AUTOMATISCHES PLANEN

ANDREAS VOGELSANG

Zusammenfassung. In dieser Ausarbeitung werden die Grundlagen für automatisches Planen vorgestellt. Dabei wird zunächst eine allgemeine Sprache zur Beschreibung von Planungsproblemen vorgestellt. Es werden Algorithmen vorgestellt, die sowohl in deterministischen als auch in bestimmten nicht deterministischen Domänen effizient Pläne finden. Dabei wird ein besonderes Augenmerk auf das Planen in Hierarchical Task Networks (HTN) gelegt. Zuletzt wird eine Möglichkeit vorgestellt, Symbolic Model Checking im HTN-Planen einzusetzen.

1. Motivation

Die Fähigkeit, ein bestimmtes Ziel automatisch und selbstständig zu erreichen, ist sicherlich ein zentraler Punkt bei der Überlegung, was unter künstlicher Intelligenz zu verstehen ist und was diese leisten soll. Nach [8] wird diese Aufgabe, eine Folge von Aktionen zu finden, die ein Ziel erreichen, als *Planen*, (oder *Planung*) bezeichnet. Programme, die Algorithmen zum Planen und Durchführen von Aufgaben nutzen, werden im Bereich der künstlichen Intelligenz *Agenten* genannt.
Vor allem bei größeren Problemdomänen ist man gezwungen, effiziente und effektive Planungsstrategien anzuwenden. Dieses ist leicht an folgendem kleinen Beispiel zu erkennen: Ein sehr simpler und stupider Planungsansatz ist es, einfach alle möglichen Aktionen, die in einem Zustand möglich sind, auszuführen und den Folgezustand mit dem Zielzustand zu vergleichen. Angenommen, es gäbe nur eine Aktion *Kaufen(id)*, mit der wir ein Produkt mit einer zehnstelligen Artikelnummer *id* erwerben können. Wenn unser Ziel ist, das Produkt mit der Artikelnummer *1234567890* zu erwerben, dann würde der naive Algorithmus zur Planung die Aktion *Kaufen(id)* für alle zehnstelligen Artikelnummern ausführen und im Folgezustand überprüfen, ob das Produkt mit der Artikelnummer *1234567890* nach Ausführung der Aktion erworben wurde. Der Algorithmus muss im schlimmsten Fall 10^{10}, also 10 Milliarden Aktionen simulieren, um festzustellen, welche Aktion zu dem gewünschten Ziel führt. Ein intelligenter Algorithmus sollte allerdings in der Lage sein, aus dem Ziel *Haben(1234567890)* direkt rückwärts auf die Aktion *Haben(1234567890)* zu schließen. Für diesen Schritt muss der Algorithmus lediglich die allgemeine Regel *Kaufen(id) führt zu Haben(id)* wissen und berücksichtigen.
Wie zu sehen ist, treten schon bei sehr kleinen Systemen, wie dem oben vorgestellten mit nur zwei Zuständen, diverse Schwierigkeiten bei der Planung auf. In Problemen der realen Welt haben Systeme oft viele Zustände und Ziele setzen sich zumeist aus vielen Einzelteilen zusammen. Ein Planungsalgorithmus braucht deshalb eine gute *Heuristikfunktion*, die ihm Auskunft darüber gibt, wie erfolgversprechend ein bestimmter Plan ist. Wenn in dem Beispiel das Ziel verfolgt werden soll, vier Produkte zu kaufen, dann gibt es schon 10^{40} mögliche Aktionsfolgen bei nur vier Schritten.

Ohne eine geeignete Heuristikfunktion wird es einem Planungsalgorithmus nicht möglich sein, effizient einen Plan zu berechnen. Eine mögliche Heuristikfunktion für unser System könnte hier zum Beispiel die Anzahl der gekauften Produkte sein. Wie wir gesehen haben, können Planungsprobleme sehr schnell, sehr große Zustandsräume aufspannen. Eine in der Informatik sehr verbreitete Technik zur Verkleinerung von Zustandsräumen ist die *Dekomposition*. Ein Planungsalgorithmus sollte in der Lage sein, ein Ziel in Teilziele zu zerlegen und Lösungen für diese auf kleineren Zuständsräumen zu berechnen. Anschließend werden diese Teillösungen zu einem Gesamtplan zusammengeführt. Die Anzahl möglicher Pläne in einem System mit n Aktionen lässt sich so, durch Dekomposition, von $O(n!)$ auf $O((n/k)! \times k)$ verkleinern, wenn das System in k gleiche Teile zerlegt werden kann.

2. Planen

Wie in der Motivation angedeutet, spielt die logische Struktur eines Problems, repräsentiert durch *Zustände*, *Aktionen* und *Ziele* eine große Rolle bei der Entwicklung effizienter Planungsalgorithmen. Auf der anderen Seite sollte ein Planungsalgorithmus für eine möglichst große Familie von Problemen Lösungen finden können. In diesem Kapitel soll daher eine Sprache vorgestellt werden, die ausdrucksstark genug ist, um eine Vielfalt an Problemen zu beschreiben, aber einschränkend genug ist, um effiziente Algorithmen zuzulassen. Die vorgestellte Sprache ist auch als *STRIPS* [1] bekannt [3].

2.1. *Planungsprobleme*

Wir werdem im Folgenden ein Planungsproblem repräsentieren als ein Tupel $P = (L, S, A, G)$, wobei L eine Menge erlaubter Literale, S eine Menge von *Zuständen*, A eine Menge von *Aktionen* (oder *Aktionsschemata*) und G eine Menge von *Zielen* darstellt.
In unserer Planungssprache drücken wir Fakten bzw. Aussagen über unser System mittels **funktionsfreier Grundliterale** aus. Insbesondere benutzen wir nur positive, aussagenlogische Literale erster Stufe. Erlaubt wäre z.B. $l = Arm(p)$, oder $l = Person(p)$, wohingegen $l = Arm(Vater(p))$ nicht erlaubt wäre.

Ein **Zustand** ist nun eine Konjunktion aus den erlaubten Literalen der Menge L. Durch die Funktionsfreiheit der Literale ist sichergestellt, dass wir einen endlichen Zustandsraum erhalten[2]. Außerdem gilt die *Annahme der geschlossenen Welt*, d.h. alle Bedingungen, die in einem Zustand nicht erwähnt werden, werden als falsch angenommen.

Ein **Ziel** ist ein partiell spezifizierter Zustand, der ebenfalls als Konjunktion von erlaubten Literalen dargestellt wird. Ein Zustand $s \in S$ erfüllt ein Ziel $g \in G$ genau

[1] STanford Research Institute Problem Solver

[2] Das ist nicht sofort ersichtlich. Ich werde aber im Abschnitt über die **Semantik** nochmal darauf zurückkommen.

dann, wenn gilt:

$$\forall l.(l \in g \rightarrow l \in s)$$

Ein Zustand also, der alle Literale aus dem Ziel enthält, erfüllt das Ziel.

