit dieser Option können Sie *SEMAW* verlassen.

§ 3. Beispiel "Tower of Hanoi"

3.1 Ziel der Simulation

Ich will in diesem Paragraphen eine Anwendung beschreiben, an der man beispielhaft erkennen kann, wie die Simulationsumgebung genutzt werden kann. Es handelt sich hierbei um eine Multi-Agenten Lösung des *Tower of Hanoi* Problems. Zuerst wird der Modellansatz beschrieben, dann erklärt in welchen Schritten dieser mit *SEMAW* umgesetzt wurde. Für die Leser, die das *Tower of Hanoi* Problem (in seiner allgemeinen Form) nicht kennen, soll diese jetzt erklärt werden. Das Szenario des *Tower of Hanoi* besteht aus m Stäben und n Scheiben, wobei die Scheiben unterschiedlich groß sind. Es besteht die Einschränkung, daß eine Scheibe nicht auf einer kleineren liegen darf. Am Anfang steht der Turm, mit der größten Scheibe unten und der kleinsten Scheibe oben, auf dem Startstab. Ziel ist es nun den Turm auf dem Zielstab zu rekonstruieren. Pro Schritt darf sich von einem Stab nur eine Scheibe erheben, d.h. parallel können maximal m Scheiben agieren.

Natürlich läßt sich das *Tower of Hanoi* Problem zentral lösen. Leicht ist ein rekursives Programm für die Lösung bei drei Stäben geschrieben. Jedoch ist zu beachten, daß der nächste Schritt bei drei Stäben leicht zu bestimmen ist; in der Regel ist nur ein Stab (als nächstes Ziel der Scheibe) sinnvoll. Hat man mehr als drei Stäbe, so wächst die Wahlmöglichkeit und der Suchraum wird exponentiell groß. Bei einer beliebigen Anzahl von Stäben ist eine optimale Strategie nicht mehr offensichtlich. Mit dem Ansatz, der hier gewählt wurde, sollte erreicht werden, daß die Lösung mit Hilfe einer lokalen Strategie, nämlich der der einzelnen Scheibe als Agent, gefunden wird. Die Hoffnung ist, mit einer relativ einfachen lokalen Strategie eine schnelle, wenn nicht sogar eine optimale Lösung zu finden. Damit würde man eine Suche durch den exponentiellen Zustandsgraphen vermeiden. Tiefer soll nicht

in die Problematik des *Tower of Hanoi* und des Multi-Agenten Ansatzes eingestiegen werden. Die aufgeführten Überlegungen dienen lediglich zum besseren Verständnis des Beispiels.

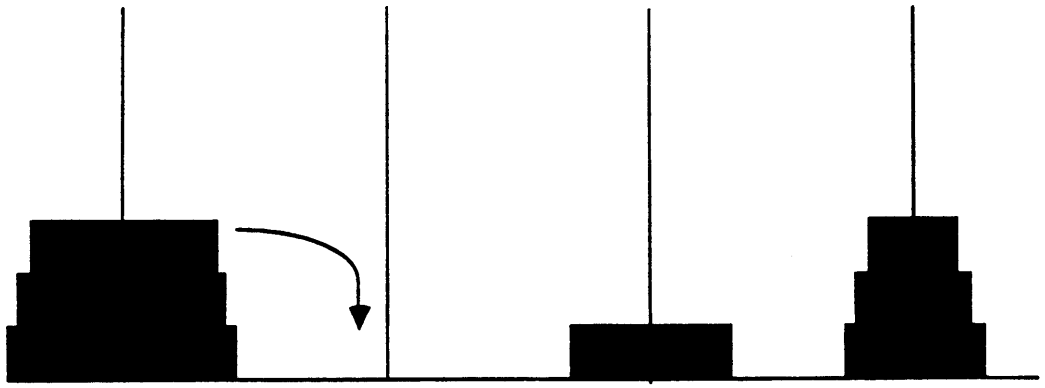
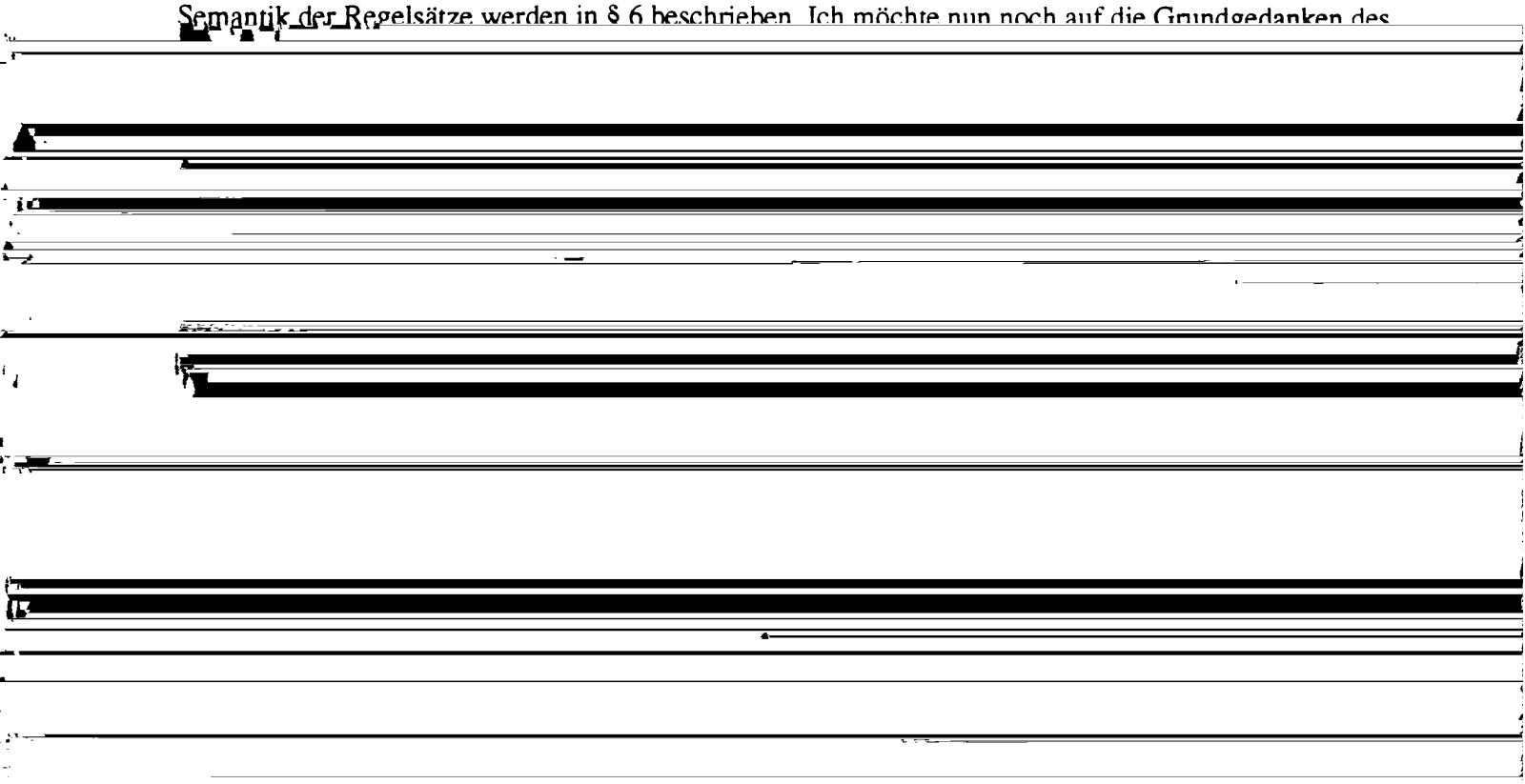


Bild 3: Tower of Hanoi

Was soll also simuliert werden? Das Szenario besteht, wie oben angedeutet, aus m Stäben und n Scheiben, wobei m und n vom Benutzer bestimmt werden. Jede Scheibe soll ein Agent sein, der simuliert wird. Die Agenten planen in jedem Schritt aufgrund ihres lokalen Wissens, welche Bewegung sie, wenn sie sich überhaupt bewegen können, als nächstes ausführen. Die Planung läuft so ab, daß der Agent sein lokales Ziel kennt, also seinen Zielstab und seine Zielposition, feststellt, ob er diese erreichen kann, und wenn dies der Fall ist, es auch versucht. Nun kann es aber sein, daß entweder die Position, die er erreichen will, besetzt ist, oder er selbst von einer anderen Scheibe

mit demselben Regelsatz geladen wird. Daher sehen Sie in Anhang A nur einen **enter-rule-set** Befehl.

Einen Regelsatz kann man als Programmiersprache für eine Komponente ansehen. Syntax und Semantik der Regelsätze werden in § 6 beschrieben. Ich möchte nun noch auf die Grundgedanken des



Das oberste Niveau des Regelsatzes dient zur Kommunikation mit dem Szenario. Während des Ablaufs, z.B. zu Beginn als Initialisierung, erhält die Komponente Nachrichten mit einer bestimmten Semantik, die sie verarbeiten muß. Insbesondere werden durch die Nachrichten *create-next-goal* und *inform-other-agents* Planung und Kommunikation (mit anderen Agenten) angestoßen. Ebenso werden auf diesem Niveau Nachrichten an die Umwelt gesendet. Die Variable **message** dient hier als eine Art Agenda.

Wenn ein Agent die Nachricht *create-next-goal* erhält, geht er wie folgt vor. Zuerst stellt er fest, wo er ist, ob er sein letztes Ziel (**next-goal**) erreicht hat, und ob er blockiert ist. Dies alles dient zur Planungsvorbereitung (**prepare-planning**). Dann erst wird getestet, ob das Ziel erreichbar oder schon erreicht ist (**test-for-goal**). In **plan-leave** wird ein nächstes Ziel (**next-goal**) ausgewählt, falls **leave**-Nachrichten vorliegen, aber noch kein Ziel geplant wurde. **Plan-leave** ist der größte und wohl auch interessanteste Teil des Regelsatzes (neben **communicate**), in dem der Agent die

Möglichkeit hat, den nächsten Schritt unter Beachtung der **leave**-Restriktionen auszuwählen (**plan-alternatives**). Wurde immer noch keine Aktion geplant, so kann der Agent in **plan-heuristically** seine Ruhezeit nutzen, um Schritte zu planen, mit denen er, ohne andere Agenten zu behindern, spätere Schritte unterstützen kann.

