New Results in Planning, Scheduling, and Design (PUK2004)

Workshop Proceedings

Edited by:
Stefan Edelkamp, University of Dortmund, Germany
Benno Stein, University of Paderborn, Germany

Located at the
27th German Conference on Artificial Intelligence
September 20-24, Ulm, Germany

WS 8

Preface

The *German Planning, Scheduling, Configuration, and Design Interest Group*, PUK for short, consists of professionals in research and practice and serves as a platform to share results and experiences. The four fields planning, scheduling, configuration, and design share many similar, often AI-based methods, with only a small communication between them. Consequently, the PUK-Workshop bridges this gap, presenting problem areas and solution concepts, together with implemented systems. This leads to possible synergies between theory and practice across the four fields to develop a common view and to find joint solutions for related problems. Besides knowledge acquisition and transfer, the main objective of the workshop is to foster the contact between different research groups and industry. In a steadily changing mix of talks from people at universities and companies, many short- and long-term cooperations have emerged. In contrast to other happenings, PUK is, therefore, highly concerned about preserving its meeting character. This year's event is the 18th PUK in a long series of exiting workshop events. Papers at PUK-2004 cover the entire scope of planning, scheduling, configuration and design. As in former events, it has a main focus, namely *Exploration with Heuristic Search*, a technique, which is used in current action planners and many systems for configuration and design. In this research area, problem solving is modelled as state space search with operators that are applied over and over again until the problem's goal is encountered. PUK-2004 encouraged students to present their Ph.D. and Masters' Theses work and selected one paper as an invited submission for the KI-Journal, namely Sebastian Kupferschmidt's summary of his Masters' Thesis.

On Thursday 23rd, we have elections and a discussion on the future of PUK. On Friday, 24th, we start with a short welcome to the 18th PUK. In the first session on *Planning and Search*, René Schumann presents his work on ABAKO and Tilman Mehler introduces incremental state space hashing. Then the intermediate results of the project group *GPS-route* are presented by Shahid Jabbar. In the second session on *Production Planning, Recommender Systems and Game Playing* Jana Köhler addresses *Planning with Workflows*, and Dietmar Jannach discusses *Product Finders and Knowledge-based Recommenders*. Subsequently, Sebastian Kupferschmidt gives insights to his *Double-Dummy Skat Solver*. After the lunch break, the third session addresses *Design and Synthesis* with a talk of Thorsten Krebs on *Dependency Analysis*. Next Benno Stein reports on the *Extensible Synthesis Framework* developed in their group. In retrospective, a report on the classical track of the *4th International Planning Competition* co-organized by Stefan Edelkamp and Jörg Hoffmann is given. Finally, we have a Panel on *Synergies in between the Fields*. We hope that, by reading this proceedings, everybody receives an impression of the the fun, importance and charme of this year's event. We wish all of you an exiting workshop!

Stefan Edelkamp and Benno Stein (co-chairs)

Program Committee at PUK 2004

Stefan Edelkamp (Co-Chair)
Lehrstuhl Informatik V
Universität Dortmund
Baroper Straße 301
GB IV / 107, D-44227 Dortmund

Benno Stein (Co-Chair)
Universität Paderborn
Faculty of Electrical Engineering,
Computer Science and Mathematics
Computer Science Department
D-33095 Paderborn

Jana Köhler (Speaker of GI Group)
IBM Research Laboratory
CH-8803 Rüschlikon
Säumerstraße 4 / Postfach
Switzerland

Jürgen Sauer (Speaker of GI Group)
Universität Oldenburg
Computer Center, HRZ
Uhlhornsweg 49-55
D-26129 Oldenburg

Lothar Hotz
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg

Thorsten Krebs
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg

Table of Contents

1st Session: Planning and Search

2nd Session: Production Planning, Recommender Systems and Game Playing

3rd Session: Design and Synthesis

ABAKO, ein agentenbasiertes Verfahren zur Koordination von Planungssystemen

René Schumann

Carl von Ossietzky Universität Oldenburg
Department für Informatik

Zusammenfassung Es wird ein Verfahren zur Koordination von Planungssystemen, das ABAKO-Verfahren (ABAKO: **a**genten**ba**sierte **Ko**ordination), vorgestellt. Bei diesem Verfahren bleibt die Autonomie der einzelnen organisatorischen Einheiten (den Planungsstellen), die ein Planungssystem für ihren Bereich einsetzen, weitgehend erhalten. Daraus folgt, dass die Planungsstellen auch nur die nötigsten Informationen für die Koordination bereitstellen. Unter diesen Bedingungen ist "nur" eine Pareto-Optimale Zuteilung der Aktivitäten auf die Planungsstellen möglich. Um weitere Verbesserungen zu erreichen, bietet das ABAKO-Verfahren die Möglichkeit Planverbesserungen durchzuführen. Dabei können die Planungsstellen Verhandlungen über Veränderungen von Zuordnungen führen, die sowohl lokal den Planungsstellen als auch dem gesamten Produktionsnetzwerk nützen.

1 Einleitung

Das Koordinationsverfahren wird im Kontext von Ablaufplanungsproblemen verdeutlicht, da es sich so gut darstellen lässt. Eine besondere Bedeutung haben Ablaufplanungsprobleme im Rahmen der betrieblichen Produktion. In diesem Bereich werden auf Grund der Komplexität oft mehrere Planungs- bzw. Optimierungswerkzeuge für einige Teilbereiche eingesetzt. Durch die Einführung solcher Systeme entstehen, insbesondere im Bereich der Fertigung, so genannte Planungsinseln, die sich etwa auf eine Maschine, einen "floor", ein Werk oder im weitesten Fall auf ein Unternehmen beschränken. Um allerdings eine Annäherung an ein Gesamtoptimum der betrieblichen Produktion (in einem Unternehmen oder einem Unternehmensnetzwerk) zu erreichen, genügt es nicht, die einzelnen Teilbereiche separat zu optimieren. Es ist vielmehr notwendig, die einzelnen Teilbereiche aufeinander abzustimmen. Die Planungsinseln müssen koordiniert werden, um so die übergeordneten Planungsziele besser zu erreichen.
Die Notwendigkeit zur Koordination von Planungssystemen, insbesondere im Bereich der Fertigung, und somit der Ablaufplanung, wird durch die Kooperation von Unternehmen in Form von virtuellen Unternehmen oder Supply Chains noch weiter verstärkt. Allerdings scheinen die herkömmlichen Koordinationsverfahren, die meist auf hierarchischen Vorgaben und deren lokaler Erfüllung beruhen, gerade bei Koordinationsaufgaben, bei denen die Koordination über Unternehmensgrenzen hinweg erfolgen soll, nur noch bedingt geeignet.

Aus diesem Grund wurde das ABAKO-Verfahren entwickelt, dass eine Koordination in solchen Unternehmensnetzwerken unter Berücksichtigung der sich dabei ergebenden Anforderungen durchführen kann.

2 Problemstellung

Eine Planungsstelle ist eine (selbstständige) organisatorische Einheit, für deren jeweiligen Bereich ein Planungssystem lokale Vorgaben erzeugt. Mehrere Planungsstellen bilden zusammen ein Unternehmens-/Produktionsnetzwerk [1], das gemeinsam Leistungen erstellen kann. Bei den Planungsstellen soll von rechtlich und (weitgehend) wirtschaftlich selbstständigen Einheiten ausgegangen werden. Da bei der Leistungserstellung des Unternehmensnetzwerks die Entscheidungen der einzelnen Planungsstellen interdependent sind, besteht ein Koordinationsbedarf zwischen den Planungsstellen. Dieser ist auf der Ebene der jeweiligen lokalen Feinplanung zu lösen, da auf den darüber liegenden Planungsebenen die Zeitraster im Allgemeinen zu grob sind und so die gemeinsame Leistungserstellung nicht effizient durchgeführt werden könnte.
Da die Koordination in einem Netzwerk von autonomen Planungsstellen durchgeführt werden soll und diese Autonomie erhalten werden soll, werden folgende Anforderungen an die Koordination gestellt:

- **Informationsoffenbarung zur Koordination ist nicht notwendig:** In einem Unternehmensnetzwerk, in dem Partner über die gleichen oder ähnliche Kompetenzen verfügen, werden die Einheiten nur dazu bereit sein die nötigsten Informationen preiszugeben. Die Koordination ist also mit möglichst wenigen und unkritischen Informationen durchzuführen.
- **Zwischen den Planungsstellen bestehen keine Hierarchien:** In einem Unternehmensnetzwerk kann nicht von einer hierarchischen Struktur ausgegangen werden. Das Koordinationsverfahren sollte sich möglichst den organisatorischen Gegebenheiten des Netzwerkes anpassen, so dass sich eine *strukturellen Analogie* des Informationssystems zur organisatorischen Ausgestaltung des Netzwerkes ergibt.
- **Die Lösung vor Ort muss berücksichtigen, dass ein Unternehmen in mehreren Netzen beteiligt sein kann:** Im Kontext von Unternehmensnetzwerken ist es immer möglich, dass eine Planungsstelle in mehreren Netzwerken gleichzeitig agiert. Eine solche Planungsstelle muss in der Lage sein sich in mehreren Produktionsnetzen mit anderen Planungsstellen zu koordinieren. Dabei muss sichergestellt sein, dass für jedes Produktionsnetzwerk die Existenz der anderen Netzwerke transparent bleibt.
- **Möglichkeit der Optimierung des Gesamtsystems unter der Berücksichtigung der lokalen Autonomie:** Die Planungsstellen haben eigene Interessen und sind evtl. für ihr wirtschaftliches Ergebnis verantwortlich. Sie werden also in erster Linie versuchen ihr lokales Ergebnis zu verbessern. Unter der Nichtbeachtung von strategischem Verhalten werden sie also nicht dazu bereit sein für eine Verbesserung eines Netzwerkergebnisses lokale Einbußen hinzunehmen.

– **Netzwerke sind dynamisch:** Netzwerke können sich nicht nur kurzfristig bilden, sie können sich auch in ihrer Ausrichtung und Zusammensetzung verändern. Es können neue Netzwerkpartner hinzukommen, die integriert werden müssen, und andere Netzwerkmitglieder können das Netzwerk verlassen. Das Koordinationsverfahren sollte entsprechend flexibel sein, um schnell an geänderte Netzwerkstrukturen angepasst werden zu können.
– **Möglichkeit der Umplanung im Netzwerk:** Um eine wirkungsvolle Koordination zu ermöglichen, sollte es auch möglich sein, eine reaktive Planung durchzuführen, so dass der netzwerkweite Plan möglichst aktuell ist und auch umgesetzt werden kann.
– **Planungssysteme als Blackbox:** Die Planungssysteme werden auf der Ebene der Planungsstellen als Blackboxen gesehen. Diese Sichtweise bietet eine Reihe von Vorteilen:
  • Anpassungen an lokale Planungssysteme könnten sonst sehr aufwändig sein, insbesondere in heterogenen Planungssystemlandschaften.
  • Das vorhandene lokale Planungssystem kann weiter verwendet werden.
  • Es ergibt sich ein modularer Aufbau bei dem das Planungsmodul leicht ausgetauscht oder verändert werden kann.

  Dem Koordinationsverfahren stehen lediglich die Eingaben (die Plandaten) des Planungssystemes und die daraus erstellten Ausgaben (der Plan) für die Koordination zur Verfügung. Das Koordinationssystem ist aber durchaus in der Lage die Plandaten zu verändern, was für die planungsstellenübergreifende Koordination auch notwendig ist.

Bisherige Koordinationsverfahren (hierarchische als auch heterarchische) sind nicht in der Lage die Koordination unter Berücksichtigung dieser Anforderungen durchzuführen. Dies wurde in der Arbeit [2] unter der exemplarischen Betrachtung der Verfahren: APS (z.B. [3]), MUST [4], marktliche Koordination (etwa [5]) und DISPOWEB [6] gezeigt.

3 Modellierung des Lösungsansatzes

3.1 Modellierung eines Produktionsnetzwerkes

Unter einer Planungsstelle wird, wie bereits erläutert, eine autonome Einheit verstanden, die ihren eigenen Zielsetzungen folgend agiert. Um diese Autonomie zu erhalten, werden die Planungsstellen mittels Software-Agenten, so genannten Planungsstellenagenten, modelliert. Der jeweilige Agent kapselt das lokal eingesetzte Planungssystem, wie dies in Abbildung 1 skizziert ist, und vertritt die Interessen der Planungsstelle im Rahmen des Koordinationsverfahrens.
Eine Planungsstelle kann in mehreren Netzwerken engagiert sein. Sie kann Produktionsnetzwerken beitreten und sie wieder verlassen. Um dies modellieren zu können, benötigt jedes Produktionsnetzwerk einen stabilen Kern, ein Grundgerüst, das einem Netzwerk fest zugeordnet ist. Aus diesem Grund wird eine weitere Komponente, der Kommunikationsserver, eingeführt. Ein Kommunikationsserver ist fest für ein Netzwerk zuständig. Er stellt die Infrastruktur für ein

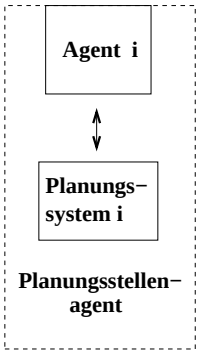

Abbildung 1. Skizze eine Planungsstellenagenten.

Netzwerk bereit. Ein Produktionsnetzwerk wird so durch eine Menge von Agenten und einem Kommunikationsserver abgebildet. Dies ist schematisch in Abbildung 2 dargestellt. Mit der Abbildung 2 soll nicht der Eindruck erweckt werden,

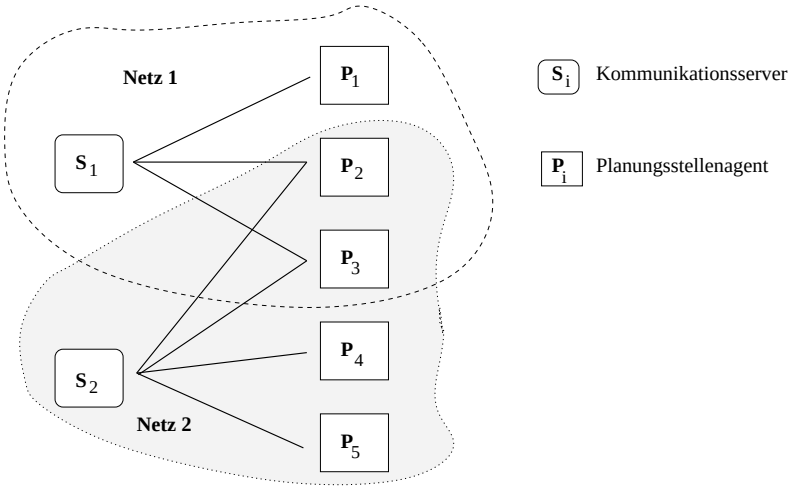

Abbildung 2. Skizze der Modellierung von Produktionsnetzwerken.

dass zwischen den Planungsstellenagenten keine Kommunikation möglich ist. Die Darstellung dieser Kommunikationsmöglichkeiten wurde in dieser Abbildungen ausschließlich aus Gründen der Übersichtlichkeit weggelassen. Ebenso soll nicht der Eindruck erweckt werden, dass die Planungsstellenagenten den Kommunikationsservern untergeordnet sind.

3.2 Aufbau eines Planungsstellenagenten

Bevor die Koordinationsverfahren erläutert werden, soll zuerst auf den internen Aufbau eines Agenten eingegangen werden. Der prinzipielle Aufbau eines Planungsstellenagenten ist in Abbildung 3 dargestellt. Die einzelnen Komponenten werden im Folgenden kurz erläutert.

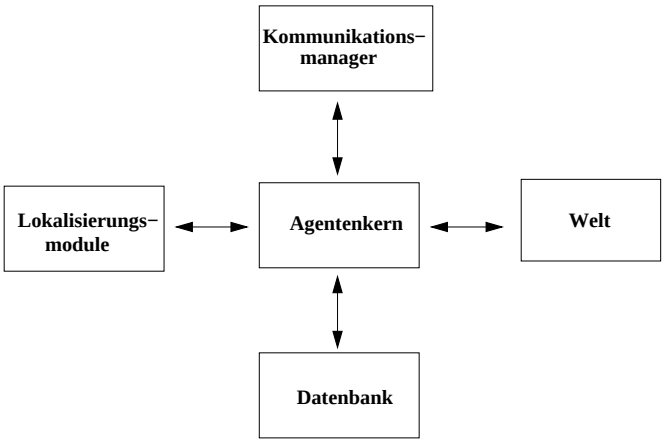

Abbildung 3. Grobe Darstellung des Agentenaufbaus.

- **Lokalisierungsmodule:** Da es einer Reihe von Anpassungen des Agenten an seine Umwelt bedarf, um seine Aufgaben im Sinne seines Eigners, einer Planungsstelle, zu erfüllen, muss dieser den Agenten an seine Bedürfnisse und seine Eigenheiten anpassen. Die Lokalisierung wird durch verschiedene Module vorgenommen, die der Agent im Rahmen seiner Initialisierung nachlädt. Es handelt sich dabei um folgende Module:
  - **Modul-Blackbox:** Dieses Modul stellt die Funktionalitäten zur Planerstellung und Bewertung zur Verfügung, und nutzt hierfür das lokale Planungssystem.
  - **Modul-Vorschlagsgenerator:** Dieses Modul erstellt im Rahmen der Planverbesserung, die später in Abschnitt 4.3 beschrieben wird, Verbesserungsvorschläge für die lokalen Planungsvorgaben.
  - **Modul-Produktionsnetz:** Damit ein Agent die Interessen seines Eigners vertreten kann, muss er diese kennen. Für jedes Netzwerk, indem die Planungsstelle engagiert ist, wird ein entsprechendes Modul angegeben, in dem die Angebotsstrategie der Planungsstelle vorgegeben wird.
- **Welt:** Jeder Agent besitzt eine interne Repräsentation seiner Umwelt. In dieser Repräsentation sind alle für den Agenten relevanten Daten gespeichert, dabei handelt es sich etwa um den Ablaufplan, die Produkte, die gefertigt werden können, oder die lokalen Aufträge.

– **Kommunikation:** Dieses Modul ist für den Nachrichtenaustausch mit anderen Komponenten (Agenten und Servern) verantwortlich.
– **Datenbank:** Die Datenbank wird zur Initialisierung und zur dauerhaften Speicherung der erzeugten Daten verwendet.
– **Der Agentenkern:** Der Agentenkern hat drei Aufgaben, nämlich die Initialisierung des Agenten, die Verarbeitung eingehender Nachrichten und die Durchführung von Planverbesserungsdiskussionen.

4 Koordinationsverfahren

Zuerst soll an dieser Stelle das genutzte Produktionsmodell vorgestellt werden:

– Für jeden Auftrag soll eine bestimmte Menge eines Produktes hergestellt werden.
– Für jedes Produkt gibt es mehrere alternative Herstellungsvarianten.
– Jede Variante besteht aus einer sequentiellen Folge von Aktivitäten.
– Jede Aktivität wird auf einer Ressource durchgeführt, wobei diese aus einer Menge alternativer Ressourcen ausgewählt werden kann, wobei sich die Bearbeitungsdauer bei den Ressourcen unterscheiden können.

Bevor die Planungsverfahren beschrieben werden, soll kurz auf die Initialisierung eines Netzwerkes eingegangen werden.
Tritt ein Agent einem Produktionsnetzwerk bei, muss er sich bei dem entsprechenden Kommunikationsserver anmelden. Hierzu übermittelt er seinen Namen und die Namen der Leistungen[1], die er in dem Netzwerk anbietet. Diese werden beim Server vermerkt, so dass der Server immer eine Liste der Agenten hat, die dem Netzwerk angehören, und über die Information verfügt, welche Leistungen diese jeweils anbieten.
Durch die Koordination der Planungsstellen wird ein netzwerkweiter Plan erstellt. Es kann also, wie bei anderen Planungsverfahren üblich, zwischen der prädiktiven und reaktiven Planung unterschieden werden.

4.1 Prädiktive Planung

Mit der prädiktiven Planung wird nach der Netzwerkbildung ein Plan erstellt. Erst wenn ein solcher initialer Plan für das Netzwerk vorhanden ist, ist dies vollständig initialisiert. In der prädiktiven Planung wird für jeden Auftrag eine Einplanung durchgeführt. Bevor jedoch eine Einplanung eines Auftrags durchgeführt werden kann, muss überprüft werden, ob der Auftrag überhaupt einplanbar ist. Hierfür muss mindestens eine Herstellungsvariante des benötigten Produktes durchführbar sein. Eine Herstellungsvariante ist durchführbar, wenn für jede Aktivität mindestens ein Agent im Netzwerk existiert, der die entsprechende Leistung anbietet. Der Ablauf für die Einplanung eines Auftrag wird im folgenden kurz beschrieben:

[1] Unter einer Leistung wird die Durchführung einer Aktivität verstanden.

1. Für jede durchführbare Herstellungsvariante des zu erstellenden Produktes wird sequentiell für jede Aktivität ermittelt welcher Planungsstellenagent diese Aktivität am besten durchführen kann. Dabei wird ein Verfahren ähnlich dem Contract-Net verwendet.
   (a) Der Kommunikationsserver fungiert als unparteiischer Auktionator. Er schreibt die jeweilige Aktivität aus, und gibt dabei ein grobes Zeitfenster an, in dem die Durchführung der Aktivität zu geschehen hat. So kann sichergestellt werden, dass die Ausführung der Aktivitäten ohne Überschneidung stattfindet. Da der Kommunikationsserver weiß, welche Aktivitäten von welchem Agenten angeboten werden, können die Ausschreibungen direkt an die entsprechenden Agenten geschickt werden.
   (b) Die Agenten, denen eine Aktivität angeboten wurde, ermitteln, ob sie diese Aktivität durchführen wollen, und wenn sie sie durchführen wollen, zu welchen Konditionen sie dies tun. Abhängig von ihrer lokalen Entscheidung können sie ein Angebot für die Durchführung der angebotenen Aktivität abgeben. In diesem geben sie ein Zeitfenster an, indem sie die Aktivität durchführen wollen.
   (c) Der Auktionator wartet eine bestimmte Zeit auf die Abgabe von Geboten und merkt sich, welches Angebot ihm für diese Aktivität am Besten erscheint.
   (d) Anhand des ausgewählten Angebots kann nun der Starttermin der folgenden Aktivität berechnet werden und bei Schritt a) für die nächste Aktivität der Herstellungsvariante fortgefahren werden.
2. Nachdem für alle durchführbaren Herstellungsvarianten eine mögliche Verteilung der Aktivitäten ermittelt wurde, kann die geeignetste/beste Variante vom Server ausgewählt werden.
3. Alle Agenten, die für die Aktivitäten der gewählten Herstellungsvariante vorgemerkt wurden, werden gebeten ihr Angebot noch einmal zu bestätigen. Dies ist notwendig, da die Einholung aller Angebote einige Zeit in Anspruch nimmt, in der sich die Situation bei den Agenten ändern kann. Bestätigen die Agenten ihr Angebot, erhalten sie den Zuschlag für ihr Angebot.
4. Die Agenten bestätigen ebenfalls noch einmal ihr Angebot zum Vertragsabschluss. Andernfalls können sie ihr Angebot zu diesem Zeitpunkt noch zurückziehen, was zu einer erneuten Verhandlung über diesen Auftrag führen würde. Bestätigen alle Agenten ihr Angebot, gilt die Verhandlung über diesen Netzwerk-Auftrag als abgeschlossen und die getroffenen Vereinbarungen werden im netzwerkweiten Plan festgehalten.

4.2 Reaktive Planung

Die reaktive Planung wird als Reaktion auf Ereignisse durchgeführt, die die Planungsdaten signifikant verändern. Dabei kann es sich zum Einen um Ereignisse auf der Ebene der Planungsstellen zum Anderen um Ereignisse auf der Netzwerkebene handeln. Treten solche Ereignisse auf, muss eine Nachverhandlung der Planungsstellenagenten erfolgen, um wieder einen gültigen Plan zu erhalten,

der alle Netzwerk-Aufträge berücksichtigt.
Für das Auslösen der reaktiven Planung kann es zwei mögliche Ursachen geben:

1. Eine Planungsstelle ändert eine getroffene Zusage im netzwerkweiten Plan, indem sie entweder einen neuen Zeitrahmen für die Tätigkeit anbietet oder die Durchführung der Aktivität ganz absagt.
2. Plandaten der Netzwerk-Aufträge verändern sich nachträglich, wodurch der netzwerkweite Plan ungültig wird. Dies kann etwa geschehen, weil das Zeitfenster eines Netzwerk-Auftrags sich verschoben hat oder ein Auftrag abgesagt wurde. Dagegen würde das Beitreten eines neuen Netzwerkmitglieds keine reaktive Planung erfordern. Der vorhandene netzwerkweite Plan könnte in einer solchen Situation zwar evtl. optimiert werden, der Erfüllungsgrad der an das Netzwerk gestellten Anforderungen würde dadurch aber nicht verändert.

Bei der reaktiven Planung kann es, abhängig vom auslösenden Ereignis, zu unterschiedlichen Abläufen im Koordinationssystem kommen, die im Folgenden vorgestellt werden:

- **Eine Aktivität wird von einer Planungsstelle zeitlich verschoben:** In diesem Fall muss der sich dadurch ergebende netzwerkweite Plan auf seine Gültigkeit hin untersucht werden. Ist der Plan noch gültig, braucht keine reaktive Planung durchgeführt werden, sondern der netzwerkweite Plan muss aktualisiert werden. Andernfalls wurde die Aktivität zeitlich so verschoben, dass dadurch eine Überschneidung mit der Vorgänger- oder Folgeaktivität entsteht. In einem solchen Fall wird die Aktivität, mit der die Überschneidung besteht, abgesagt und eine Planungsstelle gesucht, die diese Aktivität zum schellstmöglichsten Zeitpunkt durchführen kann, so dass wieder ein gültiger Plan entsteht. Eventuell müssen so nacheinander eine Reihe von Aktivitäten verschoben werden. Kann mit dieser Strategie kein gültiger netzwerkweiter Plan hergestellt werden, etwa weil die Zeitschranken des Netzwerk-Auftrags nicht eingehalten werden, muss der betroffene Auftrag insgesamt noch einmal neu verhandelt werden.
- **Eine Aktivität wird von einer Planungsstelle abgesagt:** Der netzwerkweite Plan wird dadurch ungültig. Alle eingeplanten Aktivitäten, die zu diesem Auftrag gehören, werden abgesagt, und somit aus dem netzwerkweiten Plan ausgeplant. Anschließend findet mittels der prädiktiven Planung für diesen Auftrag eine neue Verhandlung statt.
- **Verändern von Plandaten bisheriger Aufträge auf Netzwerkebene:** Wurde der Auftrag abgesagt, müssen die eingeplanten Aktivitäten abgesagt werden. Ansonsten muss geprüft werden, ob die geänderten Plandaten zu einem ungültigen Plan führen. Dies ist nicht zwangsläufig der Fall, so wird etwa durch das Verändern einer Herstellungsvariante, die bei der Einplanung der Aufträge nicht berücksichtigt wurde, der bestehende Plan nicht ungültig. Ist der netzwerkweite Plan allerdings ungültig geworden, werden die betroffenen Aufträge bzw. ihre eingeplanten Aktivitäten abgesagt. Anschließend erfolgt dann eine erneute Einplanung dieser Aufträge auf der Basis der neuen Plandaten.

– **Hinzufügen eines Netzwerk-Auftrags:** In diesem Fall wird eine Einplanung des neuen Netzwerk-Auftrages durchgeführt.

4.3 Planverbesserung

Eine Besonderheit des ABAKO-Verfahrens ist die Möglichkeit einen bestehenden Plan zu verbessern. Diese Verbesserungen werden durch Verhandlungen der Agenten über den bestehenden Plan erreicht. Diese Verhandlungen werden als Verbesserungsdiskussionen bezeichnet.
Die Grundidee der Verbesserungsdiskussionen beruht auf den folgenden Annahmen:

– Die Planungsstellen wollen ihre lokalen Pläne möglichst optimal gestalten.
– Die lokalen Optimierungspotentiale der Planungsstellen werden durch die gegenüber dem Netzwerk getroffenen Zusagen eingeschränkt.

Die erste Annahme ist offensichtlich. Die zweite Annahme ist nicht ganz so einfach nachzuvollziehen, da die Planungsstelle die Zusagen gegenüber dem Netzwerk ja freiwillig gemacht haben. Da allerdings von dynamischen Plandaten ausgegangen wird, kann es durch Veränderung der Plandaten der Planungsstelle dazu kommen, dass die gemachten Zusagen die Güte der erstellten Pläne einschränken.
Die Aufgabe der Planverbesserung besteht darin, die lokal vorhandenen Optimierungspotentiale der Planungsstellen aufzudecken. Durch diese lokalen Verbesserungen soll auch der netzwerkweite Plan insgesamt verbessert werden. Deswegen muss sichergestellt werden, dass es durch eine Verbesserung bei einer Planungsstelle nicht zu Verschlechterungen bei anderen Planungsstellen und so zu einer Verschlechterung des gesamten netzwerkweiten Plans kommt.
Um eine Verbesserung zu erreichen, muss eine Planungsstelle zuerst eine Veränderung einer getroffenen Zusage finden, die ein Verbesserungspotential freisetzt. Eine solche Veränderung bezieht sich somit immer auf eine zugesagte Aktivität. Einer Planungsstelle stehen bei der Suche nach Verbesserungen prinzipiell folgende mögliche Veränderungen einer Zusage zur Verfügung:

– Eine Planungsstelle kann versuchen die Durchführung der Aktivität auf eine andere Planungsstelle zu verschieben, um so die eigenen Ressourcen zu entlasten.
– Eine Planungsstelle kann versuchen die Durchführung der Aktivität früher zu beginnen oder später abzuschließen, um so etwa einen Engpass in der lokalen Planung zu entlasten.

Die Planungsstelle kann versuchen durch das Anwenden einer solchen Veränderungsstrategie lokale Vorgaben zu finden, deren Umsetzung einen besseren lokalen Plan ermöglichen. Die effektive Suche nach Verbesserungsvorschlägen kann durch lokales Wissen und Erfahrungen deutlich erleichtert werden, so dass die Erstellung von Verbesserungsvorschlägen durch ein Lokalisierungsmodul vorgesehen ist. Sofern ein Verbesserungsvorschlag gefunden wurde, versucht der

Planungsstellenagent diese Veränderungen im Netzwerk durchzusetzen. Die Aktivität, deren Zusage die Planungsstelle verändern möchte, besitzt allerdings Abhängigkeiten mit anderen Aktivitäten, die von anderen Planungsstellen ausgeführt werden. Aus diesem Grund muss der Planungsstellenagent mit anderen Agenten über die Veränderung der Zusage für die Ausführung der Aktivität diskutieren.

Diskussionsregeln Im Rahmen einer Verbesserungsdiskussion kann es zu ungewünschten Effekten kommen. Dabei handelt es sich etwa um

- die Diskussion aller Zusagen, was einer Neuplanung entspricht,
- die Diskussion auf Basis noch nicht beschlossener Veränderungen
- und zyklischen oder widersprüchlichen Veränderungswünschen im Verlauf einer Diskussion.

Um solch unerwünschte Effekte zu vermeiden, müssen Regeln für den Diskussionsverlauf aufgestellt werden. Folgende Regelmenge verhindert diese Effekte:

- Eine Verbesserungsdiskussion darf immer nur Aktivitäten eines Auftrags betreffen.
- In einem Produktionsnetzwerk darf zu jedem Zeitpunkt immer nur eine Verbesserungsdiskussion geführt werden.
- An einer Verbesserungsdiskussion dürfen nur die Agenten der direkten Vorgänger- bzw. Nachfolgeraktivität, der Kommunikationsserver sowie der Agent der die Diskussion initiiert hat teilnehmen.

Diese Regeln schränken die möglichen Verbesserungsdiskussionen stärker ein, als dies zur Vermeidung der unerwünschten Effekte nötig wäre. Diese restriktive Regelmenge wurde gewählt, da bisher nicht ermittelt wurde, welches die am wenigsten restriktive Regelmenge ist, die hinreichend ist, um alle unerwünschten Effekte zu vermeiden.
Um sicherzustellen, dass die Einplanung von Netzwerk-Aufträgen nicht durch Verbesserungsdiskussionen erschwert oder verhindert wird, dürfen diese in einem Netzwerk nur begonnen werden, wenn das Netzwerk vollständig initialisiert ist, d. h. alle Netzwerk-Aufträge eingeplant wurden.
Tritt während einer Verbesserungsdiskussion ein Ereignis ein, das eine reaktive Planung erfordert, muss die Verbesserungsdiskussion abgebrochen werden.

Ablauf einer Verbesserungsdiskussion Ausgangspunkt für eine Verbesserungsdiskussion bildet ein Agent, der eine Veränderung einer seiner Zusagen durchsetzen möchte. Dieser Agent initiiert eine Verbesserungsdiskussion. Je nachdem, ob im Rahmen dieser Diskussion die Durchführung einer Aktivität zeitlich oder auf eine andere Planungsstelle verschoben werden soll, ergeben sich unterschiedliche Diskussionsabläufe. Zuerst wird der Diskussionsablauf für eine zeitliche Verschiebung einer Zusage vorgestellt. Anschließend wird der Diskussionsablauf der Diskussion einer Verschiebung einer Zusage auf eine andere Planungsstelle beschrieben.

Diskussionsablauf bei einer zeitlichen Verschiebung einer Zusage
Der Diskussionsablauf soll mit Hilfe der Abbildung 4 erklärt werden.

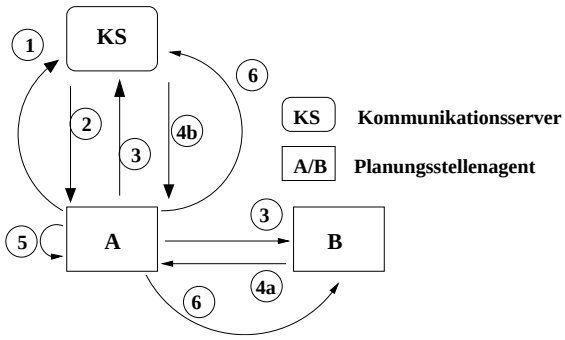

Abbildung 4. Skizze eines Diskussionsablaufs bzgl. einer zeitlichen Verschiebung.