Eine **Aktion** besteht im Allgemeinen aus drei Teilen:

- Der *Aktionsname* und eine Parameterliste (z.B. $Fliegen(p, von, nach)$)
- Die *Vorbedingung* ist eine Konjunktion aus erlaubten Literalen, die angibt, ob eine Aktion ausgeführt werden kann. Alle Variablen der Vorbedingung müssen auch in der Parameterliste vorkommen.
- Der *Effekt* ist eine Konjunktion erlaubter Literale, die den Zustand charakterisieren, der nach Ausführung der Aktion erreicht wird. Dabei werden positive Literale dem aktuellen Zustand hinzugefügt und negative Literale aus dem aktuellen Zustand gelöscht. Variablen des Effekts müssen ebenfalls in der Parameterliste vorkommen.

Bei der Ausführung müssen alle Variablen einer Aktion durch konkrete Konstanten aus dem aktuellen Zustand instanziiert werden.

Nachdem nun die Syntax eingeführt wurde, möchte ich erläutern, wie die **Semantik** einer Aktionsausführung definiert ist. Wir sagen, eine Aktion ist *anwendbar* in einem Zustand, wenn dieser die Vorbedingungen erfüllt. Andernfalls hat die Aktion keinen Effekt. Die Anwendbarkeit einer Regel erfordert also eine Substitution θ für die Variablen der Vorbedingung der Aktion durch Konstanten des Zustands. Die Ausführung einer Aktion a im Zustand s hat als Ergebnis einen Zustand s' für den gilt:

$$s' = (s \backslash \{p | \neg p \in \mathit{effect}(a)\}) \cup \{q | q \in \mathit{effect}(a)\}$$

Beispiel: Sei

$$\begin{aligned} s = & \mathit{Flugzeug}(P_1) \wedge \mathit{Flugzeug}(P_2) \\ & \wedge \mathit{Bei}(P_1, JFK) \wedge \mathit{Bei}(P_2, SFO) \\ & \wedge \mathit{Flughafen}(JFK) \wedge \mathit{Flughafen}(SFO) \end{aligned}$$

Sei außerdem

$$\begin{aligned} a = \mathit{Aktion}(&\mathit{Fliegen}(p, von, nach), \\ & \mathit{PRECOND} : \mathit{Bei}(p, von) \wedge \mathit{Flugzeug}(p) \wedge \\ & \mathit{Flughafen}(von) \wedge \mathit{Flughafen}(nach) \\ & \mathit{EFFECT} : \neg \mathit{Bei}(p, von) \wedge \mathit{Bei}(p, nach)) \end{aligned}$$

Der Zustand s erfüllt die Vorbedingung der Aktion a, da eine Substitution $\theta = \{p/P_1, von/JFK, nach/SFO\}$ existiert. Wird die Aktion a auf den Zustand s angewendet, ergibt sich der Folgezustand s' mit:

$$\begin{aligned} s' = & \mathit{Flugzeug}(P_1) \wedge \mathit{Flugzeug}(P_2) \\ & \wedge \mathit{Bei}(P_1, SFO) \wedge \mathit{Bei}(P_2, SFO) \\ & \wedge \mathit{Flughafen}(JFK) \wedge \mathit{Flughafen}(SFO) \end{aligned}$$

Ich komme an dieser Stelle nochmal auf die Überlegung zurück, warum die Funktionsfreiheit der Literale verhindert, dass der Zustandsraum unendlich groß wird. Um eine passende Substitution zu finden, ersetzt ein Planungsalgorithmus die Variablen einer Aktion durch alle in einem Zustand vorkommenden Konstanten. Die Aktion $\mathit{Fliegen}(p, \mathit{von}, \mathit{nach})$ würde beispielsweise bei zehn Flugzeugen und fünf Flughäfen in $10 \times 5 \times 5 = 250$ Regeln umgewandelt werden. Wären Funktionssymbole in den Zuständen erlaubt, könnten unendlich viele Terme erzeugt werden.

Eine **Lösung** eines Planungsproblems ist nun ein Pfad, also eine Folge von Aktionen, der, im Ausgangszustand gestartet, in einen Zustand übergeht, der das Ziel erfüllt.

Nachdem wir nun eine Sprache eingeführt haben, die uns ermöglicht, über Planungsprobleme zu sprechen, seien noch ein paar Anmerkungen über die im Folgenden vorgestellten Techniken zur Berechnung eines Plans gemacht.
Die meisten Planer zerlegen ein Problem in mehrere Unterprobleme. Dabei wird angenommen, dass die Probleme *fast zerlegbar* sind. Das bedeutet, dass die Unterziele unabhängig voneinander geplant werden können und anschließend nur noch etwas Rechenzeit benötigt wird, um diese zu kombinieren. Probleme, bei denen die Lösung für ein Unterproblem eine andere rückgängig macht, genügen dieser Annahme nicht.
Weiterhin wollen wir im Folgenden zwischen einer *uninformierten* und einer *heuristischen* Suche unterscheiden. Während wir bei den heuristischen Suchstrategien Informationen darüber haben, wie „vielversprechend“ ein Zustand in Bezug auf die Lösung des Problems ist, können wir bei uninformierten Suchstrategien lediglich zwischen Ziel- und Nichtzielzuständen unterscheiden.

2.2. *Vollständig ordnendes Planen*

Der einfachste Ansatz für einen Planungsalgorithmus basiert auf einer Zustandsraumsuche. Wie wir gesehen haben, sind Aktionen vollständig mit Vorbedingungen und Effekten spezifiziert. Das ermöglicht uns, durch das nacheinander Anwenden von Aktionen, einen Zustandsraum zu erzeugen. Dieser Raum gleicht einem Graphen, bei dem die Zustände durch Knoten und die Aktionen durch Kanten repräsentiert werden. Um eine Lösung für ein bestimmtes Problem zu finden, können wir entweder vom Ausgangszustand aus solange Aktionen anwenden, bis wir einen Zustand erreichen, der das Ziel erfüllt, oder vom Zielzustand aus solange Aktionen rückwärts anwenden, bis wir einen Ausgangszustand erreicht haben (siehe auch Abbildung 1).
Bei beiden Verfahren ist die Lösung eine streng lineare Aktionsfolge, die den Ausgangszustand direkt mit dem Zielzustand verbinden. Solche Verfahren, bei denen Lösungen eine totale Ordnung auf den Aktionen definieren, nennt man *vollständig ordnende Plansuche*. Oftmals ist es aber so, dass Teilprobleme völlig unabhängig voneinander gelöst werden können. In diesem Fall ist es egal, in welcher Reihenfolge die Aktionen der unterschiedlichen Unterziele zueinander durchgeführt werden. Der große Nachteil der vollständig ordnenden Plansuche ist also, dass sie sich nicht