Durch die Nachricht *inform-other-agents* wird der Regelsatz **communicate** aufgerufen. Im Anhang ist dieser Regelsatz in **communicate**, **com-leave1-KO**, **com-leave2-KO** und **com-inspect-place-KO** aufgeteilt. Wie in 6.3 beschrieben wird, hat man die Möglichkeit Varianten zu Regelsätzen zu definieren. In diesem Fall wurde die Fähigkeit zu kommunizieren in eine Variante verlegt. Deshalb besteht der Regelsatz **communicate** nur aus der Regel, die dem Szenario den Wunsch zu handeln mitteilt. Soll der Agent mit anderen Agenten kommunizieren, so werden die übrigen Regel (mit dem Suffix **KO**) hinzugefügt, die feststellen, ob der Agent blockiert oder seine

Die Funktion **configure-scenario** konfiguriert interaktiv das Szenario, was in Fall des *Tower of Hanoi* bedeutet, daß der Benutzer die Anzahl der Stäbe und Scheiben, sowie deren Start- und Zielposition bestimmen kann. Die Position können einmal generell erzeugt werden, d.h. es wird ein Startturm gebildet und die Scheiben bekommen als Ziel, den Turm auf einem anderen Stab zu rekonstruieren, oder der Benutzer muß jede Start- und Zielposition per Hand eingeben. Die Funktion **configure-scenario** muß die Konsistenz des Startzustandes kontrollieren, unterstützt von der Hilfsfunktion **consistent-scenario?**, die eigenen Variablen initialisieren und, dies ist unbedingt notwendig, die globale Variable ***agents***, die vom System definiert ist, korrekt initialisieren (für Details siehe 6.1).

Als nächstes folgen im Listing die Ausgabefunktionen **print-scenario-configuration** und **print-scenario. Print-scenario** ist dabei sehr technisch, da es die graphische Bildschirmausgabe realisiert. Im wesentlichen stellt es die Stäbe und Scheiben in ihrer aktuellen Position dar. Die nächste Funktion **send-world** ist die Schnittstelle zwischen Agenten und Szenario. Immer wenn Agenten eine Nachricht an die Welt schicken, wird diese der Funktion **send-world** übergeben. So erhält zum Beispiel die Funktion die Nachricht *leave* und sendet sie weiter an den adressierten Agenten. Desweiteren beantwortet **send-world** Anfragen über den Zustand des Szenarios (*where-am-i*, *look-at-place* und *look-above*). Als wichtigste Funktion (*wish-to-act*) sammelt es die Handlungswünsche der Agenten.

Es folgen die eher technischen Funktionen **perform-parallel-move**, **update-system-display** und **perform-moves. Perform-moves** wählt aus den gesammelten Handlungswünschen die durchführbaren aus. Dabei ist die Heuristik, Gruppen von Scheiben auszuwählen, die parallel agieren können. Für jede Gruppe wird **perform-parallel-move** ausgeführt, welches die Bewegungen auf dem Bildschirm ausführt und die Datenstrukturen aktualisiert. **Update-system-display** korrigiert die Ausgabe des Display-Fensters.

Die Simulation läuft in Zyklen ab, wobei die Agenten erst allein für sich Planen, dann miteinander kommunizieren, und schließlich agieren können. Ein solcher kontrollierter Ablauf muß vom Szenario selbst gesteuert werden, dazu dient die Funktion **run-scenario**. Sie erzeugt diesen Ablauf, indem sie entsprechende Nachrichten an die Agenten schickt und diese dann anstößt (mit **run-interpreter**). Zuerst werden die Agenten mit der Nachricht *initialize* initialisiert, dann mit *create next goal* wird ihre Planung angestoßen, mit *inform other agents* beginnt die Kommunikation und schließlich wird **perform-moves** aufgerufen, bevor erneut begonnen wird.

3.4 Handhaben der Menüs

Nachdem alle Programmieraufgaben gelöst wurden, kann nun das Szenario simuliert werden. Zuerst muß ein konkretes Szenario spezifiziert werden. Im Untermenü *Configure-Scenario* wählen wir den gleichlautenden Befehl aus. Das System wird initialisiert und wir können interaktiv die Anzahl der Stäbe und Scheiben auswählen. Wir geben 6 Stäbe und 8 Scheiben an, welche den Turm von Stab 1 auf Stab 2 rekonstruieren sollen. Nun müssen erst die zu den Scheiben gehörenden Agenten erzeugt werden, was mit dem Befehl *Create-Agents* des gleichen Menüs geschieht. Aufgrund der Struktur des Szenarios werden alle Agenten mit nur einer Komponente erzeugt, die (willkürlich) als "**planner**" bezeichnet wird. Das Szenario ist damit ausreichend spezifiziert.

Damit die Simulation ablaufen kann, muß das Verhalten der Agenten programmiert werden. Dazu

Regelsatz schon eingegeben wurde (siehe 3.2), können wir direkt *Load-Rule-Sets* aufrufen. Wir spezifizieren *load all agents*, und wählen die Komponente "**planner**", sowie den einzigen Regelsatz "**ToH: Local Decisions**" aus. Daraufhin müssen wir noch die Varianten **communicate**, **willing-to-help** und **knowledge-of-others** auswählen (siehe 6.3), damit das Szenario konvergiert. Haben wir dies zum ersten Mal gemacht, so wird der Regelsatz compiliert, bevor die Agenten geladen werden. Normalerweise müßte noch der Ablaufmodus ausgewählt werden. Da dieser aber auf **sequential** voreingestellt ist, brauchen wir den Befehl *Set-Running-Mode* nicht auszuführen.

Jetzt ist das Szenario ablauffertig. Mit *Run-Scenario* (Menü *Run-Scenario*) sehen wir, wie die Agenten den Turm auf Stab 2 aufbauen. Dies können wir beliebig oft wiederholen, indem das Szenario erst durch *Reset-Scenario* (Menü *Run-Scenario*) zurückgesetzt wird. In der Regel wollen

semaw:*agents*

Diese globale Variable ist eine Liste der im Szenario vorhandenen Agenten. Sie dient zur Verbindung der Agenten des Szenarios und der Agenten, die vom System erzeugt wurden.

Die Variable ist eine Liste von Einträgen der Form:

(**<agent-name> <system-pointer> <scenario-pointer>**)

Der Agentenname (**<agent-name>**) wird von der Funktion **configure-scenario** (siehe § 5) (eindeutig) vergeben und dient als Verweis in diese Liste. Dieser Name dient zur eindeutigen Referenzierung zwischen System und Szenario. Der Verweis des Szenarios (**<scenario-pointer>**) wird ebenfalls von der Funktion **configure-scenario** gesetzt und dient zum Wiederauffinden des Agenten in dem Szenario (durch die Szenario-Funktionen). Dasselbe gilt für den Verweis des Systems (**<system-pointer>**), die bei der Erzeugung der Agenten erstellt werden. Er ist nur für die Systemfunktionen gedacht.

Fenster-Variablen:

semaw:*scenario-display*
semaw:*system-display*
semaw:*scenario-window-width*
semaw:*scenario-window-height*
semaw:*system-window-width*
semaw:*system-window-height*
semaw:*system-display-offset*

Die Bildschirmausgabe des Systems ist in Interaktions-, Display- und Szenario-Fenster aufgeteilt. Das Interaktionsfenster kann durch Ein-/Ausgabe auf den ***standard-input*** / ***standard-output*** verwendet werden. Das Szenario-Fenster dient allein zur Darstellung des Szenarios und kann vollständig von den Szenario-Funktionen (insbesondere **print-scenario**) genutzt werden. Der Wert von ***scenario-display*** ist das entsprechende Fenster; ***scenario-window-width*** und ***scenario-window-height*** geben die (innere) Breite und Höhe des Fensters an. Das Display-Fenster wird sowohl vom System als auch von dem Szenario zur Ausgabe, z.B. von statistischer oder Zustandsinformation, benutzt. Es gilt dasselbe, wie für das Szenario-Fenster. Allerdings wird die obere Hälfte des Fensters durch das System genutzt. Die Ausgabe des Szenarios darf erst ab den ***system-display-offset*** erfolgen.

Funktionen:

semaw:error-message (format-string &rest args)

Diese Funktion dient zur geregelten Fehlerausgabe. Es wird empfohlen diese zu verwenden, es ist aber nicht zwingend. Die Funktionalität ist gleich der des Lisp-Befehls **format**. Die Ausgabe hat folgende Gestalt, wobei **string** den durch Anwendung von **format** auf **format-string** und **args** resultierenden String denotiert:

"~%~% ERROR: string~%"

semaw:send-agent (agent-name var message)

Wenn eine Nachricht an einen Agenten geschickt werden soll, muß die Funktion **send-agent** aufgerufen werden. **Agent-name** ist der eindeutige Name des Agenten in ***agents***. Das Argument **var** muß immer **nil** sein (es hat nur Bedeutung für das System). Die Variable **message** ist die eigentliche Nachricht, die übermittelt werden soll.

semaw:run-interpreter ()

Die Funktion **run-interpreter** sucht aktive Komponenten und läßt diese rechnen. Die Art und Weise der Rechenzeitzuteilung hängt vom Modus ab. Die Funktion endet erst, wenn keine Komponente mehr aktiv ist, also alle zu Ende gerechnet haben.

§ 5. Funktionen zur Simulation des Szenarios

Im folgenden werden die Funktionen beschrieben, die unbedingt zur Simulation des Szenarios zur Verfügung gestellt und in das Package **scenario** geladen werden müssen.

initialize-scenario ()

Diese Funktion muß das Szenario vollständig zurücksetzen, d.h in den Zustand direkt nach dem Laden des Szenarios.

configure-scenario ()

Diese Funktion muß das Szenario konfigurieren, d.h. alles notwendige tun, damit das Szenario ablauffähig ist. Zum Beispiel die Anzahl der Agenten bestimmen, insbesondere die Variable **semaw:*agents*** (siehe § 4) korrekt initialisieren.

reset-scenario ()

Diese Funktion muß das Szenario, in den von **configure-scenario** definierten Anfangszustand zurücksetzen.

print-scenario ()