1. Ein Agent findet eine Veränderung und bewertet die sich daraus ergebende Verbesserung seines lokalen Planes. Der Agent stellt beim Kommunikationsserver einen Antrag auf die Durchführung einer Verbesserungsdiskussion zur zeitlichen Verschiebung einer Aktivität.
2. Sofern nicht bereits eine andere Verbesserungsdiskussion im Netzwerk läuft, wird die beantragte Verbesserungsdiskussion zugelassen und dem anfragenden Agenten wird mitgeteilt, welche Agenten die Vorgänger- und Nachfolgeraktivität im Rahmen der Herstellungsvariante des entsprechenden Auftrags durchführen.
3. Der Agent, der die Verbesserungsdiskussion initiiert hat, muss sich nun mit den beiden Agenten, die die Vorgänger- oder Nachfolgeraktivität durchführen, über die Veränderung der Zusage einigen. Er muss diese nach ihrer jeweiligen Bewertung der beabsichtigten Veränderung befragen. Existiert keine entsprechende Vorgänger- oder Nachfolgeraktivität, handelt es sich bei der zu diskutierenden Aktivität um die erste bzw. die letzte Aktivität der Herstellungsvariante. In diesem Fall muss anstatt des entsprechenden Agenten der Kommunikationsserver zu der Bewertung der Veränderung befragt werden.
4. Es sind nun zwei Fälle zu betrachten, je nachdem ob ein Agent oder der Kommunikationsserver nach einer Bewertung gefragt wurde.
   (a) Ein Agent wurde befragt:
   Der Agent erstellt eine Bewertung des Veränderungswunsches auf Grund der Auswirkungen dieser Veränderung auf seinen lokalen Plan. Er kann anschließend auf Grund seiner Bewertung zwischen den folgenden möglichen Antworten wählen:

- Der Agent lehnt die Veränderung ab, etwa weil sie in seinem Plan Auswirkungen auf andere Aktivitäten hätte oder eine für ihn nicht akzeptable Verschlechterung seines Plans verursachen würde.
- Der Agent kann eine negative Bewertung abgeben. Er zeigt damit an, in welchem Ausmaß sich sein lokaler Plan durch die Veränderung verschlechtert, aber auch dass er bereit wäre diese Verschlechterung zu akzeptieren, wenn sich insgesamt eine Verbesserung ergibt. In einem solchen Fall könnte er dann mit einer Entschädigung rechnen.
- Der Agent gibt eine positive Bewertung ab. Er zeigt damit an, in welchem Ausmaß sich sein lokaler Plan durch die vorgeschlagene Veränderung verbessert.

(b) Der Kommunikationsserver wurde befragt:
Der Kommunikationsserver trifft seine Bewertung auf Grund der Vorgaben des Zeitfensters des entsprechenden Netzwerk-Auftrags. Der Server kann eine Veränderung ablehnen, wenn durch die Veränderung das Zeitfenster des Netzwerk-Auftrags nicht eingehalten wird. Ansonsten stimmt der Server den Veränderungen zu.

5. Nachdem die befragten Einheiten ihre Bewertungen abgegeben haben, kann die Auswertung der Diskussion durchgeführt werden. Hat ein Diskussionsteilnehmer die Veränderung abgelehnt, wird die Veränderung nicht durchgeführt, die bisher bestehende Vereinbarung bleibt weiterhin gültig. Haben alle Einheiten eine Bewertung der Veränderung abgegeben, können die Auswirkungen auf das gesamte Netzwerk ermittelt werden. Die Bewertungen der Veränderung werden vom Agenten, der die Diskussion initiiert hat, aufaddiert. Anhand der Summe dieser Bewertung sind zwei Fälle zu unterscheiden:
   - Ist die Summe der Bewertungen negativ oder gleich Null wird die Veränderung abgelehnt.
   - Ist die Summe der Bewertungen positiv wird die Veränderung angenommen.
6. Abhängig vom Ausgang der Diskussionsauswertung sind nun zwei Schritte möglich.
   (a) Die Veränderung wurde abgelehnt.
   In diesem Fall teilt der Agent, der die Verbesserungsdiskussion initiiert hatte, dem Server mit, dass die Diskussion ohne eine Veränderung beendet wurde.
   (b) Die Veränderung wurde beschlossen.
   Der Agent, der die Verbesserungsdiskussion initiiert hatte, teilt sowohl dem Server als auch den Agenten, die die benachbarten Aktivitäten durchführen, mit, dass die Veränderung durchgeführt wird. Die benachrichtigten Einheiten müssen daraufhin ihren jeweiligen Plan aktualisieren. Mit der Mitteilung der Veränderung wird gegenüber dem Server auch das Ende der Diskussion angezeigt.

Diskussionsablauf bei einer Verschiebung einer Zusage auf eine andere Planungsstelle

1. Der Agent, der eine Veränderung durchsetzen will, beantragt eine Verbesserungsdiskussion und teilt dem Server mit, dass er eine Aktivität verschieben will.
2. Wenn keine andere Diskussion zur Zeit im Netzwerk läuft, sendet der Server eine Bestätigung und eine Liste von Agentennamen, die die zu verschiebende Leistung ebenfalls anbieten. Wie bei der vorher beschriebenen Diskussion sendet der Server auch die Namen der Agenten, die die Vorgänger- und Nachfolgeraktivität der zu verschiebenden Aktivität durchführen.
3. Der Agent, der die Diskussion initiiert hat, bittet nun die ihm genannten Agenten um Angebote für eine alternative Durchführung der zu verschiebenden Aktivität.
4. Die befragten Agenten können ein Angebot für die Durchführung der Aktivität vorlegen, wenn sie sich dadurch eine Verbesserung ihres lokalen Plans versprechen. Die Agenten müssen allerdings im Rahmen der Verbesserungsdiskussion auch angeben, um wie viele Einheiten der Bewertungsmetrik sich ihr lokaler Plan dadurch verändern würde.
5. Da diese Angebote nicht im gleichen Zeitraum liegen müssen wie die zu verschiebende Aktivität, muss im weiteren Verlauf der Diskussion noch über eine zeitliche Verschiebung verhandelt werden. Der hierfür vorgesehene Diskussionsverlauf wurde bereits beschrieben. Der weitere Verlauf würde also dem vorherigen Diskussionsverlauf ab Punkt 3 entsprechen.

Auf Basis dieser beiden Diskussionsabläufe können in einem Produktionsnetzwerk Verbesserungsdiskussionen durchgeführt werden, mit denen Verbesserungen für die Planungsstellen erreicht werden können. Durch diese Verbesserungen werden aber auch Verbesserungen für das gesamte Netzwerk erreicht.

5 Resümee

Das hier vorgestellte ABAKO-Verfahren zeigt, dass eine Koordination von Planungssystemen erfolgen kann, wenn

- nur die nötigsten Informationen zur Koordination genutzt werden dürfen.
- die lokalen Planungssysteme als Blackboxen gesehen werden.
- die Autonomie der Planungsstellen anerkannt und im Koordinationsverfahren berücksichtigt wird.

Durch den Entwurf des ABAKO-Verfahrens und der im Rahmen meiner Diplomarbeit [2] erfolgten Implementierung des ABAKO-Systems konnte gezeigt werden, dass eine Koordination unter Berücksichtigung dieser Anforderungen möglich ist.

Insbesondere aus der Berücksichtigung der Anforderung, dass ein Unternehmen in mehreren voneinander unabhängigen oder gar konkurrierenden Netzwerken engagiert sein kann, resultieren weitere Probleme bei der Koordination von Planungssystemen, die nicht vollständig geklärt werden konnten. Das ABAKO-Verfahren bietet aber einen guten Ausgangspunkt, um die mit dieser neuen

Fragestellung verbundenen Untersuchungen durchzuführen und ist für daraus resultierende Erweiterungen offen.

Literatur

1. Corsten, H., Gössinger, R.: Unternehmensnetzwerke: Grundlagen - Ausgestaltungsformen - Instrumente. Schriften zum Produktionsmanagement 38, Lehrstuhl für Produktionswirtschaft Universität Kaiserslautern (2001)
2. Schumann, R.: Ein agentenbasiertes Verfahren zur Koordination von Planungssystemen. Diplomarbeit, Carl von Ossietzky Universität Oldenburg (2004)
3. Stadtler, H., Kilger, C., eds.: Supply Chain Management and Advanced Planning: concepts, models, software and case studies. 2. edn. Springer, Berlin et al. (2002)
4. Sauer, J.: Multi-Site Scheduling Hierarchisch koordinierte Ablaufplanung auf mehreren Ebenen. Habilitationsschrift, Carl von Ossietzky Universität Oldenburg (2002)
5. Schmidt, C.: Marktlich Koordination in der dezentalen Produktionsplanung Effizenz - Komplexität - Performance. Galber, Wiesbaden (1999)
6. Frey, D., Stockheim, T., Woelk, P.O., Zimmermann, R.: Integrated multi-agent-based supply chain management. In: Proc. 5th. Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise (WET ICE'03), IEEE, IEEE Computer Society Press (2003) 24 – 29

Incremental Hashing in State Space Search

Stefan Edelkamp and Tilman Mehler

Computer Science Department
Baroper Str. 301
University Dortmund
{stefan.edelkamp,tilman.mehler}@cs.uni-dortmund.de

Abstract. State memorization is essential for state-space search to avoid redundant expansions and *hashing* serves as a method to, address store and retrieve states efficiently.
In this paper we introduce *incremental state hashing* to compute hash values in constant time. The method will be most effective in guided depth-first search traversals of state space graphs, like in IDA*, where the computation of the set of successors and their heuristic estimates is extremely fast: heuristic values are often computed incrementally or retrieved from pre-computed pattern database tables, and backtracking keeps the changes in the state representation vector during the exploration small. The approach quickly decides if a given state is not present in a hash table, and accelerates successful search. It can further accelerate perfect hashing for pattern storage and look-up. If, for a better coverage of the state space, partial search methods without collision resolving is used, we establish another benefit for incremental state hashing.
We exemplify our considerations in the (n^2-1)-Puzzle, in action planning, and conduct experiments in Atomix.

1 Introduction

A *dictionary* over a universe $\mathcal{U}$ of possible keys is a partial function from a subset $\mathcal{R} \subseteq \mathcal{U}$, the *stored keys*, to some set I, the associated information. The *dictionary problem* consists of providing a data structure with the operations *insert*, *search*, and *delete*. In search applications, deletion is not always necessary. The slightly easier *membership* problem neglects any associated information. There are two major techniques for implementing dictionaries: *(balanced) search trees* and *hashing*. The former class of algorithms can achieve all operations in $O(\log n)$ worst case time and $O(n)$ storage space, where $n = |\mathcal{R}|$ is the number of stored elements.

In *state space hashing*, every state $S \in \mathcal{S}$ is assigned to a key $k(S)$, which is a part of the representation that uniquely identifies S. Note that every state representation can be interpreted as a binary integer number. Not all integers in the universe will correspond to valid states. For simplicity, in the following we will identify states with their keys. The keys are mapped into a linear array $H[0..m-1]$, called the *hash table*. The surjective mapping $h : \mathcal{S} \to \{0, \ldots, m-1\}$ is called the *hash function*. The lack of injectiveness yields *address collisions*, i.e., different states are mapped to the same address.

The time to compute the hash address $h(S)$ is often neglected for linear space backtracking search [17], but it can in fact result in a bottleneck for the exploration. For

many state-space problems, the most effective exploration algorithm is IDA* which is often tuned to generate millions of states per second [16]. In some IDA* implementations and regular domains, state space hashing is fully ignored. However, uncaught duplicates states can yield an exponential increase in the number of expanded nodes and running time [19]. Some duplicate pruning techniques operate in constant time for each generated successor state. However, they rely on a very regular structure of the state space graph. For example, *finite state machine pruning* [30] incrementally detects, whether the operator suffix of path to a node is contained in a set of forbidden operator strings, and suits well to IDA*.

Node exploration efficiency for computing the heuristic estimate of a state is obtained either by an *incremental estimate design* or by using *pattern databases* [3] that correspond to fully traversed state space abstractions prior to the search. In the tables the state-to-goal distance for each abstract state is stored, which is a lower bound for the state-to-goal distance in original space. Pattern databases are itself hash tables and our proposal for incremental hashing does apply to accelerate lookup time. Other forms of partial state hashing are encountered in *pattern search*, where *penalty* databases are built, or *deadlock* tables, which reckon that, due to unsolvable pattern states, an original state is unsolvable. Improvements for pattern hashing are available with *incremental perfect hashing* that we will discuss. For this case, all dictionary operations run in $O(1)$.

Even if all other operations to generate a successor for a given state are of constant time, a non-incremental computation of a hash function for a state vector $S = (S_1, \ldots, S_k)$ accumulates to at least $O(k)$ time. The larger the state vector the better the savings that can be obtained by incremental state hashing. If no state is stored at a given hash table address this reduces the complexity for duplicate detection from $O(k)$ to $O(1)$. If, however, a state is stored at a given hash address, most collision strategies require to compare the state descriptions of the query state and the cached draft. In either case, storage and comparison of states can be attributed to $O(k)$ time. One possible solution to this problem is to choose *partial search* methods, like bit-state hashing, which lead to much larger hash table sizes and better state space coverage. Even though completeness is sacrificed, the expected error probability is small. For partial search, incremental hashing leads to $O(1)$ dictionary operations. For *hash compaction*, the hash function that computes a *fingerprint* of a state, can also be computed incrementally.

The paper is structured as follows. We first review hash functions based on computing remainders. Then we turn to the *Rabin-Karp* algorithm that was designed to accelerate pattern matching, and extend this method to state space search. We consider sliding-tile puzzles and STRIPS action planning as the running examples. With *Partial IDA** search we address larger and faster dictionaries for duplicate detection. The case study in Atomix chooses a PSPACE hard single-agent challenge for experiments. We address incremental and perfect hashing for pattern storage, as well as a global dynamic perfect hashing scheme. Finally, we reflect related work and draw conclusions.

2 Hash Functions

A *good* hash function is one that can be computed efficiently and minimizes the number of collisions. The returned addresses for given keys should be uniformly distributed, even if the set of chosen keys in $\mathcal{S}$ is not, which is almost always the case. To achieve

a uniform distribution of the keys mapped to the set of addresses, a random experiment can be performed. The *Lehmer*-generator [20] is a pseudo random number generator on the basis of linear congruences, and is one of the most common methods for generating random numbers. An improvement is suggested in [28], while the selection of good random number generators is analyzed in [26]. A sequence of pseudo-random numbers x_i is generated according to $x_0 = b$ and $x_{i+1} = (ax_i + c)$ *mod* m for $i \geq 0$. A good choice is the *minimal standard generator* with $a = 7^5 = 16,807$ and $m = 2^{31} - 1$. For implementing multiplication without overflows, one uses a factorization of m.

If one can extend $\mathcal{S}$ to $\mathbb{Z}$, then $\mathbb{Z}/m\mathbb{Z}$ is the *quotient space* with equivalence classes $[0], \ldots, [m-1]$ induced by the relation $z \sim w$ *iff* z *mod* $m = w$ *mod* m. Therefore, a mapping $h : \mathcal{S} \to \{0, 1, \ldots, m-1\}$ with $h(S) = S$ *mod* m distributes $\mathcal{S}$ on H. For uniformity, the choice of m is important: for example, if m is even then $h(S)$ is even if and only if S is. A good choice for m is a prime which does not divide a number $r^i \pm j$ for small j. For *linear recursive hash functions* that we will consider for incremental state hashing, states in $\mathcal{S}$ are interpreted as bit strings and divided into blocks of bits. For example blocks of byte-size yield 256 different *characters*.

3 The Algorithm of Rabin and Karp

The idea of incremental hashing originates in matching a text $T[1..n]$ to a pattern $M[1..m]$. In the *algorithm of Rabin and Karp* [15], a pattern M is mapped to a number $h(M)$, which fits into a single memory cell and can be processed in constant time. For $1 \leq j \leq n - m + 1$, we will check if $h(M) = h(T[j..j + m - 1])$. Due to possible collisions, this is not a sufficient but a necessary criterion for the match of M and $T[j..j + m - 1]$. A character-by-character comparison is performed only if $h(M) = h(T[j..j+m-1])$. To compute $h(T[j+1..j+m])$ incrementally in constant time, one takes value $h(T[j..j + m - 1])$ into account, according to Horner's rule for evaluating polynomials. Let q be a sufficiency large prime and $q > m$. We assume that numbers of size $q \cdot |\Sigma|$ fit into a memory cell, so that all operations can be performed with single precision arithmetic. To ease notation, we identify characters in Σ with their order. The algorithm of Rabin and Karp performs the matching process. The input is a string T and a pattern M. The output are occurrence of M in T, or -1. The algorithms works as follows

1. Initialize $p \leftarrow t \leftarrow 0$ and $u \leftarrow |\Sigma|^{m-1}$ mod q
2. Precompute hash value of pattern: for all $1 \leq i \leq m$ set $p \leftarrow (|\Sigma| \cdot p + M[i])$ mod q
3. Precompute hash value of $T[1..n]$: for all $1 \leq i \leq m$ set $t \leftarrow (|\Sigma| \cdot p + T[i])$ mod q
4. Iterate for all $1 \leq j \leq n - m + 1$:
   (a) If hash function matches, e.g. *if* $(p = t)$, check $(M, T[j..j + m - 1])$
   (b) If pattern found at position j: return j
   (c) Otherwise, if $j \leq n - m$, use Horner's rule to compute updated hash value $t \leftarrow ((t - T[j] \cdot u) \cdot |\Sigma| + T[j + m])$ mod q

The algorithm is correct by the following observation. At the start of the j-th iteration, the invariant $t_j = (\sum_{i=j}^{m+j-1} T[i]|\Sigma|^{m-i+j-1})$ mod q is true.

As an example take $\Sigma = \{0, \ldots, 9\}$ and $q = 13$. Furthermore, let $M = 31'415$ and $T = 2'359'023'141'526'739'921$. The mapping induced by function h yields the

following sequence of hash values: $8, 9, 3, 11, 0, 1, 7, 8, 4, 5, 10, 11, 7, 9$, and 11. We see h produces collisions. The incremental computation at $j = 8$ works as follows: $14'152 \equiv (31'415 - 3 \cdot 10'000) \cdot 10 + 2 \pmod{13} \equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \equiv 8 \pmod{13}$. The computation of all hash addresses has running time of $O(n + m)$, which is also the best case overall running time. In the worst case, the matching is still of order $\Omega(nm)$, as the task of searching pattern $M = 0^m$ in text $T = 0^n$ shows.

4 Incremental Hashing

Incremental state hash functions consist of a recursive function H and an address function A so that $h(S) = A(H(S))$, $S \in \mathcal{S}$. The idea of linear recursion is that the contribution of one symbol to the hash value is independent of the contribution of the others. Let T be a mapping from Σ to R, where R is a ring, $r \in R$. Similar to the algorithm of Rabin-Karp we compute the value H of string $S_i = (s_i, \ldots, s_{i+l})$ as

$$H(S_0) = \sum_{j=0}^{l-1} T(s_j)$$
$$H(S_i) = rH(S_{i-1}) + T(s_{i+l-1}) - r^n T(s_{i-1}), \quad \text{for } 1 \leq i \leq n.$$

With this we have $H(s_j) = \sum_{i=0}^{l-1} r^{l-i+j} T(s_{i+j})$.

The *prime-division* method, that we are mainly interested in, takes $R = \mathbb{Z}$ and assigns $h(S_i) = H(S_i)$ *mod* m. Here m is chosen as a suitable prime to obtain a good distribution. For faster computation, the *two-power division* method can be an option. The computation ignores higher order bits, such that the hash address can be computed by word-level bit-operations without any modulo operation, i.e. $a * b$ maps to a AND b, $a + b$ maps to a XOR b, and a *mod* b maps to a AND $2^i - 1$.

For state space search, we often have the case that a state transition changes only a small part of the representation. In this case, the computation of the hash function can be designed incrementally. The difference to the algorithms above is that changes can now occur within the state vector, and not only at its ends.

4.1 Puzzle Solving

Take for example the Fifteen-Puzzle with $\Sigma = \{0, \ldots, 15\}$. The natural vector representation for state S is $(t_0, \ldots, t_{15}) \in \Sigma^{16}$, where $t_i = l$ means that the tile labeled with l is located at position i, and $l = 0$ is the blank. The hash value of S is $h(S) = (\sum_{i=0}^{15} t_i \cdot 16^i) \bmod q$. Let state S' with representation $(t'_0, \ldots, t'_{15})$ be a successor of S. We know that there is only one transposition in the vectors t and t'. Let j be the position of the blank in S and k be the position of the blank in v. We have $t'_j = t_k$, $t'_k = 0$, and for all $1 \leq i \leq 16$, with $i \neq j, i \neq k$ it holds that $t'_i = t_i$. Therefore,

$$\begin{aligned} h(S') &= ((\sum_{i=0}^{15} t_i \cdot 16^i) - t_j \cdot 16^j + t'_j \cdot 16^j - t_k \cdot 16^k + t'_k \cdot 16^k) \bmod q \\ &= (((\sum_{i=0}^{15} t_i \cdot 16^i) \bmod q) - 0 \cdot 16^j + t'_j \cdot 16^j - t_k \cdot 16^k + 0 \cdot 16^k) \bmod q \end{aligned}$$

$$= (h(S) + (t'_j \cdot 16^j) \bmod q - (t_k \cdot 16^k) \bmod q) \bmod q.$$

To save time, we precompute $(l \cdot 16^k) \bmod q$ for $l, k \in \{0, \ldots, 15\}$. If we store $(i \cdot 16^j) \bmod q - (l \cdot 16^k) \bmod q)$ for each value of i, j and l, k we can save one more addition. As $h(S) \in [0..q-1]$ and $(t'_j \cdot 16^j) \bmod q - (t_k \cdot 16^k) \bmod q) \in [0..q-1]$. we may substitute the last mod q operation, by addition or subtraction. The savings are larger, when the state description vector grows. For the (n^2-1)-Puzzle non-incremental hashing results in $\Omega(n^2)$ time, while in incremental hashing the efforts remain constant.

The (n^2-1)-Puzzle has underwent different designs of heuristic functions. One lower bound estimate is the *Manhattan distance (MD)*. For every two states S and S', it is defined as sum of the vertical and horizontal distances for each tile. MD is consistent, since the differences in heuristic values between two states S and S' are 1, such that $|h(S') - h(S)| = 1$ and $h(S) - h(S') + 1 \geq 0$. The heuristic can be computed incrementally in $O(1)$ time, given a two-dimensional table, which is addressed by the label, the current direction and the position of the tile that is being moved.

Incremental hashing is possible for most single-agent search problems in vector representation, including *Rubiks Cube* [18] and *Sokoban* [14]. Its effectiveness, however, relies on two factors: on the *locality*, by means on how many state variables are affected by a state transition and on the *node expansion efficiency*, e.g. running time of all other operations to generate one successor state. In *Rubik's Cube*, exploiting locality is limited. If we represent position and orientation of each sub-cube as a number in the state vector, then for each twist 8 of the 20 entries will change. In *Sokoban* the node expansion efficiency is considerably small. During move execution the set of pushable stones has to be determined in linear time to the board layout, and the (incremental) computation of the minimum matching heuristic requires quadratic time in the number of stones. In the experiment we have a closer look to the *Atomix* problem.

4.2 Planning

STRIPS *action planning* refers to a world description in predicate logic. A number of predicates AP describes what can be true or false in each state of the world. By applying operations in a world, we arrive at another world, where different atoms might be true or false. For example, in a blocks world a robot might try to reach a target state by operators that stack and unstack blocks, or place them on the table. Usually, only some atoms are affected by an operator, and most of them remain the same. The syntax and the semantics of STRIPS planning are as follows.

Definition 1. *(STRIPS Planning Problem) A* propositional planning problem *(in STRIPS notation) is a finite state space problem $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, where $\mathcal{S} \subseteq 2^{AP}$ is the set of states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states, and $\mathcal{O}$ is the set of operators that transform states into states. Operators $O = (P, A, D) \in \mathcal{O}$ have propositional preconditions P, and propositional effects (A, D), where $P \subseteq AP$ is the* precondition list, *$A \subseteq AP$ is the* add list *and $D \subseteq AP$ is the* delete list. *Given a state S with $P \subseteq S$ then its successor $S' = O(S)$ is defined as $S' = (S \setminus D) \cup A$.*

It is not difficult to devise an incremental hashing scheme for STRIPS planning that bases on the idea of the algorithm of *Rabin and Karp*. For $S \subseteq AP$ we take

$h(S) = (\sum_{a_i \in S} 2^i) \bmod q$. The hash value of $S' = (S \setminus D) \cup A$ is computed as

$$\begin{aligned} h(S') &= \left(\sum_{a_i \in (S \setminus D) \cup A} 2^i \right) \bmod q \\ &= \left(\sum_{a_i \in S} 2^i - \sum_{a_i \in D} 2^i + \sum_{a_i \in A} 2^i \right) \bmod q \\ &= \left(\left(\sum_{a_i \in S} 2^i \right) \bmod q - \left(\sum_{a_i \in D} 2^i \right) \bmod q + \left(\sum_{a_i \in A} 2^i \right) \bmod q \right) \bmod q \\ &= \left(h(S) - \left(\sum_{a_i \in D} 2^i \right) \bmod q + \left(\sum_{a_i \in A} 2^i \right) \bmod q \right) \bmod q. \end{aligned}$$

Since $2^i \bmod q$ can be pre-computed for all $a_i \in AP$, we have a running time that is of order $O(|A| + |D|)$, which is constant for most STRIPS planning problems. It is also possible to meet the complexity of $O(1)$ if we store the values $\left(\sum_{a_i \in A} 2^i\right) \bmod q - \left(\sum_{a_i \in D} 2^i\right) \bmod q$ together with each operator. Either complexity is small, when compared with the planning state size $O(|S|)$. Since the set representation is more difficult to hash, the complexity for computing the hash address of S in STRIPS planning non-incrementally turns often out to be $O(|AP|)$ instead of $O(|S|)$, as expected. STRIPS planning problems can often compactly encoded with integers to which incremental hashing also applies.

5 Partial Search

The idea of erroneous dictionaries was exploited in *Bloom filters* [1]. It was proposed for use in the web context [22] as a mechanism for identifying which pages have associated comments stored. A Bloom filter is a bit vector v of length m, together with k independent hash functions $h_1(x), \ldots, h_k(x)$. Initially, v is set to 0. To *insert* a key x, compute $h_i(x)$, for all $i = 1 \ldots k$, and set each $v[h_i(x)]$ to 1. To *search* a key, check the status of $v[h_1(x)]$; if it is 0, x is not stored, otherwise continue with $v[h_2(x)], v[h_2(x)]$, etc. If all these bits are set, report that x is in the filter. However, since they might have been turned on by different keys, the filter can make *false positive* errors. *Deletions* are not supported by this data structure, but they can be incorporated by replacing the bits by *counters* that are incremented in insertions rather than just set to 1.

For large problem spaces, it can be most efficient to apply a depth-first search strategy in combination with duplicate detection via such a membership data structure. This is also the approach of the widely used the ACM awarded software model checker SPIN [11], which applies the so-called *Supertrace algorithm*. Since the domain often contains up to 2^{30} states and more, it resorts to approximate hashing. An attempt to remedy the incompleteness of partial search is to re-invoke the algorithm several times with different hash functions to improve the coverage of the search tree. This technique, called *sequential hashing*, successively examines various beams in the search tree (up to a certain threshold depth). *Bit-state hashing*, as found in the protocol validator SPIN,

comes with an analysis for coverage prediction. *Universal hashing* has been shown to have advantages in partial search [5]. The authors showed that the *ideal* circumstances for error prediction in sequential hashing are often not fulfilled in practice, and refine the model for coverage prediction to match the observation.

Like *single* and *double bit-state hashing* [12], *hash compaction* [29] aims at reducing the memory requirements for the state table. However, it stores a compressed state descriptor in a conventional hash table instead of setting bits corresponding to hash values of the state descriptor in a table of bits. The compression function c maps a state to a b-bit number in $\{0, \ldots, 2^b - 1\}$. Since this function is not surjective, i.e., different states can have the same compression, false positive errors can arise. Note, however, that if the probe sequence and the compression are calculated independently from the state, the same compressed state can occur at different locations in the table.

The apparent aspirant for partial search is IDA* with *transposition table* [27], since, in opposite to A*, it applies depth-first search and can exploit locality of move execution. It also tracks the solution path on the stack, which allows to omit the predecessor link in the state description. When substituting the transposition table H of already visited nodes in IDA* by bit-state, multi bit-state or hash compaction we establish the *Partial IDA** algorithm [13]. Since neither the predecessor nor the f-value are present, in order to distinguish the current iteration from the previous ones, the bit-state table has to be re-initialized in each iteration of IDA*. Refreshing large bit-vector tables is fast in practice, but for shallow searches with a small number of expanded nodes this scheme can be improved by invoking ordinary IDA* with transposition table updates for smaller thresholds and by applying bit-vector exploration in large depths only.

6 Case Study: Atomix

The goal of *Atomix* [13] (cf. Figure 1) is to assemble a given molecule from atoms. The player can select an atom at a time and *push* it towards one of the four directions left, right, up, and down; it will keep on moving until it hits an obstacle or another atom. The problem is solved when the atoms form the same constellation (the "molecule") as depicted beside the board. Atomix has been proven to be PSPACE complete [10]. Recall, that $(n^2 - 1)$-Puzzle solving is NP complete, while *Sokoban* and STRIPS planning are PSPACE complete. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

For solving Atomix problems we use IDA* search, which is much faster than A* search. The number of duplicates in the search is larger than in regular domains like the $(n^2 - 1)$-Puzzle, so state storage will be essential. The design of appropriate state abstractions for pattern database storage are not trivial. The problem is that by *stopper atoms*, sub-patterns of atoms can be unsolvable while the original problem is not.

For IDA* exploration Atomix has a global state representation that contains both the board layout in form of a two dimensional table and the coordinates for the atoms in form of a simple vector. This eases many computations during successor generation. After a recursive call the changes due to a move operation are withdrawn in a backtrack fashion. Consequently – disregarding the computation time for computing the heuristic estimate, the hashing efforts (computation of the function, state comparisons and copyings), and finding the correct location for an atom in a given move direction – successor

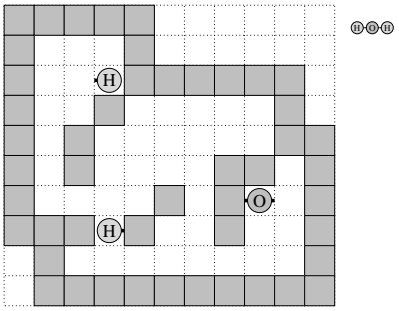

Fig. 1. One level of Atomix. The depicted problem in can be solved with 13 moves, where the atoms are numbered left-to-right in the molecule: 1 *down left*, 3 *left down right up right down left down right*, 2 *down*, 1 *right*.

generation in Atomix is a constant time operation. We will see that the three remaining aspects can all be implemented incrementally.

6.1 Incremental Heuristic

A heuristic for *Atomix* can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer position. These moves are called *generalized moves*. Note that the variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard. In order to obtain an easily computable heuristic, we also allow that an atom to slide through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an *admissible* heuristic for the original problem: The heuristic is *consistent*, since the h-values of child states can differ from that of the parent state by 0, $+1$ or -1.

Since each atom attributes one number to an overall sum it is easy to see that the heuristic estimate can be computed incrementally in constant time by subtracting the value for the currently moving atom from its start location and by adding the value for its final destination. We only need to precompute a distance table for each atom.

6.2 Incremental Successor Generation

For move execution, the implementation, we started with, looked at the set of adjacent squares into the direction of a move unless an obstacle is hit. By the distance of a move between the start and the target location, this operation is not of constant time. Therefore, we looked for alternatives. The idea to maintain a doubly linked *neighbor graph* in x and y direction fails on the first attempt. The neighbor graph would include atoms as well as surrounding walls. When moving an atom, say vertically, at the source location, the update of the link structure turns out to be simple, but when we place it at its target location, we have to determine the position in the linked list for the orthogonal direction. In both cases we are left with the problem to determine the maximal j in a

list of numbers that is smaller than a given i. With an arbitrary large set of numbers, the problem is not trivial.

However, we can take advantage that the number of atoms and walls in a given row or column are bounded by a small value (15 in our case). This yields the option to encode all possible layouts in a row or column as a bitstring with integer values in $\{0, \ldots, 2^{15}-1\}$. For each of the query positions $i \in \{0, \ldots, 15\}$ we store a table M_i of size 2^{15}, with entry $M_i[b]$ denoting the highest bit $j < i$ and $j \geq 0$ in b that is smaller than i. Each table M_i consumes 2^{15} byte or 32 KByte. These tables can be used to determine in constant time the stopping position of an atom that is pushed to the right or downwards starting in row or column i. Analogously, we can build a second set of tables for left and upward pushes. Together, the tables require $2^{15} \cdot 15 \cdot 2$ bytes (≈1MB).

6.3 Incremental Hashing

A move in Atomix will merely change the position of one atom on the board. Hence a state for a given Atomix level is sufficiently described by the positions $(p_1, \ldots, p_m)$ of atoms on the board. We exploit the fact, that only one atom is changed to calculate the hash value of a state incrementally. This constitutes a special case of the concept described in section 4. Let $s = (p_1, \ldots, p_m)$ be a state for an Atomix level. We define its hash value as $h(s) = \left(\sum_{i=1}^{m} p_i \cdot 15^{2i}\right) mod\, q$. The given atomix solver - *Atomixer* [13] - uses a hashtable of fixed size ts (40MB by default). We adjust this value to the smallest prime that is greater or equal ts and use it as q to get a better distribution for our hash function. The full calculation of the hash code is only needed for the initial state. Let s' be an immediate successor state which differs from its predecessor s only in the position of atom i. Then we have

$$h(s') = ((h(s) - ((p_i \cdot 15^{2i}) \bmod q) \bmod q) - ((p'_i \cdot 15^{2i}) \bmod q)) \bmod q$$

We can use a pre-calculated table t with $15^2 \cdot m$ entries which, for each $i \in \{1, \ldots, 15^2\}$ and each $j \in \{1, \ldots, m\}$, stores $t_{i,j} = (i \cdot 15^{2j}) \bmod q$. We can use t to substitute $p_i \cdot 15^{2i}$ with $t_{p_i,i}$ and $p'_i \cdot 15^{2i}$ with $t_{p'_i,i}$. Furthermore, we know that $0 \leq h(s) < q$ and apparently $0 \leq a \bmod q < q$ for any a. This implies, that each remaining expression taken $mod\, q$ holds a value between $-q+1$ and $2q-2$. As a consequence, each sub-term $a \bmod q$ can be substituted by $a - q$ if $a \geq q$, $q + a$ if $a < 0$ or a if $0 \leq a < q$. Now we can calculate $h(s')$ incrementally by the following sequence of program statements:

1. if ($t_{p_i,i} > h(s)$) $h(s') \leftarrow q + h(s) - t_{p_i,i}$ else $h(s') \leftarrow h(s) - t_{p_i,i}$
2. $h(s') \leftarrow h(s') + t_{p'_i,i}$
3. if ($h(s') \geq q$) $h(s') \leftarrow h(s') - q$

This way, we avoid the computationally expensive modulo operators.