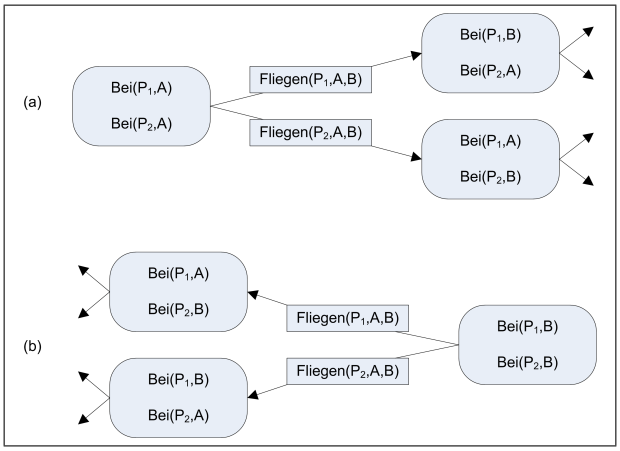

ABBILDUNG 1. Zwei Ansätze für die Suche im Zustandsraum. (a) Bei der Vorwärtssuche wird der Zustandsraum vom Ausgangszustand aus in alle Richtungen exploriert bis ein Zustand erreicht ist, der das Ziel erfüllt. (b) Bei der Rückwärtssuche werden vom Zielzustand aus Aktionen rückwärts angewendet bis ein Ausgangszustand erreicht ist [8].

den Vorteil der Dekomposition eines Problems zu Nutzen machen kann. Planungsalgorithmen, die hingegen Lösungen angeben können, in denen die Reihenfolge für bestimmte Aktionen nicht angegeben ist, nennt man *partiell ordnende Plansuche*. Welchen Vorteile das bietet wird in Abbildung 2 anschaulich dargestellt.

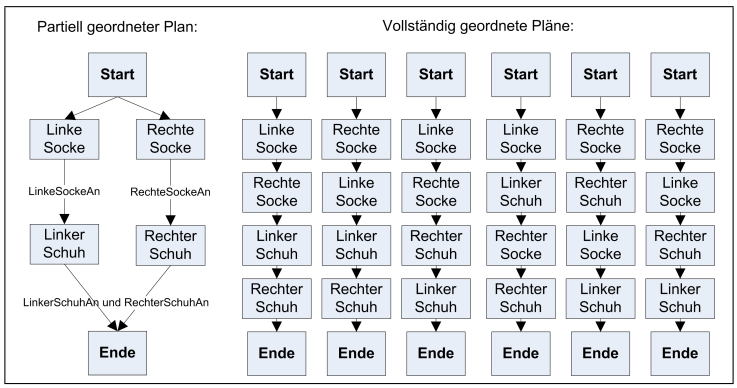

ABBILDUNG 2. Ein partiell geordneter Plan für das Anziehen von Socken und Schuhen entspricht sechs vollständig geordneten Plänen. So können viele vollständig geordnete Pläne mit einem partiell geordneten Plan abgedeckt werden [8].

2.3. *Partiell ordnendes Planen*

Im Gegensatz zu den vollständig ordnenden Plansuchen implementieren wir das partiell ordnende Planen als eine Suche im Raum partiell geordneter Pläne. Das heißt, ein Schritt in der Suche entspricht nicht einer Aktion in der realen Welt,

sondern vielmehr einer Aktion auf einem Plan. Beginnend mit dem *leeren Plan* verfeinern wir den Plan in jedem Schritt durch beispielsweise Hinzufügen einer Aktion zum Plan oder dem Festlegen einer Reihenfolge.
Im Folgenden wird der POP-Algorithmus (Partial Order Planning-Algorithmus) beschrieben, der ein Kalkül für das partiell ordnende Planen darstellt. Es wird dabei nicht darauf eingegangen, *wie* genau der Zustandsraum erkundet wird. Tatsächlich kann dafür eine Vielzahl von uninformierten oder heuristischen Suchverfahren angewendet werden [8].
Die Zustände des POP-Algorithmus sind Pläne, die größtenteils noch unvollständig sind. Jeder Plan besteht aus vier Komponenten.

Eine Menge von **Aktionen**. Diese Menge enthält die Aktionen des Planungsproblems, die im aktuellen Plan berücksichtigt werden. Der *leere Plan* enthält nur die Pseudo-Aktionen *Start* und *Ende*. Die *Start*-Aktion hat keine Vorbedingungen und hat als Effekt alle Literale, die die Ausgangssituation darstellen. Die *Ende*-Aktion hat als Vorbedingungen die Literale des Planungsziels und keine Effekte.

Eine **Ordnungsrelation**. Die Relation enthält Tupel der Form $A \prec B$, die bedeuten, dass die Aktion A irgendwann vor der Aktion B ausgeführt wird. Die Ordnungsrelation muss eine Striktordnung sein. Das heißt, sie muss transitiv, irreflexiv und antisymmetrisch sein. Ein Zyklus – wie etwa $A \prec B$ und $B \prec A$ – stellt einen Widerspruch dar.

Eine Menge **kausaler Verknüpfungen**. Diese Menge enthält Tripel der Form $A \xrightarrow{p} B$, die gelesen werden als „A *erzielt* p für B“. So sagt beispielsweise das Tripel *RechteSocke* $\xrightarrow{RechteSockeAn}$ *RechterSchuh* aus, dass die Aktion *RechteSocke* den Effekt *RechteSockeAn* erzielt, der Vorbedingung für die Aktion *RechterSchuh* ist. Insbesondere muss die Bedingung *RechteSockeAn* in der Zeit zwischen den Aktionen *RechteSocke* und *RechterSchuh* wahr bleiben. Ein Verstoß gegen diese Regel stellt einen *Konflikt* dar. Ein Konflikt mit $A \xrightarrow{p} B$ wird beispielsweise durch eine Aktion C erzeugt, die den Effekt $\neg p$ hat und für die gilt $A \prec C \prec B$ (gemäß der Ordnungsrelation).

Eine Menge **offener Bedingungen**. Eine Vorbedingung ist offen, wenn sie nicht durch eine Aktion im Plan erzielt wird. Planer versuchen die Menge offener Bedingungen auf die *leere Menge* zu reduzieren, ohne dabei einen Widerspruch zu erzeugen.