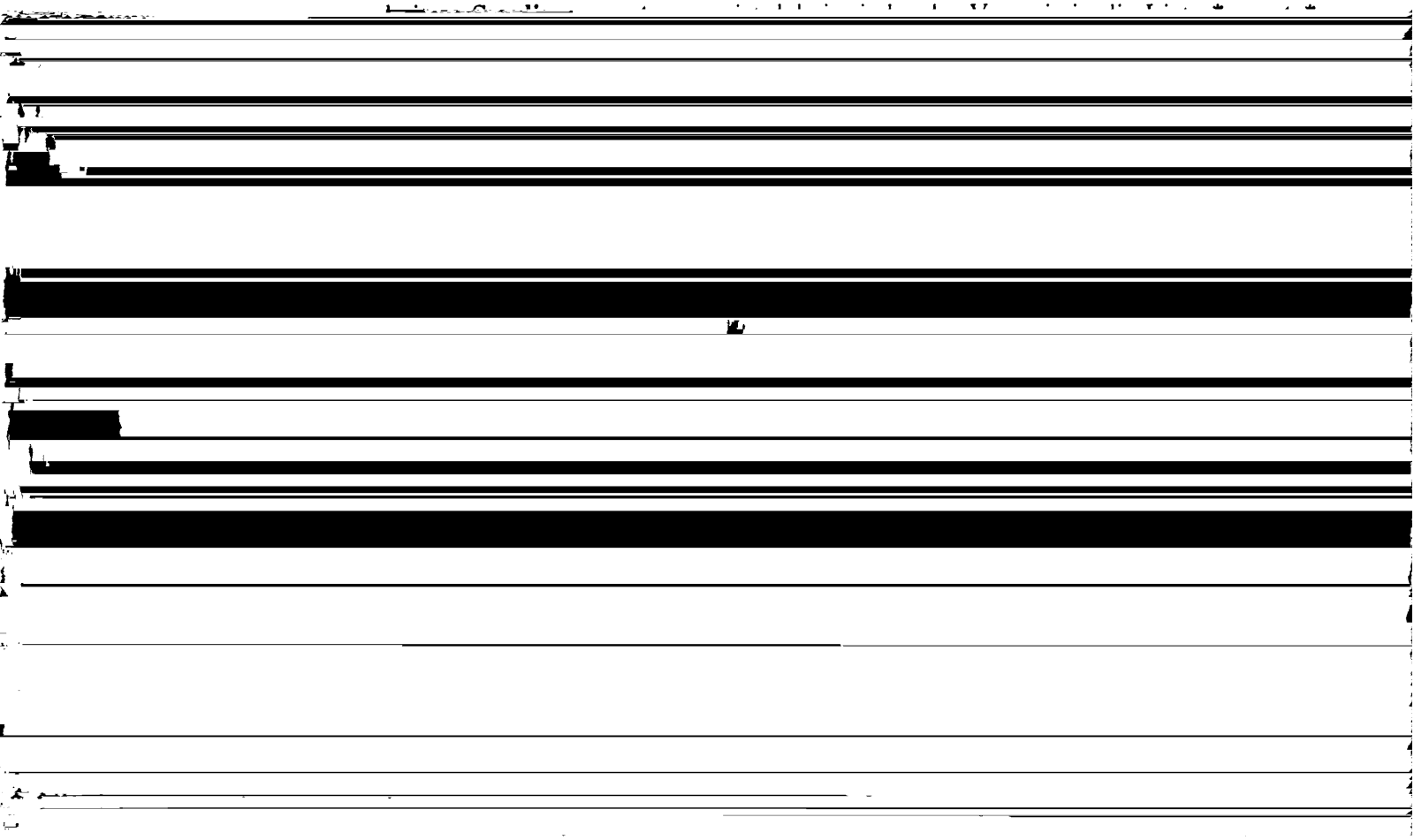
Diese Funktion stellt das Szenario (und den augenblicklichen Zustand) auf dem Bildschirm dar. Es kann dazu die in § 4 beschriebenen globalen Variablen benutzen. Es wird automatisch nach **configure-scenario** und **reset-scenario** aufgerufen, es wird nicht (und sollte in der Regel auch nicht) nach **initialize-scenario** aufgerufen werden.

print-scenario-configuration (agent-name &optional mode)

Diese Funktion soll szenario-bedingte Information auf dem ***standard-output*** ausgeben. Ist **mode** gesetzt, so soll auch zustandsbedingte Information erscheinen, z.B. augenblickliche Position.

send-world (sending-agent-name args)

Diese Funktion soll die "Nachrichten", die Agenten an die Welt (**world**) schicken,



Das Ergebnis dieser Funktion wird der entsprechenden Komponente in der Variablen **send** abgelegt.

run-scenario ()

Diese Funktion kann verwendet werden, um den Ablauf des Szenarios gezielt zu steuern. Zum Beispiel könnte diese Funktion eine Abfolge von Nachrichten an die Agenten und Aufrufen von **run-interpreter** bestehen. Man kann dadurch komplexere Simulationsabläufe realisieren. Diese Funktion wird mit der Menü-Option "run-scenario" (siehe § 2) aufgerufen.

§ 6. Syntax und Semantik der Regelsätze

6.1 Syntax

Die (rekursive) Syntax der Regelsätze ist im folgenden aufgeführt. Dabei bedeutet **<rule-set>**, daß es sich um eine grammatikalische Variable (Nicht-Terminal) handelt, der Operator * denotiert die unbestimmte, mindestens einmalige Wiederholung, | die Alternative und [ ] bezeichnen die optionale Verwendung. Alle anderen Zeichen sind Terminale.

```
<rule-set>      ::= (rule-set <rule-set-name>
                              <variable-list>
                              (rules
                                  <rule>*
                              )
```

<variable-name> kann ein (Lisp-) Symbol, welches nicht **nil** ist, sein.
<addresse> kann ein (Lisp-) String sein.
<predicate> ::= **<lisp-value>.**
<lisp-value> ist ein (Lisp-) Ausdruck, in dem die (in den Regelsätzen) definierten Variablen verwendet werden kann.

6.2 Semantik

Jeder Regelsatz kann weitere Regelsätze enthalten. Dadurch entsteht eine Baumstruktur, wobei der Verzweigungsgrad beliebig groß ist. Diese Struktur wird als *Baum* bezeichnet; der oberste Regelsatz wird *Wurzel* genannt; enthaltene Regelsätze werden als *Söhne* bezeichnet, ein Regelsatz ist *Vater* seiner Söhne. Die *Sicht* eines Regelsatz besteht aus seinen Variablen und enthaltenen Regelsätzen, sowie der Sicht seines Vaters.

Der Interpreter beginnt seine Rechnung, wenn er zum ersten Mal aufgerufen wird, an der Wurzel. Er erzeugt für die Wurzel einen *Aktivierungssatz (AS)*, in dem die Variablen des Regelsatzes instantiiert werden. Die Variablennamen müssen pro Regelsatz verschieden sein. Mit jedem Verzweigen (siehe **call** Befehl) zu einem Regelsatz (auch demselben) wird ein neuer AS erzeugt. Dadurch ist es möglich rekursive Regelsätze zu schreiben, da ein Regelsatz sich selbst aufrufen kann, ohne seinen AS zu verlieren. Mit jedem Verlassen (siehe **return** oder **break** Befehl) eines Regelsatzes wird der AS gelöscht. Die Wurzel kann nicht verlassen werden, dies führt zu einem Laufzeitfehler.

Variablen werden beim Erzeugen eines AS mit **nil** initialisiert. Sie können jeden (Lisp-) Wert annehmen; die Zuweisung kann aber nur durch den **set** Befehl durchgeführt werden. Ein Lisp-Wert (**<lisp-value>**) kann die Variablen als Argumente verwenden. Aus einem solchen Ausdruck wird eine Lisp-Funktion gemacht, der die verwendeten Variablen als Argumente übergeben werden. Ein Lisp-Wert (**<lisp-value>**) kann all die Variablen verwenden, die in seiner Sicht enthalten sind; die Verwendung der Variablen ist also statisch (orientiert sich am Programmtext). Das bedeutet zum Beispiel, daß, wenn ein Regelsatz sich selbst aufruft, die neue Instantiierung des Regelsatzes nur seine eigenen Variablen und die Variablen, die der Vater (der auch Vater des rufenden Regelsatzes ist) referenzieren kann, sieht. Wenn ein Lisp-Wert (**<lisp-value>**) ausgewertet wird, dann wird er mit den Variablen-Werten versorgt, die zu diesem Zeitpunkt gültig sind, und ausgewertet.

Wenn eine Komponente mit einem Regelsatz geladen wird, so wird die Variablenmenge der Wurzel um die Variablen **message** und **send** erweitert. Diese können also wie ganz normale Variablen verwendet werden. Beide Variablen werden vom System in bestimmter Semantik verwendet, können aber vom Regelsatz verändert werden. Wenn immer die Komponente eine Nachricht bekommt, so wird diese als erstes Listenelement in **message** eingefügt. Die Variable **message** sollte also immer so geändert werden, daß sie eine Liste bleibt. Die Variable **send** wird auf den Ergebniswert eines **send** Befehls gesetzt; alte Werte werden dabei überschrieben.

Ein Regelsatz wird in *Zyklen* interpretiert. Am Anfang des Regelsatzes wertet der Interpreter alle Prädikate (**<predicate>**) der Regeln (eines Regelsatzes) aus. Die Befehle der zu wahr (**not nil** in Lisp) ausgewerteten Prädikate werden zu einer Liste (in der Reihenfolge wie sie im Programmtext aufgelistet sind) verkettet und sequentiell ausgeführt. Nachdem alle Befehle abgearbeitet sind, beginnt der Interpreter einen neuen Zyklus. Ein Zyklus bezieht sich immer nur auf einen Regelsatz. Bei der Ausführung eines **call** Befehls wird der Zyklus des aktuellen Regelsatzes eingefroren und der neue

Regelsatz interpretiert, bis dieser verlassen wird. Dann wird im alten Zyklus fortgefahren. Es ist zu beachten, daß ein aufgerufener Regelsatz die Variablen in der Sicht des aktuellen Regelsatzes verändern kann.

Im folgenden wird die Semantik der Befehle erklärt:

set Befehl:

Die Lisp-Werte werden erst alle ausgewertet. Dann werden sie den Variablen zugewiesen. Die Zuweisung ist also parallel.

send Befehl:

Alle Lisp-Werte werden ausgewertet und zu einer Liste verkettet. Ist der Adressat (**<addressee>**) das Szenario (denotiert durch das Symbol **world**), dann wird die Nachricht an die Welt geschickt (d.h. der Funktion send-world übergeben, siehe § 5). Ansonsten wird eine Komponente (desselben Agenten) mit dem String als Namen gesucht, und die Nachricht dieser Komponente übergeben. Existiert keine Komponente mit diesem Namen, so entsteht ein Laufzeitfehler.

call Befehl:

Falls vorhanden, werden erst die Lisp-Werte berechnet. Dann wird der Regelsatz gesucht, der sich in der Sicht des rufenden Regelsatzes befinden muß. Der Regelsatz wird instantiiert und die Variablen initialisiert. Jede Variable, die im **call** Befehl einen Wert zugewiesen bekommt, wird mit dem korrespondierenden Wert initialisiert. In einem **call** Befehl können nur den Variablen des gerufenen Regelsatzes Werte zugewiesen werden. Der aktuelle Regelsatz wird an der Stelle des **call** Befehls unterbrochen, und der Interpreter verzweigt zu dem neuen Regelsatz.

return Befehl:

Die Ausführung eines **return** Befehls bewirkt, daß der Interpreter am Ende des Zyklus den Regelsatz verläßt und zu dem rufenden Regelsatz zurückkehrt (das muß nicht der Vater sein).

break Befehl:

Ein **break** Befehl bewirkt, daß der Interpreter den Zyklus abbricht und sofort zum rufenden Regelsatz zurückkehrt. Folgende Befehle werden also nicht mehr ausgeführt.

stop Befehl:

Bei einem **stop** Befehl hält der Interpreter und verzweigt zur Wurzel. Variablenwerte bleiben erhalten.