6.4 Partial Search

Atomixer supports bitstate hashing and hash compaction as described in Section 5. As the results in [13] show, optimality is preserved, while more instances can be solves within a given memory limit compared to searches that store each expanded state explicitly. Therefore, Atomixer uses hash compaction by default. The partial search methods use a second hash function to calculate the signature of a state. We calculate this signature incrementally using a procedure analogous to that described in Section 6.3.

6.5 Experiments

Experiments were performed on a Linux-based PC with a 1.8GHz CPU and 1 GB of main memory. Atomixer was run on the levels *atomix01* - *atomix30*, that came with the original source package. We used IDA* with hash compaction (the default settings of Atomixer). The search terminates, when a solution is found, or a limit of $5 \cdot 10^8$ generated states was exceeded. We do not consider explorations that last less than one second, to avoid falsification of the results by disruptive factors such as other processes running on the same machine. Table 1 shows the results as the number of generated states per second using the original hash functions (OLD), the non-incremental (FULL) and incremental (INC) hash function as described in Section 6.3. An appended '+' indicates, that we additionally applied incremental successor generation as described in Section 6.2. The last two columns show the speedup of incremental hashing compared to the original hash functions with (%+) and without (%) incremental successor generation. For improved readability, the rows are sorted in ascending order according to the value in the OLD column. In each row, the value of the best result is displayed in bold letters. First of all, our non-incremental hash function gives results, which are significantly worse than for the original hash functions. This implies, that speedups for INC, INC+ can be addressed to the incremental computation of the hash value rather than to the simplicity of the underlying hash function itself. We can observe a speed increase in almost all cases, when incremental hashing is used. Also, INC+ in general leads to better results than INC. There are a few exceptions - e.g. atomix08, atomix23 ... - where INC is better. Note that the incremental successor generation can be slower than the conventional method, for example if all pushed atoms will stop after sliding just one field in the respective direction. There is also one exception - atomix05 - where INC performs worse than OLD while INC+ performs significantly better. Interestingly, in this level INC+ also gives better results than OLD+, which implies that synergy effects between incremental hashing and incremental successor generation are possible.

To summarize the results, we can say that incremental hashing will in general lead to a speedup in the computation of the hash values. In the case of Atomix, incremental successor generation can lead to further improvements. Note however, that in all cases the best result is obtained when incremental hashing is used, but in several cases the use of incremental successor generation will not yield the best results. This implies that hashing is the more reliable incremental method.

7 Perfect Hashing

Can we find a hash function h such (besides the efforts to compute the hash function) all lookups are constant-time? The answer is "yes" - this leads to *perfect hashing*. An injective mapping of $\mathcal{R}$ with $|\mathcal{R}| = n < m$ to $\{1, \ldots, m\}$ is called a *perfect hash function*; it allows an access without collisions. The design of perfect hashing yields an optimal worst case performance of $O(1)$ accesses. Since perfect hashing uniquely determines an address, a state S can often be reconstructed given $h(S)$.

If we invest enough space, perfect hash functions are not difficult to obtain. In the example of the Eight-Puzzle for a state S in vector representation $t_0, \ldots, t_8$ we may address $h_1(S) = (((((((t_0 \cdot 9 + t_1) \cdot 9 + t_2) \cdot 9 + t_3) \cdot 9 + t_4) \cdot 9 + t_5) \cdot 9 + t_6) \cdot 9 + t_7) \cdot 9 + t_8$.

level	OLD	FULL	INC	OLD+	FULL+	INC+	%	%+
atomix17	256,673	246,805	258,704	258,526	249,118	**261,585**	0.7%	1.9%
atomix21	380,700	366,840	381,845	387,506	371,783	**391,291**	0.3%	2.7%
atomix25	473,978	440,722	502,366	501,625	455,502	**512,211**	5.9%	8%
atomix20	540,044	508,817	548,329	544,466	513,658	**552,809**	1.5%	2.3%
atomix24	657,609	600,564	**671,618**	661,804	605,019	667,316	2.1%	1.4%
atomix14	713,663	659,291	766,154	755,423	691,725	**796,558**	7.3%	11.6%
atomix08	733,417	663,429	**753,647**	640,073	620,393	747,272	2.7%	1.8%
atomix19	796,444	739,710	808,041	797,549	743,240	**814,451**	1.4%	2.2%
atomix13	1,412,668	1,267,683	1,418,434	1,486,279	1,276,591	**1,684,517**	0.4%	19.2%
atomix10	1,530,034	1,307,702	**1,598,567**	1,493,518	1,293,226	1,567,692	4.4%	2.4%
atomix15	1,661,902	1,433,116	1,672,912	1,688,618	1,542,067	**1,887,148**	0.6%	13.5%
atomix28	1,703,547	1,434,778	**1,782,075**	1,605,560	1,376,060	1,676,364	4.6%	-1.5%
atomix16	1,819,438	1,586,244	1,961,707	1,971,064	1,653,603	**2,019,222**	7.8%	10.9%
atomix07	1,869,857	1,621,060	2,022,653	2,031,446	1,689,760	**2,082,812**	8.1%	11.3%
atomix22	1,886,223	1,641,820	**2,055,244**	1,921,155	1,585,590	1,996,486	8.9%	5.8%
atomix29	1,933,725	1,720,128	**2,016,535**	1,881,462	1,667,881	**2,016,535**	4.2%	4.2%
atomix05	1,949,013	1,430,778	1,784,694	2,015,316	1,667,778	**2,181,120**	-8.4%	11.9%
atomix09	2,004,407	1,769,055	2,085,507	2,039,956	1,626,776	**2,206,088**	4%	10%
atomix04	2,064,244	1,760,354	2,212,310	2,145,608	1,781,590	**2,232,961**	7.1%	8.1%

Table 1. Experimental Results

Function h_1 yields at most $9^9 = 387,420,489$ different hash addresses, requires about 46,18 MByte space and, as already shown, can be used for incremental hashing.

A better hash function h_2 would be to compute the rank of the permutation in some given ordering, resulting in 9! states or 44,29 KByte[1]. By looking closer at a permutation tree T_k for the symmetric group S_k, it is easy to see that such a hash function h_2 exists. Leaves in T_k are all permutations and at each node in level i, the i-th value is selected, reducing the range of the remaining set of available values. This leads to an $O(k^2)$ algorithm to compute the lexicographical rank, where k is the number of index positions in the state vector. However, it is possible to compute a permutation index in linear time. The recursive algorithm *rank*, due to [23], is called with parameter k, permutation π, and its inverse π^{-1}:

1. If $(k = 1)$ then return 0
2. Set $t \leftarrow \pi[k-1]$; swap $\pi[k-1]$ and $\pi[\pi^{-1}[k-1]]$; swap $\pi^{-1}[t]$ and $\pi^{-1}[k-1]$
3. Return $t \cdot (k-1)! +$ *rank*$(k-1, \pi, \pi^{-1})$

The inverse π^{-1} of π can be computed by setting $\pi^{-1}[\pi[i]] = i$, for all $i \in \{0, \ldots, k-1\}$: Algorithm *rank* will yield a perfect hash function for permutation games that covers the entire set of reachable states. It is also possible to compile a rank back into a permutation in linear time. The procedure *unrank*, initialized with the identity permutation, works as follows. It is called with the parameters k, r, and π:

[1] Note that since only every second board layout is reachable, there are exactly $n^2!/2$ solvable instances to the (n^2-1)-Puzzle, thus reducing the minimum space needed by a half.

1. If $(k = 0)$ stop
2. Set $t \leftarrow \lfloor r/(k-1)! \rfloor$; swap $\pi[k-1]$ and $\pi[t]$
3. *unrank*$(k-1, r - t \cdot (k-1)!, \pi)$

If the algorithm is terminated at the k-th step then the positions $k-l, \ldots, k-1$ hold a random l-permutation of $\{0, \ldots, k-1\}$. One drawback of perfect functions of type h_2 is that it is not simple to devise an incremental constant time version. While π and π^{-1} are possible to be maintained in constant time, the incremental update for the recursive procedure *rank* is involved. One space-consuming option is to use $O(n^2 \cdot n!)$ memory to precompute a rank table for each possible transposition that can occur to infer a index-to-index mapping for each available move. In the (n^2-1)-Puzzle, where each tile transposes with the blank only, we reduce the complexity to $O(n \cdot n!)$ space.

In STRIPS planning, for a state S with propositions a_i, $i \in \{1, \ldots, |AP|\}$, a coarse perfect hash function is defined as $h(S) = \sum_{a_i \in S} 2^i$ (now without modulo operation) and has values in the interval$[0, 2^{|AP|} - 1]$. Reconstruction of S given $h(S)$ is trivial. It can be used for incremental hashing.

7.1 Pattern Addressing

As the usage of perfect hash functions for large state space problems is limited, we may apply perfect hashing to *pattern database* construction and usage. Recall, that pattern databases [3] are hash tables for fully explored abstract state spaces, storing with each abstract state the shortest path distance in the abstract space to the abstract goal. Pattern databases are constructed in a traversal of the inverse search space graph. The distance value stored in the hash table is a lower bound on the solution length in original space and serves as a heuristic. Different pattern databases can be combined and the effort to determine a value are small. Pattern databases work, if the abstraction function is a homomorphism, so that each path in the original state space has a corresponding one in the abstract state space. In difference to search in original space, the entire or at least a large fraction of the abstract space [31] has to be looked at. Since pattern databases are itself hash tables we can use incremental hashing for these tables, too. Note that the addressing is also important for *deadlock* and *penalty* hash tables [14].

For p selected tiles in the (n^2-1)-Puzzle, we have $(n^2)!/p!$ different positions, that can be perfectly addressed by the hash function $h(S) = \sum_{i=0}^{p-1} t_i \cdot (n^2)^i$ with at most n^{2p} addresses. As said, algorithm *rank* runs in $O(p)$ time to compute a hash address for one of the $(n^2)!/p!$ boards. The ratio of n^{2p} and $(n^2)!/p!$ for smaller patterns p is not as bad as with a full board, e.g. for $p = 8$ and $n^2 = 16$ it resolves to 8.27.

If we restrict the exploration in STRIPS planning to some certain subset of propositions $R \subseteq AP$, we generate an abstract state space [6] with states $S_A \subseteq R$. Abstractions of operators (P, A, D) are defined as $(P \cap R, A \cap R, D \cap R)$ and $\mathcal{I}_A = \mathcal{I} \cap R$. If we re-index the propositions $a_j \in R$ we have a perfect hash function $h(S) = \sum_{a_j \in R} 2^j$ with values in $[0, 2^{|R|} - 1]$.

7.2 Universal, FKS and Cuckoo Hashing

At least in theory, there are dynamic perfect hash functions on small space, and for which we need to introduce universal hash functions. *Universal hashing* requires a

set of hash functions to have on average a good distribution for any subset of stored keys. Let $\{0, \ldots, m-1\}$ be the set of hash addresses and $\mathcal{S} \subseteq I\!N$ be the set of possible keys. A set of hash functions $\mathcal{H}$ is universal, if for all $x, y \in \mathcal{S}$ we have $|\{h \in \mathcal{H} \mid h(x) = h(y)\}|/|\mathcal{H}| \leq 1/m$. One idea in the design of universal hash functions is to include a suitable random number generator inside the hash computation. Universal hash functions lead to a good distribution of values on the average. If h is drawn randomly from $\mathcal{H}$ and S is the set of keys to be inserted in the hash table, the expected cost of each search, insert and delete operation is bounded by $(1 + |S|/m)$.

With a hash function h of a class $\mathcal{H}$ of universal hash functions, we can easily obtain constant lookup time if we don't mind spending a quadratic amount of memory. Say we allocate a hash table of size $m = n(n-1)$. Since there are $\binom{n}{2}$ pairs in $\mathcal{R}$, each with a chance $1/m$ of colliding with each other, the probability of a collision in the hash table is bounded by $\binom{n}{2}/m \leq 1/2$. In other words, the chance of drawing a *perfect hash function* for the set of stored keys is $1/2$. While for each given hash function there is a worst set of stored keys that maps all of them into the same bin, the crucial element of the algorithm is a *randomized rehashing*: if the chosen h actually leads to a collision, just try again with another hash function drawn with uniform probability from $\mathcal{H}$.

The so-called *FKS hashing scheme* [8] (named after the initials of the inventors Fredman, Komlós and Szemerédi) ended a long dispute in research about whether it is also possible to achieve constant access time with a *linear storage size* of $O(n)$. The algorithm uses a two-level approach: First hash into a table of size n, which will produce some collisions. Then, for each resulting bin, use universal hash function set $\mathcal{H}$ for rehashing, squaring the size of the bin to get zero collisions. Unfortunately, the *FKS* hashing scheme is only applicable to the static case, where the hash table is created once with a fixed set of keys, and no insertions and deletions are allowed afterwards. Later the algorithm was generalized to allow for update operations [4]. It can be shown that this dynamization scheme uses $O(n)$ storage; *search* and *delete* operations are executed in $O(1)$ worst case access time, while insertions need $O(1)$ amortized expected time. However, it remains unclear, whether the approach can be made incremental. For the first hash function this is possible, but for the selection of a universal hash function for each bucket a solution can be involved.

Therefore, we propose an alternative conflict strategy. With *cuckoo hashing*, [25] implemented a dictionary with two hash tables, T_1 and T_2, and two different hash functions, h_1 and h_2. Each key, k, is either in $T_1[h_1(k)]$ or $T_2[h_2(k)]$. If an item produces a collision in the first table the detected synonym is deleted and inserted to the other table. There is a small probability that the *cuckoo-process* may not terminate at all and loop forever. The analysis shows that such a situation is rather unlikely, so that one can pick a fresh hash functions and reorganize the entire structure. Although reorganization costs linear time it contributes a small amount to the expected time. *Cuckoo hashing* has worst case constant access time and amortized worst case insertion time. Both hash values can be computed incrementally for each inserted state.

8 Conclusion

Incremental hashing is an important option to improve the performance of backtrack searchers like IDA*. The strategy is very general and applies to most state space prob-

lems. It often reduces the time complexity to compute the hash value of a given state from an expression linear in the state description length to a mere constant. In domains, where successor generation and heuristic evaluation are constant time operations this resolves a bottleneck for the exploration. In the case study for the Atomix problem we have shown that many of these assumption can be made true even in challenging domains. For this problem we design both incremental successor generation and heuristic function evaluation in $O(1)$.

Nonetheless, state insertions into the dictionary in case of an unsuccessful search and state comparisons in case of a successful one, still have to access the entire state description. On the one hand, these actions require fast memory compare and copy actions. On the other hand, for perfect hashing functions, as often met during pattern database construction and search, incremental state hashing is time optimal. To push the envelope to dynamic incremental storage, we addressed popular constant time hashing schemes and choose cuckoo hashing as our favorite.

We expect the idea to be especially important in software model checking, where huge state descriptions with only little changes are to be stored. In form of *recursive hashing* [2], incremental hashing variants are implemented in software validation tools like the model checker SPIN and the the program model checker JPF (Java PathFinder [9]). In both cases, however, the computation is not incremental and traverses the entire state vector. The savings in the hash address computation are based on using polynomials that lead to a linear number of cyclic shifts instead of a linear number of multiplications. In the future we will try to implement the incremental hashing function in our experimental c++ model checker StEAM [21].

Acknowledgments The work is supported by *Deutsche Forschungsgemeinschaft* (DFG) in the projects *Heuristic Search* (Ed 74/3) and *Directed Model Checking* (Ed 74/2).

References

1. B. Bloom. Space/time trade-offs in hashing coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.
2. J. D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320, 1997.
3. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.
4. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23:738–761, 1994.
5. J. Eckerle and T. Lais. Limits and possibilities of sequential hashing with supertrace. In *Formal Description Techniques for Distributed Systems and Communication Protocols, Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer, 1998.
6. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.
7. D. Fotakis, R. Pagh, and P. Sanders. Space efficient hash tables with worst case constant access time. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 271–282, 2003.
8. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *Journal of the ACM*, 3:538–544, 1984.

9. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2002.
10. M. Holzer and S. Schwoon. Assembling molecules in atomix is hard. Technical Report 0101, Institut für Informatik, Technische Universität München, 2001.
11. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
12. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
13. F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. In *German Conference on Artificial Intelligence (KI)*, pages 229–243, 2001.
14. A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.
15. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
16. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
17. R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
18. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.
19. R. E. Korf, M. Reid, and S. Edelkamp. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1–2):199–218, 2001.
20. H. D. Lehmer. Mathematical methods in large-scale computing units. In *Symposium on Larg-Scale Digital Calulating Machienery*, pages 141–146. Cambridge, Massachusetts, Harvard University Press, 1949.
21. P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in c++ with the assembly-level model checker StEAM. In *Workshop on Model Checking Software (SPIN)*, pages 39–56, 2003.
22. H. Marais and K. Bharat. Supporting cooperative and personal surfing with a desktop assistant. In *ACM Symposium on User Interface Software and Technology*, pages 129–138, 1997.
23. W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.
24. A. Östlin and R. Pagh. Uniform hashing in constant time and linear space. In *ACM Symposium on Theory of Computing*, pages 622–628, 2003.
25. R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA)*, pages 121–133, 2001.
26. S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.
27. A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
28. L. Schrage. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, 5(2):132–138, 1979.
29. U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, Aachen, 1996.
30. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.
31. R. Zhou and E. Hansen. Space-efficient memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI)*, 2004. To appear.

On-Line Navigation in GPS-Route

Heiner Ackermann, Mohamed Bettahi, René Brüntrup, Maik Drozdzynski, Stefan Edelkamp, Vanessa Faber, Andreas Gaubatz, Thomas Härtel, Seung-Jun Hong, Shahid Jabbar, Miguel Liebe, Tilman Mehler, Anne Scheidler, Björn Schulz und Feng Wang

Fachbereich Informatik
Baroper Straße 301
Universität Dortmund

Zusammenfassung In der Projektgruppe *GPS-Route* wird ein On-Line Navigationssystem konzipiert und implementiert, das aus einer Menge von GPS-Spuren eine mit Fahrtzeiten und Fahrtmöglichkeiten annotierte Karte generiert, diese vorverarbeitet, speichert und darin möglichst effizient nach kürzesten Wegen sucht. Die eigentlichen Daten werden sowohl mit einem GPS Empfänger als auch in einer Simulationsumgebung erzeugt. Projektziel ist eine Client/Server Architektur, in der statische als auch dynamische GPS-Spurinformationen in Datenbanken verwaltet und zur effizienten und mehrfachen kürzesten- und schnellsten-Wege Suche von einem PDA aus genutzt werden. Dabei werden verschiedene Beschleunigungstechniken eingesetzt. Als Grundlage der Simulation und der Visualisierung dienen topographische Karten des Landesvermessungsamt NRW, deren Pixeldarstellung halbautomatisch in einen Straßengraphen konvertiert werden. Durch geeignete Clusteralgorithmen werden die GPS-Spuren inkrementell zu einer geeigneten Anfragestruktur kombiniert.

1 Problembeschreibung

Der Ansatz zur Routenplanung mit dynamisch erzeugten Spurdaten eignet sich besser zur Navigation als ein statischer Graph des Wegeplanes. Aktuelle Informationen können mit Hilfe von GPS-Aufnahmegeräten bereitgestellt werden. Dabei wird der personelle und technische Aufwand zur elektronischen Kartographierung minimiert. Zudem sind kommerziell vertriebene elektronische Karten ungenau und die gespeicherte Information für ein *on-board* Navigationssystem durch stete Realwelt-Änderungen oft veraltet.

Der Vorteil einer an der Universität angesiedelten Entwicklung ist der gezielter Einsatz von aktuellen Algorithmen und Datenstrukturen. Zur Datenreduktion empfehlen sich z.B. Verfahren des *geometrischen Rundens*. Für die statistische Datenverbesserung auf Basis zusätzlicher Informationsquellen sind geeignete *Filter* angemessen. Für die reduzierten und gefilterten Daten bietet sich die *automatische Erkennung von Kreuzungen* an. Innerhalb des entstehenden Graphen werden mit verschiedenen *Explorationsalgorithmen* längen- oder zeiteffiziente Routen berechnet und dem Anwender fahrtbegleitend zur Verfügung gestellt. Anfragen werden mit effizienten Punktlokalisationsverfahren an den bestehenden Graphen angepasst. Zur effizienten Suche kann der Graph zudem auf alle Schnitt- und Endpunkte *reduziert* werden.

Die Implementierung einer Benutzeroberfläche auf einen PDA (z.B. einem Pocket PC mit GPRS Internetverbindung) stellt hohe Anforderungen an die Effizienz der Kartenvisualisierung und an die Minimierung der Kommunikation zum Server.

Topographische Karten, z.B. aus den der Landesvermessungsämtern, sind sehr genau vermessen und lassen mit mit GPS Informationen verknüpfen. Zur Vektorisierung der Karten, d.h. zur Straßenextraktion und Graphgenerierung, benötigt man verschiedene graphische *Skelettierungs-* und *Trackingalgorithmen*. Auf diesen Graphen kann man dann eine Verkehrssimulation durchführen. Solche Simulationsumgebungen dienen desweiteren zur *Protokollierung des Verkehrsflusses* und des *Fahrzeugverhaltens*.

Kurzum, das Thema birgt viele algorithmische Fragestellungen und ist für ein Informatik-Hauptstudiumsprojekt (12 Personen und einem Jahr Entwicklungszeit) geeignet. Im Folgenden fassen wir den Zwischenbericht des Projekts zusammen. Dabei fangen wir bei den Vorarbeiten und der zu Verfügung gestellten Infrastruktur an. Dann rekapitulieren wir die Themen des Einstiegsseminars. Darauf folgend beschreiben wir die Aufteilung in Kleingruppen und berichten über erste Implementationserfolge. Zum Schluss verweisen wir auf den ausgearbeiteten Zwischenbericht der Projektteilnehmer und bieten einen Ausblick auf mögliche Ziele in naher Zukunft.

2 Vorarbeiten

Ein Prototyp für ein spurbasierten Routenplanungssystem wurde im Rahmen der Diplomarbeit von Shahid Jabbar entwickelt und der Projektgruppe zur Verfügung gestellt. Er ermöglicht das Einlesen von GPS-Spuren, die Berechnung des Schnittpunktgraphen, grundlegende Graphkompressionsverfahren, Punktlokalisation und kürzeste-Wege-Suche. Ansätze zur Effizienzsteigerung wurden entwickelt und dynamischen Veränderung der Karten vorgeschlagen. Aktuelle GPS Spurdaten wurden z.B. durch Rad- oder Taxi-Fahrten oder durch protokollierte Spaziergänge bereitgestellt. Die Darstellung der erstellten Information, sprich die Visualisierung einzelner GPS-Sequenzen, erwies sich als nicht weiter schwer. In der Arbeit wird entweder durch eigene oder durch referenzierte Resultate nachgewiesen, dass alle im Quelltext detailliert beschriebenen Algorithmen und Verfeinerungen optimale Pfade liefern. Dabei ist die Verwendung von Heuristiken in Form von unteren Schranken essentiell. Die Beweise zur Zulässigkeit und Konsistenz der Schätzungen in den vorgestellten insbesondere statischen Modellen begründen die Optimalität von A* als Variante des Algorithmus von Dijkstra und haben somit starken Einfluss auf die Laufzeitresultate.

Die genutzten Erkenntnisse aus dem Bereich der *Algorithmischen Geometrie* beinhalten auch neuere Ergebnisse zum Aufbau und zur Verwaltung effizienter Anfragestrukturen. Die durchgeführten Implementierungen reichen von der Eingabe über verschiedene Vorverarbeitungsschritte, der Anwendung von mitunter gerichteten Suchverfahren, bis hin zur Ausgabe der GPS-Spuren. Es werden die Grenzen existierender elektronischer Routenplanungssysteme diskutiert und die Besonderheit des entwickelten Systems hervorgehoben. Dabei werden Grundlagen zu den Themen der geodätischer Datenerhebung und Informationsverarbeitung angeboten.

Zur Datenreduktion empfiehlt die Arbeit als Verfahren des geometrischen Rundens den *Douglas-Peuker* Algorithmus, der den Linienzug einer Spur rekursiv vereinfacht. Auch wenn der Ansatz zu den einfachsten Verfahren gehört, ist er als Repräsentant dieser Klasse gut ausgewählt. Für die statistische Datenverbesserung auf Basis zusätzlicher inertialer Informationsquellen werden *Kalman* Filtermethoden vorgeschlagen. Auch wenn in der Praxis der Arbeit kein Inertialsystem verwendet wurde, ist die beschriebene Öffnung in die Richtung konsequent. Für die reduzierten und gefilterten

Daten nutzt der Ansatz zur Erkennung von Spurkreuzungen den *Algorithmus von Bentley und Ottmann*, der aus einer Menge von Sequenzen durch das *scan-line* Prinzip den assozierten gewichteten Graphen, inklusive der Schnittpunkte effizient berechnet. Zur Suche wird der Graph zudem auf alle Schnitt- und alle Endpunkte reduziert. Die gefundenen kürzesten Wege im kompakten Graphen werden dann wieder eindeutig dekomprimiert. Anfragen werden mit effizienten Punktlokalisationsverfahren basierend auf einer *Delaunay-Triangulation* (bzw. einem *Voronoi-Diagramm*) der GPS Punktmenge als Struktur nächster Nachbarn an den bestehenden Graphen angepasst. Innerhalb dieses Graphen werden mit verschiedenen Kürzeste-Wege Verfahren längen- oder zeiteffiziente Routen berechnen und dem Anwender fahrtbegleitend zur Verfügung gestellt werden. Die Implementatierung des Systems *GPS-Route* fußt auf ein unübliches, aber sehr flexibles und prägnantes Programmierparadigma: die *Mixin* Programmierung. Hier werden die Basisklassen als `c++`-Templates genutzt. Das Einlesen von GPS-Spuren und die Berechnung des Schnittpunktgraphen wurde mit Hilfe der LEDA-Bibiliothek verwirklicht. Das System *GPS-Route* kann über eine Internet-Schnittstelle aufgerufen und gestartet werden. Dabei wurden zwei Optionen voneinander getrennt: Die textuelle Verarbeitung von Daten über ein CGI/Perl Skript und die graphische Interpretation in dem System VEGA. Beide Ansätze sind noch beschränkt in der Größe der Datenmenge, die sie einlesen bzw. darstellen können. Das System verfügt noch nicht über die vorgeschlagene Netzwerk-Komponente, in der mehrere Anfragen und GPS-Aktualisierungen von verschiedenen Benutzern an ein und denselben Server gestellt werden.

3 Seminarphase

Es wurde ein Einstiegsseminar zu den folgenden Themen veranstaltet.

– *External Memory Point Localization*

  Zusammenfassung und ergänzende Erklärungen zum Artikel *I/O-efficient Point Location using Persistent B-Trees* von Lars Arge, Andrew Danner und Sha-Mayn Teh. Die Autoren beschreiben eine neue Variante persistenter B-Bäume, die im externen Speichermodell sowohl theoretisch als auch praktisch effizient ist. Sie setzt keine totale Ordnung auf den Elementen voraus, benötigt linearen Platz $O(N/B)$, und kann Updates in logarithmisch vielen I/Os ausführen. Unter Verwendung dieser Datenstrukturn kann das Punktlokationsproblem I/O-effizient gelöst werden.

– *Geometric Containers*

  Es wird angenommen, dass, wie bei Routenplanungssystemen üblich, ein statischer, gewichteter Graph vorliegt, auf dem eine ganze Reihe von Kürzeste-Wege-Anfragen gestellt werden. Es stehen also wenige Änderungen vielen Anfragen gegenüber, wobei die Idee naheliegend ist, ein Preprocessing durchzuführen, um die dann folgenden Anfragen schneller beantworten zu können. Eine Speicherung aller kürzesten Wege kommt aufgrund des quadratischen Speicherplatzbedarfs von im Allgemeinen nicht in Frage. Dann wird der Fall eines dynamischen Graphen betrachtet. Bei Änderungen der Gewichtungen im Graphen, sollen nicht alle Ergebnisse des Preprocessing verworfen, sondern möglichst effizient aktualisiert werden.

– *Dynamic Shortest Paths*

  Es wird die dynamische Version des Kürzesten-Wege-Problems mit einem Startknoten betrachtet. Das Problem besteht darin, die Informationen über die kürzesten Wege wieder herzustellen, nachdem sich der Graph verändert hat. Die Informationen über die kürzesten Wege sollen dabei nicht komplett neuberechnet werden, sondern möglichst von der älteren Version übernommen werden. Die häufigsten Änderungen an einem Graphen, die in diesem Zusammenhang vorgenommen werden, sind das Erhöhen und Verringern der Kantengewichte und das Hinzufügen und Löschen von Kanten. Wenn eine beliebige Folge dieser Operationen auf einem Graphen erlaubt sein soll, bezeichnet man das Problem als *dynamisch*.

– *Geometric Data Structures*

  Bei der orthogonale Bereichssuche geht es um Suchanfragen innerhalb von Datenbanken. Viele dieser Suchen können als geometrische Suchen interpretiert werden. Um diese Suche als geometrisches Problem darzustellen, stellen wir die Daten als Punkte im zweidimensionalen Raum dar. Wir stellen *KD-Bäume* und verschiedene Varianten von *Bereichsbäumen* vor. Beim *Windowing* stellen wir uns ein Navigationssystem im Auto vor, von welchem wir uns im näheren Umkreis unseres Aufenthaltsortes die Straßenkarte anzeigen lassen möchten. Befinden wir uns z.B. in Deutschland, möchten wir uns einen Ausschnitt aus der gesamten Strassenkarte von diesem Land anzeigen lassen. Das Navigationssystem muss also mithilfe einer Suchanfrage und unserem Standort diesen Kartenabschnitt ausgeben. Hierfür werden drei Datenstrukturen vorgestellt: *Intervallbäume*, *Prioritäts-Suchbäume* und *Segmentbäume*.

– *Geometric Filtering*

  Die Kombination von GPS mit INS (inertial navigation system) und dem Kalman-Filter ermöglicht eine wesentliche Verbesserung der Navigation. Der Kalman-Filter, der die statistischen Modelle beider Systeme berücksichtigt, kann die Vorteile beider nutzen, um den Einfluß ihrer nachteiligen Merkmale optimal zu minimieren. Der Kalman Filter löst das allgemeine Problem der Schätzung und Vorhersage des zukünftigen Verhaltens eines Prozesses, der als lineares, dynamisches System modelliert ist. Die Untersuchung und Vorhersage des Prozesses erfolgt diskret über die Zeit und rekursiv. Bei jedem Iterationsschritt wird der Zustand des Prozesses neu berechnet, wobei die Matrizen, die die Historie des Prozesses, die Varianzen und die Kovarianzen der Prozessvariablen beschreiben, aktualisiert werden. Man nimmt eine dem Prozessrauschen zugrundeliegende Gaussverteilung an.

– *Geometric Rounding*

  Beim geometrischen Runden wird die Komplexität geometrischer Objekte reduziert. Dies bedeutet, dass insbesondere die zur Darstellung der geometrischen Objekte benötigten Datenmengen verkleinert werden. Geometrische Objekte werden einerseits durch einen kombinatorischen und einen numerischen Datenanteil beschrieben. Als Beispiel sei ein Graph genannt, bei dem der kombinatorische Teil beschreibt in welcher Weise die Knoten des Graphen verknüpft sind und der numerische Teil die Koordinaten der Knoten in der Ebene darstellt. Als erster Rundungsalgorithmen werden die Algorithmen von *Green und Yao*, *Snap Rounding* und *Polygonzugvereinfachungen*, wie der *Douglas-Peuker Algorithmus* vorgestellt.

– *Map Generation*

  Die Aufgabe zur *Kartenerzeugung/Kartenverbesserung* kann in nacheinander folgenden Schritten aufgeteilt werden. *Sammeln* unbearbeiteter GPS Daten von Fahrzeugen, die entlang von Strassen fahren. *Filtern und Resamplen* von Spuren, um GPS-Störungen zu reduzieren. *Aufteilung der Spuren* in Sequenzen von Segmenten, indem die Spuren an eine Initialisierungskarte anpasst werden. Diese Initialisierungskarte könnte z.B. eine kommerziell erhältliche digitale Karte sein. Alternativ kann auch durch einen Algorithmus, eine Netzwerkstruktur allein durch Spuren erstellt werden. Für jedes Segment wird eine Straßenmittellinie generiert, die ungefähr die Form der Straße erfasst. Diese dient uns dann später auch als Referenzlinie für die einzelnen Spuren auf einer Straße. Die Anzahl der Spuren auf einer Straße wird bestimmt, indem ein Cluster-Algorithmus benutzt wird.

– *Electronic Maps and GPS Devices*

  Es wird erläutert, wie ein GPS-System aufgebaut ist. Dabei wird erklärt, wie man seine Position mit Hilfe von Referenzzeiten, die mittels elektromagnetischen Wellen ausgesendet werden, bestimmen kann. Der eindimensionalen Fall wird schrittweise bis zur dritten Dimension fortgeführt. Unterschieden wird anfänglich nach synchronen und asynchronen Uhren. Elektronische Karten lassen sich grob in zwei Kategorien einordnen. *Rasterkarten* bestehen aus einem Rasterbild, wie z.B. Bitmap, und einer Positionsreferenzdatei. Diese Datei enthält Informationen, um das Rasterbild in die Welt einzuordnen. Eine einfache Möglichkeit ist für die obere linke und untere rechte Ecke des Bildes die Koordinaten anzugegeben. Dafür muss das Bild rechtswinklig angeordnet sein. Es sind auch weitere, komplexere Referenzen möglich. *Vektorkarten* bestehen aus geometrischen Objekten. Hierbei werden nur die Positionen der Objekte gespeichert. Die Visualisierung kann diese nun geeignet darstellen und somit ist auch ein uneingeschränkter Zoom möglich.