Ein Plan, der aus den vier genannten Komponenten besteht, nennen wir einen *konsistenten Plan*, wenn es keine Zyklen in der Ordnungsrelation und keine Konflikte in den kausalen Verknüpfungen gibt. Ein konsistenter Plan, der keine offenen Vorbedingungen mehr hat, ist eine Lösung des Planungsproblems.
Ein Schritt im POP-Algorithmus entspricht nun einer Verfeinerung eines bestehenden Plans. Der Ausgangsplan besteht dabei nur aus den Aktionen *Start* und *Ende*. Die Ordnungsrelation enthält nur die Bedingung *Start* $\prec$ *Ende*, es gibt keine kausalen Verknüpfungen und alle Vorbedingungen von *Ende* sind offene Vorbedingungen.

Die Nachfolgerfunktion des POP-Algorithmus wählt ein beliebiges p aus der Menge der offenen Vorbedingungen für eine Aktion B und erstellt einen Nachfolgerplan für jede Aktion A, die in konsistenter Weise p für B erzielt. Dabei wird das Tripel $A \xrightarrow{p} B$ in die Menge der kausalen Verknüpfungen und das Paar $A \prec B$ in die Ordungsrelation aufgenommen. Ein möglicher Konflikt einer Aktion C zu der kausalen Verknüpfung $A \xrightarrow{p} B$, also eine Aktion, die $\neg p$ als Effekt hat, wird dadurch aufgelöst, dass entweder $C \prec A$ oder $B \prec C$ in die Ordnungsrelation aufgenommen wird.
Wenn A eine Aktion ist, die vorher noch nicht im Plan enthalten war, dann fügen wir diese dem Plan hinzu und erweitern die Ordnungsrelation zusätzlich noch um $\mathit{Start} \prec A$ und $A \prec \mathit{Ende}$.
Ein Zieltest prüft, ob der aktuelle Plan eine Lösung darstellt. Da durch die Nachfolgerfunktion sichergestellt ist, dass nur konsistente Pläne erzeugt werden, reicht es zu prüfen, ob die Menge der offenen Bedingungen leer ist.
Die Ausführung eines berechneten Plans entspricht dann einfach der wiederholten Ausführung einer beliebigen möglichen Aktion im Plan. Diese Flexibilität bei der Ausführung erweist sich für einen Agenten als sehr hilfreich, wenn die Welt nicht kooperiert, oder auch, wenn mehrere kleine Pläne zu größeren Plänen zusammengefügt werden sollen, da die flexible Ausführung Spielraum zur Auflösung von Konflikten beim Zusammenführen lässt. Besonders der letzte Punkt ist wichtig im Hinblick auf das im nächsten Kapitel vorgestellte Planen mit Hierarchical Task Networks.

3. HTN - Hierarchical Task Networks

3.1. *Einführung*

Die im ersten Kapitel vorgestellten Planungstechniken liefern uns schon gute Algorithmen, um Pläne für ein gegebenes Planungsproblem zu finden. Vor allem die Methode des partiell ordnenden Planens liefert eine flexible Möglichkeit, die effizient einen Plan berechnen kann. Doch Planungsprobleme in der realen Welt erreichen im Allgemeinen eine Größe, die es unmöglich macht, das Problem als Ganzes mit einer der vorgestellten Techniken zu umfassen. Im Folgenden wird das Planen mit Hilfe von *Hierarchical Task Networks (HTN)* vorgestellt. Diese Technik versucht, Planungsprobleme mit Hilfe von hierarchischer Zerlegung handhabbar zu machen. Die Grundidee hinter der hierarchischen Dekomposition ist die Zerteilung eines Problems in eine kleine Anzahl von Unterproblemen, deren Komplexität deutlich geringer ist als die Komplexität des gesamten Problems.
Beim HTN-Planen wird der ursprüngliche Plan, der das Problem beschreibt, als Beschreibung auf höchster Ebene betrachtet, die angibt, was zu tun ist. Eine Aktionszerlegung verfeinert diesen Plan nun in mehrere Unterpläne, die beschreiben, wie das Oberziel umgesetzt werden kann. Ist der urspüngliche Plan beispielsweise, ein Haus zu bauen, so könnte dieser Plan darauf reduziert werden, eine Genehmigung einzuholen, einen Bauunternehmer zu beauftragen und den Unternehmer zu bezahlen. Pläne werden solange konkretisiert, bis schließlich nur noch *primitive*

Aktionen übrig bleiben, die der Agent selber sofort ausführen kann. Je nach Agent können diese primitiven Aktionen sehr unterschiedlich sein.
Wir sehen, dass das „reine“ HTN-Planen einer Konkretisierung des ursprünglichen Problems entspricht. Im Gegensatz zum vollständig, oder auch partiell ordnenden Planen, bei denen jeweils ein Plan, beginnend mit dem leeren Plan, *aufgebaut* wird, ist beim HTN-Planen immer schon ein kompletter Plan für das Problem vorhanden. Wir werden in Kapitel 3.3 sehen, dass sich das HTN-Planen gut in das partiell ordnende Planen integrieren lässt.

3.2. *Aktionszerlegung*

Der Preis, den man zahlen muss, um ein bestimmtes Planungsproblem mittels eines HTN zerlegen zu können, ist eine sogenannte *Planbibliothek (Plan Library)*. In ihr muss Domänenwissen gespeichert sein, das angibt, wie eine Aktion zerlegt werden kann.
Die Planbibliothek besteht aus Methoden der Form $zerlegen(a, d)$, die besagt, dass eine Aktion a in einen partiell geordneten Plan d zerlegt werden kann. Der Plan d besteht dabei, wie in Kapitel 2.3 beschrieben, aus einer Aktionsmenge, einer Ordnungsrelation, einer Menge kausaler Verknüpfungen und einer Menge offener Bedingungen. Ein Beispiel für eine solche Methode zeigt Abbildung 3. Im HTN-Planen werden die Effekte der *Start*-Aktion auch als *externe Vorbedingungen* und die Vorbedingungen der *Ende*-Aktion auch als *externe Effekte* bezeichnet.

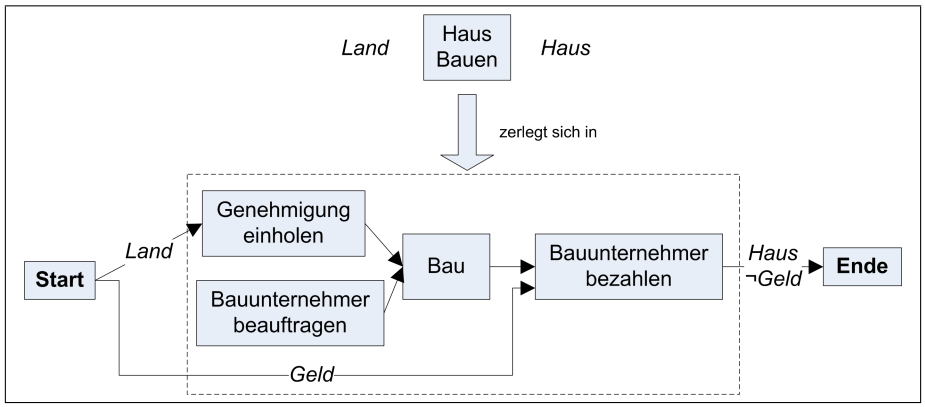

Abbildung 3. Eine mögliche Zerlegung der Aktion *HausBauen* in mehrere Unterziele mit zusätzlichen Vorbedingungen und Effekten [9].