6.3 Eingabe der Regelsätze

Regelsätze müssen erst in das System eingegeben werden, bevor Agenten mit ihnen geladen werden können. Die Eingabe der Regelsätze erfolgt mit folgender Funktion:

enter-rule-set (rule-set &optional add-list)

rule-set ist ein gewöhnlicher Regelsatz. Die **add-list** dient zur Modifizierung des Regelsatzes beim Laden einer Komponente. Wenn im Menü (siehe § 2) ein Regelsatz ausgewählt wird, dann kann der Benutzer noch Varianten aus der **add-list** (sofern diese vorhanden sind) auswählen. Diese ausgewählten Varianten modifizieren den Regelsatz, wie im folgenden erklärt wird.

<add-list>	::=	((**<option-name>** **<modify-command>***)*).
<modify-command>	::=	(**<command>** (**<rule-set-name>***) **<exchange-part>**).
<command>	::=	add-rule |
		add-rule-set |
		replace-rule |
		replace-rule-set.

<option-name> ist der Name, unter der die Variante im Menü erscheint. Es muß ein (Lisp-) String sein. Die Liste der Regelsatznamen definiert den Regelsatz, in dem die Operation ausgeführt werden soll. Eine leere Liste denotiert die Wurzel (sie muß in keinem Pfad angegeben werden). Ansonsten geben die Namen (von links nach rechts) an, in welchen Sohn abgestiegen werden soll.

Add-rule und **replace-rule** erhalten als Argument eine Regel, die textuell (dem spezifizierten Regelsatz) hinzugefügt wird, bzw. die eine Regel mit demselben Namen ersetzt. **Add-rule-set** und **replace-rule-set** erhalten Regelsätze als Argumente. Sie arbeiten auf dieselbe Art und Weise. Die Veränderung ist rein textuell; nach der Veränderung muß der Regelsatz wieder korrekt sein.

Anhang A

```
;;;-*- Syntax: Common-Lisp; Base: 10; Package: SEMAW; Mode: LISP -*-

(in-package 'semaw)

; ************************************************************
; **     rule-sets for Tower of Hanoi
; ************************************************************

; The abilities of an agent are seperated according to the following scheme:
;
;  no knowledge about the goal position      (GG)
;  ability to communicate                    (CO)
;  willingness to help                       (WH)
;  knowledge of other agents                 (KO)
;  knowledge of the global task              (KT)

; the spacial position of an agent is modelled by a list: ("stick" "height of agent's position")
; this is usually abbreviated by (place position)

(enter-rule-set
 `(rule-set "ToH: Local Decisions"
   (variables agent              ; number of agent (his priority is: number-of-agents - agent)
              number-of-places
              blocker            ; number of agent that blocks this agent
              blocker-sent-off   ; keep in mind if blocker was sent off already
                                 ; all positions have the following syntax:
                                 ;   ('number-of-place' nil/'number-of-position')
              now                ; actual position
              goal               ; final goal
              next-goal          ; place agent wants to move to next
              wish               ; intention to act
                                 ; 'leaves' has the following syntax:
                                 ;   ('agent' 'leave place' 'don't go to place')
              leaves
                                 ; 'alternatives' has the following syntax:
                                 ;  - a list with length `number-of-places'
                                 ;  - each element is either nil or TRUE
              alternatives
              act-flag           ; agent wants to act
              wait-flag          ; agent does not participate in negotiations
                                 ; auxiliary variables
              reached            ; used by rule set reached
              reachable          ; used by rule set reachable
              new-goal           ; used by rule set plan-alternatives
   )
   (rules
    (rule0
     (equal (caar message) "initialize")
                                 ; syntax of message: ("initialize" agents-number number-of-places
                                 ;                                  goal-place goal-position
     (set agent                  (nth 1 (car message))
          number-of-places       (nth 2 (car message))
          now                    nil
          goal                   (list (nth 3 (car message)) (nth 4 (car message)))
          blocker                nil
```

```
        blocker-sent-off    nil
        next-goal           nil
        wish                nil
        message             nil
        leaves              nil
        alternatives        (do ((i 1 (+ i 1))
                                 (r nil (cons nil r))
                                )
                                ((> i (nth 2 (car message))) r)
                            )
      )
      (stop)
    )
    (rule1
      (equal (caar message) "create next goal")
      (set message (cdr message))
      (call "prepare-planning")
                              ; plan
      (call "test-for-goal")                ; can you reach goal
      (call "plan-leave")                   ; if not, did you get any leave-messages
      (call "plan-heuristically")           ; if not, do anything heuristically
                                            ;  (to enhance the finding of the solution)
    )
    (rule2
      (equal (caar message) "inform other agents")
      (set message (cdr message))
      (call "communicate")
    )
    (rule3
      (equal (caar message) "will to act")
      (set message (cdr message))
      (send world "will-to-act" next-goal)
    )
    (leave-receive
      (equal (caar message) "leave")
                              ; "leave" `place` `priority of sending agent`
      (set message (cdr message))
    )
    (leave-send
      (equal (caar message) "send-leave")
      (set message (cdr message))
    )
    (rule
      (null message)
      (stop)
    )
  )
  (rule-sets
    (rule-set "prepare-planning"
      nil
      (rules
        (initialize-planing
          'TRUE
```

```
   (send world "where-am-i")          ; what is my present place?
   (set now (list (car send) (cadr send)))
   (call test-for-success)            ; did I already reach my next-goal?
   (send world "look-above")          ; check if agent is blocked by a slice
   (set blocker send)
   (set blocker-sent-off (if          ; there is a blocker, a blocker was sent off
                                      ;  and they are the same agent
                     (and blocker blocker-sent-off (= blocker-sent-off blocker))
                     blocker
                     nil
                  ))
   (return)
  )
 )
 (rule-sets
  (rule-set test-for-success
   nil
   (rules
    (rule
                          ; if next-goal was reached then initialize variables
                          ;  (to plan a new next goal)
                          ; it does not matter if position was reached
     (equal (car now) (car next-goal))
     (set blocker-sent-off nil
        next-goal      nil
        leaves         nil
        alternatives    (do ((l alternatives (cdr l))
                        )
                        ((null l) alternatives)
                        (rplaca l nil)
                    ))
    )
    (rule
      'TRUE
     (return)
    )
   )
  )
 )
)
(rule-set "test-for-goal"
 (variables state)                  ; state sequentializes firing of rules !
```

```
  (rule
   (null state)
   (call "reached" (place goal))        ; check if goal was reached
   (call "reachable" (place goal))      ; or is reachble
   (set state t)
  )
  (rule
   (and state reached)
                                        ; stop acting and negotiating
   (set act-flag  nil
      wait-flag t)
  )
```

```
    (rule
     (and state (not reached) reachable)
                                ; try to reach goal
     (set next-goal goal
          act-flag  (if (equal reachable 'directly) t nil)
          wait-flag t)
    )
    (rule
     state
     (return)
    )
   )
)
(rule-set "plan-leave"
 nil
 (rules
  (rule
    'TRUE
   (return)
  )
 )
)
(rule-set "plan-alternatives"
                                ; first determine your options and save them into sets
                                ;  (find-heuristically-best-alternative)
                                ; then select the best (or second best) one (by random)
 (variables sets)               ; elements of set:
                                ;  0 : places that are empty
                                ;  1 : places the agent can go to
                                ;  2 : places the agent could go to if the top agent would flee
                                ;  3 : the flattest place
                                ;  4 : height of flattest place
 (rules
  (rule
    'TRUE
   (set sets (do ((i 0 (1+ i))              ; initialize sets
                  (r nil (cons nil r))
                 )
                 ((>= i 5) r)
             ))
   (call "find-heuristically-best-alternative" (i 1))
   (set sets (progn         ; height of flattest place is no longer needed
              (rplacd (nthcdr 3 sets) nil)
              sets
             ))
   (call "select-heuristically-best-alternative" (set-number (position-if-not #'null sets)))
   (return)
  )
 )
 (rule-sets
  (rule-set "find-heuristically-best-alternative"
                                ; order the alternatives heuristically (into sets)
   (variables i place)
   (rules                       ; a loop counting the places
    (rule
```

```
    (<= i number-of-places)
    (send world "look-at-place" i)
    (set place send)
    (set sets (cond ((not (nth (1- i) alternatives))        ; it is not an alternative
                     sets
                    )
                    ((null place)                           ; it is empty
                     (rplaca (nthcdr 0 sets) (cons i (nth 0 sets)))
                     sets
                    )
                    ((and place                             ; agents has lower priority
                          (< (cadar place) agent))
                     (rplaca (nthcdr 1 sets) (cons (list i (cadar place)) (nth 1 sets)))
                     sets
                    )
                    ((or (and (cdr place)                   ; if top agent flees
                              (>= (cadar place) agent)
                              (< (cadadr place) agent))
                         (and (null (cdr place))
                              (>= (cadar place) agent)))
                     (rplaca (nthcdr 2 sets) (cons (list i (cadar place)) (nth 2 sets)))
                     sets
                    )
                    ((and (cdr place)       ; a smaller place than the present flattest place
                          (>= (cadar place) agent)
                          (>= (cadadr place) agent)
                          (or (null (nth 4 sets)) (<= (length place) (nth 4 sets))))
                     (rplaca (nthcdr 3 sets) (list i))
                     (rplaca (nthcdr 4 sets) (length place))
                     sets
                    )
                    ('TRUE
                     sets
                    )
                )
          i   (1+ i))
    )
    (rule
     (> i number-of-places)
     (return)
    )
   )
)
(rule-set "select-heuristically-best-alternative"
  (variables set-number)
  (rules
    (rule
     (and (or (= set-number 0) (= set-number 3)))       ; choose one
     (call "pick-one" (place (nth (random (length (nth set-number sets)))
                                  (nth set-number sets)))
                (set-no set-number))
    )
    (rule
     (and (or (= set-number 1) (= set-number 2)))
                          ; choose the one with the lowest number of slices
```