– *Traffic Models*

  Hier werden Verkehrsmodelle, sowie der GPS-Simulator von Horst und Ralf Lichtenheld vorgestellt. Als erstes wird erklärt, was Verkehrsmodelle sind und welche Unterstützung hierbei die Mikrosimulation leistet. Danach wird der Blick in die Zukunft der Verkehrsmodellierung geworfen und untersucht, welche Fragestellungen relevant sind, um ein solches Modell auf spezielle Bedürfnisse auszurichten. Als nächstes werden repräsentativ sechs Modelle vorgestellt und charakterisiert. Der letzte Punkt befasst sich mit einem GPS-Simulator, der auf Möglichkeiten und Funktionen untersucht wird.

– *Algorithm Animation*

  VEGA steht für **V**isulisation **E**nvironment for **G**eometric **A**lgorithms und stellt ein verteiltes System dar, welches sich zur Aufgabe gemacht hat geometrische Algorithmen (Triangulation, Point Localization, Sweep Line Techniques etc.) zu Visualisieren und graphisch zu animieren. Mit anderen Worten: Die Arbeitsweise von geometrischen Algorithmen wird mit Vega auf dem Bildschirm sichtbar gemacht. Das Vega System besteht im Wesentlichen aus dem Vega-Server, dem Vega-Client und einer Algorithmenbibliothek. Es wird ebenfalls ermöglicht, die Bibliotheken LEDA und CGAL einzubinden und zu nutzen.

– *LEDA*

  LEDA ist die Abkürzung für *Library of Efficient Data Types and Algorithms*. LEDA wird seit 1988 am Max-Planck-Institut für Informatik in Saarbrücken entwickelt. Das Grundprogramm von LEDA wird in vier Pakete, *Basis*, *Graph*, *Geometrie* und *GUI*, unterteilt. Es handelt sich hierbei um eine C++ Klassenbibliothek, die eine beträchtliche Anzahl von effizienten Datentypen und Algorithmen beinhaltet: Keller, Schlangen, Listen, Mengen, Wörterbücher, gerichtete, ungerichtete und planare Graphen, Linien, Punkte und Ebenen; dazu kommen viele Algorithmen der Graphen- und Netzwerktheorie sowie der algorithmischen Geometrie. Zu dem Basispaket von LEDA existieren zahlreiche Erweiterungspakete, die sogenannten LEPs, die die Funktionalität von LEDA um die gewünschten Anwendungsbereiche erweitern. LEPs gibt es z.B. für die Bereiche *abstrakte Voronoi Diagramme*, *dynamische Graphenalgorithmen* und *erweiterte Geometrie*.

– *Multi-Level Shortest Path*

  Der Multi-Level Graph ist ein hierarchisches Konzept, bei dem der Originalgraph auf geeignete Weise in Schichten zerlegt wird. Aus dem Orginalgraphen konstruieren wir einen neuen Graphen $M(G, S_1, S_2, ..., S_l)$ durch Einfügung von zusätzlichen Kanten zwischen benachbarten Leveln des Graphen. Eine Schicht besteht aus einer Knotenmenge S_i, welche eine Untermenge der Knotenmenge des Originalgraphen ist. Für die Knotenmenge S_i gilt: $V \supset S_1 \supset S_2 \supset ... \supset S_l$. Das Gewicht einer neuen Kante zwischen zwei Schichten entspricht die Länge des kürzesten Weges zwischen den beiden Knoten auf dem Originalgraphen. Für jede Kundenanfrage wird anhand des Start- und Zielknotens aus dem Multi-Level Graphen ein Teilgraph aufgebaut und mit Hilfe des Algorithmus von Dijkstra der kürzesten Weg zwischen Start und Zielort berechnet.

Die Ausarbeitungen zu den Themen lieferten die Grundlage der folgenden Implementierungen und einen großen Bestandteil des Zwischenberichts[1]. Die folgenden Themen wurden von den Betreuern der PG präsentiert: *External Memory Graph Search*, *System GPS-Route* und *Mobile Programming*.

Als technisches Inventar wurde ein T-Mobile MDA mit GPS-Mouse, eine Telefonkarte und 512 MByte SD Speicherchip zur Verfügung gestellt. Desweiteren konnte ein GARMIN GPS Empfänger in Verbindung mit einem Laptop zur Aufzeichnung genutzt werden. Bei der Kartenervektorisierung und Simulation wurde der DTK-Kartensatz von Dortmund des Landesvermessungsamts NRW (Bonn) genutzt, von dem ca. 300 *Kacheln* der Größe 1 km^2 im TIFF Format zum Stückpreis von ca. 1 Euro pro Kachel eingekauft wurden. Die genauere Ausgangsbasis sind 18×16 Kartenelemente, die in der Auflösung von 2000 mal 2000 Pixeln vorliegen.

4 Kleingruppen

Bei der Aufteilung in Kleingruppen haben sich vier Gruppen gebildet:

[1] Ausgewählte Literaturangaben zu den Themen sind auf der Projektgruppeninternetseite `http://ls5-www.cs.uni-dortmund.de/~edelkamp/gpsroute` zu finden.

– Simulation: *Kartenvektorisierung* und Erstellen einer *Simulationumgebung* zur automatischen Erzeugung von GPS-Spuren
– Map Generation: *Dynamische Kartengenerierung* auf Basis von vorhandenen GPS-Spurdateien.
– Routenplanung: Datenstrukturen und Algorithmen zur *Punktlokalisation* und *Kürzeste-Wege Berechnung* auf dem Server, basierend auf den erstellten Karten.
– PDA: *PocketPC* Frontend zur *Karten- und Routendartellung, GPS Spuraufnahme* und *Schnittstelle zum Server* zur Verarbeitung von *Start/Ziel Anfragen*

Jede Kleingruppe hat eine Ausarbeitung über die ersten Implementationserfolge geschrieben. Diese Arbeiten schließen den Zwischenbericht ab und werden im folgenden zusammengefasst.

4.1 PDA Programmierung

Ziel der PDA-Gruppe ist es, ein Programm zu entwickeln, das die Nutzung des Navigationssystems ermöglicht. Die Anforderungen an das Programm unterteilen sich in vier verschiedene Bereiche. Zunächst wird ein *Rahmenprogramm* erstellt, das es erlaubt, die Straßenkarten, die errechneten Routen und den aktuellen Standort zu visualisieren. Um die errechneten Routen zu empfangen, ist eine *Netzwerkanbindung* erforderlich. Diese wird weiterhin benötigt, um die von der GPS-Maus empfangenen und gespeicherten Daten zu versenden. Hieraus ergibt sich der dritte Aufgabenbereich, die *Filterung und Speicherung* der empfangenen GPS-Daten. Der vierte und letzte Aufgabenbereich stellt die *Umrechnung* der GPS-Koordinaten in Gauss-Krüger-Koordinaten dar. Diese Konvertierung ist wichtig, um empfangene Daten, die im GPS-Format vorliegen, auf dem Kartenmaterial darzustellen.

Man kann sich in der Karte bewegen, indem man den Navigationsknopf des PDA betätigt. Durch mittiges Drücken auf diesen Knopf wird in vier verschiedenen Auflösungen zyklisch gezoomt, drückt man auf den rechten/linken/oberen/unteren Rand des Navigationsknopfs, kann man die Karte scrollen (vergl. Abbildung 1). Bei Auswahl des Menüpunktes „Neue Route“ öffnet sich ein Fenster, in welchem der Modus der Routenberechnung ausgewählt werden kann (schnellster/kürzester Weg) und die Start- und Zielangaben gesetzt werden können. Klickt man nun auf den Knopf *Suchen*, wird die Karte aufgerufen, auf welcher man den Start- und Zielpunkt mit Hilfe von Fähnchen setzen kann (vergl. Abbildung 2).

4.2 Kartenvektorisierung

In der Teilgruppe *Simulation* waren die Vektorisierung der Karteninformation und der Aufbau des Simulationsgraphen die zentralen Themen im ersten Semester. Die geeignete Verkehrssimulation selbst wird im zweiten Semester behandelt.

Der Startpunkt, das Kartenrohmaterial, besteht aus Pixeldateien. Jedem Pixel (Position i,j) ist ein Farbwert zugeordnet. Die Farbe selbst ist als Tripel von rotem, grünem und blauem Farbanteil gegeben (RGB-Farbskala). Zu unserem sich später herausstellenden Vorteil sind die gegebenen Karten nicht verrauscht, was bedeutet, dass sich die farbigen Flächen nicht aus einer Vielzahl unterschiedlicher RGB-Werte ergeben, sondern alle einen eindeutigen Farbwert haben. Aufgrund dieser Eigenschaft ist es einfach,

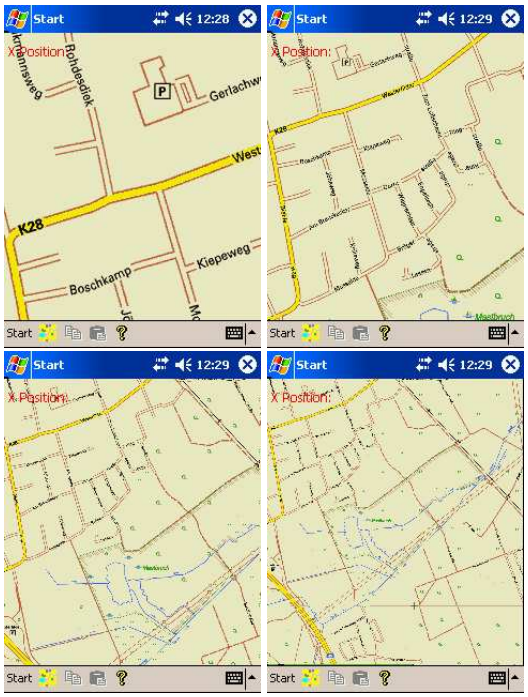

Abbildung 1. Skalieren der Karte.

die für die Straßengraphgenerierung interessanten Straßenflächen vom Rest zu trennen. Zu diesem Zwecke müssen vom Benutzer lediglich die Farbwerte der gewünschten Straßen angegeben werden. Ein einfacher Algorithmus setzt die Farbwerte derjenigen Pixel auf schwarz, welche die vorgegebenen Farben haben. Alle Pixel, die nicht eine der vordefinierten Straßenfarben haben, werden schließlich auf weiß gesetzt. Bei diesem einfachen sogenannten Filter treten allerdings Schwierigkeiten auf.

Zum einen gibt es schwarze Schriftzüge, welche Straßennamen, Ortsteile und andere wichtige Orte angeben und innerhalb der Straßen verlaufen oder Straßen schneiden. Darüberhinaus sind auch alle Straßenbahn- und Eisenbahnstrecken schwarz markiert.

Man könnte nun einerseits die Farbe schwarz als Straßenfarbe definieren. In diesem Fall wählt man allerdings auch gleichzeitig alle Bahnstrecken als Straßenfläche aus. Andererseits könnte man die Farbe schwarz nicht auswählen, würde somit aber weiße Lücken auf den Straßenflächen erhalten. Der größte Teil der Straßen ist in den TIFF-Dateien als weiße Fläche gegeben. Bei der Auswahl dieser Farbe werden dann allerdings auch Sportplatze und Parkplatzmarker und andere kleinere Dinge ausgewählt. Die Extraktion der Straßenflächen it also nicht ganz sauber und bedarf manueller und auch automatisierter Nachbearbeitungen. Die manuelle Nachbearbeitung besteht aus einfachen Mal- bzw Radierwerkzeugen, wie sie aus jedem Bildbearbeitungsprogramm bekannt sind. Zu den automatisierten Nachbearbeitungen zählen: Erosion, Dilatation,

Abbildung 2. Festlegen der Route.

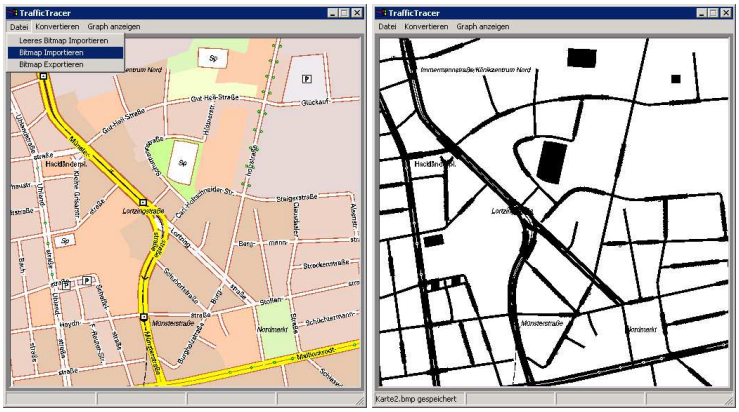

Abbildung 3. Extraktion der Straßenflächen.

Morphologisches Öffnen, Morphologisches Schließen, Lücken Schliessen und Fragmente Entfernen.

Im folgenden sollen diese sechs automatisierten Bildbearbeitungsalgorithmen anhand von Screenshots erklärt werden. Grundlage für die ersten *Säuberungen* der Straßenflächen durch Filterung sind morphologische Operationen. Die Basis der mathematischen Morphologie ist die Mengenlehre. Neben den geläufigen Mengenoperationen wie Vereinigung Durchschnitt, Differenz und Komplement spielen auch Reflektion ($\hat{A} = \{w \mid w = -a, a \in A\}$) und Translation ($(A)_z = \{c \mid c = a + z, a \in A\}$) eine wichtige Rolle.

Erosion und Dilatation Für zwei Mengen A und B in Z^2 ist die Erosion von A und B definiert als $A \ominus B = \{z \mid (B)_z \subseteq A\}$. Eine Erosion von A und B besteht also aus allen Punkten z, für die gilt, dass B um z verschoben in A enthalten ist. Die Menge B wird in diesem Zusammenhang als *strukturierendes Element* bezeichnet. Die Form und Größe des strukturierenden Elementes ist für das Erosionsergebnis entscheidend. Aufgrund der Aufgabenstellung ist die Wahl eines symmetrischen struk-

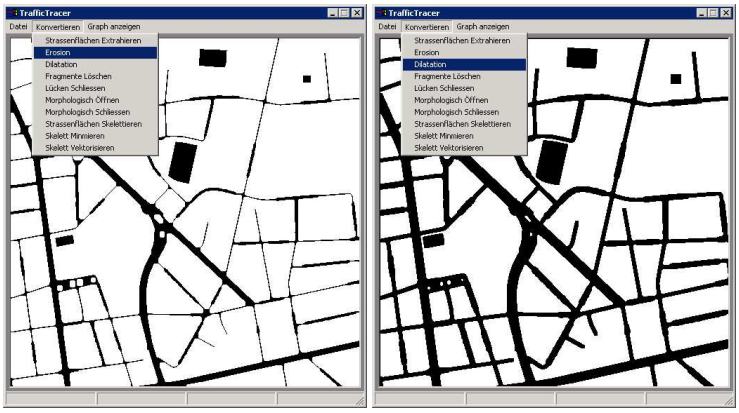

Abbildung 4. Erosion und Dilatation.

turierenden Elementes notwendig, da ansonsten die Straßenflächen *verzerrt* würden. Da durch die Erosion Pixel (vom Rand der schwarzen Straßenflächen aus) gelöscht werden, darf das strukturierende Element darüber hinaus nicht zu groß sein, da es sonst passieren kann, dass *dünne* Straßenflächen komplett gelöscht werden. Aus diesem Grunde wird ein 3x3-Pixel großes strukturierendes Element gewählt, wobei der Benutzer durch mehrfache Erosion die Straßenflächen stetig verdünnen kann. Auf diese Weise wird bereits ein Großteil der störenden Schriftzüge, sowie Straßen- und Eisenbahnschienen eliminiert:

Für zwei Mengen A und B in Z^2 ist die Dilatation definiert als $A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \oslash\}$. Somit sind Erosion und Dilatation komplementäre Operationen und es gilt die Gleichung: $(A \ominus B)^c = A^c \oplus \hat{B}$. Auch bei der Dilatation wird ein 3x3-Pixel großes strukturierendes Element gewählt. Bei der Benutzung des Dliatationsfilters muss der Anwender darauf achten, dass eng beieinander liegende Straßenflächen nicht zu einer großen Straßenfläche verschmolzen werden. Allerdings können kleine Lücken innerhalb der Straßenflächen verkleinert oder geschlossen werden.

Morphologisches Öffnen und morphologisches Schliessen Das morphologische Öffnen einer Menge A durch das strukturierende Element B (beide aus Z^2) ist definiert als $A \circ B = (A \ominus B) \oplus B$. Es ist also nichts anderes, als eine Erosion von A und B, unmittelbar gefolgt von einer Dilatation des Erosionsergebnisses. Durch diese Operation werden schmale Verbindungen zwischen schwarzen Flächen aufgebrochen, die Konturen werden abgerundet und kleine Vorsprünge eliminiert.

Das morphologische Schließen einer Menge A durch das strukturierende Element B (beide aus Z^2) ist definiert als $A \bullet B = (A \oplus B) \ominus B$. Durch diese Operation werden ebenfalls Konturen geglättet, allerdings werden im Gegensatz zum morphologischen Öffnen schmale Verbindungen zwischen schwarzen Flächen verstärkt, und kleine Löcher und Einbuchtungen gefüllt:

Lücken Schliessen Ein Problem welches nicht allein durch Dilatationen gelöst werden kann ist die Eliminierung von ´Lücken´ innerhalb breiter Straßenflächen, z.B. bei Autobahnen. Zwar könnten durch mehrfache Dilatationen solche Lücken geschlossen

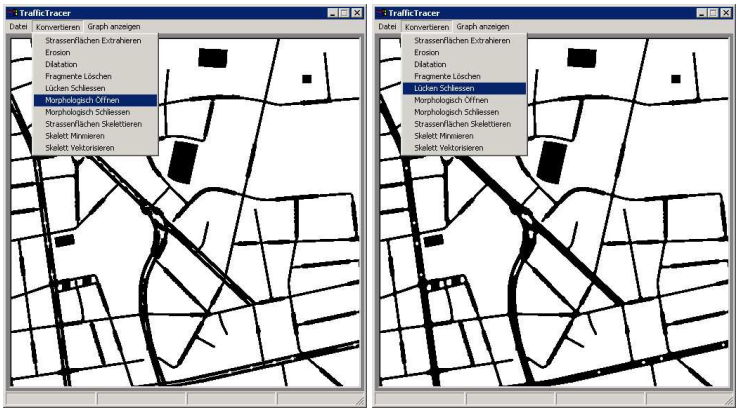

Abbildung 5. Morphologisches Öffnen und Schließen.

werden, wobei dann allerdings viele Straßenflächen verschmolzen würden (s.o.). Abhilfe schafft in diesem Zusammenhang der Algorithmus *Lücken Schließen*, welcher lediglich für alle weißen Pixel überprüft, ob dieses mindestens $n \in \{1, 2, ..., 8\}$ schwarze Nachbarpixel hat. Die Anzahl n soll vom Benutzer festgelegt werden können, wobei sich ein Wert von $n = 5$ als praktikabel erwiesen hat.

Fragmente Entfernen Durch die Erodierung werden die Straßenbahnschienen leider nicht ganz eliminiert. Außerdem befinden sich unter anderem Sportplätze und verschiedene Kartensymbole auf dem 0/1 - Bitmap. Ziel dieses Algorithmus ist es, diese schwarzen Stellen zu finden und weiß zu färben.

Es wird jedes Pixel des Bitmaps untersucht. Findet sich ein schwarzes Pixel *x* , wird in weiteren Schritten die Umgebung von *x* analysiert. Es wird die Eigenschaft ausgenutzt, dass Straßenzüge stets aufeinanderfolgende schwarze Pixel besitzen. Sind nun alle umgebenden Pixel um *x* weiß, kann man davon ausgehen, dass *x* isoliert von anderen schwarzen Pixeln ist und es weiß gefärbt werden kann.

In der Methode wird ein ungerader Ganzzahlwert *i* größer 3 übergeben, der die maximale Größe eines Quadrates angibt, welches um *x* gezogen wird. In einer Schleife wird iterativ das Quadrat von Kantenlänge 3 bis *i* vergrößert. Bestehen die Kanten eines Quadrates zu einem Zeitpunkt nur aus weißen Pixeln, so wird der Flächeninhalt weiß gefärbt und die Iteration abgebrochen. Ist dies nicht der Fall wird das Quadrat entsprechend der Iteration (bis zur maximalen Kantenlänge *i*) vergrößert. Randpixel des Bitmaps müssen in Sonderfällen betrachtet werden.

Die Skelettierung einer Pixeldatei basiert auf der Verwendung morphologischer Operatoren. Nachdem die Straßenflächen durch die vorgestellten Filter sowie durch manuelle Nachbearbeitung sauber extrahiert wurden führt ein Skelettierungsalgorithmus zu einer Darstellung aus welcher dann im nächsten Schritt ein Straßengraph entwickelt werden kann. Das Skelett einer s/w-Pixeldatei ist, vereinfacht und anschaulich gesagt, eine Menge von dünnen Linien, welche die *Mittelachsen* der ursprünglichen schwarzen Flächen darstellen. Auf die Straßenflächen bezogen bedeutet dies, dass das Straßenskelett die Mittellinien der ursprünglichen Straßenflächen darstellen soll. In der

mathematischen Morphologie ist das Skelett von A definiert als $S(A) = \bigcup_{k=0}^{k \leq K} S_k(A)$ mit $S_k(A) = (A \ominus B) \circ B$, wobei B das strukturierende Element ist und $(A \ominus kB) = (\ldots(A \ominus B) \ominus B) \ominus B \ominus \ldots) \ominus B$ gilt. K ist der letzte iterative Schritt bevor A zu einer leeren Menge erodiert, also $K = \max\{k \mid (A \ominus kB)\} \neq \oslash$. Eine erste Implementierung dieser klassischen Definition eines Skelettes hat nicht das gewünschte Ergebnis gebracht. Durch obige Definition wird nicht der Zusammenhang des Skeletts garantiert. Dies hatte zur Folge, dass gerade an Straßenkreuzungen (und anderen Stellen) das Skelett zerbricht und eine Grapherstellung aus dem Skelett sehr schwierig oder unmöglich wird. Eine Lösung bietet ein heuristischer Ansatz, der von Blum 1967 vorgestellt wurde [Digital Image Processing, Rafael C. Gonzales, Second Edition, Kap. 11.1.5, S.650]. Er basiert auf der so genannten Mittelaxentransformation (MAT). Die MAT einer Region R mit Grenze G ist folgendermaßen definiert. Für jeden Punkt p in R finde man alle nächsten Nachbarn auf der Grenze G. Falls p mehr als einen nächsten Nachbarn hat gehört er zu der Mittelachse, also dem Skelett. Eine noch anschaulichere Definition einer MAT kann über ein Steppenfeuer gemacht werden. Man stelle sich eine zusammenhängende Fläche (homogenen und) trockenen Steppengrases vor, welches gleichzeitig an seinen Grenzen entzündet wird. Die MAT dieser Grasregion besteht aus allen Punkten, welche von mindestens zwei Brandfronten gleichzeitig erreicht wird. Eine direkte Umsetzung dieser Idee führt zu ineffizienten Algorithmen, da die Distanzen von jedem inneren Punkt zu allen Randpunkten berechnet werden müssten. Der in der Implementierung benutzte Algorithmus geht folgendermaßen vor: Die zu skelettierende s/w-Grafik wird in 2 Basisschritten bearbeitet, welche dann sukzessive wiederholt werden, bis keine Veränderungen mehr gemacht werden.

Eine direkte Generierung des Graphen aus dem Skelett, welches mit obigem Algorithmus generiert wurde, ist immer noch problematisch, da das Skelett nicht maximal verdünnt ist. Ein maximal verdünntes Skelett sei definiert als ein Skelett, welche zerfallen würde wenn man einen weiteren Pixel mit mindestens 2 Nachbarn löscht. Diese Bedingung ist Voraussetzung dafür, dass durch die lokale Untersuchung der Umgebung eines Pixel auf dem Skelett, dieser eindeutig als Kreuzungspixel, als Sackgaßenendpixel oder als ein Pixel im Straßenverlauf identifiziert werden kann. Der Algorithmus zur Skelettverdünnung muss mit 2^8 = 256 Möglichkeiten für die Nachbarschaftskonstellation umgehen können und die richtige Entscheidung über das Löschen oder Nichtlöschen eines schwarzen Pixels treffen. Auch dieses Problem wurde *heuristisch* gelöst:

Nachdem das Skelett auf geeignete Weise vorbereitet wurde kommt nun ein Trackingalgorithmus zum Einsatz, welcher einen zusammenhängenden, und gerichteten Graphen erzeugt. Der Algorithmus basiert auf der Sweepline-Technik. Die Pixel werden spaltenweise (Sweepline) betrachtet. Sobald ein Pixel gefunden wird, welches zum Skelett gehört, wird eine Teilroutine aufgerufen, welche die Zusammenhangskomponente bearbeitet, zu der der gefundenen Pixel gehört.

Dabei ist zu beachten, dass nicht alle in den Listen gespeicherten Nachfolger eines Kreuzungspixels aufgerufen werden müssen. Es kann vorkommen, dass man durch die Traversierung des Skelettes auf diese Weise, zu Kreuzungsknoten gelangt, die bereits zuvor besucht wurden, und deren Nachbarliste noch nicht vollständig abgearbeitet wurde. Eine spezielle Markierung der noch nicht abgearbeiteten Nachfolgerpixel eines Kreuzungspixels löst dieses Problem. Erst wenn auf der durch den Sweeplinealgorithmus gefundenen Zusammenhangskomponente der Graph erzeugt wurde, traversiert die Sweepline weiter das Bild und sucht die nächste Komponente. Auf diese Weise können

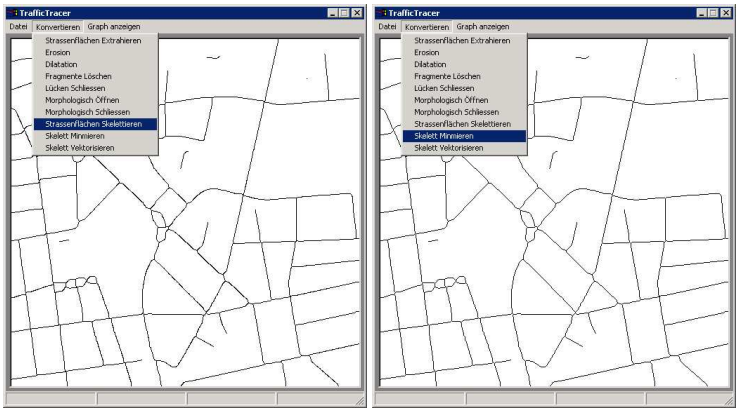

Abbildung 6. Skelettierung und Minimierung.

auch isolierte schwarze Flächen (wie sie in den Karten u.a. bei Sportplätzen vorkommen) eliminiert werden.

4.3 Kartengenerierung

Die *Map Generation* Gruppe hat sich im ersten Semester der Projektgruppe mit der Erstellung eines flexiblen, leicht erweiterbaren Prototypen zur Generierung von Wegekarten (z.B. Straßenkarten) beschäftigt. Im folgenden wird ein kurzer Überblick über dessen Funktionsweise geliefert und im Weiteren näher spezifiziert. Der Prototyp verlangt als Eingabe GPS-Spuren im NMEA-Format. Diese werden über das Internetprotokoll TCP/IP entgegengenommen. Die Daten werden in ein für dieses Programm lesbares Format gebracht und zwischengespeichert. Bei entsprechenden Voraussetzungen - genug Systemressourcen - wird diese gespeicherte Spur weiterverarbeitet. Dazu müssen bereits verarbeitete Spuren geladen und mit der aktuellen Spur zusammengeführt werden. Hierfür wird ein Clusteralgorithmus verwendet. Die Verwaltung der bereits verarbeiteten Spuren ist durch eine Datenbank mit vorgeschaltetem Cache realisiert. Die Datenbank enthält somit immer die gerade aktuelle Version der Karte und kann nach evtl. Konvertierung zur Routenplanung weiterbenutzt werden.

Die Wegekarte ist ein Graph mit zeitannotierten Kanten. Dieser Graph wird jedoch nicht in klassischer Art gespeichert. Es wurde hier eine Zwei-Komponenten-Struktur implementiert: Die Knoten, die annähernd den GPS-Punkten entsprechen, werden in konstant großen Kacheln gespeichert, die durch entsprechende Zugriffsmethoden eine schnelle Nachbarsuche in einem zu spezifizierenden Umfeld ermöglicht. Momentan ist die Nachbarsuche nur durch den Brute-Force-Ansatz realisiert, d.h. alle Punkte auf einer Kachel werden getestet. Die dazugehörigen Kanten werden jeweils in einer listenähnlichen Struktur als Folge von Knoten-IDs gespeichert, wobei eine Kante nur aus Knoten mit Grad zwei bestehen darf, ausgenommen Start- und Endknoten. Somit stellt eine Kante ein Wegestück zwischen zwei Kreuzungen dar. Das zugrunde liegende Koordinatensystem ist das der geographischen Länge, Breite und Höhe.

Abbildung 7. Vektorisierung und Grapherzeugung.

Die Verwaltung der Daten auf der Festplatte übernimmt eine Datenbank, die sowohl unter Windows, als auch Linux bzw. Unix angesteuert werden kann. Dazu wird unter Windows das native *ODBC-Interface* benutzt, während unter Linux/Unix der Zugriff über *unixODBC* abgewickelt wird. Die Struktur der Datenbank ist sehr einfach gehalten. Es ist ein Server realisiert worden, der standardmäßig auf Port 9000 lauscht und bei entsprechenden Anfragen reagiert. Er nimmt die GPS-Spur, die laut NMEA-0183 Standard im ASCII-Format vorliegen, entgegen und beendet nach Übermittlung die Verbindung. Der empfangene String wird in einzelne GPS-Punkte umgewandelt und in zeitlich korrekter Reihenfolge zu einer linearen Liste verknüpft.

Für die Verwaltung und das Anstoßen der Weiterverarbeitung der Traces ist der *TileManager* verantwortlich. Liegt mindestens ein neuer Trace vor, so wird er aktiv: Ein Trace wird nach vorgegeben Kriterien (z.B. neuester Trace) ausgewählt, die von diesem Trace beeinflussten Kacheln werden gesperrt und ein neuer *TraceProcessor* wird für diese Spur angelegt und gestartet. Der *TraceProcessor* kombiniert im weiteren den neuen Trace mit der bestehenden Wegekarte, indem er entweder bestehende Kanten verschiebt oder neue Kanten anlegt.

4.4 Routing

Die Hauptaufgabe der Routinggruppe bestand in der Entwicklung und Implementierung einer Schnittstelle, über die Benutzer Routinganfragen stellen können. Die Schnittstelle sollte Geräteunabhängig (Handheld, Laptop, ...) sein. Außerdem sollten geeignete Datenstrukturen und Speed-Up Techniken erweitert und implementiert werden, so dass Routinganfragen schnell beantwortet werden können.

Die Schnittstelle wird über eine Client/Server-Architektur basierend auf dem TCP/IP-Protokoll realisiert. Als Grundlage für die Datenstrukturen wird die LEDA-Bibliothek in Version 3.1.9 verwendet, sowie verschiedene *Template*-Klassen, die jeweils unterschiedliche Aspekte und Speed-Up Techniken bezüglich des Algorithmus von Dijkstra

zur Berechnung kürzester Weg implementieren. Diese *Template*-Klassen wurden an der Universität Konstanz entwickelt.

Der *Routinggraph* ist ein gerichteter Graph, der alle zum Routen benötigten Informationen enthält. Darunter verstehen wir Kreuzungen, Fahrzeiten zu verschiedenen Tageszeiten und Straßenlängen. Eine Kreuzung wird dabei nicht notwendigerweise durch einen einzelnen Knoten repräsentiert. Der Routinggraph abstrahiert vom konkreten Straßenverlauf. In einer Datenbank existiert deshalb zu jeder Kante des Routinggraphen eine Folge von GPS-Koordinaten, die den genauen Straßenverlauf beschreiben. Der genaue Straßenverlauf wird beim Berechnen eines kürzesten Weges nicht benötigt, so dass eine getrennte Datenhaltung geeignet erscheint. Der Routinggraph wird von der Map-Generation-Gruppe in Abhängigkiet von GPS-Traces erzeugt und in einer Datei abgelegt. Diese Datei wird in regelmäßigen Abständen (z.B. alle 24 Stunden) gelesen und einen Graph für Routingfunktionen erzeugt. Die Graph-Information wird als ASCII-Datei gespeichert. Routinganfragen, die unterstützt werden sind:

1. Berechnung eines *kürzesten* (S, T)*-Weges*. Um diese Berechnungen ausführen zu können, werden lediglich die Straßenlängen benötigt.
2. Berechnung eines *schnellsten* (S, T)*-Weges*.
   (a) Berechnung eines schnellsten (S, T)-Weges zu einer *vorgegebenen Abfahrtszeit* t *am Knoten* S.
   (b) Berechnung eines schnellsten (S, T)-Weges zu einer *vorgegebenen Ankunftszeitzeit* t *am Knoten* T.

   Um diese Berechnungen ausführen zu können, werden die Fahrzeitinformationen benötigt. Leicht einzusehen ist, dass die Varianten 2a und 2b symmetrisch sind. Variante 2b entspricht Variante 2a im Graphen G', indem alle Kanten umgedreht worden sind.

Eine Routinganfrage setzt sich nun wie folgt zusammen:

1. Start- und Zielkoordianten (S, T).
2. Die Angabe der Bewertungsfunktion *Strecke* oder *Zeit*. Im Falle der Zeitfunktion muss zusätzlich zwischen Variante 2a und 2b gewählt werden, wobei auch ein Abfahrts- bzw. Ankunkftszeitpunkt t angegeben werden muss.

Alle Routinganfragetypen haben gemeinsam, dass zunächst zwei Knoten (S', T') im Routinggraphen bestimmt werden müssen, von denen aus ein kürzester/schnellster Weg berechnet wird, da man nicht davon ausgehen kann, dass die Knoten (S, T) im Routinggraphen enthalten sind. Eine naheliegende Lösung für diese Problem ist es, diejenigen Punkte (S', T') zu bestimmen, die minimalen Abstand zu (S, T) haben. Gegebenenfalls kann es auch sinnvoll sein, alle von den Knoten (S', T') ausgehenden Kanten aus der Datenbank zu laden und zu untersuchen, an welchen Kanten die Knoten (S, T) am nächsten liegen.