Eine Methode der Planbibliothek sollte nur *korrekte* Zerlegungen enthalten. Eine Methode $zerlegen(a, d)$ ist dann korrekt, wenn d ein vollständiger, konsistenter, partiell geordneter Plan ist, der mit den Vorbedingungen von a die Effekte von a erzielt.
Eine Aktion kann mehrere mögliche Zerlegungen in der Planbibliothek enthalten, die sogar voneinander verschiedene Vorbedingungen und Effekte haben können. Dabei übernimmt man für die Aktionsbeschreibung der Aktion auf höchster Ebene nur die Schnittmenge der Vorbedingungen und Effekte aller möglichen Zerlegungen auf der darunter liegenden Ebene. Die so „versteckten“ Vorbedingungen und Effekte

können auch als Implementierungsdetails einer bestimmten Instanziierung gesehen werden. Vorbedingungen und Effekte, die innerhalb einer Instanziierung erzielt und verbraucht werden, werden ebenfalls gegenüber der höheren Ebene „versteckt“. Dieses sogenannte *Information Hiding* ist ein entscheidender Faktor bei der Reduzierung der Komplexität. Allerdings stellt uns das Information Hiding auch vor die Herausforderung, darauf zu achten, dass interne Bedingungen nicht zu Konflikten mit anderen Aktionen führen, da auf höheren Ebenen nicht mehr alle Informationen sichtbar sind. Wie diese Konflikte aufgelöst werden können, ist in [5] gezeigt worden.

3.3. *HTN im partiell ordnenden Planen*

Wie schon angedeutet, werde ich in diesem Kapitel eine Möglichkeit vorstellen, wie das HTN-Planen in das partiell ordnende Planen integriert werden kann. Dazu erweitern wir den vorgestellten POP-Algorithmus um die Möglichkeit, Zerlegungsmethoden auf den aktuellen Plan P anzuwenden. Dabei wird eine nicht primitive Aktion a' aus P ausgewählt und durch $d' = SUBST(\theta, d)$ ersetzt. Diese Ersetzung wird für alle Methoden $zerlegen(a, d)$ aus der Planbibliothek ausgeführt, bei der a und a' mit der Substitution θ unifizierbar sind. Jede Anwendung einer Zerlegungsmethode resultiert in einem eigenen Nachfolgerplan.

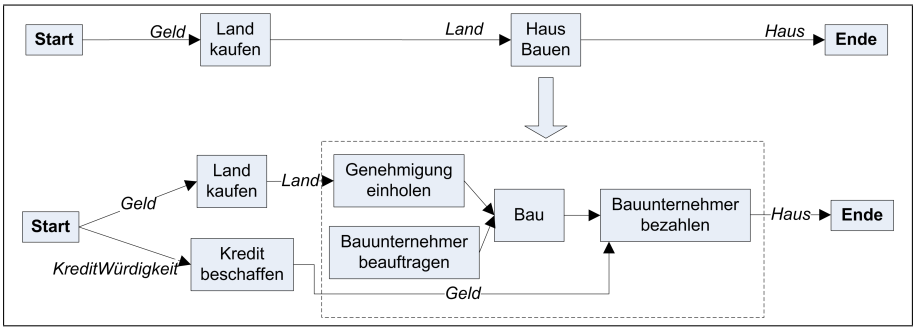

Abbildung 4. Verfeinerung eines bestehenden Plans mit Hilfe einer Aktionszerlegung. Die zusätzliche Vorbedingungen *Geld* muss durch eine neue Aktionen *Kreditbeschaffen* erzielt werden. Subtask Sharing ist in diesem Fall nicht möglich [9].

Abbildung 4 zeigt beispielhaft eine mögliche Zerlegung. Die Änderungen im aktuellen Plan nach einer Zerlegung werden in drei Schritten durchgeführt, die für jeden Nachfolgerplan ausgeführt werden müssen:

(1) Zunächst wird die alte Aktion a' aus dem Plan entfernt und durch die neuen Aktionen aus d' ersetzt. In der Regel sind die einzelnen Schritte aus d' neu zu instanziierende Aktionen im Plan P. Es kann aber auch vorkommen, dass es bereits eine Aktion im ursprünglichen Plan gibt, die mit einer der neuen Aktionen aus d' unifizierbar ist, also die gleichen Vorbedingungen und Effekte aufweist. In diesem Fall müssen wir keine neue Instanz der Aktion aus d' bilden, sondern können die bereits bestehende Aktion verwenden. Wir sprechen dann von einer *gemeinsamen Nutzung von Teilaufgaben (Subtask Sharing)*.

(2) Als nächstes muss die Ordnungsrelation angepasst werden. Dazu müssen die Ordnungsbedingungen für a' im ursprünglichen Plan mit den Schritten in d' verbunden werden. Stellen wir uns eine Ordnungsbedingung der Form $B \prec a'$ im ursprünglichen Plan vor. Ein primitiver Ansatz ist es, alle Bedingungen der Form $\mathit{Start} \prec s$ in d' für einen Schritt s durch die Bedingung $B \prec s$ zu ersetzen. Das könnte aber eine zu strenge Forderung sein, da in einem verfeinerten Plan möglicherweise nicht jeder Schritt zwangsläufig nach B ausgeführt werden muss. Besser ist es, für jede Ordnungsbeziehung einen *Grund* mitzuführen, so dass die Ordnungsbedingungen möglichst weit gelockert werden können. Analog behandeln wir Ordnungsbedingungen der Form $a' \prec C$

(3) Als letzten Schritt müssen wir die Menge der kausalen Verknüpfungen anpassen. Dazu ersetzen wir alle kausalen Verknüpfungen der Form $B \xrightarrow{p} a'$ im ursprünglichen Plan durch eine Menge von kausalen Verknüpfungen von B zu allen Schritten in d', die die externe Vorbedingung p haben. Das heißt, die eine Vorbedingung haben, die durch die *Start*-Aktion in d' bereitgestellt wird.