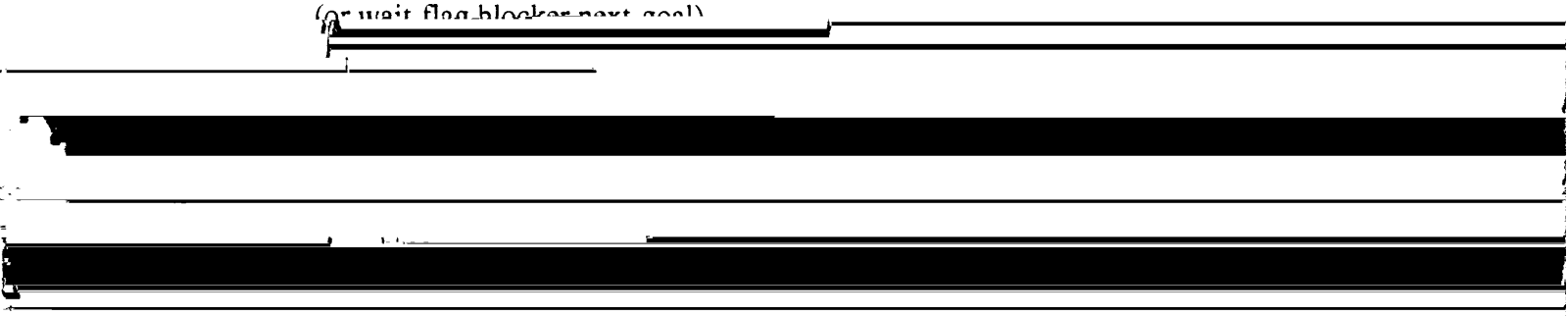
```
        (call "pick-one" (place (car (rassoc (list (apply #'max (mapcar #'cadr
                                                                  (nth set-number sets))))
                                             (nth set-number sets) :test 'equal)))
                         (set-no set-number))
       )
       (rule
         'TRUE
        (return)
       )
      )
      (rule-sets
       (rule-set "pick-one"
        (variables place set-no)
        (rules
         (rule
          place
          (set new-goal (list place nil))
         )
         (rule
           'TRUE
          (return)
         )
        )
       )
      )
     )
    )
  )
  (rule-set "plan-heuristically"
   nil
   (rules
    (rule
      'TRUE
     (return)
    )
   )
  )
  (rule-set "communicate"
   nil
   (rules
    (rule
     (null next-goal)
     (set wait-flag t)
     (break)
    )
    (rule
     (and (not blocker) act-flag)
     (send world "wish-to-act" (car next-goal) (cadr next-goal))
    )
    (rule
      'TRUE
     (return)
    )
   )
  )
)
```

```
(rule-set "reached"
                              ; "reached" sets variable reached to true iff
                              ;  position is unspecified and place has been reached
                              ; or
                              ;  place and position have been reached
 (variables place)
 (rules  (is-place-reached?
          'TRUE
         (set reached (or (and (null (cadr place)) (= (car now) (car place)))
                          (and (cadr place)       (equal now place))))
         (return)
        )
 )
)
(rule-set "reachable"
                              ; "reachable" sets variable reachable to true iff
                              ;  position is unspecified and agent could go on top of the place
                              ;                          or only the top agent would have to flee
                              ; or
                              ;  position is top or one above and the same predicate
                              ;       as above holds true
 (variables place stick)
 (rules  (rule
          (null stick)
          (send world "look-at-place" (car place))
          (set stick (nconc send (list (list 0 0)))
             reachable nil)
         )
         (rule
          (and (cdr stick) (or (null (cadr place))
                               (= (caar stick) (cadr place)))
                           (< (cadadr stick) agent))
          (set reachable 'nondirectly)
         )
         (rule
          (and stick (or (null (cadr place))
                         (= (caar stick) (1- (cadr place))))
                     (< (cadar stick) agent))
          (set reachable 'directly)
         )
         (rule
          stick
          (return)
         )
 )
)
)
)
(list
 '(communicate        (replace-rule        ()
   (leave-receive
    (equal (caar message) "leave")
                              ; "leave" 'place' 'priority of sending agent'
    (set leaves (list (caddar message) (cadddr (car message)) (cadar message))
       message (cdr message))
```

```
  )
                    )
                    (replace-rule       ()
  (leave-send
   (equal (caar message) "send-leave")
   (send world "leave" (cadar message) (caddar message) (cadddr (car message)))
   (set message (cdr message))
  )                 )
 )
'(willing-to-help     (replace-rule-set  ()
  (rule-set "plan-leave"
   (variables state)          ; state sequentializes firing of rules !
   (rules
    (rule
     wait-flag                ; this means next-goal is already planned
     (break)
    )
    (rule
     (not leaves)             ; no reason to leave
     (break)
    )
    (rule
     (and blocker blocker-sent-off)    ; if I am blocked, I cannot move anyhow
                                       ;  and the blocker has already been sent off
     (break)
    )
    (rule
     (null state)                      ; if leave-messages exist then collect your alternatives
                                       ;  and plan next move
     (set alternatives (do ((l alternatives (cdr l))          ; which places are not forbidden
                            (i 1 (1+ i))
                           )
                           ((null l) alternatives)
                           (rplaca l (if (or (= i (cadr leaves)) (= i (caddr leaves))) nil 'TRUE))
                       ))
     (call "plan-alternatives")
     (set state t)
    )
    (rule
     (and state                        ; if we have no next-goal then take new-goal
          (null next-goal))            ; check if it is reachable (then set act-flag)
     (set next-goal new-goal)
     (call "reachable" (place next-goal))
     (set act-flag (if (equal reachable 'directly) t nil))
    )
    (rule
     (and state                        ; decide whether to take new-goal or next-goal
          next-goal
          (not blocker))
     (call "reachable" (place new-goal))
     (set next-goal (if (equal reachable 'directly)          ; take only new-goal if it is
                                                             ; reached directly
                        new-goal
                        next-goal)
         act-flag  (if (equal reachable 'directly) t nil))
```

```
      (call "reachable" (place next-goal))              ; this may be double work
      (set act-flag  (if (equal next-goal new-goal) act-flag
                         (if (equal reachable 'directly) t nil)))
     )
     (rule
      state
      (set wait-flag t)
      (return)
     )
    )
   )
)              )
'(knowledge-of-others     (add-rule    ("communicate")
  (rule
   (and blocker (not blocker-sent-off)
        (or act-flag leaves))
   (set act-flag nil
        message (nconc message (list (list "send-leave" blocker (car next-goal) (car now))))
        blocker-sent-off blocker)
  )                    )
                       (add-rule          ("communicate")
  (rule
   (and (not blocker) (not act-flag))
                       ; why does not agent want to act?
   (send world "look-at-place" (car next-goal))
   (call "inspect-place" (place send))
  )                    )
                       (add-rule-set      ("communicate")
  (rule-set "inspect-place"
   (variables place agent-on-pos agent-under-pos)
   (rules
    (rule              ; position specified and position occupied
     (and (cadr next-goal) (>= (length place) (1- (cadr next-goal))) (not agent-on-pos))
     (set agent-under-pos (if (> (cadr next-goal) 1)
                              (cadr (assoc (1- (cadr next-goal)) place :test '=)))
         agent-on-pos    (cadr (assoc (cadr next-goal) place :test '=)))
    )
    (rule
     (and agent-on-pos (or (= (cadr next-goal) 1) (< agent-under-pos agent)))
                       ; position is specified, send agent at position off,
                       ;  if lower agent could bear sending agent
     (set message (nconc message (list (list "send-leave" agent-on-pos
                                             (car now) (car next-goal)))))
    )
    (rule
     agent-on-pos
     (return)
    )
    (rule
     (null (cadr next-goal))           ; no position specified
                                       ; determine first agent that would bear sending agent
     (set agent-under-pos (do ((l place (cdr l)))
                              ((or (null (cdr l)) (< (cadadr l) agent)) (cadar l))))
    )
    (rule
```

```
      (and (null (cadr next-goal)) agent-under-pos)
                        ; send this agent off
      (set message (nconc message (list (list "send-leave" agent-under-pos
                                              (car now) (car next-goal)))))
      (return)
     )
    )
   )                    )
 )
'(global-knowledge (replace-rule-set  ()
  (rule-set "plan-heuristically"
                        ; the heuristic is to free a slice which has to move before
                        ;  oneself bymoving to the place
                        ; from which the next slice will move (it will leave an empty place)
   (variables scenario      ; which slices are on which place
          first-slice       ; the first slice to move next
          last-slice        ; the last slice (according to lower priority) which
                            ;  could move with first-slice in parallel
          able-to-move      ; is first-slice able to move
          blocking          ; am I blocking one of the slices that should move
   )
   (rules
    (rule
     (or wait-flag-blocker-next-goal)
```



```
    )
    (rule
      'TRUE
     (call "look-at-scenario" (i 1))
     (call "determine-variables")
     (call "find-free-place")
     (return)
    )
   )
   (rule-sets
    (rule-set "look-at-scenario"
                        ; this rule-set determines the actual position of the slices
     (variables i)
     (rules
      (rule
       (<= i number-of-places)
       (send world "look-at-place" i)
       (set scenario (cons send scenario)
           i (1+ i))
      )
      (rule
       (> i number-of-places)
       (set scenario (nreverse scenario))
       (return)
      )
     )
    )
    (rule-set "determine-variables"
     nil
```

```
  (rule
    'TRUE
    (set first-slice  (do ((goal-place (reverse (nth (1- (car goal)) scenario))
                                       (cdr goal-place))
                           (i 1 (1+ i))
                          )
                          ((or (null goal-place)
                               (/= (caar goal-place) (cadar goal-place))) i)
                    ))
    (set able-to-move  (let ((goal-place (nth (1- (car goal)) scenario))
                             (first-place (do ((l scenario (cdr l))
                                               )
                                              ((or (null l) (member first-slice (car l)
                                                                    :test '= :key ,#'cadr))
                                               (car l))
                                          ))
                             )
                             (and (or (and (null goal-place) (= first-slice 1))
                                      (and goal-place (= (1- first-slice) (caar goal-place))))
                                  (= (cadar first-place) first-slice))
                       ))
    (set last-slice   (min (1- agent)
                           (if able-to-move
                               (+ first-slice (- number-of-places 2))
                               (+ first-slice (- number-of-places 3)))))
    (set blocking     (do ((my-place (nth (1- (car now)) scenario) (cdr my-place))
                           (b nil (and (>= (cadar my-place) first-slice)
                                       (<= (cadar my-place) last-slice)))
                           )
                          ((null my-place) b)
                       ))
    (set alternatives  (do ((l alternatives (cdr l))
                            (i 1 (1+ i))
                            )
                           ((null l) alternatives)
                           (rplaca l (if (= i (car goal)) nil      ; set to nil if place is goal place or
                                                                   ; the place of the first-slice
                                         (do ((l (nth (1- i) scenario) (cdr l))
                                              (b nil (or b (= (cadar l) first-slice)))
                                              )
                                             ((null l) b)
                                         ))
                           ))
    (return)
  )
 )
)
(rule-set "find-free-place"
 nil
 (rules
  (rule
   (and able-to-move blocking)
   (call "plan-alternatives")
   (call "reachable" (place new-goal))
```

```
            (set next-goal (if (equal reachable 'directly) new-goal next-goal)
                act-flag  (if (equal reachable 'directly) t act-flag))
           )
           (rule
             'TRUE
             (return)
           )
          )
         )
        )
       )                    )
     )
    '(no-global-position (replace-rule       ()
      (rule0
       (equal (caar message) "initialize")
                            ; syntax of message: ("initialize" agents-number number-of-places
                            ;                          goal-place goal-position
       (set agent                    (nth 1 (car message))
          number-of-places           (nth 2 (car message))
          now                        nil
          goal                       (list (nth 3 (car message)) nil)
          blocker                    nil
          blocker-sent-off           nil
          next-goal                  nil
          wish                       nil
          message                    nil
          leaves                     nil
          alternatives               (do ((i 1 (+ i 1))
                                          (r nil (cons nil r))
                                         )
                                         ((> i (nth 2 (car message))) r)
                                     )
       )
       (stop)
      )                     )
     )
    )
)
```

Anhang B

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Base: 10; Package: SCENARIO -*-

(in-package 'scenario)

; ********************************************
; **  scenario functions for Tower of Hanoi
; ********************************************


; ************************
; **  global variables
; ************************

(defvar *no-success* nil)
(defvar *will-to-act?* nil)                ; does any agent want to act?