Die drei von uns unterstützten Routingvarianten erfordern alle die Lösung eines kürzesten Wege Problemes mit verschieden Bewertungsfunktionen. Variante 1 entspricht dem allgemein bekannten *Single-Source-Single-Target Shortest-Path* Problem und kann mit dem Algorithmus von Dijkstra gelöst werden. Die Varianten 2a und 2b sind ebenfalls *Single-Source-Single-Target Shortest-Path* Probleme. Dabei ist zu berücksichtigen, dass die Bewertungsfunktion zeitabhängig ist. Mit einer entsprechenden Anpassung können auch diese Varianten mit dem Algorithmus von Dijksta gelöst werden.

Um die Berechung eines kürzsten Weges zwischen einem Knotenpaar (s, t) zu beschleunigen, stehen zwei Vorverarbeitungsmöglichkeiten zur Auswahl. *Lösung des All-Pair-Shorstest-Path Problem*: Dieser Ansatz ermöglicht, dass Routinganfragen prinzipiell sehr schnell beantwortet werden können. Ein Nachteil dieses Ansatz ist allerdings, dass er viel Speicherplatz benötigt. *Verwendung von Containern:* Die zentrale Idee dieses Ansatz ist es, zu jeder Kante (s, v) eine Datenstruktur zur Verfügung zu stellen, in der alle Knoten t gespeichert werden, so dass ein kürzester Weg von s nach t über die Kante (s, v) führt. Speed-Up Techniken zur Berechnung schnellster Wege sind zum jetzigen Zeitpunkt noch nicht realisiert. Erste Experimente für das kürzeste Wege Problem haben gezeigt, dass die Vorverarbeitung zur Berechnung der Container auf sehr grossen Graphen extrem zeitaufwendig ist.

Um aktuelle Verkehrsinformationen aufzunehmen und dadurch Kantengewichte im Routinggraphen anzupaßen, soll ein Web-Interface entwickelt werden. Im Web-Interface sollen Staus, sowie Straßensperrungen durch einen prozentualen Faktor zwischen mehreren Knoten im Graphen angegeben werden. Die Einträge zur temporären Kantengewichtsänderung können für unbestimmte Zeit, als auch für eine festgelegte Zeitperiode eingetragen werden. Die Eintragungen erfolgen in einer Datenbank, die vom Routingsystem in bestimmten Zeitabständen durchsucht wird, um den Routinggraphen zu aktualisieren. Einträge, die sich über Ihre Existenzzeit hinaus in der Datenbank befinden, werden vom Routingsystem automatisch gelöscht.

5 Ausblick

Der Zwischenbericht der Projektgruppe dokumentiert den Stand nach einem von zwei Semestern. Die Projektgruppe ist ihrem Mindestziel bedeutend näher gekommen. Die Grundbausteine des Systems wurden bereits verwirklicht. So ist eine Vektorisierung und Straßengraphgenerierung der Karten genauso möglich, wie die Darstellung selbiger in verschiedenen Vergrößerungen auf dem PDA. Die Kartengenerierungsgruppe ist dabei, die Überlagerung von per TCP/IP eingefügten GPS Spuren zu verarbeiten; die Datenbankanbindungen stehen. Kürzeste Wege können in kleinen Datensätzen gefunden werden und auch die Punklokalisation funktioniert.

Allerdings sind viele Implementierungen noch nicht stabil und für den Praxiseinsatz geeignet. Desweiteren sind die Schnittstellen zwischen den Gruppen noch nicht ausreichend getestet.

Abschliessend einige, aber längst nicht alle möglichen Erweiterungsmöglichkeiten:

- Ein Ziel ist es, die *praktische Akzeptanz* deutlich zu erhöhen und Firmenkontakte aufzubauen. Ein großeres Manko für die *Autonavigation* ist derzeit allerdings der Mangel an Informationen über die Straßentyp und Ausstattung (Autobahn, Landstraße, Einbahnstraße und Ampel). Außerdem ist hier die industrielle Konkurenz recht groß.
- Unser *universitäres Interesse* gilt der Entwicklung, Ausarbeitung und Verwirklichung von von effizienten, insbesonderer geometrischen Algorithmen. So soll die Einbindung dynamischer Routeninformation z.B. über nicht (mehr) passierbares Gelände und stockenden Verkehrsfluss ermöglicht werden. Desweitern ist die Integration weiterer Messdaten, die Implementierung von Externspeicherplatzsuchverfahren, sowie die effektive Zusammenfassung und Bereinigung von GPS-Daten von Interesse.

Danksagung Die Arbeit wird von der *Deutsche Forschungsgemeinschaft* in den Projekten *Heuristische Suche* (Ed 74/3) und *Gerichtete Modellprüfung* (Ed 74/2) unterstützt.

Literatur

1. H. Ackermann, M. Betahi, R. Brüntrup, M. Drodzynski, V. Faber, A. Graubatz, A. Härtel, S.-J. Hong, M. Liebe, A. Scheidler, B. Schulz, F. Wang, S. Edelkamp, S. Jabbar, and T. Mehler. Zwischenbericht PG 452: GPS-Route. Technical report, Universität Dortmund, 2004.
2. R. K. Ahuja, K. Mehlhorn, J. B. Orbin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, pages 213–223, 1990.
3. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28:643–647, 1979.
4. D. H. Douglas and T. K. Peuker. Algorithms for the reduction of the number of points. *The Canadian Cartographer*, 10:112–122, 1973.
5. S. Edelkamp, S. Jabbar, and T. Willhalm. Accelerating heuristic search in spatial domains. In *Workshop on Planning and Configuration (PUK)*, pages 1–20, 2003.
6. S. Edelkamp, S. Jabbar, and T. Willhalm. Geometric travel planning. In *International Conference on Intelligent Transportation Systems (ITSC)*, 2003.
7. S. Edelkamp and S. Schrödl. Route planning and map inference with global positioning traces. In *Computer Science in Perspective*, Lecture Notes in Computer Science, pages 128–151. Springer, 2003.
8. S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, pages 153–174, 1987.
9. S. Gutmann. Markov-kalman localization for mobile robots. In *In International Conference on Pattern Recognition (ICPR)*, 2002.
10. C. A. Hipke and S. Schuierer. Vega—a user-centered approach to the distributed visualization of geometric algorithms. In *Computer Graphics, Visualization and Interactive Digital Media (WSCG)*, pages 110–117, 1998.
11. S. Jabbar. GPS-based navigation in static and dynamic environments. Master's thesis, Universität Freiburg, 2003.
12. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
13. Navigation Technologies, Sunnyvale, CA. *Software Developer's Toolkit*, 5.7.4 Solaris edition, December 1996.
14. S. Schroedl, S. O. Rogers, and C. K. H. Wilson. Map refinement from GPS traces. Technical Report RTC Report Number 6/2000, DaimlerChrysler Research and Technology North America, Palo Alto, CA, November 2000.
15. D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. Technical report, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 2003.
16. D. Wagner, T. Willhalm, and C. Zaroliagis. Dynamic shortest path containers. Technical report, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, Computer Technology Institute, and Department of Computer Engineering & Informatics University of Patras, 2003.
17. J. Wang, S. Rogers, C. Wilson, and S. Schroedl. Evaluation of a blended DGPS/DR system for precision map refinement. In *Proceeedings of the ION Technical Meeting 2001*, Institute of Navigation, Long Beach, CA, 2001.

Planning with Workflows - An Emerging Paradigm for Web Service Composition

Biplav Srivastava
IBM India Research Laboratory
Block 1, IIT, New Delhi 110016, India
sbiplav@in.ibm.com

Jana Koehler
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
koe@zurich.ibm.com

Abstract

In a previous work, we had analyzed the gaps in the prevalent approaches (i.e., Semantic Web Services and WSDL-described Web Services) for the problems of modeling, composing, executing, and verifying Web services, and derived challenges for the AI planning community. The challenges were in representation of complex actions, handling of richly typed messages, dynamic object creation and specification of multi-partner interactions. An important question that constantly arose was how the goals for automatic composition would be derived. In this paper, we revisit this issue in the light of new trends in software engineering towards Model Driven Architecture and early deployment of Web service composition solutions. We argue that Web services composition can not be seen as a one-shot plan synthesis problem defined with explicit goals but rather as a continual process of manipulating complex workflows, which requires to solve synthesis, execution, optimization, and maintenance problems as goals get incrementally refined. We then identify additional issues that become important in applying planning techniques.

Introduction

Web services have received much interest in academia and industry due to their enormous potential in facilitating seamless business-to-business or enterprise application integration. The business world has defined Web Services with the WSDL specification (Christensen *et al.* 2001), their composition into flows with the BPEL4WS specification (Curbera & others 2002), and their invocation with the SOAP protocol (Box *et al.* 2000). The Semantic Web community has propounded the DAML-S specification (Ankolenkar & others 2002) where semantic annotations describing web resources are explicitly declared with terms, which are precisely defined in ontologies including behavioral details of their preconditions and effects. For the composition of Web services, the Semantic Web community draws on the goal-oriented inferencing techniques from planning.

In a previous work (Srivastava & Koehler 2003), we had analyzed the prevalent approaches (i.e., Semantic Web Services and WSDL-described Web Services) for the problems of modeling, composing, executing, and verifying Web services. We found that the challenges for the AI planning community in applying their techniques are in the representation of complex actions, the handling of richly typed messages, the dynamic creation of objects, and the specification of multi-partner interactions. An important practical question that constantly arises is how the goals for automatic composition can be derived. So far, most approaches see composition as a one-shot plan synthesis problem defined with explicit goals. However, in many practical problems where Web service composition is needed (most of these are centered around business process and/or enterprise application integration), no explicit goals are given. We are observing that the starting point for Web service compositions are, quite often, abstract workflows, which are derived from business models. We will discuss example scenarios for this and derive what planning should deliver in the context of these scenarios. In particular, we dicuss that workflow expansion and feasibility testing, flow decomposition and composition as well as selecting the best web service instances for efficiency (optimization) are required. Composition itself must be a continual process that involves synthesizing, executing, optimizing and maintaining complex workflows as their structure gets incrementally refined.

The paper is organized as follows. We start with a brief summary of our previous observations for WSDL-described Web Services and Semantic Web Services. Next, we discuss *Model-Driven Architecture* (MDA) as the dominant paradigm for business process and enterprise application integration and what it means for composing web and grid services. We then describe the nature of planning for this application and identify new research topics in plan storage and retrieval, plan analysis, and continual optimization.

Background

The literature on composition of both WSDL-described (Staab & others 2003) and Semantic Web services (McIlraith, Son, & Zeng 2001) is quite comprehensive. [1] Much of it deals with resolving discrepancies in the description of Web services, the syntax and semantics of their composition and how they could be executed. Planning is being explored for automatic Web services composition [2] (McDermott 2002; Blythe & others 2003; Srivastava 2002) and

[1] See the survey at http://www.public.asu.edu/ jfan4/wssurvey.htm

[2] See papers of the ICAPS 2003 Planning for Web Services workshop at http://www.isi.edu/info-agents/workshops/icaps2003-p4ws/program.html.

many areas apart from classical planning could be relevant: distributed planning (DesJardins *et al.* 1999), planning as model checking (Giunchiglia & Traverso 1999), planning with extended goals and actions (Dal-Lago, Pistore, & Traverso 2002), and HTN planning (Erol, Hendler, & Nau 1994).

It is worth while to highlight an important characteristic about the Web service composition problem that is often overlooked. When one refers to Web services, one can refer to either the abstract web service type or to one (or more) of the several instances of a particular Web service. In the AI literature, the Web service composition problem is presented as the problem of synthesizing the complex Web service type and the scenarios are too small to have multiple Web service instances for a given type. The resulting composition is a plan - a simplified form of a workflow[3]. However, in practice, there can be a choice among many Web service instances (e.g., due to competing service providers) and the instantiated composite service will be published, deployed and made available to the clients. Since a deployed composite Web service is a long running process, it is crucial that the Web service instances are chosen carefully for performance. Industry has been primarily concerned with the instantiation/performance of the composite web service because Web services are synthesized and deployed less frequently than they are accessed. But if the benefit of running a deployed composite service reduces substantially compared to other available alternatives over time, new compositions will be tried.

We describe a realistic but simple Web service composition scenario and highlight the challenges of planning with WSDL-described and Semantic Web Services.

Example

Consider a scenario with three **types** of Web services: there is an *AddressBookService* which can return the address of a person given her name, a *DirectionService* which can return the driving directions between two input addresses, and a *GPSDirectionService* which can return the driving directions between the locations of two people given their names (see also Figure 1). The available Web service **instances** along with their invocation cost are shown in Table 1. The problem is to implement a composite Web service that can provide the driving directions between the location of any pair of persons. In practice, many metrics for a web service instance are collected, e.g., like the response time and throughput, see a detailed description in (Nanda & Karnik 2003). Any web services composition algorithm has to not only find a feasible plan (containing the relevant Web service types), but also decide how to optimize performance by invoking the best Web service instances for each type.

In Figure 2, two plans for composing the available Web service types are shown to get the driving directions. The plans can be instantiated with specific Web services listed in Table 1. In Plan1, the two *AddressBookService* services can be instantiated to the same or different Web service instances. If the runtime cost of the composite Web service has to be less than 100 units, only Plan1 with the two *AddressBookService* instantiated to AD1, AD2 or AD3 and *DirectionService* to DD1 can satisfy the constraint (total cost 75 units). Note that over time, the plans themselves may not change but the invocation cost estimate of the Web service instances can change and that can make another composition (different plan, instantiation or both) the better choice.

[3]See http://tmitwww.tm.tue.nl/research/patterns/.

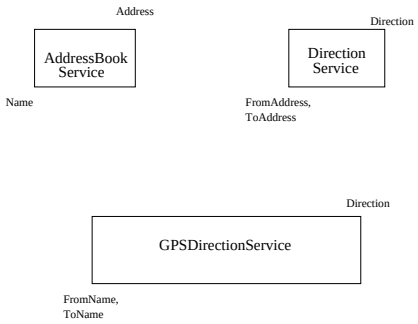

Figure 1: The input and output descriptions of the example Web services.

Service Type	Service Instances
AddressBookService	AD1 (25), AD2 (25), AD3 (25), AD4 (50)
DirectionService	DD1 (25), DD2 (80)
GPSDirectionService	GPS1 (200), GPS2 (200)

Table 1: A simple example listing the Web service types and available instances for each type. The cost of invoking each Web service instance is given in brackets.

WSDL-Described Web Service

The WSDL-described Web services specifications are primarily syntactical: the Web service interface resembles a remote procedure call and the interaction protocol is manually written. WSDL gives only the functional (input and output)

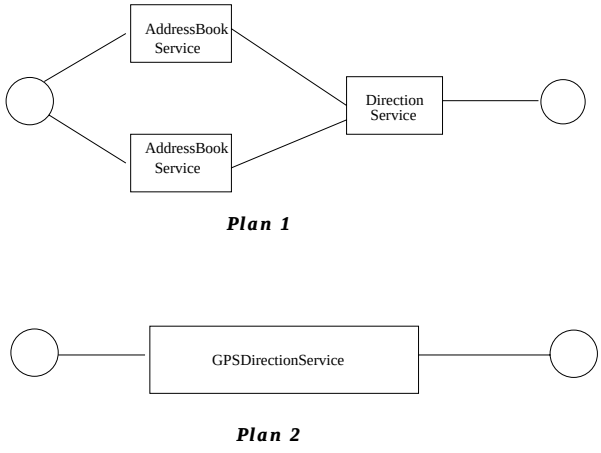

Figure 2: Two different plans for the composite Web service.

specification of Web services, not the process-centric specification of context (precondition and postconditions) to the overall computation being performed. As a very simple example, consider a Web service that manipulates two numerical expressions. The result type of the manipulation can be specified, but not the precedence rules, which determine the expression's correctness - an operator O_i is only appropriate if no operator O_j of higher precedence is pending evaluation in its immediate vicinity in any sub-expression involving O_i.

Due to the restrictiveness of the representation, most work on the automatic composition of WSDL-described Web services has focused on how to combine *instances* of Web services to serve a well specified request (output type). For example, a BPEL4WS specification for a composite service to provide driving directions given two names can be defined as shown below. Note that it identifies the exact web service instances (AD1, AD2, DD1) to use for the invocation.

```
...
<sequence>
   <receive name="receiveRequest" .../>
   <invoke name="invokeAD1"
     partnerLink="callTo"
     portType="AddrBook-PT"
     operation=".."
     inputVariable=".."
     outputVariable=".."/>
   ...
   <invoke name="invokeAD2" .../>
   <invoke name="invokeDD1" .../>
   <reply name="sendReply" .../>
</sequence>
...
```

As long as such BPEL4WS specifications are linear sequences of atomic service invocations, explicit goals are given, and each service is annotated with its predefined precondition and goals, an AI planner can be easily used to generate a plan. However, as we will show next, realistic scenarios may look quite differently from this ideal setting.

The normal situation today is that these BPEL4WS specifications are manually composed by IT experts using, for example, graphical editors to avoid writing the XML code by hand. Similarly, UML can be used to create BPEL4WS (Amsden *et al.* 2003). The BPEL4WS approach looks at composite services mainly from the runtime perspective of functions, data and control flow. The only information available for reasoning about a service are inputs, outputs and exception handlers. Schemas define and restrict the format of data and define their relationships. The only flexibility is in dynamically selecting a Web service binding details or to execute specific branches in the specified workflow.

Semantic Web Services

The Semantic Web (Berners-Lee, Hendler, & Lassila 2001) views the World Wide Web as a globally linked database where web pages are marked with semantic annotations. At the core, semantic annotations are assertions about web resources and their properties (example, "A is subclass of B") expressed in the Resource Description Format (RDF) (RDF 1999). The Semantic Web Services are dynamic web resources represented in the DAML-S representation (Ankolenkar & others 2002). The Web service here is described by *ServiceProfile* to advertise the service, *ServiceModel* to express its behavior and *ServiceGrounding* to find its implementations. The preconditions and postconditions of the Web service are explicitly declared in *ServiceModel* using terms from pre-agreed ontologies.

```
<rdf:RDF ...>
 <daml:Ontology>
 ...
 </daml:Ontology>
 <!-- Atomic Process : AddressBookService --
>
 <daml:Class rdf:ID="AddressBookService">
  <daml:subClassOf
   rdf:resource=".../Process.daml#AtomicProcess"/>
 </daml:Class>
 <!-- Inputs and Outputs -->
 <daml:Property rdf:ID="Name">
  <daml:subPropertyOf
   rdf:resource=".../Process.daml#input" />
  <daml:domain ... />
  <daml:range rdf:resource="...#Name" />
 </daml:Property>
 <daml:Property rdf:ID="Address">
  <daml:subPropertyOf
   rdf:resource=".../Process.daml#output" />
  <daml:domain ... />
  <daml:range rdf:resource="...#Address" />
 </daml:Property>
</rdf:Property>
```

In the above DAML-S specification, a Web service of type *AddressBookService* is described as an atomic process. The properties are defined by extending the DAML-S Process ontology (Process.daml) extensively. The Web service instances can be described by extending the properties of the corresponding Web service type.

The specification of the composite route finding service is given below which invokes concrete instances of the atomic process types.

```
<daml:Class rdf:ID="Route_Process">
 <daml:subClassOf rdf:resource=
     ".../Process.daml#CompositeProcess"/>
  <daml:subClassOf>
   <daml:Restriction>
   ...
     <process:listOfInstancesOf ...>
       <daml:Class rdf:about="#AD1Service"/>
       <daml:Class rdf:about="#AD2Service"/>
       <daml:Class rdf:about="#DD1Service"/>
     </process:listOfInstancesOf>
   ...
   </daml:Restriction>
  </daml:subClassOf>
</daml:Class>
```

The planning approaches in the Semantic Web are focused on the process-centric description of services as actions that are applicable in states. State transitions are defined based on the preconditions and postconditions of actions that mimic the intended behavior of the Web services.

Executing an action (a service) triggers a transition that leads to a new state where the effects of the action are valid. This approach relies on the representation of state, actions, goals, events and optionally, an ontology of standard terms. The plan can be adapted both offline and online. There is more flexibility in terms of considering different choices of services (plans) based on goals, but the goals are explicitly given.

Discussion

The key hurdles in applying AI planning technology to the service composition problem are the following:

- Complex actions. Action specifications traditionally consist of preconditions and effects. However, a Web service modeled as an action can have complex control structures in its specification involving loops and nondeterministic choices. Furthermore, the assumption of explicit preconditions and effects seems to be rather unrealistic.
- Rich data types. The "objects" manipulated by the Web services (actions) are typed messages which may contain identifiable parts that can be arbitrarily complex descriptions. Planning methods have traditionally worked with simple types.
- Dynamic objects. Processes using Web services can lead to new object being created at runtime. In planning, all objects are assumed to be known in the initial state.
- The execution of a workflow very often needs coordination among the partners, which are involved in an interaction. This is in contrast to the traditional on-off invocation by the executor assumed in AI planning. Real-time interleavings between a planning and an execution unit have not been studied very widely so far.

Trends Influencing Web Services

Web services, like other software building blocks, are affected by the changing practices in software engineering. In this section, we discuss two trends that will have an important influence on web services and their composition:

- Model-Driven Architecture and Model-Driven Development
- Continual Optimization of Processes and on-Demand Composition of Workflows

Model-Driven Architectures

Model-driven architectures (MDA) have been proposed by the OMG (The-Object-Management-Group 2003) to reinforce the use of an enterprise architecture strategy and to enhance the efficiency of software development. The key to MDA lies in three factors:

- the adoption of precise and complete semantic models to specify the structure of a system and its behavior,
- the usage of data representation interchange standards,
- the separation of business or application logic from the underlying platform technology.

Each of these key factors poses a challenge of its own, but the model representation seems to be the most critical among them. Models can be specified from different views, from the business analyst's or the IT architect's view, and they can be represented at different levels of abstraction. MDA distinguishes between platform-independent (PIM) and platform-specific (PSM) models, but we can expect that many different PIM and PSM models will be used to describe the different aspects of a system. Representing a model of a system (be it a PIM or a PSM) is only one side of the coin. In order to be useful, a model must be further analyzable by algorithmic methods. For example, the fundamental properties of the behavior of a system should be verifiable in a model, and different models of the same system should be linked in a way that relevant changes in one model can be propagated to the other models to keep the model set consistent.

The OMG has defined and is defining a set of standards that will help to implement model-driven architectures and model-driven development. At the core of these standards are the Unified Modeling language (UML), the Meta-Object Facility (MOF)—a unique object-oriented representation for models, the XML Meta-Data Interchange (XMI) to persist MOF models in an XML dialect, and the Common Warehouse Metamodel (CWM), which standardizes data formats.

Besides the OMG standards, a number of competing modeling approaches are under development. A particular emphasis is on modeling distributed enterprises and their processes. Graphical rendering tools such as Microsoft Visio or Powerpoint have been very popular so far. However, recently the focus moved from standalone models to business-process models, which are suitable to automate the transition from a business process model to the executable runtime of a system. To generate code means to start from a much more structured representation with a (hopefully) well-defined semantics. Two prominent representatives are ARIS (Scheer *et al.* 2002) and Holosofx (Deborin & others 2002),—just to name a few. The latter is forming the foundation of IBM's recent Websphere Business Modeler product, which is part of an integrated solution that supports modeling, code generation and monitoring of the deployed code. Interestingly, these modeling approaches come all from a Petri-net background. They focus on data-and control flow, but state information is never made explicit. Trying to apply AI planning to Web service composition therefore means to face the clash of two different worlds: Petri nets vs. State machines.

In the following, we briefly illustrate this problem by reviewing two major scenarios in which Web service composition occurs:

- Model-driven Business Integration, and
- Service Composition from Conversation Specifications.

Model-driven Business Integration

The automation of business processes and the integration of processes and applications is a current major driver for IT investments. Workflow systems on the basis of Message Oriented Middleware (MOM) products are often used to achieve this goal. A workflow defines a set of activities and their control- and data flow. Even in the simplest

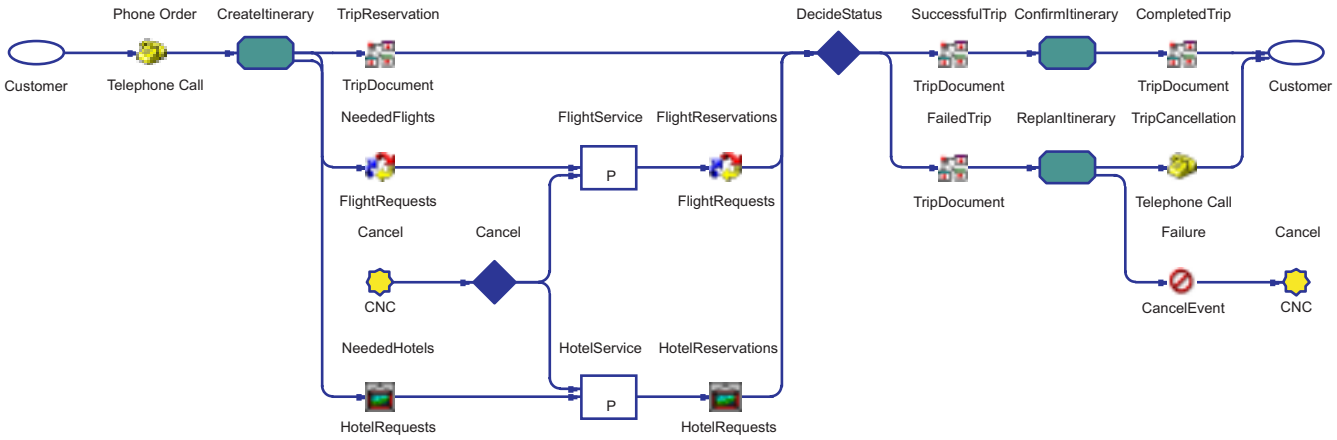

Figure 3: ADF representation of a trip reservation business process.

setting, a workflow usually contains branching and iterations in the control flow. In contrast to this, in AI planning, the notion of a plan has traditionally been simpler–a plan is a sequence of steps where each step can have a set of parallel action instances. The generation of plans involving more complex control flows has only been considered by a few approaches (Dal-Lago, Pistore, & Traverso 2002; Stephan & Biundo 1996). Interestingly, the standardization efforts in AI planning defined a unique domain representation format, but not a standardized format for plans. Adopting the upcoming BPEL4WS standard could fill this gap and would also nicely align plans with workflow standards.

In Model-Driven Business Integration, workflows ("plans") are assumed to be generated from business process models. Figure 3 shows the specification of a plan in the ADF formalism by Holosofx.

In this example representation, the customer is an external entity (depicted with a white oval) initiating the trip-handling process. Three main elements are used to capture the structure of the process: (1) Grey octagons depict the elementary process steps, which are named as *CreateItinerary*, *ConfirmItinerary*, and *ReplanItinerary*. (2) White boxes containing a "P" letter depict sub-processes and refer to the *FlightService* and *HotelService*. (3) Black diamonds show binary decision steps in the process. Typed information entities flow between the various process steps. In the example model, we distinguish between five different types of information entities named *TelephoneCall*, *TripDocument*, *FlightRequests*, *HotelRequests*, and *CancelEvent*. Each of them is depicted with a separate so-called Phi symbol. Below each Phi symbol, we find the type and above the symbol, we find the Phi name (the particular instance of this type). A star-shaped goto symbol allows to model cyclic processes. In our example, the Phi *Failure* of type *CancelEvent* is sent from the *ReplanItinerary* step back to the *FlightService* and *HotelService* sub-processes via the CNC goto symbol. Which of the services has to be notified is decided in the decision step receiving the Phi.

The task of the 'planning system' is to transform this representation into an executable BPEL4WS specification. The problem lies not in determining the set of actions and their execution order, but in deriving the correct structure of the BPEL4WS process, its required interactions with partners and the Web service interfaces based on the specified data flow. A particular challenge lies in transforming unstructured cyclic control flows into well-structured optimized workflow code. For simple cases, graph-based transformations can be used (Koehler *et al.* 2003), for more complicated cases, more sophisticated splitting and optimization techniques must be used, but so far there is not much need for search-based planning techniques. However, there is an upcoming need for verification due to the increasing complexity of the flows (in particular when the combination of concurrency and loops occurs).

Service Composition from Conversation Specifications

A second scenario, which seems to require more planning-like techniques is the problem of service composition from a given conversation specification. Business process models often only concentrate on specifying the process itself, but do not pay much attention to specify the process context. For a Web service-based process, it is of particular importance to specify the interactions between a process and its partners providing additional Web services. This specification can be formulated using the Web Service Conversation Language (WSCL) (Banerji & others 2002), a proposed standard submitted to the W3C, which is based on activity diagrams that describe the allowed interactions. The WSCL diagram for the trip-handling example considered above, may look like in Figure 4.

The abstract BPEL4WS specifications (also sketched as WSCL diagrams) of the three partners of this process, the customer, the flight agency, and the hotel booking service, may look like those shown in Figures 5, 6, and 7.

The task of a planner is to generate a BPEL4WS process based on this information that when executed will correctly interact with the given partner processes and implement the conversations specified in the WSCL diagram. This problems resembles much more a planning problem, because it

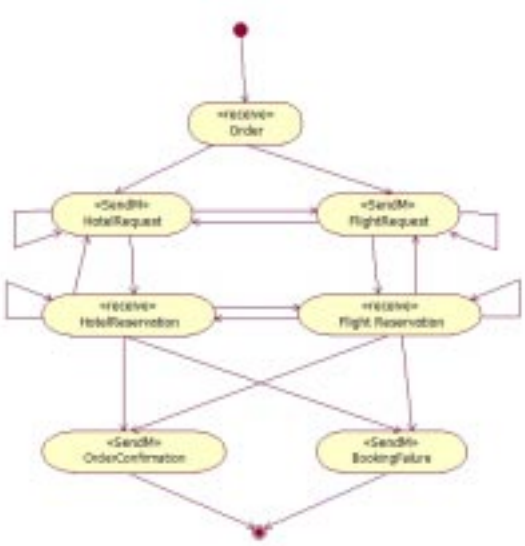

Figure 4: A WSCL Diagram for the Trip-Handling Process

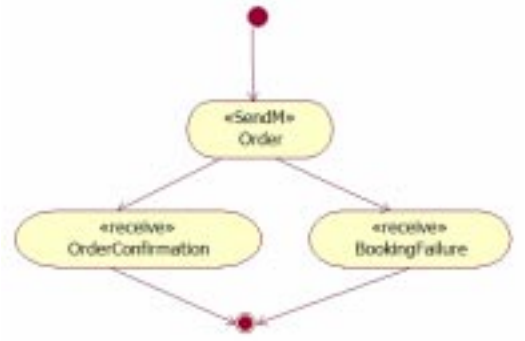

Figure 5: Conversational Behavior of the Customer

requires to select the necessary activities and their ordering. The difference is that again no explicit pre- and postconditions are given, but the conversational behavior of the various BPEL4WS activity types is known. It is also obvious that the plan must contain control structures to enable the cyclic conversations to be implemented.

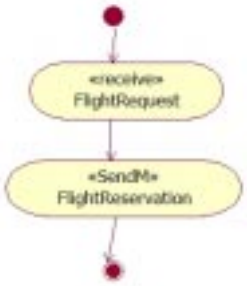

Figure 6: Conversational Behavior of the Flight Service

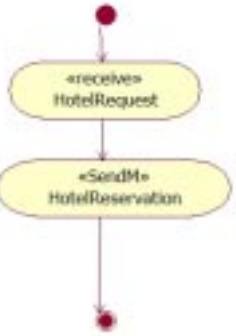

Figure 7: Conversational Behavior of the Hotel Service

Continual Optimization of Processes and on-Demand Composition on Workflows

There are two key concerns while composing Web services. Since applications are built for specific business requirements (which may not necessarily be explicitly stated), the composition of services has to be semantically compatible with the business domain in order to be useful. Another consideration is that the composed Web service should be executed efficiently by leveraging the inherent optimization opportunities, e.g., concurrency, in the continually running web service flow, cf.(Nanda, Chandra, & Sarkar 2003). These concerns were illustrated with a very simple example in Table 1.

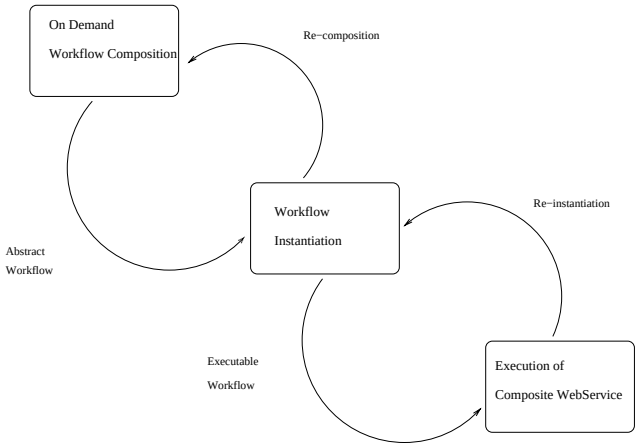

Figure 8: 2-stage View of Web Service Composition

Web service composition can be viewed as a 2-step process (see Figure 8) that addresses these concerns separately. In the first stage, the workflow representing the composition of the Web service composition may be produced on demand using planning techniques. The planning process can be implemented as a synthesis from scratch, as plan reuse involving a revision of previous cases, or as a semi-automated refinement of abstract workflows through the mixed-initative of planner and user. In the second stage, a composite process representing an instantiation of the workflow has to be pub-

lished, deployed and made available to the clients, so that it can be used just like any other atomic Web service.

However, the stages are not independent. While the composite web service is running, various runtime metric will be monitored and alternative instantiations or workflows, or both may offer better execution performance. The user may also change the requirements for the composition, which can be an additional driver for the generation of an alternative and workflow.

In planning, decoupling of causal reasoning and resource reasoning has been investigated earlier (Srivastava, Kambhampati, & Do 2001) where constraints were used to provide feedback between independent action selection and sequencing phases that iterated alternatively. This may also serve as a framework for the continual optimization of workflow compositions.

New Areas

In applying planning for Web service composition, some new areas have to be tackled.