Mit diesen drei Schritten haben wir unseren POP-Algorithmus vollständig um die Aktionszerlegung erweitert. Für die Suche nach Lösungen müssen allerdings noch weitere Schwierigkeiten bedacht werden. So kann es sein, dass ein nicht auflösbarer Konflikt auf einer höheren Ebene, also ein Konflikt einer Aktion mit einer kausalen Verknüpfung, die nicht vor oder nach dieser eingeordnet werden kann, nach einer Zerlegung wieder aufgelöst werden kann. Unsere ursprüngliche Suche hätte solche Zweige mit nicht auflösbaren Konlikten nicht weiter untersucht. Bei der HTN-Erweiterung könnten uns so aber Lösungen verloren gehen.

3.4. *Diskussion*

„Reines“ HTN-Planen, also die Suche nach einer Lösung durch bloßes Verfeinern eines Plans ist im Allgemeinen unentscheidbar, obwohl der Zustandsraum endlich ist. Schuld an diesem Umstand ist, dass es rekursiv definierte Zerlegungsregeln geben kann. Diese zunächst etwas ernüchternde Erkenntnis kann aber schnell abgeschwächt werden, denn wir können Entscheidbarkeit leicht erreichen, wenn wir beispielsweise die Rekursion, die ohnehin nur in speziellen Domänen verwendet wird, verbieten, oder tiefenbeschränken. Entscheidbarkeit können wir außerdem erreichen, wenn wir den im letzten Abschnitt vorgestellten hybriden Ansatz von partiell ordnendem Planen in Verbindung mit HTN-Planen verwenden, da partiell ordnendes Planen allein entscheidbar ist (siehe [9]).
Bei den Überlegungen zur Unentscheidbarkeit stellt sich die Frage, woher die Effizienzsteigerung kommen soll, die wir uns vom HTN-Planen erhofft haben. Dazu wollen wir zwei der größten Faktoren zur Effizienzsteigerung betrachten.
Zum einen kann das Subtask Sharing einen Plan erheblich effizienter machen. Der Sinn des Subtask Sharings ist, bereits erzielte Effekte für mehrere Aktionen wiederzuverwenden. Schauen wir uns dazu das Beispiel an, wie Compiler den Ausdruck $\sin(x) - \tan(x)$ kompilieren. Die meisten Compiler nutzen dazu zwei Subroutinen, die jeweils $\sin(x)$ bzw. $\tan(x)$ berechnen. Dabei finden alle Aktionen der

n	HTN	nicht-hierarchisch
10	6	59049
100	74559107	$5,1537 * 10^{47}$
1000	$2,7586 * 10^{79}$	$1,3220 * 10^{477}$

TABELLE 1. Vergleich der Anzahl zu untersuchender Pläne in einem nicht hierarchischen Planer und einem HTN-Planer für $d = b = 3$ und $k = 7$.

sin-Berechnung vor den Aktionen der tan-Berechnung statt. Betrachten wir nun die folgende Taylor-Folgen-Annäherung:

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}; \quad \tan(x) \approx x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315}$$

Ein HTN-Planer mit Subtask Sharing könnte viele Schritte der sin-Berechnung in der tan-Berechnung wiederverwenden.

Viel entscheidender bei der Betrachtung der Komplexität ist aber die folgende Überlegung. Angenommen wir wollen einen Plan mit n Aktionen konstruieren. Wenn jede Aktion b erlaubte Folgeaktionen hat, dann gibt es $O(b^n)$ Pläne, die ein nicht hierarchischer Zustandsraumplaner untersuchen müsste.

Nehmen wir an, für jede nicht primitive Aktion gäbe es d mögliche Zerlegungen in jeweils k Unteraktionen. Wie viele Zerlegungsbäume gibt es in diesem Fall, die ein HTN-Planer untersuchen müsste? Da die gewünschten n Aktionen auf der primitiven Ebene liegen müssen, also den Blatt-Knoten des Zerlegungsbaumes entsprechen, muss die Anzahl der Ebenen im Baum unterhalb der Wurzel gleich $\log_k n$ sein. Das heißt die Anzahl der inneren Knoten im Baum ist $1 + k + k^2 + \ldots k^{\log_k n} = (n-1)/(k-1)$. Da jeder innere Knoten d mögliche Zerlegungen hat, ergeben sich insgesamt $d^{(n-1)/(k-1)}$ mögliche Zerlegungsbäume. Wie sich diese Formeln bei ähnlichen b und d auswirken zeigt Tabelle 1.

Gerade auf Grund dieses enormen Einsparungspotentials sind heute fast alle Planer für große Probleme in der Praxis HTN-Planer. HTN-Planer ermöglichen den Experten, über die Planbibliothek zusätzliches Domänenwissen mit in die Planung einzubinden. Besonders intelligente Agenten sind dabei zusätzlich in der Lage, neue Methoden zu *lernen* und diese in ihre Planbibliothek zu übernehmen. Dazu werden einmal mühsam erstellte Pläne *verallgemeinert*, also von probleminstanz-spezifischen Details wie Namen, befreit.

HTN-Planen ist nicht zuletzt auch deshalb so erfolgreich, weil es auf verblüffende Art und Weise dem menschlichen Vorgehen beim Planen gleicht. Es ist nur schwer vorstellbar, dass ein Mensch, der nicht nach einer ähnlichen Strategie verfährt wie ein HTN-Planer, wesentlich kompetenter planen kann.

4. Planen in nicht deterministischen Domänen

Bisher haben wir uns lediglich *deterministische* Domänen angeschaut. In deterministischen Domänen sind die Effekte einer Aktion immer klar und von Anfang an bekannt. So konnten die bisher vorgestellten Planer zunächst einen Plan erstellen,

ihn danach einfach durchführen und sicher sein, dass das Ziel anschließend erreicht ist.
Wesentlich realistischer für Probleme der realen Welt sind aber *nicht deterministische* Domänen. Also Domänen, in denen man die Effekte bestimmter Aktionen nicht vorhersagen kann. Wir unterscheiden in diesem Zusammenhang die *begrenzte Indeterminanz* und die *unbegrenzte Indeterminanz*. Bei der begrenzten Indeterminanz können Aktionen zwar unvorhersagbare Effekte haben, diese können aber in der Aktionsbeschreibung aufgelistet werden. Der Effekt einer Aktion *Münze werfen* ist nicht vorhersagbar, kann aber nur *Kopf* oder *Zahl* sein. Bei unbegrenzter Indeterminanz ist die Menge der Effekte entweder unendlich, oder so groß, dass diese nicht mehr in der Aktionsbeschreibung aufgelistet werden kann.
Wir wollen im Folgenden zwei Ansätze untersuchen, die Planen in nicht deterministischen Domänen realisieren. Dabei setzen wir die Annahme der *begrenzten Indeterminanz* und der *vollständig beobachtbaren Welt* voraus. In einer vollständig beobachtbaren Welt können wir jederzeit den Zustand der gesamten Domäne erfassen.