(defvar *display-message* nil)             ; predicate, whether a message between agents is
displayed on the screen
(defvar *flip-time-period* 100)            ; how many times a slice is flipped (see flip-slice)

; data structure that holds all information about the status of the scenario
; *scene*
;  number of places
;  number of slices
;  list of slices
;     name
;     initial position          (#place #position)
;     position                  (#place #position)
;     goal                      (#place #position)
;     breadth
;  list of places
;     list of slices
;  act wishes
;     (slice place pos)
(defvar *scene* nil)

; window variables

(defvar *window1*)                         ; scenario display
(defvar *window2*)                         ; system display

(defvar *sc-x1*)                           ; variables for print-scenario
(defvar *sc-y1*)
(defvar *sc-x2*)
(defvar *sc-y2*)
(defvar *distance*)
(defvar *width*)
(defvar *height*)

; statistical variables

(defvar *steps*)                           ; number of planning cycles
(defvar *moves*)                           ; number of moves
(defvar *slice-moves*)                     ; number of moved slices
(defvar *messages*)                        ; number of send messages
```

```
; macros to access *scene*

(defmacro get-scene-number-of-places (scene)
  `(car ,scene)
)

(defmacro get-scene-number-of-slices (scene)
  `(cadr ,scene)
)

(defmacro get-scene-slices (scene)
  `(caddr ,scene)
)

(defmacro get-scene-places (scene)
  `(cadddr ,scene)
)

(defmacro get-scene-act-wish (scene)
  `(cadddr (cdr ,scene))
)

(defmacro set-scene-slices (scene slices)
  `(rplaca (cddr ,scene) ,slices)
)

(defmacro set-scene-places (scene places)
  `(rplaca (cdddr ,scene) ,places)
)

(defmacro add-scene-act-wish (scene wish)
  `(rplaca (cddr (cddr ,scene)) (cons ,wish (cadddr (cddr ,scene))))
)

(defmacro set-scene-act-wish (scene wishes)
  `(rplaca (cddr (cddr ,scene)) ,wishes)
)

(defmacro get-slice-name (slice)
  `(car ,slice)
)

(defmacro get-slice-initial-position (slice)
  `(cadr ,slice)
)

(defmacro get-slice-position (slice)
  `(caddr ,slice)
)

(defmacro set-slice-position (slice position)
  `(rplaca (cddr ,slice) ,position)
)
```

```
(defmacro get-slice-goal (slice)
  `(cadddr ,slice)
)

(defmacro get-slice-breadth (slice)
  `(cadddr (cdr ,slice))
)

; ********************************
; ** initialization functions
; ********************************

(defun initialize-scenario ()
  (setq *scene* nil)
)

(defun reset-scenario ()
  (dolist (l (get-scene-slices *scene*))
        (set-slice-position l (get-slice-initial-position l))
  )
  ; consistent-scenario? also has an effect on *scene*
  (consistent-scenario? (get-scene-slices *scene*))
  (setq *steps*              0
        *moves*              0
        *slice-moves*        0
        *messages*           0)
  ; delete act-wishes
  (set-scene-act-wish *scene* nil)
)

; ********************************
; ** configuration functions
; ********************************

(defun consistent-scenario? (slice-list)
; this function test whether the actual positions of the slices are consistent
; with the scenario constraints
; it also updates the list of places (of *scene*)
  (let ((places-list nil)
        )
      (dotimes (i (get-scene-number-of-places *scene*)) (push nil places-list))
      (do* ((slice slice-list (cdr slice))
            (place (car (cadar slice))
                   (car (cadar slice)))
            (pos (cadr (cadar slice))
                 (cadr (cadar slice)))
            (list-of-pos (nreverse (nth (- place 1) places-list))
                         (if place (nreverse (nth (- place 1) places-list))))
           )
           ((null slice)
           )
           ; create missing positions
           (dotimes (i (- pos (length list-of-pos)))
                    (push nil list-of-pos)
           )
```

```
        (rplaca (nthcdr (- place 1) places-list) (nreverse list-of-pos))
        (rplaca (nthcdr (- pos 1) (nth (- place 1) places-list)) (car slice))
      )
      ; check if places-list is consistent
      (do ((place places-list (cdr place))
           (bool 'TRUE)
          )
          ((null place)
           (if bool
               (set-scene-places *scene* places-list))
           bool
          )
          (do ((pos (car place) (cdr pos))
               (found-nil nil)
              )
              ((null pos)
              )
              (cond ((null (car pos))
                     ; an empty position has been found
                     (setq found-nil 'TRUE)
                    )
                    ((and (car pos) (cadr pos)
                          (> (get-slice-breadth (car pos)) (get-slice-breadth (cadr pos))))
                     ; both positions are occupied and the lower slice is smaller
                     (setq bool nil)
                    )
                    ('TRUE
                     ; there is a slice on that position: if an empty position
                     ; was detected before, the slice has no support
                     (if found-nil
                         (setq bool nil)
                     )
                    )
              )
          )
      )
    )
  )

(defun configure-scenario ()
  (let ((places 0)
        (slices 0)
       )
       (format t "~%  Configuration menue of TOWER of HANOI:~% ------------------------------
--------~%~%")
       (format t "  Number of places: ")
       (setq places (read-from-string (read-line)))
       (format t "~%  Number of slices: ")
       (setq slices (read-from-string (read-line)))
       (format t "~%")
       (when (and (numberp places) (numberp slices))
             (setq *scene* (list places slices nil nil nil)
                   *steps*        0
                   *moves*        0
                   *slice-moves*  0
```

```
      *messages*        0)
    (format t "  Standard scene configuration? ")
    (if (equal (read-line) "y")
      ; ask for starting and goal place
      (do ((start nil)
           (end nil)
          )
          ((and (numberp start) (numberp end)
                (> start 0) (> end 0) (<= start places) (<= end places))
           ; update slice slots
           (do* ((i 1 (+ i 1))
                 (agent-list nil)
                 (slice-list nil)
                 (name "slice1" (format nil "slice~A" i))
                )
                ((> i slices)
                 (set-scene-slices *scene* slice-list)
                 (consistent-scenario? slice-list)
                 (setq semaw:*agents* (nreverse agent-list))
                )
                (push (list name (list start i) (list start i) (list end i) i) slice-list)
                (push (list name nil (car slice-list)) agent-list)
           )
          )
          (format t "~%  Configurating two towers:~%  Starting from place ")
          (setq start (read-from-string (read-line)))
          (format t "  to place ")
          (setq end (read-from-string (read-line)))
          (cond ((not (and (numberp start) (numberp end)))
                 (semaw:error-message "  Input not numbers. Please try again.~%")
                )
                ((not (and (> start 0) (> end 0) (<= start places) (<= end places)))
                 (semaw:error-message "  Numbers out of range. Please try again.~%")
                )
          )
      )
      ; ask for every slice position
      (do* ((i 1 (+ i 1))
            (agent-list nil)
            (slice-list nil)
            (name "slice1" (format nil "slice~A" i))
            (start-place nil)
            (start-pos nil)
            (goal-place nil)
            (goal-pos nil)
           )
           ; check if configuration is consistent: every slice must
           ; be supported by ground or another larger slice
           ((and (> i slices) (consistent-scenario? slice-list))
            (set-scene-slices *scene* slice-list)
            (setq semaw:*agents* (nreverse agent-list))
           )
           (when (> i slices)
                 (semaw:error-message "  The configuration was inconsistent. Please try
again.~%")
```

```
                    (setq i 1
                        agent-list nil
                        slice-list nil
                        name "slice1"
                    )
                )
                (format t "~%  Slice ~A start from place  " name)
                (setq start-place (read-from-string (read-line)))
                (format t  "                    at position ")
                (setq start-pos (read-from-string (read-line)))
                (format t "        with goal at place       ")
                (setq goal-place (read-from-string (read-line)))
                (format t "                    at position ")
                (setq goal-pos (read-from-string (read-line)))
                (format t "~%")
                (cond ((not (and (numberp start-place) (numberp start-pos)
                            (numberp goal-place) (numberp goal-pos)))
                    (semaw:error-message "  Input not numbers. Please try again.~%")
                    (setq i (- i 1))
                    )
                    ((not (and (> start-place 0) (> goal-place 0)
                            (<= start-place slices) (<= goal-place slices)
                            (> start-pos 0) (> goal-pos 0)))
                    (semaw:error-message "  Numbers out of range. Please try again.~%")
                    (setq i (- i 1))
                    )
                    ('TRUE
                    (push (list name (list start-place start-pos) (list start-place start-pos)
                            (list goal-place goal-pos) i) slice-list)
                    (push (list name nil (car slice-list)) agent-list)
                    )
                )
            )
        )
    )
 )
)

; **********************
; ** print functions
; **********************

(defun print-scenario-configuration (slice &optional flag)
 (if flag
    (format t "    Slice has initial position ~A,~%            current position ~A and~%
goal position ~A.~%"
            (get-slice-initial-position slice)
            (get-slice-position slice)
            (get-slice-goal slice))
    (format t "    Slice has initial position ~A,~%              goal position ~A.~%"
            (get-slice-initial-position slice)
            (get-slice-goal slice))
 )
 (format t "    Slice has priority ~A.~%"
        (get-slice-breadth slice))
```

```
)