Storage and retrieval of plans: As has been found in other applications of planning, users would want to use plans produced from previous planning episodes in guiding new compositions. The storage and retrieval of plans is related to the storage of workflows, but additional meta-data is available in planning about the objective and functionality of the plan. For example, information about how the plan was obtained (from scratch, with a user's participation or by reuse) is an important annotation for retrieving plans. In data integration, this is called the problem of *data provenance* and for plan storage, it may be termed *plan provenance*. Previous work on case retrieval for plans is relevant but there, the retrieval was performed in the context of a specific planner.

Plan Analysis and Optimization: Since a plan may be the best selection in one context and need not remain the same over time, techniques are required to analyze plans over an uncertain future (Garland & Lesh 2002). Optimization is of prime importance in web services because there can be many available Web service instances with drastically different invocation costs. Optimization has been given limited focus in planning: it was introduced with PDDL 2.1, but the metrics are rather simple. In this paper, we discussed with the help of a simple example that the continual optimization is the practical challenge even in the presence of simple objectives.

Plan Execution Monitoring: Previously, work has been done in monitoring of actions and overall plan execution. For Web services, the change is that the same plan or Web service composition will be executed continuously and the environment can evolve over time.

Conclusion

Building on our analysis of the gap between the needs of Web services composition for business applications like enterprise application integration and the current status of AI planning techniques, we argue that planning cannot be seen as a one-shot plan synthesis problem defined with explicit goals. Rather, it is a continual process of synthesizing, executing, optimizing, and maintaining complex workflows as goals get incrementally refined. We discuss example scenarios that illustrate new challenges for planning and identify additional areas that may become important for applying planning techniques.

References

Amsden, J.; Gardner, T.; Griffin, C.; Iyengar, S.; and Knapman, J. 2003. UML profile for automated business processes with a mapping to BPEL 1.0. IBM Alphaworks http://dwdemos.alphaworks.ibm.com/ wstk/ common/ wstkdoc/ services/ demos/ uml2bpel/ docs/ UMLProfileForBusinessProcesses1.0.pdf.

Ankolenkar, A., et al. 2002. DAML services. http://www.daml.org/services/.

Banerji, A., et al. 2002. WSCL: The web services conversation language. http://www.w3.org/TR/wscl10/.

Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The semantic web. Scientific American, May issue.

Blythe, J., et al. 2003. The role of planning in grid computing. Proc. ICAPS.

Box, D.; Ehnebuske, D.; Kakivaya, G.; Layman, A.; Mendelsohn, N.; Nielsen, H.; Thatte, S.; and Winer., D. 2000. Simple object access protocol (soap) 1.1. http://www.w3.org/TR/SOAP/.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2001. The web services description language WSDL. http://www-4.ibm.com/ software/ solutions/ webservices/ resources.html.

Curbera, F., et al. 2002. Business process execution language for web services. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Dal-Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In Dechter, R.; Kearns, M.; and Sutton, R., eds., *Proceedings of the 20th National Conference of the American Association for Artificial Intelligence*, 447–454. AAAI Press.

Deborin, E., et al. 2002. *Continuous Business Process Management with HOLOSOFX BPM Suite and IBM MQSeries Workflow*. IBM Redbooks.

DesJardins, M.; Durfee, E.; Ortiz, C.; and Wolverton, M. 1999. A survey of research in distributed, continual planning. *AI Magazine* 20(4):13–22.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In Hammond, K., ed., *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, 249–254. AAAI Press, Menlo Park.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *Eighteenth national conference on Artificial intelligence*, 461–467. American Association for Artificial Intelligence.

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In Biundo, S., ed., *Proceedings of the 5th European Conference on Planning*, LNAI. Springer.

Koehler, J.; Hauser, R.; Kapoor, S.; Wu, F.; and Kumaran, S. 2003. A model-driven transformation method. In *Proceedings of the 7th International IEEE Conference on Enterprise Distributed Object Computing (EDOC)*. IEEE Press.

McDermott, D. 2002. Estimated-regression planning for interactions with web services. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling*. AAAI Press, Menlo Park.

McIlraith, S.; Son, T.; and Zeng, H. 2001. Semantic web services. In IEEE Intelligent Systems (Special Issue on the Semantic Web), March/April 2001.

Nanda, M. G., and Karnik, N. 2003. Coordinating components in decentralized composite web services. In *Proceedings of the Association of Computing Machinery International Symposium on Applied Computing, Melbourne, Florida, USA*.

Nanda, M. G.; Chandra, S.; and Sarkar, V. 2003. Decentralizing composite web services. In *Proceedings of the Tenth International Workshop on Compilers for Parallel Computers (CPC), January 8-10, 2003, Amsterdam, The Netherlands.*

RDF. 1999. RDF: Resource description framework. http://www.w3.org/RDF/.

Scheer, A. W.; Abolhassan, F.; Jost, W.; and Kirchner, M. 2002. *Business Process Excellence - ARIS in Practice*. Springer.

Srivastava, B., and Koehler, J. 2003. Web service composition - current solutions and open problems. ICAPS 2003 Workshop on Planning for Web Services, Trento, Italy.

Srivastava, B.; Kambhampati, S.; and Do, M. B. 2001. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in realplan. *Artif. Intell.* 131(1-2):73–134.

Srivastava, B. 2002. Automatic web services composition using planning. In *Proceedings of 3rd Internation Conference on Knowledge-Based Computer Systems*, 467–477.

Staab, S., et al. 2003. Web services: Been there, done that? *IEEE Intelligent Systems, Jan-Feb issue.* 72–85.

Stephan, W., and Biundo, S. 1996. Deduction-based refinement planning. Proc. of AIPS-96, pages 213–220. AAAI Press, 1996.

The-Object-Management-Group. 2003. OMG model driven architecture. http://www.omg.org/mda.

Preference-based Treatment of Empty Result Sets in Product Finders and Knowledge-based Recommenders

Dietmar Jannach

Institute for Business Informatics & Application Systems
University Klagenfurt, A-9020 Klagenfurt, Austria
dietmar.jannach@ifit.uni-klu.ac.at

Abstract. "Your search returned 0 results" is an undesirable message for customers using an online product-finder or a web-based sales advisory system. Such a situation typically occurs when a filter-based recommender system is used and the user's requirements are unrealistic or inconsistent. In this paper, we present two orthogonal techniques for dealing with such situations, whereby the work is based on a general model of this class of recommendation systems.
First, an algorithm for priority-based filter relaxation is presented where the end user can actively participate in the problem resolution process by specifying the importance of the predefined advisory rules that led to the empty result.
Second, we adapt an algorithm from the field of model-based diagnosis and repair in order to compute a set of action alternatives for the customer. Each of these alternatives corresponds to a possible (small) revision of the originally inconsistent requirements, such that a product proposal can be made.
The paper finally describes practical results from experiences in different real-world application domains and discusses domain-specific heuristics and techniques for further search time improvements for complex problems and multi-user environments.

Introduction

Due to the huge variety of available products and services, many companies run product finders or more intelligent systems like recommender systems or sales advisory systems on their corporate web-sites. Based on the users' inputs, these systems filter out those products that fulfil the given specifications or match the needs and preferences of the customer. In particular for technical domains, where recommendations are not based on the concepts of "quality" and "taste", knowledge-based recommender systems have their specific advantages compared to other prominent approaches based on, e.g., collaborative filtering. Once the knowledge of the expert is made explicit and encoded in a knowledge base, the recommender system will behave like an experienced sales assistant and the quality of the recommendation will be constantly high, even if there are new products or new users involved. Even more, when adopting a knowledge-based approach, such systems will also be able to "explain" their recommendations to the customer. One of the main criticisms of such filter-based approaches [1] is that the undesirable situation can arise

where all products are filtered out and no proposal can be made. "Your search returned 0 results" is the only response of many online systems in such situations. An experienced sales assistant, however, would explain to his customer why there are no results, i.e., which advisory guidelines and rules he did obey. Even more, he could inform the customer what he can do about the situation, for instance reconsidering the potentially conflicting requirements, or he could even propose alternative solutions that satisfy most of the given customer requirements.

In this paper, we present two orthogonal techniques for dealing with such situations. First, we show how filter-relaxation based on priorities can be exploited in order to take the different importance levels of the given advisory guidelines into account and help the user in understanding the proposals. Second, we describe an algorithm for computing a suitable set of alternatives that tries to minimize the differences between the original requirements and other slightly changed requirements such that a solution is possible and respects the strict time restrictions of online recommendation systems.

Example and definitions

For demonstration purposes, we shall use a small example from the domain of digital cameras, a typical problem area where hundreds of different products are available and where online sales advisory systems are already used to assist the customer in finding the right product. We use the following very general model of a filter-based recommender system. Each *product* in the knowledge base is characterized by a set of attributes, whereby in many domains also set-valued attributes must be supported. Next, there is a set of *variables* that correspond to the user preferences, e.g., direct inputs or indirectly computed characteristics. Finally, a set of *filtering rules* describes the relation between the customer characteristics and the fitting product properties.

Our digital cameras are described by the property set *P*, whereby P = {weight, price, interfaces} and the following products exist in the product knowledge-base

PKB = {
{id(p1). weight(100). price(80). interface(usb)}
{id(p2). weight(100). price(150). interfaces(usb). interfaces(firewire).}
{id(p3). weight(200). price(200). interfaces(usb). interfaces(firewire). interfaces(dockingstation).}
}

During the advisory session the customer can specify his preferences by assigning values to the variables from the set *V* that contains *pref_weight (*with the domain "*low, medium, irrelevant")*, *pref_class (cheap, middle, high, premium, irrelevant)* and *pref_interfaces (standard, advanced)*[1]. We represent the actual customer requirements in a set *REQ* of positive ground literals that use the predicate symbols from *V,* e.g.,

REQ = {pref_class(premium). pref_weight(low). pref_interfaces(advanced).}

[1] Note that in many technical domains it is advantageous to question the customer about his preferences and not directly about product properties, in particular when the knowledge level of the customers can be low [2].

The expert's filter rules $FR = \{f1, f2, f3\}$ determining the contents of the result set RS are as follows, whereby the trivial axiom $RS \subseteq PKB$ has to hold.

$\boldsymbol{f1}$: $pref\_weight(low) \in REQ \Rightarrow \forall X,P: P \in RS: weight(X) \in P \wedge X < 150.$
$\boldsymbol{f2}$: $pref\_interfaces(advanced) \in REQ \Rightarrow \forall P: P \in RS: interface(firewire) \in P.$
$\boldsymbol{f3}$: $pref\_class(premium) \in REQ \vee pref\_class(high) \in REQ$
$\Rightarrow \forall X,P: P \in RS: interfaces(dockingstation) \in P \wedge (price(X) \in P \wedge X > 180).$

Given that the customer wants a premium class camera with an advanced set of interfaces and a model that also has a low weight, the application of the rules will result in no suitable product. One approach to deal with the problem is to assign priorities to each filter rule in advance and to iteratively retract the rules until a sufficient number of products remains. Retracting the weight-rule *f1* results in product *p3*, retracting *f2* alone will not help, and retracting *f3* leads to product *p2*. If we assume that the domain expert annotates the filter rules with initial priorities $f3 > f1 > f2$, because from experience he knows that the Firewire – requirement (*f2*) is not that important for most customers, the system will initially come up with solution *p3*, i.e., the rules *f2* and *f1* will be relaxed. If we assume that for each rule an explanatory text is maintained both for the case that the rule is applied and for the case it is relaxed, the system could explain:

- (f3 - positive): Based on your preferences, I propose the premium or high class model "p3" that also ships with a convenient docking station.
- (f1 - negative): Due to your other requirements, I included a camera in the proposal that does not fulfill your requirements with respect to a low weight.

Note that although filter *f2* was retracted, the conditions of the filter are fulfilled for product *p3*. As such, we can include the positive explanation for *f2*.

- (f2 - positive): This camera fulfils your requirements on advanced interfaces.

After this initial proposal and the explanation, the customer can be given the possibility to dynamically change the priorities of the rules, if they differ from the domain expert's predefined priorities and trigger the computation of another result.

Nonetheless, there are domains where there are strict rules that should never be relaxed or situations where the customer interactively states that he explicitly stipulates the application of specific rules. Consequently, a situation with an empty result set still can arise. In this case, it would be helpful for the customer if the system provides a set of alternatives of slight changes in the requirements such that a product can be recommended. If we assume that all filter rules in our example are strict ones, some *good* options for the customer could be as follows.

- (1) If you change your weight requirement from "low" to "middle", I can propose the following products: p3.
- (2) If you change your class requirement from "premium" to "middle", I can propose the following products: p2.

Theoretically, there are lots of other alternatives that differ from the previous ones in their "quality". For instance, all repair-alternatives that contain (1), as well as additional changes in the requirements can be seen as suboptimal. Furthermore,

alternatives where requirements are completely taken back, i.e., changes where the removal of a predicate from *REQ* leads to a solution, are also not optimal.

After the description of the algorithm for preference-based relaxation in the next section, we will describe a technique for efficient computation of suitable repair alternatives in cases where no result exists.

We will use the following general definition of knowledge-based recommendation problems for the subsequent algorithms[2].

***Definition**: A Knowledge-based Recommendation Problem (KBRP) is a tuple <P, V, PKB, FR, REQ>, where P and V are sets of predicate symbols that are used for describing product properties and customer requirements. PKB represents the available products and is a set of sets of positive ground literals using the predicate symbols from P. FR is a set of logical sentences with the structure of "filter rules". REQ is a set of positive ground literals using the symbols from V and correspond to actual customer requirements.*
A"filter rule" is a logical sentence in form of an implication, where the antecedent is a logical expression where predicate symbols from V are allowed, and where the consequent describes restrictions on elements of PBK by the usage of predicate symbols from P.

***Definition**: Given a Knowledge-based Recommendation Problem KBRP<P, V, PKB, FR, REQ>, KBRP is a called a "Valid KBRP" if there exists a set REQ (including the empty set) such that there exists a subset RS of PKB where FR ∪ REQ ∪ RS is satisfiable.*

Generally, a *knowledge-based recommender* in this implementation independent problem formulation is a software system capable of computing a valid recommendation *RS*, given the parameters *PKB, FR,* and *REQ*, or reporting failure of finding a solution.

Priority-based relaxation

The algorithm for priority-based relaxation is straight-forward and from the basic idea similar to the Hierarchical Constraint Satisfaction approach [3]. We extend the KBRP<*P, V, PKB, FR, REQ*> with a function Φ that relates each filter rule from *FR* with a priority value, i.e., a positive integer, whereby higher values stand for lower priorities and zero means that a rule should not be relaxed. The algorithm takes KBRP, Φ and a search limit as input and returns a set of products that remain after a successful relaxation process or otherwise an empty set. In addition, for each product in the result set, the sets of applied and removed filter rules for the explanation

[2] We use first order logic as a representation language in order to facilitate a clear and precise presentation.
In this context, we use the more general term "knowledge-based recommendation problem"; in the literature, the terms "filter-based recommendation" and "filter-based product retrieval" are also used to describe this class of systems.

process is constructed, whereby we include those filter rules in the set of applied filters that were originally removed due to a higher priority, but would hold for a given product, see filter rule *f2* in the example section.

Algorithm RELAX(*PKB, FR, REQ,* Φ, limit)

```
result = ∅                                  // initialize the result
allrelaxed = ∅                              // remember everything we relaxed
relaxable = subset of elements f of FR where Φ(f) > 0
while (| result | < limit ∧ | relaxable | > 0)   // as long as there are relaxable filters
                                            // and the limit is not reached.
   priority = highest value of Φ(f), where f ∈ relaxable
                                            // determine the set of filters to remove.
   relaxset = subset of relaxable, were Φ(f) = priority
   relaxable = relaxable \ relaxset         // remove the set from the original set.
   allrelaxed = allrelaxed ∪ relaxset       // remember the removal
   result = filter(PKB, relaxable , REQ);   // call the recommender/product finder.
end while
// compute the correct explanation sets
// variable "explanations" contains triples of products, applied, and relaxed filters
explanations = computeExplanations(relaxable, allrelaxed, result)
return result;
```

Algorithm 1. Priority-based relaxation

Algorithm COMPUTEEXPLANATIONS(applied, relaxed, result)

```
// The algorithm computes a set of positive and negative explanations
// for each product in the result set by testing each filter in the applied
// and relaxed sets.
explanations = ∅;
for each p ∈ result
   // prepare the real sets for the explanations
   pos_arguments = applied
   neg_arguments = relaxed
   for each f ∈ relaxed                     // test all relaxed individually
      rs = filter(p, f , REQ)               // call the recommender, check if the
                                            // current product fulfils the filter rule
      if (| result | > 0)                   // if filter consistent for that product
         pos_arguments = pos_arguments ∪ f  // update the explanation sets, e.g.,
         neg_arguments = neg_arguments \ f  //  filter f2 in the example
   end for
   // add the new explanation tuple.
   explanations = explanations ∪ <p, pos_arguments, neg_arguments >
end for
return explanations
```

Algorithm 2. Computing adequate explanations

With regard to complexity of the algorithm, the computation of the result set without the explanations takes at most as many iterations as there are different values in the range of Φ. For each element in the result set, the number of iterations for the explanations is |*result*| * |*relaxed*|. Note that in a practical setting we will not pre-compute all explanations in advance but rather on demand, when a user asks for the explanation for a specific product.

Computing repair alternatives

In domains where some of the rules are strict and cannot be relaxed definitely, a situation with an empty proposal can still arise. Then, it would be helpful if the system can propose the customer a set of "repair" actions which corresponds to slight changes and revisions of the initial requirements. In principle, two basic approaches are possible. First, the underlying recommendation knowledge base can be extended with additional domain knowledge, i.e., a set of explicitly modelled repair rules like shown in [4] for the domain of product configuration and reconfiguration. Such approaches, however, are costly in terms of knowledge acquisition and in particular maintenance as for each change in the knowledge base also the repair rules have to be checked and possibly adapted. Even more, such knowledge bases tend to get complex because of the strong interdependencies between the rules. On the other hand, model-based (diagnosis) approaches like, e.g., [5] or [6], that try to minimize the amount of additional needed domain knowledge for the repair task, face the problem of large search spaces for the computation of possible repair alternatives. Applied to real-world problem settings, such systems rely on domain-specific heuristics for discriminating between the alternatives or different forms of (structural) abstractions in order to produce adequate results within limited resource bounds.

If we follow such a model-based approach, the search complexity in our domain is determined by the size of the domain of the variables (i.e., the user inputs) and the property, whether these variables can be multi-valued or not. For each single-valued variable with a domain-size of n, we have $n+1$ possible assignments when we include a special *null*-value with the meaning that the user did not specify a requirement for a variable. For each multi-valued variable, there are 2^n possible answer combinations. If we assume that there are three single-valued and three multi-valued variables involved in the remaining strict rules and we have an average number of four possible answers, there are theoretically more than 60.000 possible answer combinations. Obviously, an exhaustive search for those combinations is not feasible, given the tight time limits of an online recommendation system[3].

In the following, we present algorithms and techniques for efficient computation of a suitable set of alternatives where we use domain-independent heuristics as well as application-dependent variants in order to reduce the required search times for practical settings.

Diagnosing the requirements. Depending on the application domain it can be sufficient that the system presents the user a list of requirements that should be

[3] An offline pre-computation for *all* possible input combinations is not possible, given the vast search space for, e.g., twenty to twenty-five questions in a realistic scenario.

completely retracted for a possible solution, e.g., "*If you skip your requirements on the weight, I can propose the following solution...*" In such cases, the computation of alternatives can be performed by using the standard *Hitting-Set* algorithm that is commonly used in the field of model-based diagnosis ([7], [8]). The algorithm is used in a way that the nodes of the Hitting-Set DAG[4] are labelled with *conflicts* which are in our case subsets of user requirements that cause the result set to be empty. If there is no conflict generation support, it will be the union of all variables that are used in the non-relaxable filters that have to be applied in the current situation.

Definition. *Given a Knowledge-based Recommendation Problem KBRP<P, V, PKB, FR, REQ>, a "conflict" for KBRP is a set* $C \subseteq REQ$ *such that there is no set* $RS \subseteq PKB$ *for which* $FR \cup RS \cup C$ *is satisfiable.*

A conflict is "minimal", if there is no proper subset C' of C such that C' is also a conflict for KBRP<P, V, PKB, FR, REQ>.

The outgoing edges of the DAG are labelled with variable names from the nodes which are retracted in the current search phase. The main advantage of this approach is that the diagnoses are computed in ascending order with respect to their cardinality, which is reasonable because we assume that alternatives with fewer changes in the requirements are preferred by the users. Even more, the tree pruning techniques from [7] can be exploited to reduce the search space, as we are typically only interested in *minimal diagnoses*. As an example, if we found that omitting the *weight* requirement results in a solution, we can remove branches in the search space that involve the *weight* requirement and any additional requirement.

As a simple heuristic which helps us to find a first solution more quickly is to try those variables first which are involved in more non-relaxable filter rules.

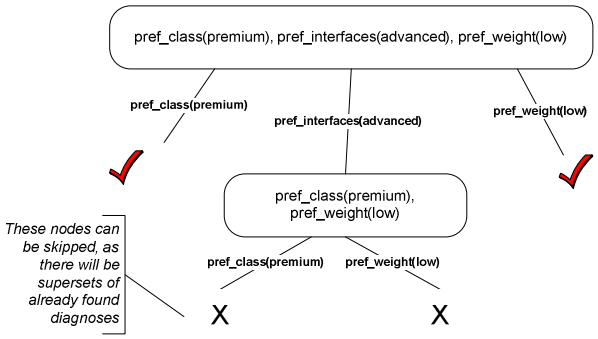

Fig. 1. A simple example for HS-DAG construction

During the breadth-first HS-DAG construction (Fig. 1), each call to the *Theorem Prover (TP)* [7] corresponds to a search for products performed by the underlying knowledge-based recommender, where we leave out all requirements that are on the path from the root node of the DAG to the current node. If this search results in a

[4] Directed acyclic graph.

solution, we can close this node *n* and add *H(n)*[5] [7] to the set of diagnoses. Note that for each TP-call all the other current user requirements that are not involved in the current set of non-relaxable filters have also to be taken into account. Therefore, in order to guarantee that the algorithm will always find at least one solution, we make the assumption that there are no filter rules in the knowledge base whose antecedent defines a condition on the *non-existence* of a requirement, like

$$\forall X: pref\_x(...) \notin REQ \Rightarrow \ldots.$$

If this – for many domains realistic assumption – holds, we can guarantee that by retracting user requirements no additional filter rules will get active during the search for solutions and we can thus safely prune the search tree.

Optimizations.

1. In general, we cannot assume that the underlying recommender system is capable of generating (minimal) *conflict sets*, such that we start with one single conflict that includes all problematic, i.e., conflicting user requirements. Nonetheless, during the HS-DAG construction, we remove elements from this conflict and check if a solution exists. If we encounter a situation where the removal of one input does not lead to a solution, we know that the remaining user inputs alone cause a conflict. As an online recommendation system will be used by many customers and subsequent customers may have a similar set of requirements, we can cache these already minimized conflicts and reuse them in the next session where a similar repair is needed. A safe reuse of such a conflict is possible if the cached conflict is a subset of the requirements of the next customer (compare, e.g., to [9]).
2. Because we allow disjunctions of predicates in the filters' pre-conditions, it can be the case that there are unassigned variables in the union set of the variables of the non-relaxable filters. Therefore, these variables can be removed from the initial *conflict* – which consequently reduces the search space – such that these variables do not have to be considered during the HS-DAG construction process.

Searching for alternative solutions. In some application domains, just presenting a set of requirements to be removed might not be satisfying for the customer. Even more, when individual requirements are fully removed, the customer might get a proposal that is not "near" to his original preferences. Fully removing a "low price" preference, for instance, could result in the proposal of premium priced products if no other filter rules constrain the price limit. Consequently, it would be preferable if the user can choose among several variants and interactively decide which of his requirements he is willing to give up or relax. In order to cope with such situations, the following value-based variant of the first approach can be used (see Fig. 2). Similar to the first approach, we proceed in a breadth-first approach. However, upon creation of a new node, all alternative settings for the variable under examination are tested individually except for the conflicting assignment. In cases, where more than one variable is tested (e.g., at the second level), we theoretically check all possible combinations of these variables, i.e., the Cartesian product of the possible values.[6]

[5] *H(n)* equals the set of all edge-labels from the root of the DAG to *n*.

[6] At the moment, the value-level approach handles user requirements with finite domains of enumeration values. For free-input requirements like a price limit we reduce the search space to respecting the requirement or not but do not search for alternative values.

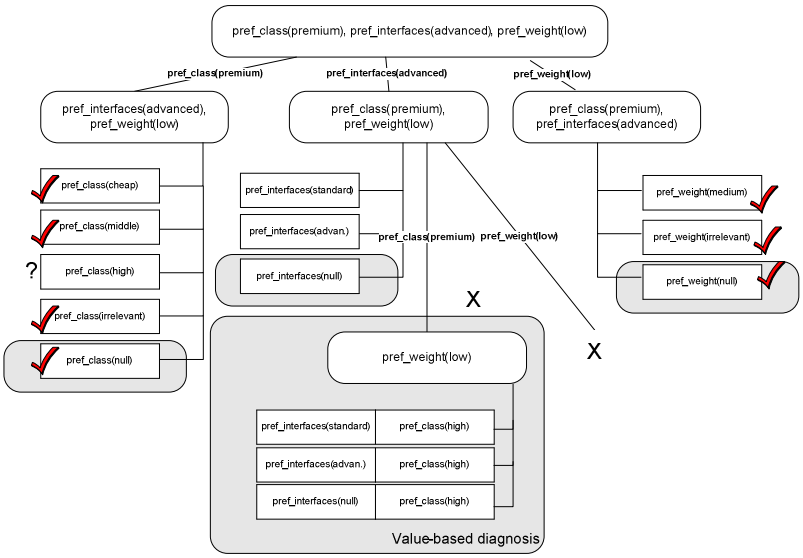

Fig. 2. Value-based algorithm

Note that in the value-based approach, we add a special "null" value (marked grey in Fig. 2) to the domain of each variable as the full removal of a requirement could be a possible option in some domains.

The main advantage of the breadth-first approach is that alternative solutions with fewer changes are found first. In practice, proposing alternatives with more than three or four variables changed does not help the users too much anyway, as the differences to the original preferences are not satisfying for the users. Therefore, in practical situations, the search is stopped when the first few alternatives are found.

When using the value-based approach, pruning has also to be performed on the value level. As we can see in Fig. 2, the node with the path (*pref_interfaces*, *pref_class*) cannot be immediately closed like in the first algorithm. Although we already found several solutions by changing the value of *pref_class* alone, there was no solution when we tried the value "*high*" (indicated by the question mark). Nonetheless, as we make no restrictions on the filter rules except for those on the structure described in the definitions, it could be the case that there exists a combination of *pref_class(high)* and some value of *pref_interfaces* such that a solution with changes in both variables is possible. However, note that we only have to check the combinations with *pref_class(high)* because combinations with other values of *pref_class* would result in a superset of an already found diagnosis. On the other hand, nodes that would involve *pref_weight* can be immediately closed since all alternative assignments to *pref_weight* result in a suboptimal solution.

Nonetheless, additional pruning approaches can be applied to reduce the search space, but come at the cost of loss of solutions. First, we could omit such cases as described in the last paragraph and prematurely prune the node with the path (*pref_interfaces*, *pref_class*), when we have domain-specific knowledge about the

filter constraints. In addition, for each node we can stop examining the possible value combinations, once we found a first solution, e.g., after trying the "cheap" value for *pref_class* we do not check the other alternatives. Such an approach can be useful in cases where we are only interested repair alternatives that involve single changes.

In general, the diagnosis algorithm will always find at least one solution given a valid recommendation problem, as the search space is exhaustively searched and no minimal solutions get lost by the value-based pruning techniques.

Discriminating between diagnoses. In cases when there are lots of solutions, i.e., repair alternatives returned by the search alternatives, these proposals should be ranked such that the customer can choose the alternative that matches his preferences best. A quite intuitive ranking will be based on the number of changes that are required in the requirements. Depending on the domain, other application-specific orderings can be used in order to rank the alternatives with the same number of changes. First, we can prefer those alternatives that maximize the number of products that can be proposed if the changes are applied. Another ordering can take the differences to the original requirements on the value level into account. In many domains, the possible values for a specific variable have an implicit ordering, like a price preference (low, medium, high etc.). As a consequence, from a practical perspective, it can even be better to promote an alternative where two of the requirements are changed to a neighbouring value, than to propose a single-change alternative where a customer's preference has to be inverted, e.g., changed from "yes" to "no". In principle, an initial "cost model" for changes can be computed without further domain knowledge, if we assume that the possible values for a variable are defined in such an implicit order. Before the results are presented to the user, the system can rank the alternatives based on these costs which are determined by the *distance* to the original requirement and the overall number of the possible values[7]. For multi-valued variables, changes where the original requirement statement and some more or fewer values lead to a solution can be for instance preferred over others that have complete different values.

If desired, such a model of "change costs" can also be explicitly defined in the knowledge base. The definition of such a model however can require significant knowledge acquisition efforts as these cost functions have to be defined manually for each variable. Nonetheless, compared with approaches where "repair rules" are explicitly defined, the definition of cost functions has the advantage that changes in the model only have local effects and the overall consistency of the knowledge base is not affected.

Finally, when such a cost model exists, it can be exploited during the search phase in settings where we perform an incomplete search and stop at a certain threshold, e.g., when a defined number of alternatives is found. As an example, we could stop examining other possible values for a price preference, when we found that a change from "low" to "medium" results in a solution, as we know that increasing the price limit further will only result in a suboptimal alternative with respect to the costs, i.e., the quality of the repair alternative.

---

[7] The number of possible values is important, because the *distance* from "yes" to "no" is small, but there are no other alternatives, so the *cost* estimate should be correspondingly high.

Experimental results

The described algorithms were implemented as add-on to the ADVISOR SUITE [10] framework, a research toolkit for rapid development of personalized online sales advisory systems. For the evaluation process, knowledge bases from several real-world problems from different application domains were tested. The application domains range from complex areas like investment alternatives to technical items like digital cameras or skis, up to "quality and taste" domains like wine or cigars.

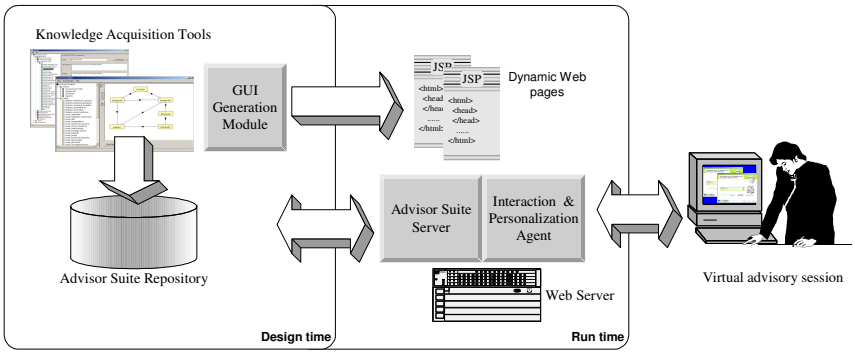

Fig. 3. Overall architecture of ADVISOR SUITE

Figure 3 depicts the overall architecture of the ADVISOR SUITE framework. The required knowledge for recommendation and personalization is captured using graphical knowledge acquisition tools and stored in a knowledge base which is built on top of a relational database system. In the Java-based ADVISOR SUITE SERVER module, the core recommendation logic is implemented; the individual advisory sessions are managed by the INTERACTION AND PERSONALIZATION AGENT. Short response times and high overall performance are key issues for such online Web-based systems. In general, the knowledge-based ADVISOR SUITE framework thus extensively caches and pre-loads the contents of the knowledge base and pre-compiles the recommendation rules into a compact internal format, such that a fast rule-evaluation process is possible.

Problem size. In our real-world advisory problems, the knowledge-base typically comprises twenty to thirty questions (variables) whereby – based on a personalization process – not every user is asked every question [2]. The number of filtering rules that determine the product selection mostly remained manageable and ranges from twenty to sixty expert rules[8]. The number of available products varies with the domain, from a few dozens when "services" are recommended, to several hundred when the domain is investment advisory or digital cameras.

Priority-based relaxation. The computation of explanations in terms of applied and relaxed advisory rules (Algorithm 1) is not problematic, as the search complexity

[8] The overall knowledge base is more complex, as it contains additional knowledge for the personalization of the dialogue flow, the adaptive user interface, as well as for the personalized ordering of the results records.

is linear with the number of priority levels, typically not more then twenty or thirty. Each check whether a sufficient number of results will be available upon relaxation of the filter rules on a certain level, can be performed within 10 to 20 milliseconds on a standard PC and database system depending on the number of available products, the overall response times for all tested cases were below one second. From the end-user perspective, however, this feature was highly appreciated by users in all domains where such an advisory application was implemented. First of all, the existence of a personalized explanation significantly increases the customers' confidence in the proposal, i.e., when reading the natural language explanations of the applied advisory rules and the justification for the products the user implicitly "learns" things about the domain. Furthermore, the possibility to stipulate the application of individual rules or decrease the priority of a rule lets the user express his real preferences in an intuitive and comprehensible way, in cases where the priorities which are predefined by the domain expert do not match his personal interests. Nonetheless, we also found that some of the users were overwhelmed when they are asked to assign relative priorities to the rules, such that in most cases we followed an approach where the only possibilities are to force the application of a rule or completely ignore it and to undo these choices.

Search for alternatives. The search for possible alternatives is computationally complex and has to be fine-tuned for each application domain. In typical applications, the list of explanations of the expert rules leading to the empty result set is presented to the user, followed by the list of repair alternatives. Depending on the domain, the complexity of the problem, and also the skill-level of the users, we have to tune the parameters of the diagnosis process: In some cases, only single-step or two-step repairs are comprehensible for the users; in other domains users want to have a choice of many different and possibly complex alternatives. In most cases, the maximum cardinality of the desired repairs is three or four as repairs that involve more changes are hard to comprehend and asses for the end user. If for instance five or more of the originally fifteen requirements have to be changed, the customer's confidence in the proposal might be low, because the product that is proposed after the change does not match a large part of his preferences.In our test cases, the computation of diagnoses up to cardinality three with single-valued attributes showed acceptable response times below one second for the computation of all possible repair alternatives without any further optimization, even in cases where nearly all of the customer requirements were involved in a conflict. For the computation of diagnoses with higher cardinality, we introduced several optimizations described as follows, such that the response times stay within the acceptable limit of two or three seconds for the hardest real-world problems with cardinality five.