4.1. *Vorwärtssuche*

Eine Möglichkeit für das Planen in nicht deterministischen Domänen haben Kuter und Nau [6] vorgestellt. Sie haben die bereits in Kapitel 2.2 vorgestellte Vorwärtssuche für nicht deterministische Domänen angepasst. Dazu wollen wir kurz rekapitulieren, wie die Vorwärtssuche im deterministischen Fall aussah.
Beginnend mit dem leeren Plan fügen wir bei der Vorwärtssuche in jedem Schritt eine beliebige Aktion, die im aktuellen Zustand anwendbar ist, zum Plan hinzu. Diesen Schritt wiederholen wir solange, bis wir einen Zustand erreicht haben, der das Ziel erfüllt. Die Auswahl einer Aktion ist dabei die entscheidende Drehschraube für die Optimierung des Algorithmuses. Wird die Vorwärtssuche beispielsweise in Verbindung mit HTN-Planen eingesetzt, wird die Auswahl der Aktion entscheidend durch die Methoden der Planbibliothek beeinflusst. In ihr können effiziente Strategien und Heuristiken codiert sein.
Im nicht deterministischen Fall müssen wir die Abbruchbedingung der Vorwärtssuche etwas anpassen. Während bisher die Anwendung einer Aktion immer genau *einen* Folgezustand hatte, liefert die Übergangsfunktion im nicht deterministischen Fall immer eine Menge von möglichen Zuständen. Der Unterschied ist, dass wir vorher lediglich überprüfen mussten, ob der letzte Zustand des Plans das Ziel erfüllt. Jetzt müssen wir für jeden möglichen Zustand, der in einer Zustandsmenge des Plans auftritt, einen Pfad in unserem Plan haben, der das Ziel erfüllt. Diese Notwendigkeit ist in Abbildung 5 dargestellt.
Der große Vorteil des HTN-Planens – die effiziente Exploration des Suchraums durch geeignete Heuristiken in Form von Aktionszerlegungen– kann aber in Domänen, in denen es keine oder nur wenige Möglichkeiten gibt, eine Heuristikfunktion anzugeben, nicht ausgenutzt werden. Dann wird der Suchraum sehr schnell sehr groß und die Vorwärtssuche stößt vor allem bei der expliziten Darstellung der Zustände an ihre Grenzen.
Dieser Nachteil wird in [7] sehr schön anhand der Hunter-Prey Domäne erklärt. In dieser Domäne gibt es einen Jäger (Hunter) und eine Beute (Prey), die zufällig auf

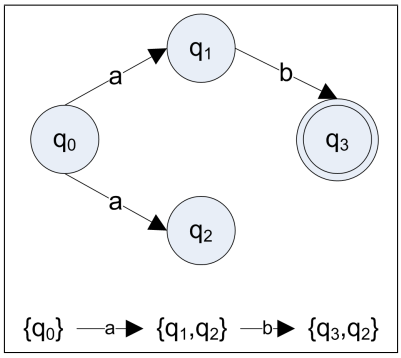

ABBILDUNG 5. Bei der nicht deterministischen Vorwärtssuche müssen alle Zustände einer Zustandsmenge einen Pfad zu einem Zielzustand haben. q_3 ist ein Zielzustand, aber für den Zustand q_2 gibt es keine Aktion, die q_2 in den Zielzustand überführt. Wenn die Aktion a in Zustand q_0 das System in den Zustand q_2 überführt, führt der angegebene Plan nicht in einen Zielzustand. Es ist notwendig, dass es eine Aktion gibt, die q_2 in einen Zielzustand überführt. In der deterministische Variante reicht es aus, wenn der letzte Zustand in einem Plan den Zielzustand erfüllt.

einem $n \times n$-Grid platziert sind. Beide Spieler können sich in einem Schritt entweder ein Feld nach *oben*, *unten*, *rechts* oder *links* bewegen. Der Jäger hat zusätzlich eine Aktion *catch*, um die Beute zu fangen und die Beute hat eine Aktion *stand*, um auf dem Feld stehen zu bleiben. Ziel ist es, einen Plan für den Jäger zu entwickeln, die Beute zu fangen. Die Indeterminanz in dieser Domäne ist gegeben durch die Unvorhersagbarkeit der Bewegung der Beute. Dieses Beispiel wird uns auch in den folgenden zwei Abschnitten noch begleiten.
Da diese Domäne sehr wenig Spielraum für effiziente Strategien lässt – es läuft darauf hinaus zu schauen, wo sich die Beute befindet und sich dann in die Richtung zu bewegen – wird der Zustandsraum und die Darstellung der Zustände bei der Vorwärtssuche sehr groß und schon bei einem 10×10-Grid bekommen aktuelle Implementierungen Schwierigkeiten, das Problem noch im Speicher darzustellen.

4.2. *Symbolic Model Checking*

Model Checking hat in den letzten Jahren immer mehr an Popularität gewonnen und wird immer häufiger dazu eingesetzt, automatisch zu prüfen, ob ein System bestimmte Eigenschaften erfüllt. Da die Algorithmen für Model Checker inzwischen in der Lage sind, auch sehr große Zustandsräume zu prüfen (bis zu 10^{200} Zustände [2]), ist diese Technik auch für das automatische Planen interessant geworden.
Beim Model Checking wird ein System als eine *Kripke Struktur* dargestellt, bestehend aus einer Menge von Zuständen, einer Menge von initialen Zuständen, einer Übergangsfunktion und einer Labeling-Funktion, die in jedem Zustand einer Menge von Variablen den Wert *true* oder *false* zuweist. Ein Model Checker bekommt nun als Eingabe eine solche Struktur eines Systems und eine zu prüfende Eigenschaft, die beispielsweise als temporallogische Formel formuliert sein kann. Als Ergebnis

liefert der Model Checker *true*, wenn die Bedingung im System erfüllt wird, oder *false*, wenn die Eigenschaft nicht erfüllt wird.
Da die Zustandsräume in solchen Systemen ebenfalls sehr groß werden können, verfolgt das *Symbolic Model Checking* den Ansatz, die Zustände in einer kompakteren Weise darzustellen. Symbolic Model Checking basiert hauptsächlich auf zwei Grundideen:

(1) Beim Symbolic Model Checking werden Mengen von Zuständen untersucht und nicht mehr nur einzelne Zustände.
(2) Das Symbolic Model Checking wird symbolisch ausgeführt. Die Zustandsmengen und die Übergänge zwischen Zustandsmengen werden durch logische Formeln repräsentiert.