(defun print-scenario ()
 (setq *window1* semaw:*scenario-display*
     *window2* semaw:*system-display*)
 (scl:send *window1* :clear-region 0 0 semaw:*scenario-window-width*
                                       semaw:*scenario-window-height*)
 (scl:send *window1* :set-mouse-position 0 0)
 (scl:send *window2* :clear-region 0 0 semaw:*system-window-width*
                                       semaw:*system-window-height*)
 (scl:send *window2* :set-mouse-position 0 0)

 (graphics:draw-string "Tower of Hanoi" 0 semaw:*scenario-window-height*
                       :stream *window1*)

 (graphics:draw-string (format nil "Number of Places: ~A"
                                   (get-scene-number-of-places *scene*))
               5 semaw:*system-display-offset*  :stream *window2*)
 (graphics:draw-string (format nil "Number of Slices: ~A"
                                   (get-scene-number-of-slices *scene*))
               5 (+ semaw:*system-display-offset* 12) :stream *window2*)
 (graphics:draw-string (format nil "Number of satisfied slices:    ~A" 0)
               5 (+ semaw:*system-display-offset* 30) :stream *window2*)
 (graphics:draw-string (format nil "Number of planning cycles:     ~A" 0)
               5 (+ semaw:*system-display-offset* 42) :stream *window2*)
 (graphics:draw-string (format nil "Number of action steps:        ~A" 0)
               5 (+ semaw:*system-display-offset* 54) :stream *window2*)
 (graphics:draw-string (format nil "Number of moved slices:        ~A" 0)
               5 (+ semaw:*system-display-offset* 66) :stream *window2*)
 (graphics:draw-string (format nil "Number of messages:            ~A" 0)
               5 (+ semaw:*system-display-offset* 78) :stream *window2*)

 ; draw the scenario
 (setq *sc-x1* 10
     *sc-y1* 20
     *sc-x2* (- semaw:*scenario-window-width* 10)
     *sc-y2* (- semaw:*scenario-window-height* 25)
     *distance* (floor (/ (1+ (- *sc-x2* *sc-x1*)) (get-scene-number-of-places *scene*)))
     *width*   (/ *distance* (get-scene-number-of-slices *scene*) 2)
     *height*  (/ (1+ (- *sc-y2* *sc-y1*)) (get-scene-number-of-slices *scene*)))
 (setq *window1* *window1*)
 (setq *window2* *window2*)
 (graphics:draw-line *sc-x1* *sc-y2* *sc-x2* *sc-y2* :stream *window1*)
 (dotimes (i (get-scene-number-of-places *scene*))
       (graphics:draw-line (+ *sc-x1* (floor (/ *distance* 2)) (* i *distance*)) *sc-y1*
                   (+ *sc-x1* (floor (/ *distance* 2)) (* i *distance*)) *sc-y2*
                   :stream *window1*)
 )
 ; draw slices
 (do ((l (get-scene-places *scene*) (cdr l))
    (i 0 (1+ i))
   )
   ((null l) )
   (do ((k (car l) (cdr k))
       (j 0 (1+ j))
```

```
      )
      ((null k) )
      (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                  (floor (* (1- (get-slice-breadth (car k))) *width*)))
                               (- *sc-y2* (floor (* j *height*)))
                               (- (+ *sc-x1* (* (1+ i) *distance*))
                                  (floor (* (1- (get-slice-breadth (car k))) *width*)))
                               (- *sc-y2* (floor (* (1+ j) *height*)))
                               :stream *window1*)
   )
 )
)

; ****************************
; ** simulation functions
; ****************************

(defun perform-parallel-move (act-list)
 ; this function shows a parallel move on the screen and
 ;  updates *scene*
 ; delete slice from places-list
 (do ((l (get-scene-places *scene*) (cdr l))
   )
   ((null l) )
   (when (assoc (car (last (car l))) act-list)  ; select only slices which are at the top
      (rplaca l (nreverse (cdr (nreverse (car l)))))
   )
 )
 ; change scenario-display: move slices upwards
 (dolist (l act-list)
      (let ((i (1- (car (get-slice-position (car l)))))
          (j (1- (cadr (get-slice-position (car l)))))
         )
         (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                     (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                  (- *sc-y2* (floor (* j *height*)))
                                  (- (+ *sc-x1* (* (1+ i) *distance*))
                                     (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                  (- *sc-y2* (floor (* (1+ j) *height*)))
                                  :stream *window1* :alu :flip)
         (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                     (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                  *height*
                                  (- (+ *sc-x1* (* (1+ i) *distance*))
                                     (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                  0
                                  :stream *window1* :alu :draw)
      )
 )
 (dolist (l act-list)
      (let ((i (1- (car (get-slice-position (car l)))))
         )
         (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                     (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                  *height*
```

```
                        (- (+ *sc-x1* (* (1+ i) *distance*))
                           (floor (* (1- (get-slice-breadth (car l))) *width*)))
                        0
                        :stream *window1* :alu :flip)
      )
  )
  ; determine new place
  (dolist (l act-list)
      (rplaca (nthcdr (1- (cadr l)) (get-scene-places *scene*))
          (nreverse (cons (car l) (nreverse (nth (1- (cadr l)) (get-scene-places *scene*))))))
      (set-slice-position (car l) (list (cadr l) (length (nth (1- (cadr l))
                                                              (get-scene-places *scene*)))))
  )
  ; move to new place
  (dolist (l act-list)
      (let ((i (1- (car (get-slice-position (car l))))))
          )
          (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   *height*
                                   (- (+ *sc-x1* (* (1+ i) *distance*))
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   0
                                   :stream *window1* :alu :draw)
      )
  )
  ; move slices downwards
  (dolist (l act-list)
      (let ((i (1- (car (get-slice-position (car l)))))
            (j (1- (cadr (get-slice-position (car l))))))
          )
          (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   *height*
                                   (- (+ *sc-x1* (* (1+ i) *distance*))
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   0
                                   :stream *window1* :alu :flip)
          (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   (- *sc-y2* (floor (* j *height*)))
                                   (- (+ *sc-x1* (* (1+ i) *distance*))
                                      (floor (* (1- (get-slice-breadth (car l))) *width*)))
                                   (- *sc-y2* (floor (* (1+ j) *height*)))
                                   :stream *window1* :alu :draw)
      )
  )
  ; redraw the sticks (of the places)
  (dotimes (i (get-scene-number-of-places *scene*))
      (graphics:draw-line (+ *sc-x1* (floor (/ *distance* 2)) (* i *distance*)) *sc-y1*
                          (+ *sc-x1* (floor (/ *distance* 2)) (* i *distance*)) *sc-y2*
                          :stream *window1*)
  )
)
```

```
(defun update-system-display ()
 (do ((l (get-scene-slices *scene*) (cdr l))
    (satisfied 0 (if (equal (get-slice-position (car l)) (get-slice-goal (car l)))
                (1+ satisfied)
                satisfied
        ))
    )
    ((null l)
    (scl:send *window2* :clear-region 240 semaw:*system-display-offset*
                        semaw:*scenario-window-width* semaw:*scenario-window-height*)
    (graphics:draw-string (format nil "Number of satisfied slices:    ~A" satisfied)
                5 (+ semaw:*system-display-offset* 30) :stream *window2*)
    (graphics:draw-string (format nil "Number of planning cycles:     ~A" *steps*)
                5 (+ semaw:*system-display-offset* 42) :stream *window2*)
    (graphics:draw-string (format nil "Number of action steps:        ~A" *moves*)
                5 (+ semaw:*system-display-offset* 54) :stream *window2*)
    (graphics:draw-string (format nil "Number of moved slices:        ~A" *slice-moves*)
                5 (+ semaw:*system-display-offset* 66) :stream *window2*)
    (graphics:draw-string (format nil "Number of messages:            ~A" *messages*)
                5 (+ semaw:*system-display-offset* 78) :stream *window2*)
    )
 )
)

(defun perform-moves ()
 ; this function makes the moves of the slices
 ;  it determines (in cycles) a group of slices which may move in parallel
 ; WARNING: this function sometimes works with equality of cons-cells (lists
 ;        are copied, but the members stay the same!)
 (let ((act-list (get-scene-act-wish *scene*))
      (wait-list nil)
     )
     ; slices may have a cyclic dependancy: one may move if the other one moves and vice versa
     ; thus the set of moving slices is repeatedly checked
     (loop
      ; find all slices for one parallel move
      (do* ((l (copy-list act-list) (cdr l))
          (place (unless (null l) (reverse (nth (1- (cadar l)) (get-scene-places *scene*))))
               (unless (null l) (reverse (nth (1- (cadar l)) (get-scene-places *scene*)))))
          (b nil)  ; boolean variable tells whether a conflict has occured
          )
          ((and (null l) (if (and b act-list)
                        (progn
                         ; start anew to find a slice which cannot act
                         (setq l (copy-list act-list)
                              place (unless (null l) (reverse (nth (1- (cadar l))
                                                                   (get-scene-places *scene*))))
                              b nil)
                         nil
                        )
                        'TRUE
                    ))
          )
          ; check whether slice could possibly move
          (cond (; is slice (car l) still willing to act?
```

```
 (member (car l) wait-list)
 )
 (; is slice blocked?
 (let ((pos (get-slice-position (caar l)))
      )
     (get-slice-breadth (nth (cadr pos) (nth (1- (car pos))
                                             (get-scene-places *scene*))))
 )
 ; then forget it
 (setq act-list (delete (car l) act-list))
  (setq b 'TRUE)
 )
 (; can slice move to place?
 (and (null (caddar l))
     (or (and (cdr place) (< (get-slice-breadth (caar l))
                             (get-slice-breadth (cadr place))))
        (and place (< (get-slice-breadth (caar l)) (get-slice-breadth (car place)))
             (not (assoc (car place) act-list)))))
 (setq act-list (delete (car l) act-list))
 (push (car l) wait-list)
  (setq b 'TRUE)
 )
 (; can slice move to place at specified position?
 (and (caddar l)
     (not (and (= (length place) (caddar l))
          (or (null (cdr place)) (> (get-slice-breadth (caar l))
                                    (get-slice-breadth (cadr place))))
          (assoc (car place) act-list)))
     (not (and (= (length place) (1- (caddar l)))
          (or (null place) (> (get-slice-breadth (caar l))
                              (get-slice-breadth (car place))))
          (not (assoc (car place) act-list)))))
 (setq act-list (delete (car l) act-list))
 (push (car l) wait-list)
  (setq b 'TRUE)
 )
 ('TRUE
 ; is there another slice which wants to go to the same place?
 ; slice (car l) has to be compared only with those who have been checked sofar
 (dolist (k (copy-list act-list))
       (if (eq k (car l))
          (return)
       )
       (when (= (cadr k) (cadar l))
          (setq b 'TRUE)  ; remark that we have to start anew
          ; delete one of the slices
          (cond ((and (caddr k) (caddar l))
               (if (< (get-slice-breadth (car k)) (get-slice-breadth (caar l)))
                  (progn
                   (push (car l) wait-list)
                   (setq act-list (delete (car l) act-list))
                  )
                  (progn
                   (push k wait-list)
                   (setq act-list (delete k act-list))
```