1. *Result caching*. For each node in the search tree, we have to check whether changing a value results in a solution or not which corresponds to a call to the recommender system. In the ADVISOR SUITE framework, the results of previous sessions (together with explanations and repairs) are compactly stored in memory and on disk such that the system constantly "learns" new relations between

requirements and solution[9]. Our experiences show that only a fragment of the theoretically possible input combinations actually occur and many users have same or similar requirements such that the space requirements for this cache are limited. A typical example for such a "pattern" in user requirements is that customers who specify a high price-limit also have a high preference for brand products. We encode such previous results (and also those that origin during the diagnosis process) compactly, such that the check for a solution for an already known combination is then less then one millisecond. Finally, we also cache the possible repair alternatives which can subsequently be accessed without reasoning, when the same set of requirements occurs a second time. The experiences of an application with about fifteen thousand online advisory sessions in the first week of deployment shows that a high "cache-hit" rate can be achieved that justifies the limited overhead of managing such a cache.

2. *Conflicts re-using*. The same principle of such a system-wide cache can also be applied for conflicts, as in particular the computation of minimal conflicts can be a time-consuming task.
3. *Domain-dependent heuristics*. Finally, domain-dependent heuristics can help us in reducing the search space, in particular for multi-valued requirements that significantly enlarge the size of the search space. In many domains, we know for each multi-valued attribute, whether more values in such a requirement reduce the set of possible products or broaden this set, e.g., when the requirements have an implicit "one-of" semantics. Therefore, depending on the semantics, we can limit the search to alternatives with more or fewer values in the requirements and prune out all other constellations, which will not result in good solutions.

In general, the diagnostic approach for the search of repair alternatives has to be more parameterized and fine-tuned for a specific application than the relaxation technique. In particular, special attention has to be paid that the underlying complexity is hidden from the user and the user is not overwhelmed by a large set of complex alternatives. From the user interface perspective it is therefore important that we for instance aggregate the alternatives on the same variable set (e.g., "change the weight requirement to *middle* or *irrelevant*") and give the user the chance to accept and apply the alternative in a single interaction.

Conclusions

We have presented two techniques for dealing with empty result sets in knowledge-based recommender systems, whereby we based our work on a general model of this class of recommender systems that in practice base the search on product filtering.

The technique described for finding alternative requirements follows the tradition of approaches described in ([6],[8],[11], or [15]), where similar algorithms were used for computing action repair actions both for the domains classical model-based

[9] The pre-computation of all possible user input combinations and corresponding results is not feasible both with respect to time and space requirements. The cache has to be invalidated upon the periodic changes in the knowledge base, e.g., when new filter rules are introduced.

diagnosis (e.g., electronic circuits) as well as for the domains of re-configuration and debugging of software and knowledge-bases. The algorithms in this paper are designed in a way that they can be implemented non-intrusively as add-on to existing advisory systems and product finders. The only requirements are that filter rules can be selectively removed and added to the knowledge base (priority-based relaxation) and user inputs can be removed or changed dynamically (search for repair alternatives). The underlying implementation of the product finder has not to be changed or known for the application of the described techniques.

From the end user perspective, the usage of one or the other of the techniques adds a preference aspect ([12], [13]) to the advisory system, where the personalization of proposals is not only limited to preferences expressed on initial user requirements; the end user is also given a certain degree of freedom in his choice of repairs (compromises in the conflicting requirements) and the prioritization of advisory rules. The experiences show that the acceptance of the system and the confidence in the proposal increase, when the end user is able to understand and manipulate the results of the advisory process in these ways.

The presented work also strongly corresponds with the field of "Cooperative Answering" [18] in database systems, where the problem exists, that direct answers to database queries like "yes" or "no" are not always the best answers, i.e., an intelligent system would e.g. allow for the usage of "soft constraints" or preferences and cooperatively provide extra or alternative information. In the work of [19], for instance, automated analysis of the individual parts of a failing query is proposed, such that the system can provide an explanation which parts of the query caused the failure. While the approach in [19] involves high costs for identifying such sub-queries, this division in sub-queries in our application domain is indirectly given by the filter constraints themselves. In addition, our approach allows us to define natural-language explanations for failed sub-queries as well as interactive, preference-based, and non-technical selection of sub-queries to be applied.

Our future work includes the analysis whether similar techniques can be applied to the class of recommendation systems based on Case-based Reasoning ([16], [17]) in particular when the *case base* has to be updated after changes of the recommendation rules or when new products are available. Vice-versa, it will be analyzed if techniques from the CBR field can be adopted for filter-based recommender systems. In addition, we will further evaluate successful approaches from the areas of configuration and constraint satisfaction; in particular techniques based on *local search* [14] whose principle of iterative solution improvement matches the requirements of our application domain. Finally, further work will be spent on personalizing and improving the quality the resulting explanations such that they contain no spurious elements (see, e.g., [20]) or are adapted according to the current user's skills or preferences.

References

[1] D. Bridge. Product recommendation systems: A new direction. In R. Weber and C. Wangenheim, editors, Procs. of the Workshop Programme at the Fourth International Conference on Case-Based Reasoning, p. 79-86, 2001.

[2] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, M. Zanker, and R. Schäfer. A framework for the development of personalized, distributed web-based configuration systems. AI Magazine, 24(3):97-110, 2003.

[3] T. Schiex, H. Fargier, and G. Verfaille. Valued constraint satisfaction problems: Hard and easy problems. In International Joint Conference on Artificial Intelligence, p. 631-639, Montreal, Canada, 1995.

[4] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen. Framework and Conceptual Model for Reconfiguration. In Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05. AAAI Press, 1999, p. 59-64.

[5] S. Srinivas and E. Horvitz, Exploiting System Hierarchy to Compute Repair Plans in Probabilistic Model-Based Diagnosis, In: Proceedings of Eleventh Conference on Uncertainty in Artificial Intelligence, Montreal, 1995, Morgan Kaufmann, p. 523-531.

[6] G. Friedrich, G. Gottlob, and W. Nejdl: Formalizing the Repair Process - Extended Report. Annals of Mathematics and Artificial Intelligence, Vol. 11(1-4): 187-201 (1994)

[7] R. Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1), Elsevier, 1987, p. 57-95.

[8] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner. Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence, 152(2), p. 213-234, 2004.

[9] R. Greiner, B.A. Smith, R.W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence, 41(1), Elsevier, 1989, p. 79-88.

[10] D. Jannach and G. Kreutler, Building on-line sales assistance systems with ADVISOR SUITE, In Proceedings: 16th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE'04), Banff, CAN, 2004.

[11] M. Stumptner and F. Wotawa, Reconfiguration using Model-based Diagnosis, in: Proceedings of the International Workshop on Diagnosis (DX99), June 1999.

[12] U. Junker, Preference-Based Search for Scheduling. In Proceedings: AAAI/IAAI, Austin, TX, USA, 2000. p. 904-909.

[13] U. Junker, Preference programming for configuration, In Proceedings IJCAI'01 – Workshop on Configuration, Seattle, 2001.

[14] R. Sosic and J. Gu. Efficient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. IEEE Transactions on Knowledge and Data Engineering, Vol. 6, 5, p. 661-668, Oct 1994.

[15] L. Console, G. Friedrich, D. T. Dupré: Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. IJCAI 1993, Chambéry, France, p. 1494-1501.

[16] R. Burke, The Wasabi Personal Shopper: A Case-Based Recommender System, In Proceedings: AAAI/IAAI, Orlando, Florida, 1999, p. 844-849.

[17] R. Burke, Knowledge-based Recommender Systems. In A. Kent (ed.), Encyclopedia of Library and Information Systems. Vol. 69, Supplement 32. Marcel Dekker, 2000.

[18] T. Gaasterland, P. Godfrey, and J. Mincker, An overview of Cooperative Answering, Journal of Intelligent Information Systems Vol. 1(2), pp. 123-157, Kluwer, 1992.

[19] J.M. Janas. On the Feasibility of Informative Answers. In: Gallaire et al., Advances in in Database Theory, Vol. 1. Plenum Press, 1981

[20] G. Friedrich, Elimination of spurious explanations, Proc. of 16th European Conference on Artificial Intelligence, Valencia, Spain, 2004.

Entwicklung eines Double Dummy Skat Solvers

Sebastian Kupferschmid

Albert-Ludwigs-Universität Freiburg
Email: kupfersc@informatik.uni-freiburg.de

Zusammenfassung Dieser Artikel präsentiert, wie man mit einem $\alpha\beta$-Algorithmus zusammen mit einer *Monte-Carlo-Simulation* Skat spielen kann. Der Schwerpunkt liegt dabei auf Erweiterungen des $\alpha\beta$-Algorithmus, die diesen drastisch beschleunigen. Unter anderem wird eine Erweiterung von M. Ginsbergs *Partition Search* vorgestellt, die es erlaubt Symmetrien beim Skat auszunutzen. Desweiteren stelle die Arbeit eine skatspezifische *forward pruning*-Methode vor, die es ermöglicht in bestimmten Situationen vorzeitig den Spielbaum zu beschneiden, ohne die Korrektheit des Algorithmus zu verletzten. Dadurch lässt sich der Algorithmus signifikant beschleunigen. Diese Methode kann auch auf andere Spiele angewandt werden, wenn diesen ein ähnliches Punktesystem wie dem Skatspiel zugrundeliegt.

1 Einleitung

Für Spiele, in denen vollständige Informationen vorliegen, kann der spieltheoretische Wert bzw. eine optimale Strategie mit dem Minimax-Algorithmus bestimmt werden. Allerdings funktioniert die Anwendung dieses Algorithmus in Spielen mit unvollständiger Information nicht mehr.

Im Folgenden wird am Beispiel des Skatspiels gezeigt, wie man dieses Problem mit einem so genannten *Monte-Carlo-Ansatz* (MC) umgehen kann. Die Verwendung des MC-Ansatz, um Spiele mit unvollständiger Information zu lösen, ist nicht neu. Er kam unter anderem in M. Ginsbergs GIB[1] [3], einem Bridgeprogramm, zur praktischen Anwendung.

Bridge und Skat sind sich in vielen Punkten recht ähnlich, aber es gibt auch wesentliche Unterschiede. Beim Bridge entscheidet z. B. nur die Anzahl der gewonnenen Stiche über den Ausgang des Spiels, während diese Größe beim Skat nur eine untergeordnete Rolle spielt. Hier ist die Summe der Punktwerte der Karten in den gewonnenen Stichen spielentscheidend. Zusätzlich kompliziert wird das Problem dadurch dass, der Zusammenhang zwischen dem Punktwert und der Ranghöhe der Karte nicht proportional ist, z. B. ist die ranghöchste Karte (♣J) nur 2 Punkte wert, während jedes Ass 11 Punkte wert ist. Hierdurch gibt es weniger Symmetrien als beim Bridge, aber es gibt viele Situationen, die *fast* symmetrisch sind. Unter anderem zeige ich in diesem Artikel, wie M. Ginsbergs *Partition Search* erweitert werden kann, um diese *Pseudo*-Symmetrien auszunutzen.

[1] Goren In a Box oder Ginsberg's Intelligent Bridgeplayer

Dieser Artikel ist folgendermaßen strukturiert: nach einer kurzen Vorstellung des MC-Ansatzes in Abschnitt 2 wird damit eine Lösung für ein vereinfachtes Skatspiel (ohne Reizen und Drücken) präsentiert. In Abschnitt 3 wird das Herzstück des MC-Ansatzes, ein so genannter *Double Dummy Skat Solver* (DDS), vorgestellt. Ein DDS ist ein Programm, das für jede Position in einem Skatspiel, in dem mit offenen Karten gespielt wird, eine Karte bestimmen kann, deren Ausspiel den Gewinn maximiert. Der DDS ist durch einen $\alpha\beta$-Algorithmus mit einigen Erweiterungen realisiert worden, auf die ich ebenfalls eingehen werde. In Abschnitt 4 werden kurz Nullspiele besprochen. Das Problem des Reizens werde ich in Abschnitt 5 behandeln.

Im Wesentlichen besteht dieser Artikel aus Auszügen aus meiner Diplomarbeit [5], in der auch kurz die Skatregeln erklärt werden. Für ein verbindliches Regelwerk verweise ich auf die *Internationale Skatordnung* [4].

2 Monte Carlo

Im Zusammenhang mit Spielen besteht der MC-Ansatz darin, die unbekannten Informationen durch geratene Informationen zu ersetzen. Für das daraus resultierende Problem mit vollständiger Information wird dann der bzw. die besten Züge ermittelt. Durch wiederholtes Ersetzen der unvollständigen Informationen und anschließende Berechnung kann ein Spielzug gefunden werden, der in möglichst vielen Fällen der erfolgreichste Zug ist.

Angewandt auf das Ausspielen einer Karte in einem Skatspiel ergibt sich damit folgender Algorithmus:

1. Konstruiere eine Menge D von Kartenbelegungen, die konsistent mit den bereits gesammelten Informationen sind.
2. Berechne mit dem DDS für jede Kartenbelegung $d \in D$ und jede nach den Skatregeln spielbare Karte c den Wert des Spiels, den man erzielen kann, wenn diese Karte gespielt wird.
3. Spiele die Karte, die die meisten simulierten Spiele gewonnen hat.
4. Haben mehrere Karten dieselbe Anzahl an Spielen gewonnen, dann wird diejenige Karte davon gespielt, die die höchste kumulative Punktzahl in diesen Spielen erreicht hat.
5. Gibt es mehrere solche Karten, dann wird zufällig eine davon gewählt.

2.1 Auswahl der besten Karte

Die beste Karte wird aufgrund der in der Simulation gewonnenen Spiele bzw. der dort erreichten Punktzahlen ermittelt. Wird zu diesem Zweck nur eines der beiden Kriterien verwendet, dann ergeben sich daraus folgende unerwünschte Spielweisen:

– Angenommen es würde nur die höchste durchschnittliche Punktzahl als Bewertungskriterium verwendet werden, dann kann es passieren, dass eine Karte sehr wenige Spiele gewinnt und diese dafür sehr hoch. Solch eine Karte

würde dann gewählt werden, obwohl die Wahrscheinlichkeit für einen Sieg gering ist. Diese Vorgehensweise führt zu einem riskanten Spieler.
– Wählt man diejenige Karte, die die meisten simulierten Spiele gewonnen hat, dann kann sich folgende pathologische Situation ergeben: alle Karten haben dieselbe Anzahl an Spielen gewonnen. Jetzt ist unklar, welche Karte gespielt werden soll. Unter Umständen wird dann eine wertvolle Karte ausgespielt, die aber von der anderen Partei gestochen wird. Diese Spielweise hat zur Folge, dass das Spiel sehr knapp gewonnen wird. Bedingt durch die Ungenauigkeit der MC-Simulation kann es in diesem Fall sogar passieren, dass ein sicherer Sieg vergeben wird.

2.2 Probleme mit Monte Carlo

Der MC-Ansatz bietet die Möglichkeit, Algorithmen, wie z. B. $\alpha\beta$ in Domänen anzuwenden, in denen nur unvollständige Informationen vorliegen. Allerdings ist diese Herangehensweise auch problematisch.

Der offensichtlichste Nachteil ist, dass auf diese Weise nie ein Zug gemacht wird, durch den nur Informationen gesammelt werden sollen. In gewissen Situationen würde ein Skatspieler z. B. eine Karte ausspielen — und möglicherweise verlieren — um sich ein Bild über die gegnerische Kartenverteilung zu verschaffen. Solche Züge werden mit dem MC-Ansatz vermieden. Das liegt daran, dass das reale Spiel mit unvollständiger Information auf den Fall mit vollständigem Weltwissen reduziert wird. Sind alle Informationen bekannt, dann muss so ein Zug nämlich nie gemacht werden.

Allgemein ist die Optimalität der Züge bei dieser Herangehensweise nicht gewährleistet. Selbst wenn alle möglichen Ersetzungen der unvollständigen Informationen untersucht werden, kann dieser Ansatz zu falschen Ergebnissen führen, wie D. Basin und I. Frank in [1] gezeigt haben. Allerdings spielt M. Ginsbergs GIB mit einem MC-Ansatz auf hohem Niveau Bridge.

Das in meiner Diplomarbeit entstandene Programm hat mit diesem Ansatz zumindest XSkat[2] geschlagen. Leider kann ich ansonsten nichts über die Qualität des entstandenen Programms sagen, da mir bis jetzt die Gelegenheit fehlte, es gegen einen Experten spielen zu lassen.

3 Der Double Dummy Skat Solver

Durch den MC-Ansatz lassen sich, wie gezeigt, Spiele mit unvollständigen Informationen auf Spiele mit vollständigem Weltwissen reduziert. Aus diesem Grund werden ab jetzt nur noch Skatspiele mit offenen Karten betrachtet.

Da ein Skatspiel, wie sich gezeigt hat, im Mittel einen verhältnismäßig kleinen Spielbaum hat, kann dieser vollständig durchsucht werden. Das hat den Vorteil, dass nicht auf Heuristiken zurückgegriffen werden muss, die das Ergebnis der Suche beeinflussen. Dadurch erhält man den tatsächlichen, spieltheoretischen

[2] XSkat 3.4 von Gunter Gerhardt `http://www.gulu.net/xskat/`

Wert des Spiels. Allerdings ist eine Implementierung des DDS[3] durch reine $\alpha\beta$-Suche in der Praxis zu langsam, wie folgendes Beispiel in Abbildung 1 zeigt:

Es soll für jede Karte von Spieler 0, der in folgendem ♣-Spiel mit neun Karten der Alleinspieler ist, ausgerechnet werden, wie hoch das erreichbare Endergebnis ist, wenn diese Karte als erstes ausgespielt wird. Für diese Aufgabe benötigte eine reine $\alpha\beta$-Implementierung 187,88 s[4].

Spieler 0	♣7	♣8	♣Q	♣K	♣A	♢7	♢10	♢A	♠A
Spieler 1	♡J	♢8	♢9	♢Q	♠9	♡7	♡8	♡9	♡Q
Spieler 2	♢J	♣9	♡A	♠7	♠8	♠Q	♠K	♠10	♢K

Abbildung 1. Ein Skatspiel mit neun Karten pro Spieler

Um in annehmbarer Zeit eine statistisch signifikante Anzahl an Spielen berechnen zu können, muss der $\alpha\beta$-Algorithmus durch Erweiterungen beschleunigt werden. Der von mir entwickelte DDS ist eine *Nullfenster-Suche* (*Minimal Window Search*) mit *Transpositionstabelle* — einer Hashtabelle, in der Zwischenergebnisse zur Wiederverwendung gespeichert werden — und Zuganordnung. Hinzu kommen noch drei skatspezifische Erweiterungen, die die Suchzeit weiter senken und auf die ich später noch eingehen werde.

3.1 Nullfenstersuche

Statt mit einem initialen Fenster von $[-\infty, +\infty]$ zu starten, verwendet der Algorithmus das Intervall $[\delta, \delta + 1]$. Hierdurch steigt die Wahrscheinlichkeit, dass Knoten beschnitten werden. Wenn $\delta = 60$, dann ist ein Suchergebnis größer als δ gleichbedeutend mit einem Gewinn des Alleinspielers, analog verliert er bei einem Ergebnis kleiner als $\delta + 1$. Auf diese Weise kann zwar festgestellt werden, ob das Ausspielen einer Karte zum Gewinn des Spiels führt, aber nicht, welche Karte den höchsten Gewinn erzielt.

Durch diesen Ansatz erhält man statt der numerischen Werte aus dem $\alpha\beta$-Algorithmus ein boolsches Resultat. Liefert der Algorithmus *true*, dann ist die entsprechende Position eine Gewinnposition für MAX, den Alleinspieler.

3.2 Vorzeitiger Spielabbruch

Da ein Skatspiel i. A. nicht erst im letzten Stich entschieden wird, kann dieser Algorithmus um einen Test erweitert werden, der überprüft, ob die für den Gewinn

[3] Der Name Double Dummy Solver ist übrigens aus der Bridge-Welt entlehnt. Dort bezeichnet er ein Programm, das unter Kenntnis der Karten aller Spieler und ohne Reizung Bridge spielt. Der Partner des Alleinspielers wird beim Bridge als Dummy bezeichnet. Nach dem ersten Ausspiel legt dieser seine Karten offen auf den Tisch.

[4] Alle Laufzeiten wurden auf einem Athlon-XP-2100+ mit 256MB Ram und gcc-3.2 gemessen.

notwendigen Punkte schon erzielt wurden. Wenn dem so ist, so kann sofort das entsprechende Ergebnis zurückgeliefert werden. Auf diese Weise muss der Suchbaum in den meisten Fällen nicht bis zur vollen Tiefe evaluiert werden, es kann sogar vorkommen, dass nie ein Blatt evaluiert werden muss. Dieser Algorithmus ist in Abbildung 2 zu sehen. Hierbei gibt `bound` den Wert an, mit dem MAX das Spiel gewinnt (normalerweise 61). Zuerst wird überprüft, ob eine Partei bereits die für einen Sieg notwendigen Punkte erzielt hat ($p.good\_points$ bezeichnet die bisher erzielte Punktzahl des Alleinspielers in p und $p.bad\_points$ die der Gegenspieler). Wenn dem so ist, wird das entsprechende Ergebnis zurückgeliefert. Wenn nicht, dann werden die Nachfolgepositionen untersucht. Eine Position p ist eine Gewinnposition für MAX, falls mindestens eine Nachfolgeposition von p ein Gewinn für MAX ist. Sind alle Nachfolger von p für MAX verloren, dann ist auch p eine verlorene Position, analog für MIN-Knoten.

Es lässt sich zeigen, dass

$$\forall \delta \in \{0, \ldots, 120\} \quad mws(p, \delta) = true \iff \alpha\beta(p, \delta, \delta + 1) \geq \delta + 1$$

In der Implementierung dieses Algorithmus fällt auf, dass es keine gesonderte Behandlung für Blätter des Spielbaumes gibt. Diese ist nicht notwendig, da spätestens mit dem letzten Zug eine Position erreicht wird, in der eine der beiden Parteien gewonnen hat.

```
def mws(p, bound):
    if p.good_points > bound:         # Hat Alleinspieler
        return True                   # schon gewonnen?
    elif p.bad_points >= 120 - bound: # Haben Gegner schon
        return False                  # gewonnen?

    if p isa MAX_Node:
        result = False
    else:
        result = True

    for q in succ(p):                 # Untersuche alle
        succ_win = mws(q)             # Nachfolgepositionen
        if p isa MAX_Node:            # Alleinspieler am Zug
            if succ_win == True:
                return True
        elif p isa MIN_Node:          # Gegner am Zug
            if succ_win == False:
                return False

    return result
```

Abbildung 2. *mws* mit vorzeitigem Spielabbruch

3.3 Memory Enhanced Test Driver

Der im letzten Abschnitt vorgestellte Algorithmus kann zwar bestimmen, mit welchen Karten ein Spieler gewinnen kann, gibt aber keine Auskunft über die Höhe des Gewinns. Wie in Abschnitt 2 erläutert, benötigt der MC-Ansatz aber u. U. die exakte Punktzahl, die durch das Ausspielen einer bestimmten Karte erreichbar ist.

In diesem Abschnitt wird das von A. Plaat, J. Schaeffer et. al. [9] entwickelte Rahmenprogramm *Memory enhanced Test Driver* (MTD) vorgestellt, mit dem sich dieses Problem umgehen lässt.

MTD benutzt einen Algorithmus zur Tiefensuche mit Nullfenster, um damit eine Bestensuche zu implementieren. Wie der Name schon sagt, wird dazu ein Algorithmus verwendet, der seine Zwischenergebnisse in einer Transpositionstabelle speichert. Die beste Aktion wird durch wiederholtes Suchen ermittelt. Um die Suche effizient zu gestalten, wird hierbei auf die Teilergebnisse von vorangegangenen Suchen zugegriffen. Wird z. B. zur Tiefensuche $\alpha\beta$ verwendet, dann kann mit diesem Ansatz der spieltheoretische Wert einer Position in kürzerer Zeit berechnet werden, als das ohne dieses Framework möglich wäre. Der Name MTD geht übrigens auf J. Pearls *Test-Procedure* zurück [8]. Auf die Nullfenster-Suche *mws* aus Abbildung 2 angewandt, ergibt sich der Algorithmus aus Abbildung 3.

```
def mtd(p, f):
    upper_bound = 120
    lower_bound = 0

    while lower_bound < upper_bound:
        if f == lower_bound:
            f += 1
        if mws(p, f) == True:
            f += 1
            lower_bound = f
        else:
            f -= 1
            upper_bound = f
    return f
```

Abbildung 3. *mtd* zusammen mit *mws*

Hierbei bezeichnet f den Startwert für die Suche nach dem höchsten Endergebnis, das von p aus erreichbar ist. Für eine Anfangsposition beim Skatspiel liegt dieser Wert zwischen 0 und 120. Liefert $mws(p,\ f)$ das Ergebnis *true* zurück, dann kann ausgehend von p ein Ergebnis besser als f erreicht werden. Infolge dessen wird die untere Schranke `lower_bound` angepasst. In einer darauffolgenden Suche kann dann mit einem höheren Wert für f nach dem besten Wert

gesucht werden. Analoges gilt für den Fall, dass *false* zurückgeliefert wird. Die Suche endet, wenn `lower_bound == upper_bound` gilt. In diesem Fall ist f der beste Wert, der von der Position p aus erreichbar ist.

3.4 Transpositionstabelle

Ohne den Einsatz einer Transpositionstabelle hat dieser Algorithmus natürlich eine länger Laufzeit als $\alpha\beta$. Beim Skatspiel wie auch bei anderen Spielen ist der Suchraum ein Graph, d. h. es gibt Knoten, die mehrere Eltern haben. Im Gegensatz dazu sind Minimax-basierte Algorithmen besser zur Baumsuche geeignet. Die Rechenzeit dieser Algorithmen kann drastisch reduziert werden, indem man die wiederholte Expansion eines Knotens vermeidet. Zu diesem Zweck werden in einer Transpositionstabelle die Ergebnisse bereits evaluierter Teilbäume gespeichert und bei Bedarf wiederverwendet.

Das Problem bei der Wiederverwendung von Teilergebnissen beim Skatspiel ist, dass den Blättern im Spielbaum akkumulierte Punkte zugeordnet werden. Betrachtet man einen Knoten p im Spielbaum, dann hängt dessen Bewertung vom Spielergebnis — der Bewertungen der Blätter in dem Teilbaum mit Wurzel p — ab. Damit aber ein Ergebnis wiederverwendbar ist, darf es nicht von den bisher erzielten Punkten abhängen, sondern nur von den Punkten, die in den Stichen im Teilbaum mit Wurzel p erzielt werden. Abbildung 4 visualisiert diese Problematik. Hierbei seien die Restspiele, die in p und q beginnen gleich.

Formal: Sei P die Menge aller Spielpositionen und $T : P \to \mathbb{N}$ die Funktion, die einer Spielposition p die Zahl der Punkte zuordnet, die MAX bisher erzielt hat (Punkte aus Skat + bisher gewonnene Stiche). Weiter bezeichne $t(p)$ die Zahl der Punkte, die MAX in p erzielt hat, nachdem er oder einer der Gegner die letzte Karte eines Stichs ausgespielt hat. Wenn $(p_1, \ldots, p_n = p)$ ein Pfad von der Wurzel zu p ist, dann gilt $T(p) = \sum_{i=1}^{n} t(p_i) + skat$.

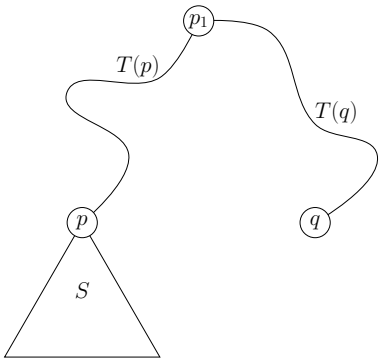

Abbildung 4. Problematik akkumulierter Punkte. Die Restspiele in p und q seien gleich.

Der Einfachheit halber wird hier beschrieben, wie sich dieses Problem beim Minimax-Algorithmus lösen lässt. Eine Anwendung auf $\alpha\beta$ funktioniert ähnlich.

Satz 1 *Sei P die Menge aller Spielpositionen, $p \in P$ und $succ(p)$ die Menge der unmittelbaren Nachfolgepositionen der Position p. T und t seien wie oben definiert. Mit $M(p)$ werde die durch den Minimax-Algorithmus definierte Funktion*

$$M(p) = \begin{cases} T(p) & \textit{falls } p \textit{ Blatt} \\ \max\limits_{p' \in \mathrm{succ}(p)} M(p') & \textit{falls } p \textit{ MAX-Knoten} \\ \min\limits_{p' \in \mathrm{succ}(p)} M(p') & \textit{falls } p \textit{ MIN-Knoten} \end{cases}$$

bezeichnet. Sei $m : P \to \mathbb{R}$ wie folgt definiert:

$$m(p) = \begin{cases} 0 & \textit{falls } p \textit{ Blatt} \\ \max\limits_{p' \in \mathrm{succ}(p)} (t(p') + m(p')) & \textit{falls } p \textit{ MAX-Knoten} \\ \min\limits_{p' \in \mathrm{succ}(p)} (t(p') + m(p')) & \textit{falls } p \textit{ MIN-Knoten} \end{cases}$$

Dann gilt

$$\forall p \in P \quad M(p) = m(p) + T(p).$$

Diese Behauptung ist mit vollständiger Induktion leicht zu beweisen. Somit lässt sich für die Berechnung von $M(p)$ die Berechnung der verbleibenden Punkte $m(p)$ von den bereits erzielten Punkten $T(p)$ entkoppeln.

Speicherplatzbedarf Mit dem Einsatz einer Hashtabelle ergibt sich sofort die Frage nach dem benötigten Speicherplatz für selbige.

Die Anzahl der verschiedenen Positionen, die im Verlauf eines beliebigen Skatspiels auftreten können, lässt sich durch folgenden Summe nach oben abschätzen:

$$\underbrace{3 \cdot \sum_{i=0}^{9} \binom{10}{i} \binom{10}{i+1}^2}_{\substack{\#\text{Positionen mit einer} \\ \text{Karte auf Tisch}}} + \underbrace{3 \cdot \sum_{i=0}^{9} \binom{10}{i}^2 \binom{10}{i+1}}_{\substack{\#\text{Positionen mit zwei} \\ \text{Karten auf Tisch}}} + \underbrace{3 \cdot \sum_{i=1}^{9} \binom{10}{i}^3 + 2.}_{\substack{\#\text{Positionen ohne} \\ \text{Karten auf Tisch}}}$$

Die Auswertung dieses Ausdrucks ergibt 316.581.176. Die 2 im letzten Summanden steht für die Anfangs- und die Endposition. Es lässt sich zeigen, dass es genügt, nur Positionen, bei denen keine Karten auf dem Tisch liegen in der Tabelle zu speichern. Hierdurch kann die Zahl der Einträge in der Transpositionstabelle durch $3 \cdot \sum_{i=1}^{9} \binom{10}{i}^3 + 1 = 114.495.775$ nach oben abgeschätzt werden.

Natürlich benötigt man für das Speichern in der Hashtabelle eine effiziente Repräsentation für solche Spielpositionen. Da es genau 32 Spielkarten gibt, ist es geschickt diese durch je ein Bit in einem 32-Bitwort zu repräsentieren. Eine

Menge von Karten kann so eindeutig durch die bitweise Disjunktion der Repräsentationen ihrer Karten dargestellt werden. Auf diese Weise lassen sich die Karten, die ein Spieler zu Beginn des Spieles hatte in einer *Maske* speichern. Da sich diese Informationen während des Spiels nicht ändern, kann ein Spielstand durch die Repräsentation der noch nicht ausgespielten Karten dargestellte werden. Möchte man wissen, welche Karten ein Spieler in einer bestimmten Spielposition noch auf der Hand hält, genügt es, seine Maske mit der Repräsentation des Spielstandes bitweise zu konjugieren. Da die beiden Bits, die den Skat repräsentieren, sich während des Spiels auch nicht ändern, kann man in diese beiden Bits speichern, welcher Spieler die nächste Karte ausspielen muss.

Durch diese Repräsentation und die Tatsache, dass der $\alpha\beta$-Algorithmus im Mittel etwa 1.500.000 in die Transpositionstabelle schreibt, kommt man mit relativ wenig Speicher aus.

3.5 Äquivalenzklassen

Bisher wurden nur Erweiterungen von $\alpha\beta$ betrachtet, die man nicht nur beim Skat anwenden kann (außer vielleicht der vorzeitige Spielabbruch). In den folgenden Abschnitten werden zwei skatspezifische Erweiterungen vorgestellt, die die Laufzeit des Programms weiter beschleunigen.

Der bisher betrachte Algorithmus berechnen immer nur den Wert für eine Spielposition. Die Werte solcher Positionen werden zwar in der Transpositionstabelle gespeichert, doch kann man diese Ergebnisse nur dann wiederverwerten, wenn man bei der Suche genau diese Positionen ein zweites Mal analysiert.

Es lässt sich allerdings zeigen, dass man mit diesen Ergebnissen den Wert für andere, noch nicht besuchte Positionen nach oben und unten abschätzen kann. Dadurch wird die Suche erheblich beschleunigt. Solche Situationen treten z. B. auf, wenn ein Spieler in einer Position $\diamondsuit$ bedienen muss. Nun kann er sich u. a. für $\diamondsuit 8$ oder $\diamondsuit 9$ entscheiden. Der bisherige Algorithmus hätte dann zuerst den Wert für $\diamondsuit 8$ und dann für $\diamondsuit 9$ ausgerechnet. Mit Hilfe des Satzes über Äquivalenzklassen, der hier vorgestellt wird, wird eine dieser beiden Berechnungen überflüssig, da genau der gleiche Wert erzielt wird.

Diese Vorgehensweise geht auf *Partition Search* von M. Ginsberg [2] zurück. Im Prinzip ist Partition Search ein $\alpha\beta$-Algorithmus mit Transpositionstabelle. Der Unterschied besteht darin, dass in der Transpositionstabelle nicht einzelne Positionen, sondern Mengen von Positionen gespeichert werden.