Ein Zustand wird dabei als Konjunktion von Variablen, die im aktuellen Zustand den Wert *true* haben, dargestellt. Eine Zustandsmenge ist dann die Disjunktion aller in ihr enthaltenen Zustände. Genauso wird auch die Übergangsfunktion in eine Disjunktion von Übergängen übersetzt. Dabei wird eine Kripke-Struktur K mit einer Zustandsmenge S_K und einer Übergangsrelation R_K folgendermaßen in eine Formel φ_{s_0} bzw. φ_R übersetzt:

$$s_0 \in S_K \Leftrightarrow \varphi_{s_0} = \bigwedge_{v \in var(s_0)} v$$

$$(s, s') \in R_K \Leftrightarrow \varphi_r = \varphi_s \wedge \varphi_{s'} \in \varphi_R$$

$$\varphi_R = \bigwedge_{r \in R_K} \varphi_r$$

Durch diese Darstellung lassen sich die Zustandsmengen und die Übergangsrelation mittels logischer Operationen stark reduzieren oder miteinander verknüpfen und sind einer expliziten Darstellung der Zustände deutlich überlegen. Für die genaue Umsetzung der Zustände und der Übergangsfunktion sei auf [4] verwiesen. Eine Möglichkeit, das Symbolic Model Checking effizient zu implementieren, bieten die sogenannten *Binary Decision Diagrams (BDD)*, die ebenfalls in [4] näher beschrieben sind.
Diese kompakte Darstellung von Zuständen ermöglicht effizientes Planen auch in sehr großen Zustandsräumen. In der bereits vorgestellten *Hunter-Prey Domäne* findet ein auf Symbolic Model Checking basierender Algorithmus Pläne auch noch für 50×50-Grids und darüber hinaus [7].
Der Nachteil des Symbolic Model Checkings ist allerdings, dass es keine Möglichkeit gibt, wie im HTN-Planen effiziente Suchstrategien oder Heuristiken zu berücksichtigen. Diese Schwäche wird deutlich, wenn wir die Hunter-Prey Domäne etwas abwandeln und statt einer einzigen Beute mehrere Beuten auf dem Spielfeld verteilen, die alle gefangen werden müssen. Dabei sagen wir, dass sich eine Beute nicht auf ein Feld bewegen darf, das sich direkt neben dem einer anderen Beute befindet. Während wir beim HTN-Planen eine Strategie wie „Verfolge zunächst eine bestimmte Beute und fange danach die anderen“ berücksichtigen können, ist das beim Symbolic Model Checking nicht möglich. Hinzu kommt, dass die Bewegungen abhängig voneinander sind und somit die Zustände des Problems nur geringfügig

reduziert werden können. Auf einem 3×3-Grid liegt deshalb die Laufzeit eines Symbolic Model Checking-Planers schon bei einer Anzahl von 4 Beuten deutlich über der eines HTN-Planers [7].

4.3. *Kombination von HTN-Planen und Symbolic Model Checking*

Kuter, Nau, Pistore und Traverso haben 2005 einen Algorithmus vorgstellt, der die Vorteile von HTN-Planen und Symbolic Model Checking miteinander kombiniert. Dabei bildet eine HTN-basierte Vorwärtssuche die Grundlage. Allerdings werden jetzt nicht nur Zustandsmengen untersucht, wie es im Kapitel 4.1 vorgestellt wurde, sondern der Algorithmus arbeitet auf sogenannten *situations*. Eine *situation* ist ein Tupel (S, w), das eine Zustandsmenge S in einem bestimmten Tasnetzwerk w darstellt. Beginnend mit der initialen situation (I, w) generiert der Algorithmus rekursiv Mengen von *situations*, bis eine Lösung des Planungsproblems gefunden wird.
Der Vorteil der *situations* ist, dass diese bei der Berechnung der Nachfolger einer *situation* zusammengefasst und so kompakt dargestellt werden können. An dieser Stelle kommt die Idee des Symbolic Model Checking ins Spiel. Der genaue Algorithmus ist in [7] nachzulesen.
In der Hunter-Prey Domäne zeigt sich besonders schön, wie der vorgestellte Algorithmus die Vorteile des HTN-Planens und des Symbolic Model Checkings verbinden kann. Der Algorithmus schlägt sowohl in der Variante mit nur einer Beute als auch in der Variante mit mehreren Beuten auf beliebig großen Grids beide zuvor vorgestellten Algorithmen.

5. Fazit und Ausblick

In dieser Ausarbeitung wurden die Grundlagen des automatischen Planens vorgestellt. Dabei haben wir uns zunächst auf die allgemeine Beschreibung von Planungsproblemen konzentriert und Lösungen vorgestellt, wie Pläne in deterministischen Domänen erstellt werden können. Im letzten Kapitel haben wir gesehen, welche Ansätze es gibt, auch in nicht deterministischen Domänen Pläne effizient zu finden. Dabei haben wir uns zunächst auf vollständig beobachtbare Welten und eine begrenzte Indeterminanz beschränkt. Für Ansätze mit unbegrenzter Indeterminanz sei auf [9] hingewiesen. Ein weiteres wichtiges Feld im Bereich der automatischen Planung ist das Multiagenten-Planen, bei dem es andere Agenten in der Umgebung gibt, mit denen kooperiert, koordiniert oder konkurriert wird. Ein möglicher Ansatz in diesem Bereich ist das partiell ordnende Mehrkörper-Planen [1].

Literatur

[1] C. Boutilier, R. I. Brafman. Partial order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, (14):105–136, 2001.
[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In: *Information and Computation, 98(2)*, S. 142–170. 1992.
[3] R. Fikes, N. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. In: *Artificial Intelligence, 2*, S. 189–208. 1971.

[4] M. Ghallab, D. Nau, P. Traversi. *Automated Planning - theory and practice*, Kap. C: Model Checking, S. 561–572. Morgan Kaufmann, San Francisco, 2004.

[5] S. Kambhampati, A. D. Mali, B. Srivastava. Hybrid planning for partially hierarchical domains. In: Aaai Press (Hg.), *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, S. 882–888, Wisconsin, 1998. Madison.

[6] U. Kuter, D. Nau. Forward-chaining planning in nondeterministic domains. Technischer bericht, American Association for Artificial Intelligence, 2004.

[7] U. Kuter, D. Nau, M. Pistore, P. Traverso. A hierarchical task-network planner based on symbolic model checking. Technischer bericht, American Association for Artificial Intelligence, 2005.

[8] S. Russel, P. Norvig. *Künstliche Intelligenz - Ein moderner Ansatz, 2*, Kap. 11: Planen, S. 466–483. Pearson, München, 2004.

[9] S. Russel, P. Norvig. *Künstliche Intelligenz - Ein moderner Ansatz, 2*, Kap. 12: Planen und Agieren in der realen Welt, S. 515–533. Pearson, München, 2004.