```
                                    )
                                  )
                                 )
                                 ((caddr k)
                                  (push (car l) wait-list)
                                 (setq act-list (delete (car l) act-list))
                                 )
                                 ((caddar l)
                                  (push k wait-list)
                                 (setq act-list (delete k act-list))
                                 )
                                 ('TRUE
                                 (if (< (get-slice-breadth (car k)) (get-slice-breadth (caar l)))
                                     (progn
                                      (push (car l) wait-list)
                                      (setq act-list (delete (car l) act-list))
                                     )
                                     (progn
                                      (push k wait-list)
                                      (setq act-list (delete k act-list))
                                     )
                                  )
                                 )
                            )
                       )
                  )
                 )
             )
         )
        ; if act-list is empty nobody can act and we are done
        (if (null act-list)
           (return)
        )
        (setq *moves* (1+ *moves*)
            *slice-moves* (+ *slice-moves* (length act-list)))
        ; act-list contains slices which may act in parallel
        (perform-parallel-move act-list)
        ; otherwise start once more with the slices which haven't acted sofar
        (setq act-list wait-list
            wait-list nil)
      )
  )
  (update-system-display)
  ; delete act-wishes
  (set-scene-act-wish *scene* nil)
)

; the functions set-pattern, flip-slice and print-sentence are used when
; *display-message* is set

(defun set-pattern (slice set)
; this function changes the pattern of a slice
  (let ((i (1- (car (get-slice-position slice))))
        (j (1- (cadr (get-slice-position slice))))
        (bitmap (tv:make-binary-gray 8 8 '(#b10000001
```

```
                                #b01000010
                                #b00100100
                                #b00011000
                                #b00011000
                                #b00100100
                                #b01000010
                                #b10000001)))
      )
    (if set
      (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                  (floor (* (1- (get-slice-breadth slice)) *width*)))
                               (- *sc-y2* (floor (* j *height*)))
                               (- (+ *sc-x1* (* (1+ i) *distance*))
                                  (floor (* (1- (get-slice-breadth slice)) *width*)))
                               (- *sc-y2* (floor (* (1+ j) *height*)))
                               :stream *window1*
                               :pattern bitmap
      )
      (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                  (floor (* (1- (get-slice-breadth slice)) *width*)))
                               (- *sc-y2* (floor (* j *height*)))
                               (- (+ *sc-x1* (* (1+ i) *distance*))
                                  (floor (* (1- (get-slice-breadth slice)) *width*)))
                               (- *sc-y2* (floor (* (1+ j) *height*)))
                               :stream *window1*
                               :alu :draw
      )
    )
  )
)

(defun flip-slice (slice)
; this function makes the slice flip its pattern
  (let ((i (1- (car (get-slice-position slice))))
        (j (1- (cadr (get-slice-position slice))))
       )
       (dotimes (n (* 2 *flip-time-period*))
        (graphics:draw-rectangle (+ *sc-x1* (* i *distance*)
                                    (floor (* (1- (get-slice-breadth slice)) *width*)))
                                 (- *sc-y2* (floor (* j *height*)))
                                 (- (+ *sc-x1* (* (1+ i) *distance*))
                                    (floor (* (1- (get-slice-breadth slice)) *width*)))
                                 (- *sc-y2* (floor (* (1+ j) *height*)))
                                 :stream *window1* :alu :flip
                                 :pattern t
        )
       )
  )
)

(defun print-sentence (addressed-agent sending-agent text)
; displays a string, makes the slices flip its pattern, and deletes the string
  (set-pattern addressed-agent t)
  (let ((i (1- (car (get-slice-position sending-agent))))
       )
```

```
    (graphics:draw-string text (* i *distance*) 10 :stream *window1*)
  )
  (flip-slice sending-agent)
  (let ((i (1- (car (get-slice-position sending-agent))))
      )
      (graphics:draw-string (do ((i 1 (1+ i)) (s "" (concatenate 'string " " s)))
                                ((> i (length text)) s))
                    (* i *distance*) 10 :stream *window1*)
  )
  (set-pattern addressed-agent nil)
)

(defun send-world (sending-agent args)
  (cond
      ((equal (car args) "leave")
       ; "leave" `addressed agent' `place to leave' 'your own place'
       (when *display-message*
           (print-sentence (caddr (nth (1- (cadr args)) semaw:*agents*))
                (assoc (semaw:agent-name sending-agent) (get-scene-slices *scene*) :test 'equal)
                (format nil "Hey, leave your place and don't go to place ~A!" (caddr args))
           )
       )
       ; send message to the addressed agent
       (semaw:send-agent (cadr (nth (1- (cadr args)) semaw:*agents*))
                nil
                (list "leave" (caddr args)
                             (get-slice-breadth (assoc (semaw:agent-name sending-agent)
                                                       (get-scene-slices *scene*) :test 'equal))
                             (cadddr args)
                ))
       (setq *messages* (1+ *messages*))
       nil
      )
      ((equal (car args) "wish-to-act")
       ; store act wish
       (let ((entry (assoc (assoc (semaw:agent-name sending-agent)
                                  (get-scene-slices *scene*) :test 'equal)
                           (get-scene-act-wish *scene*)))
          )
          (if entry
             (rplacd entry (list (cadr args) (caddr args)))
             (add-scene-act-wish *scene*
                      (list (assoc (semaw:agent-name sending-agent)
                                   (get-scene-slices *scene*) :test 'equal)
                            (cadr args)
                            (caddr args)))
          )
       )
       nil
      )
      ((equal (car args) "where-am-i")
       (get-slice-position (assoc (semaw:agent-name sending-agent)
                                  (get-scene-slices *scene*) :test 'equal))
      )
      ((equal (car args) "look-at-place")
```

```
      (do ((place (nth (1- (cadr args)) (get-scene-places *scene*)) (cdr place))
           (i 1 (+ i 1))
           (r nil (cons (list i (get-slice-breadth (car place))) r))
          )
          ((null place)
           r
          )
      )
      )
      ((equal (car args) "look-above")
       (let ((pos (get-slice-position (assoc (semaw:agent-name sending-agent)
                                             (get-scene-slices *scene*) :test 'equal)))
          )
          (get-slice-breadth (nth (cadr pos) (nth (1- (car pos)) (get-scene-places *scene*))))
       )
      )
      ((equal (car args) "will-to-act")
       (if (cadr args)
          (setq *will-to-act?* t)
       )
      )
 )
)

(defun run-scenario ()
 (dolist (agent semaw:*agents*)
       (semaw:send-agent (cadr agent) nil
                  (list "initialize" (get-slice-breadth (caddr agent))
                                                  (get-scene-number-of-places *scene*)
                              (car (get-slice-goal (caddr agent)))
                                                  (cadr (get-slice-goal (caddr agent)))
                  ))
 )
 (semaw:run-interpreter)

 ; simulation is run in cycles
 ;  create next goal
 ;  inform other agents
 ;  test if agent wants to act
 ;  perform moves
 ;  test on success
 (loop
  (dolist (agent semaw:*agents*)
       (semaw:send-agent (cadr agent) nil
                  (list "create next goal")))
  (semaw:run-interpreter)

  (setq *steps* (1+ *steps*))

  (dolist (agent semaw:*agents*)
       (semaw:send-agent (cadr agent) nil
                  (list "inform other agents")))
  (semaw:run-interpreter)

  (dolist (agent semaw:*agents*)
```

```
        (semaw:send-agent (cadr agent) nil
                (list "will to act")))
    (semaw:run-interpreter)

    ; if nobody wants to act, we are done
    (unless *will-to-act?*
        (return)
    )

    (perform-moves)

    (setq *no-success* nil)
    (dolist (l (get-scene-slices *scene*))
        (if (not (equal (get-slice-position l) (get-slice-goal l)))
            (setq *no-success* 'TRUE)
        )
    )
    (unless *no-success*
        (return)
    )
  )
)
```

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters: The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka: Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos: The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein: A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder: Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger: Integrated Plan Generation and Recognition - A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer: ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit: Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne: Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne: Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz: Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz: Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström: On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-01
Werner Nutt: Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

RR-92-02
Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg: Π_{ODA}: The Paper Interface to ODA
53 pages

RR-92-03
Harold Boley: Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark: Feature based Integration of CAD and CAPP
19 pages

RR-92-07
Michael Beetz: Decision-theoretic Transformational Planning
22 pages

RR-92-08
Gabriele Merziger: Approaches to Abductive Reasoning - An Overview -
46 pages

RR-92-09
Winfried Graf, Markus A. Thies: Perspektiven zur Kombination von automatischem Animationsdesign und planbasierter Hilfe
15 Seiten

RR-92-11
Susane Biundo, Dietmar Dengler, Jana Koehler:
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

RR-92-13
Markus A. Thies, Frank Berger:
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

RR-92-14
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   Markus A. Thies, Frank Berger

22 pages

RR-92-15
Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-17
Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18
John Nerbonne: Constraint-Based Semantics
21 pages

DFKI Technical Memos

TM-91-01
Jana Köhler: Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt:
Clamping, COKAM, KADS, and OMOS:
The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley (Ed.):
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel , Rainer Bleisinger:
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

TM-91-06
Johannes Stein: Aspects of Cooperating Agents
22 pages

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Bussmann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang:
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

DFKI Documents

D-91-01
Werner Stein , Michael Sintek: Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-02
Jörg P. Müller: Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining
127 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause: RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp: Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 Seiten

D-91-09
David Powers, Lary Reeker (Eds.):
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS: Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

D-91-15
Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, Frank Steinle:
µCAD2NC: A Declarative Lathe-Worplanning Model Transforming CAD-like Geometries into Abstract NC Programs
100 pages

D-91-16
Jörg Thoben, Franz Schmalhofer, Thomas Reinartz:
Wiederholungs-, Varianten- und Neuplanung bei der Fertigung rotationssymmetrischer Drehteile
134 Seiten

D-91-17
Andreas Becker:
Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Abeitsplanung
86 Seiten

D-91-18
Thomas Reinartz: Definition von Problemklassen im Maschinenbau als eine Begriffsbildungsaufgabe
107 Seiten

D-91-19
Peter Wazinski: Objektlokalisation in graphischen Darstellungen
110 Seiten

D-92-01
Stefan Bussmann: Simulation Environment for Multi-Agent Worlds - Benutzeranleitung
50 Seiten

Simulation Environment for Multi-Agent Worlds
- Benutzeranleitung

Stefan Bussmann

D-92-01
Document