Definition 1. *Auf Spielkarten lässt sich wie folgt eine partielle Ordnung definieren: Haben zwei Karten c und c' die gleiche Farbe, dann gilt $c \prec c'$ falls c von c' gestochen werden kann. Eine Karte c kann von einer Karte c' gestochen werden, falls c' die gleich Farbe hat wie c, oder Trumpf ist und nach den Regeln des aktuellen Skatspiels die höhere Karte ist.*

Das ist im Prinzip die natürliche Ordnung, die durch die Regeln des Skatspiels induziert wird.

Definition 2. *Sei C eine Menge von Karten und die disjunkte Vereinigung von C_1 und C_2. Eine Äquivalenzklasse $E_{C_1}^{C_2}(c)$ auf C, ist die größte Teilmenge von C_1 mit folgender Eigenschaften:*

- $c \in E_{C_1}^{C_2}(c)$
- $\forall c' \in E_{C_1}^{C_2}(c)$ *gilt: c' hat dieselbe Farbe wie c und für jede Karte $d \in C_2$ mit derselben Farbe wie c und c' gilt: $c \prec d \iff c' \prec d$*

Zwei Karten c und c' sind äquivalent bezüglich der Mengen C_1 und C_2, in Zeichen $c \overset{C_2}{\underset{C_1}{\sim}} c'$, falls $c' \in E_{C_1}^{C_2}(c)$. Da diese Äquivalenzklassen repräsentantenunabhängig sind, ist das gleichbedeutend mit $c \in E_{C_1}^{C_2}(c')$.

Diese Definition soll an folgendem Beispiel illustriert werden.

Beispiel 1 Sei $C_1 = \{\diamondsuit 8, \diamondsuit Q, \diamondsuit A\}$ und $C_2 = \{\diamondsuit 7, \diamondsuit K, \diamondsuit 10\}$. Die Äquivalenzklasse $E_{C_1}^{C_2}(\diamondsuit Q)$ ergibt sich zu $\{\diamondsuit Q, \diamondsuit 8\}$. Beide Karten, $\diamondsuit 8$ und $\diamondsuit Q$, können die gleichen Karten aus C_2 stechen, nämlich $\diamondsuit 7$. Von den restlichen Karten aus C_2 werden beide Karten gestochen. Das ist auch die größte Menge dieser Art, denn $\diamondsuit A$ kann von keiner Karte aus C_2 gestochen werden.

Hat man den spieltheoretischen Wert eines Repräsentanten einer Äquivalenzklasse E berechnet, dann kann mit Hilfe des nächsten Satzes, eine obere und untere Schranke für die Werte der anderen Karten aus E angegeben werden.

3.6 Satz über Äquivalenzklassen

Skat kann als 2-Personenspiel aufgefasst werden. Hierbei ist Spieler 1 der Alleinspieler und Spieler 2 wird von den beiden Gegnern gebildet. Im Folgenden sei $G = (X, Y, A)$ ein Skatspiel in strategischer Form im Sinne der Spieltheorie [7] mit beliebiger, aber fester Kartenverteilung. Das Spiel sei ein Farb- oder Grandspiel. Hierbei sei X die Menge der Strategien für Spieler 1 und Y die Menge der Strategien für Spieler 2. Eine Strategie $x \in X$ ist eine Funktion $x : P \to C$. Hierbei ist P die Menge aller Spielpositionen, und C ist die Menge der Spielkarten. Analog ist $y \in Y$ definiert. Sei weiter $A : X \times Y \to \mathbb{R}$ die Bewertungsfunktion. A gibt dabei den Wert des Spiels aus Sicht von Spieler 1 an, d. h. $A(x, y) = r$ bedeutet, dass wenn Spieler 1 mit x und Spieler 2 mit y spielt, Spieler 1 r Punkte erzielt.

Satz 2 (Satz über Äquivalenzklassen) *Sei p eine beliebige Position in einem Skatspiel. C_1 bezeichne die Menge der Karten, die Spieler 1 in p auf der Hand hält und C_2 die Menge der restlichen Karten, die in p noch im Spiel sind, einschließlich der Karten im aktuellen Stich. Spieler 1 habe die Karten c und c' mit $c \overset{C_2}{\underset{C_1}{\sim}} c'$ auf der Hand. Sei weiter $v(p, c) = \max\limits_{\substack{x \in X \\ x(p)=c}} \min\limits_{y \in Y} A(x, y)$ Dann gilt:*

$$|v(p, c) - v(p, c')| \leq d(c, c')$$

Hierbei bezeichnet $d(c, c') = |val(c) - val(c')|$ *und* $val(c)$ *den Wert der Karte* c *z. B.* $val(\clubsuit K) = 4$.

Hierbei bezeichnet $v(p, c)$ den besten Wert, den Spieler 1 erreichen kann, wenn Spieler 1 in p die Karte c spielt.

Mithilfe dieses Satzes lässt sich der bisher betrachtete Algorithmus weiter beschleunigen. Statt für jede Karte den spieltheoretischen Wert auszurechnen, kann man somit auf schon errechnete Werte zurückgreifen und Abschätzungen vornehmen.

In der Implementierung des Double Dummy Solvers wird dieser Satz nur auf Karten c und c' angewendet mit $d(c, c') = 0$. D. h. es kommen nur die vier Buben, alle 7er, 8er und 9er in Betracht. Erste Versuche mit allgemeineren Äquivalenzen zeigten nicht den gewünschten Effekt.

3.7 Ergebnisse

Wendet man diesen Satz auf den *mws*-Algorithmus mit Transpositionstabelle an, so reduziert sich die Zahl der zu betrachtenden Knoten um über die Hälfte. Allerdings sinkt die Ausführungszeit nicht in gleichem Maß. Das liegt daran, dass in diesem Fall zusätzlich noch die Äquivalenzklassen berechnet werden müssen.

I. A. ist eine Erweiterung immer zweischneidig: zwar reduziert sie die Zahl der zu betrachtenden Knoten, sie benötigt aber zusätzliche Rechenzeit. Der Einsatz einer Erweiterung ist demnach nur dann gerechtfertigt, wenn dadurch die Ausführungszeit insgesamt sinkt.

3.8 Sichere Stiche

Je näher an der Wurzel das Niveau ist, auf dem im Spielbaum Positionen erreicht werden, in denen eine der beiden Parteien gewonnen hat, desto schneller ist der Algorithmus. Es liegt daher nahe, einen Test einzusetzen, mit dem die noch zu erreichenden Punktzahl einer Partei nach unten abgeschätzt werden kann.

Zu diesem Zweck werden zu Beginn jedes neuen Stichs für die Partei, die den Stich eröffnet, die Punkte abgeschätzt, die diese bei forcierender Spielweise im Restspiel noch erreichen kann. Es werden alle Stiche betrachtet, die von dieser Partei ohne Unterbrechung gewonnen werden. Die erzielte Punktzahl aus diesen Stichen wird als Ergebnis zurückgeliefert. Das Problem bei diesem Test ist, dass sichergestellt werden muss, dass die Gegenpartei optimal spielt, da sonst das zurückgelieferte Ergebnis die erreichbaren Punkte überschätzen könnte.

Wird nicht Trumpf angespielt und kann die Gegenpartei einen Stich nicht für sich entscheiden, dann ist klar, dass jeder Spieler dieser Partei — kann er Farbe bekennen — die niedrigste Karte dieser Farbe ausspielt. Ist ein Spieler auf dieser Farbe blank, dann wird aus Gründen der Performance nicht mit $\alpha\beta$ berechnet welche Karte zu spielen ist, sondern es wird für jeden Spieler gezählt, wie oft er nicht bedienen konnte. Am Ende des Tests gibt ein Spieler für jedes Nichtbedienen die wertloseste Karte ab.

In Farbspielen ist für die Gegenpartei beim Ausspielen von Trumpf nicht klar, ob zuerst ein Bube oder eine andere Karte mit höherem Punktwert gespielt werden soll. Dieses Problem kann umgangen werden, indem die Punktwerte der Trumpfkarten, die ein Spieler der Gegenpartei auf der Hand hält so vertauscht werden, dass die i-t höchste Trumpfkarte den i-t höchsten Punktwert erhält. Es lässt sich zeigen, dass die forcierende Partei mit dieser Wertvertauschung, höchstens so viele Punkte erreichen kann wie ohne Vertauschung.

3.9 Die MIN-Spieler

Bilden die MIN-Spieler die ausspielende Partei und somit der Alleinspieler die Gegenpartei, so soll so gespielt werden, dass der Alleinspieler nie einen Stich bekommt. Es reicht also, dass einer der MIN-Spieler den Stich gewinnt. Dazu reicht es, dass ein Spieler forcierend spielt, während der andere von seinen spielbaren Karten diejenige mit der höchsten Punktzahl spielt.

3.10 Ergebnisse

Auf den vorstehenden Überlegungen basierend wurde ein Algorithmus zur Abschätzung der sicheren Punkte der jeweils ausspielenden Partei implementiert. Ein Test von 1.000 zufälligen Farb- und Grandspielen hat ergeben, dass die durchschnittliche Differenz von sicheren Punkten für eine Anfangsposition und den tatsächlichen Wert 8,67 beträgt. Hierbei wurde die Vorhandposition zufällig an einen der drei Spieler vergeben.

4 Nullspiele

Beim Skatspiel hat das Nullspiel eine Sonderstellung; es ist das einzige Spiel, bei dem es keinen Trumpf gibt. Das Ziel des Alleinspielers ist es, überhaupt keinen Stich zu gewinnen. Die Punkte, die in einem Stich erzielt werden, sind also uninteressant. Der Alleinspieler muss versuchen jeden Stich zu verlieren. Da die Punkte nicht wichtig sind, darf man ein Spiel im Gegensatz zu den anderen Varianten auch dann nicht abbrechen, wenn eine Partei mehr als 60 Punkte erreicht hat.

Doch auch bei Nullspielen kann die Suche vor dem Erreichen von Blättern im Spielbaum abgebrochen werden. Die erste Situation, in der man ein Spiel abbrechen kann, folgt direkt aus den Spielregeln, nämlich dann, wenn der Alleinspieler einen Stich gewonnen hat.

Die zweite Situation, in der das Spiel vorzeitig beendet ist, tritt ein, wenn der Alleinspieler ein *sicheres Blatt* auf der Hand hat. Ein sicheres Blatt ist ein Blatt, mit dem man bei optimalem Spiel und beliebiger Kartenverteilung bei den Gegnern immer gewinnt. Mit dieser kleinen Modifikation läßt sich der *mws*-Algorithmus 2 auch für Nullspiele verwenden.

Es stellte sich heraus, dass Nullspiele im Mittel mit sehr wenigen Erweiterungen sehr schnell gelöst werden können. Ein Test mit 1.000 zufälligen Nullspielen,

bei denen auch das Ausspielen der ersten Karte randomisiert wurde, ergab, dass die mittlere Knotenzahl pro Spiel 21.000 ist und die durchschnittliche Laufzeit 0,019 s beträgt. Aufgrund der Einfachheit gehe ich auf das Nullspiel nicht weiter ein. Für eine ausführlichere Abhandlung über Nullspiele verweise ich auf meine Diplomarbeit [5].

5 Reizen & Drücken

Ein Skatspiel beginnt mit dem Reizen. Hierbei muss jeder Spieler entscheiden, ob er mit seinen zehn Karten und den zwei unbekannten Karten, die im Skat liegen, gegen die beiden anderen Spieler gewinnen kann. Auch dieses Problem ließe sich mit einem DDS lösen. Folgender Algorithmus kann das bewerkstelligen:

- Konstruieren eine Menge D von möglichen Kartenbelegungen für die beiden anderen Spieler und den Skat.
- Berechne für jede dieser Kartenbelegungen und für jedes Spiel (♢, ♡, ♠, ♣, Grand und Null), ob mit diesen Karten gewonnen werden kann.
- Reize auf das Spiel, das die meisten dieser Spiele gewinnt.

Für diesen Ansatz müssen sehr viele Spiele berechnet werden. Angenommen D habe nur fünf Elemente. Für jede dieser Kartenbelegungen müssen dann alle sechs möglichen Spiele berechnet werden. Pro Spiel gibt es $\binom{12}{10} = 66$ Möglichkeiten um zwei Karten zu drücken. Insgesamt müssten in diesem Fall 1.980 Spiele berechnet werden. Für diesen Ansatz benötigt man einen extrem schnellen DDS.

Leider ist der von mir entwickelte DDS für diese Aufgabe zu langsam. Aus diesem Grund habe ich mit einem *Least-Mean-Square*-Algorithmus (siehe etwa [6]) ein Klassifikator gelernt, der bei Eingabe von zehn Karten und einem Spiel (♢, ..., Grand, Null) sagen kann, ob der Alleinspieler mit diesen zehn Karten das übergebene Spiel gewinnen kann.

Mit diesem Klassifikator lässt sich auch das Drücken realisieren:

- Konstruiere aus den zwölf Karten des Alleinspielers alle 66 mögliche Kombinationen für den Skat.
- Wende jeweils auf die restlichen zehn Karten den Klassifikator an. Drücke die Karten, für der Klassifikator das höchste positive Ergebnis zurückgeliefert hat und spiele das entsprechende Spiel.

Die Qualität dieses Klassifikators ist leider nicht besonders hoch. Zum Teil werden offensichtliche Fehler gemacht. Es kann vorkommen, dass eine blanke Zehn nicht gedrückt wird und stattdessen die Sieben der gleichen Farbe. In manchen Fällen wird sogar Trumpf gedrückt.

Auch fällt auf, dass der Klassifikator nur sehr wenige Karten als spielbar einstuft. Dieses Problem ist meiner Meinung nach mit den Trainingsdaten verbunden. Zum Lernen wurden nämlich nur die Ergebnisse von Spielen mit offenen Karten verwendet.

Ein besserer Klassifikator könnte auf einer handverlesenen Menge von Expertenregeln aufbauen, allerdings ist aus algorithmischer Sicht das Lernen natürlich viel interessanter.

6 Zusammenfassung

In diesem Artikel wurde gezeigt, wie man mit Hilfe des MC-Ansatzes mit einem $\alpha\beta$-Algorithmus ein Programm schreiben kann, das Skat spielt.

Die wichtigste Komponente bildet in diesem Ansatz der DDS. Durch mehrere Verbesserungen konnte die Zeit, die für das Bestimmen der besten Anfangskarte benötigt wird, stark verkürzt werden. Ein $\alpha\beta$-Algorithmus mit Transpositionstabelle ($\alpha\beta\_tt$) benötigt für diese Aufgabe durchschnittlich 25 Sekunden und evaluierte dabei 85.000.000 Knoten. Folgende Tabelle zeigt die Historie der Leistungssteigerung:

Algorithmus	**Durchschnittswerte**	
	Knoten	**Laufzeiten**
$\alpha\beta\_tt$	85.000.000	25,00 s
mws_tt	2.500.000	2,40 s
mws_equiv	1.100.000	1,40 s
mws_ss	500.000	0,14 s
mws_mo	150.000	0,06 s

Hierbei bezeichnet mws_tt die Nullfenstersuche mit Transpositionstabelle, und mws_equiv nutzt zusätzlich noch den Satz über Äquivalenzklassen aus. Mit mws_ss kommen noch die "sicheren Stiche" zur Anwendung. Der letzte Algorithmus enthält zusätzlich noch eine Zuganordnung, auf die in dieser Ausarbeitung allerdings nicht näher eingegangen wurde.

Damit aus dem bisher entstandenen Programm ein richtiges Skatprogramm werden kann, muss vor allem das Reizen und Drücken verbessert werden. Zusätzlich sollten auch alle Spielvarianten wie z. B. *ouvert* oder *Schneider* gespielt werden können. Im Moment denke ich darüber nach, wie man das Reizen und Drücken verbessern kann. Wenn dieses Problem gelöst ist, soll ein GUI implementiert werden, damit das Programms besser mit menschlichen Skatspielern verglichen werden kann.

Literatur

1. Frank, Ian und David A. Basin: *Search in Games with Incomplete Information: A Case Study Using Bridge Card Play.* Artificial Intelligence, 100(1-2):87–123, 1998.
2. Ginsberg, Matthew L.: *Partition Search.* In: Shrobe, Howard und Ted Senator (Herausgeber): *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2*, Seiten 228–233, Menlo Park, California, 1996. AAAI Press.
3. Ginsberg, Matthew. L.: *GIB: Steps Toward an Expert-Level Bridge-Playing Program.* In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Seiten 584–589, 1999.
4. Internationale Skatordnung. Deutscher Skatverband e.V. (DSkV) und International Skatplayers Association e.V. (ISPA-World), Vom *27. Deutschen Skatkongress,* `www.skat.com/dskv/skatgericht/isko.pdf`, (23. Juli 2003). 1998.

5. Kupferschmid, Sebastian: *Entwicklung eines Double Dummy Skat Solvers — mit einer Anwendung für verdeckte Skatspiele.* Diplomarbeit, 2003.
6. Mitchell, Tom M.: *Machine Learning.* McGraw-Hill International Editions, 1997.
7. Osborne, Martin J. und Ariel Rubinstein: *A Course in Game Theory.* MIT Press, 1994.
8. Pearl, Judea: *Heuristics — Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Co., Reading, MA, 1984.
9. Plaat, Aske, Jonathan Schaeffer, Wim Pijls und Arie de Bruin: *Best-First Fixed-Depth Game-Tree Search in Practice.* In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Band 1, Seiten 273–279, Montreal, Canada, 1995.

Dependency Analysis and its Use for Evolution Tasks

Lothar Hotz[1], Thorsten Krebs[2], and Katharina Wolter[3]

[1] HITeC c/o Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
hotz@informatik.uni-hamburg.de
[2] LKI, Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
krebs@informatik.uni-hamburg.de
[3] LKI, Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
kwolter@informatik.uni-hamburg.de

Abstract. During the life-cycle of products, evolution of the knowledge used for configuring these products is demanded. Change operations are introduced to capture modifications to the configuration model. Preconditions and impacts of these operations are explained and motivate a dependency analysis for supporting change operations on the configuration models. We give a detailed discussion of the dependency analysis for structure-based configuration models.

1 Introduction

Structure-based configuration is used to configure software-intensive systems in the European project *ConIPF (Configuration of Industrial Product Families)*. Since configuration models are used to configure families of products over time, it is apparent that new functionality is introduced for being able to create new products. This has an impact on the configuration model: the knowledge has to evolve in parallel with the components used for product derivation.

Change operations are introduced to capture modifications to the configuration model. But since modifications may lead to inconsistent states of the model, the need for complex change operations (i.e. a concatenation of basic change operations that are dependent on another) arises. Such complex change operations are built to avoid inconsistent states within the configuration model. The impacts of certain modifications have to be known for constructing a sequence of basic operations that fulfills the task of the complex operation and therefore avoids inconsistencies.

Impacts of modifications can be anticipated by knowing dependencies between the diverse knowledge entities of the modeling facilities of structure-based configuration. These dependencies can be computed by analyzing the model. Therefore, a dependency graph can be generated that helps in predicting impacts of modifications. Evolution tasks can be supported with such dependency graphs.

In this paper, we focus on the dependency analysis for the structure-based configuration tool KONWERK [3]. The modeling facilities of this configuration approach and their dependencies are introduced, analyzed and discussed in detail. As an example for applying the dependency analysis we show how evolution of the configuration model can be enhanced by using dependency graphs. Further potential application areas are mentioned in the summary.

The remainder of this paper is organized as follows: in Section 2 we describe the basic modeling facilities and the configuration procedure of structure-based configuration. In Section 3 evolution of the configured product family and the configuration model is introduced. Change operations are identified as a mechanism to capture modifications to the model. Section 4 presents the general idea behind dependency analysis, lists and discusses concrete dependencies we identified for structure-based configuration and describes how these dependencies can be computed automatically. In Section 5 we give the evolution of configuration models as an example for applying dependency analysis. Finally, we conclude with the summary in Section 6.

2 Structure-based Configuration

2.1 Modeling Facilities

In structure-based configuration, product components and the mapping between these are formalized in a configuration model. This model is represented with the *Configuration Knowledge Modeling Language (CKML)*[4] and is used for automated product configuration. Components of diverse types can be represented with concepts provided by the modeling facilities of structure-based configuration. A brief definition is given in the following (see [2] for a more detailed view):

- Each concept has a name which represents a uniquely identifiable character string.
- Attributes of concepts can be represented through parameters. A parameter is a tuple consisting of a name and a value descriptor. Diverse types of domains are predefined for value descriptors - e.g. integers, floats, sets and ranges.
- Concepts are related to exactly one other concept in the *taxonomic relation*. Thus, the latter concept is the superconcept of the former one – the subconcept. Properties (i.e. parameters and relations) of the superconcept are inherited by the subconcept. The superconcept subsumes the subconcept, i.e. all properties of the subconcept are subsets of those of the superconcept.
- *Partonomies* are generated by modeling compositional (has-parts) relations. Such a relation definition contains a list of parts (i.e. other concept definitions) that are identified by their names. Each of these parts is assigned with a minimum and a maximum cardinality that together specify how many instances of these concepts can be instantiated as parts of the aggregate.
  Has-parts relations are a class of compositional relations. Thus, own relations with new names can be modeled – e.g. a concept can contain a has-hardware

[4] CKML is an extension of the language BHIPS used for KONWERK

definition next to a has-software definition to emphasize on the difference between the parts in both relations.

Concepts describe classes of objects from which multiple concept instances can be generated during the configuration process. The concept is a static description that is not altered at any time while a concept instance of such a concept definition is dynamically created during the configuration process and configured until its properties are fully specified in terms of the task specification. A concept instance always is instance of exactly one concept definition.

Constraints are used to describe restrictions between properties of distinct concepts. There are three layers for describing constraints: the *conceptual constraint* consists of a precondition that has to hold in order to be executed. *constraint relations* define the restriction itself. A constraint relation is reusable for an arbitrary number of conceptual constraints. A conceptual constraint can call an arbitrary number of constraint relations. Constraint relations can be of different types (e.g. functions, extensional, etc.). Instances of constraint relations *(constraints)* are automatically generated during configuration and represent restrictions between concept instances. See Figure 3 for an example.

By analyzing the configuration model necessary decisions that have to be made are inferred during the configuration process. Each decision is represented by a configuration step. There are 4 types of configuration steps:

1. *parameterization* represents a decision of setting a parameter.
2. *specialization* represents a decision of specializing a concept instance to a more specific concept according to the taxonomy.
3. *decomposition* represents a decision of decomposing a concept into its parts (i.e. top-down reasoning). During decomposition appropriate instances of the parts are generated (instantiation is seen as a sub step of decomposition).
4. *integration* represents a decision of integrating a part into an aggregate (i.e. bottom up reasoning). During integration appropriate instances of the aggregates are generated (instantiation is also seen as a sub step of integration.)

Consistency of the model is defined as follows (compare [7]):

Specialization-related: given a super- and a subconcept, all values of the subconcept's properties have to be subsets of the corresponding property (identified by name) of the superconcept.

Composition-related: given an aggregate and its parts, each part has to be defined as a concept.

Constraint-related: given a constraint and the participating concept properties, the constraint may only use value ranges defined in the model. Furthermore, only subsets of values of the concept properties are allowed as propagation results.

2.2 Example Domain

As an example we give the application domain of Car Periphery Supervision (CPS) systems introduced by [8]. A CPS system consists of automotive systems that are based on sensors installed around the car to monitor its local

environment. Sensor measurement methods and evaluation mechanisms provide information for various kinds of high-level comfort and safety related applications like Parking Assistance and Pre-Crash Detection. In CPS systems sensors are mounted on the vehicle (e.g. ultrasonic sensors hidden in the bumper).

In Figure 1, a small configuration model taken from the CPS domain is presented. Besides the specialization and decomposition one constraint is given (see Figure 3), which ensure that the parameters `Type` of `Main` and `Sensor-Configuration` are equal. This (probably simplifying) example is used in the following sections for illustrating the notions behind dependency computation.

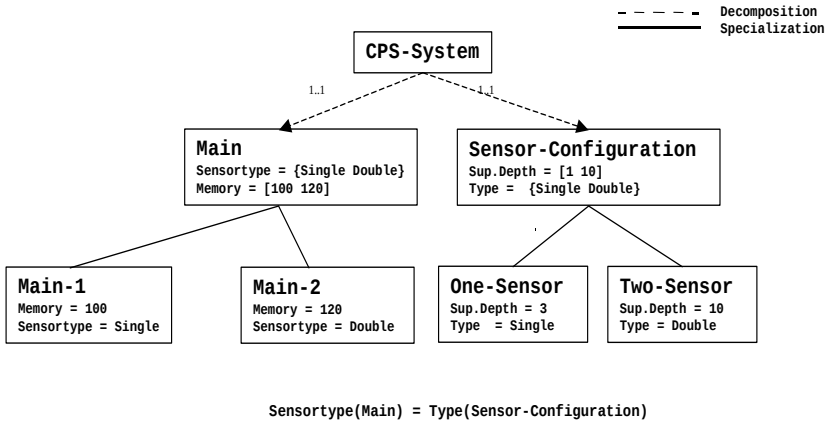

Fig. 1. Example of a configuration model

Given a configuration model the configuration steps to be resolved during the configuration process are automatically computed from it. In Figure 2, all configuration steps are listed (incl. definition of later used short-cuts) which are identified when the configuration model shown in Figure 1 is given. All configuration steps have to be determined (i.e. configured) in one configuration process. A terminal value (i.e. a value not further specifiable) for each configuration step has to be computed by a value determination method (like automatic inferencing, asking the user, invoking a function etc.). Computing of the dependencies of configuration steps is presented in the following.

3 Operations for Evolving Configuration Models

Evolution of products and product components is inevitable throughout their life cycle – driven by advancing technology, increasing customer requirements or bug fixes. New versions and variants are built and therefore also modeled. It is common that evolution is anticipated to a certain extent, e.g. by modeling planned features [4]. But it is impossible to predict all future developments (like bug fixes) and therefore the model has to be repaired at some time.

```
Instantiate CPS-System (Instant(CPS-System))
Decompose CPS-System (Decompose(CPS-System))
Instantiate Main (Instant(M))
Instantiate Sensor Configuration (Instant(SC))
Specialize Main (Specialize(M))
Specialize Sensor Configuration (Specialize(SC))
Determine the parameter Sensortype of the Main (Param(Type,M))
Determine the parameter Memory of the Main (Param(Memory,M))
Determine the type of the Sensor Configuration (Param(Type,SC))
Determine the parameter Supervision Depth of the Sensor Configuration
(Param(Sup.Depth,SC))
```

Fig. 2. Configuration Steps of the Example with Short-cuts

Concept
name: M-SC-Equal
precondition:
?c **name:** CPS-System
?s **name:** Sensor-Configuration
relation: part-of ?c
?m **name:** Main
relation: part-of ?c
constraint-call:
equal (?m sensortype) (?s type)

Fig. 3. A Conceptual Constraint

Evolving the configuration model can have diverse effects on its consistency and on existing or potentially derivable products. Different scenarios like richer or smaller variety of products (leading to different configuration solutions) are conceivable. In the following, we confine ourselves to impacts that modifications to the configuration model can have and how consistency of the model can be guaranteed (see Section 2).

Modifications to the configuration model are captured through *change operations*. *Basic operations* are those that cannot be further partitioned – i.e. simple modifications like adding a parameter value. *Complex operations* are composed of multiple basic operations or include some additional knowledge about the modification. They provide a mechanism for grouping basic operations into a logical unity (compare [6]). This is needed because an operation may lead to an inconsistent state of the configuration model and thus should be followed by further operations that correct this situation. Hierarchies of change operations

can be formulated to exploit the inheritance mechanism for specifying common properties.

Standard operations e.g. are adding, deleting and modifying the diverse knowledge entities. Adding a parameter value and adding a compositional relation have similar characteristics and impacts while they are fundamentally different from deleting these knowledge entities. Thus, it is obvious that a hierarchy of change operations is reasonable. As an example, in the following we show our hierarchy of operations for the complex change operation *modifying concept parameters* (operations are bold, impacts are in italics):

- **Renaming a parameter**
  - *in case of existing subconcepts: also change the name in those*
  - *in case of constraints binding this parameter: change the name in the precondition of the conceptual constraint*
- **Changing a parameter value**
  - *in case of existing subconcepts: check that the corresponding parameter values of subconcepts are subsets of the new value*
  - *in case of constraints binding this parameter: check that the constraint relation still computes reasonable values*
  - **Enlarging value range**
    - *no impacts on inheritance have to be addressed – subconcepts will keep subsets of values after this operation*
  - **Scaling down value range**
    - *impacts on inheritance have to be addressed – the new value might be a subset of the former subconcept's values*
  - **Changing value type**
    - *check that values are still reasonable*

To find out which combination of basic change operations is needed to perform a more complex operation, a dependency analysis is needed. Impacts of modifications are examined and indicate the need for further modifications to reach a consistent state of the knowledge base.

4 Dependency Analysis

Dependency detection is performed on the configuration model as a whole. It can be used for computing the impacts a change of the configuration model would have.

When the configuration model is specified, all possible dependencies are already present, i.e. it is not necessary to *model* dependencies, but only to compute them from the configuration model. Besides the configuration model the dependencies are of course affected by the semantics of CKML, which is implemented in the inference machine of KONWERK. We identified rules that are based on the configuration model on the one hand, and on the inference machine that interprets the configuration model on the other hand. An example rules is:

A specialization determines parameterization of all parameters which are new or changed in c in respect to its superconcept in the taxonomical hierarchy.

There are rules between configuration steps, conceptual constraints and constraint relations, and those which are established in the constraint net by instances of constraint relations. However, the rules are not related to a specific domain (like CPS). Thus, when an arbitrary configuration model expressed in CKML is given with these rules the dependencies can be computed automatically. All identified rules are given in Section 4.2. But first we present the general concept of dependency identification.

4.1 General Idea

Two general types of dependencies can be identified: those things that are *prerequisites* for other things, and those things that *determine* other things.

Looking at the example in Figure 1 the concept Sensor-Configuration can only be instantiated after the aggregate named CPS-System is decomposed. Thus, the decomposition of the aggregate is a *prerequisite* of the instantiation. However, the value of the instantiation is not determined by such a dependency. On the other side the parameterization of the parameter Type of an instance of concept Main determines the specialization of this instance, e.g. when Type = Double is given, a Main instance would be specialized to Main-2. Thus, when deciding a parameter value of an instance the specialization of that instance may be determined. This type of dependency is called *determination*. Hereby a value in the partial configuration is changed.

Also for constraints, prerequisites and determination dependencies can be identified. For conceptual constraints, only if the patterns listed in the condition part of a conceptual constraint (see Figure 3) are fulfilled, the constraint calls specified in the constraint-calls part of the conceptual constraint are instantiated. Thus, the patterns are prerequisites for the constraint-relation instance. The constraint itself (in our example the constraint equal) *determines* the values of the involved parameters.

In Figure 4 the dependencies of the example are presented in a *dependency graph*. An arrow means the node at the arrowhead *depends on* the node at the end of this arrow. Dependencies of type prerequisite are depicted as *v-edges*. Dependencies of type determination are depicted as *d-edges*.

Given such a dependency graph, when a product is configured, not all dependencies are used for each task. Thus, the dependencies for one configuration depend on the given task specification. However, one configuration process is a walk through the general (i.e. not task-specific) dependency graph.

In Figure 5 the decomposition of the CPS-System concept and the specialization of the Sensor-Configuration is given by the user. The inference machine computes the other configuration steps. This is done by the indicated walk through the dependency graph. When the values for memory of Main and for Type of Sensor-Configuration are given by the task specification, the equal constraint concerning the parameters Type of Main and Sensor-Configuration might indicate a

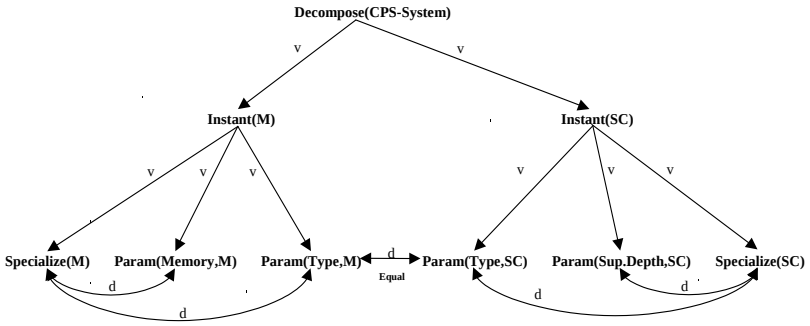

Fig. 4. Dependencies of the Example with Prerequisites and Determinations

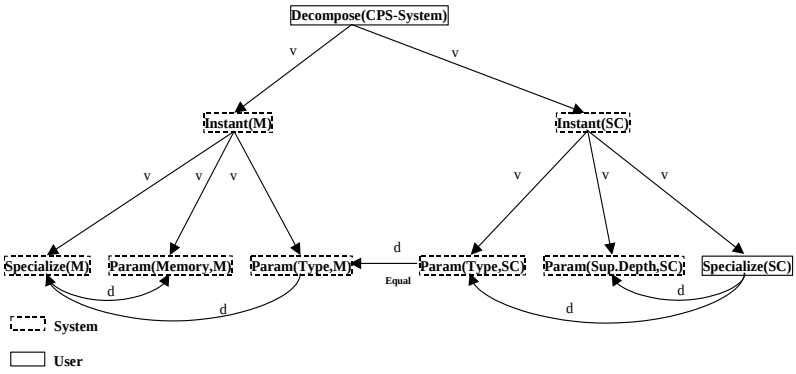

Fig. 5. A Configuration

conflict, i.e. the case when the memory is 100 and the supervision depth is 10 (see Figure 6). In general when a node has two incoming *d-edge* dependencies (like Specialize(M), Param(Type,M), Param(Type,SC), Specialize(SC)) a conflict at this node can probably occur.[5]

The general idea is, when a configuration model is given, all possible dependencies can be computed in advance and independently of a given task. Thus, a dependency graph can be directly computed from a configuration model, and can be used for dependency analysis. For operations that are related to a specific task (like "give me the parts of the model, that are considered for this specific product") the necessary parts of the dependency graph can be activated. The prerequisites (*v-edges*) are considered as conditions for the activation of a sub graph. Thus, for a given task specification those conditions can or cannot be fulfilled, which leads to an activation or non-activation of the following sub graph. Furthermore, patterns of conceptual constraints are used as prerequisites in the dependency graph. This is presented in detail in the next section.

[5] Not considering user interactions which can cause conflicts at any node.

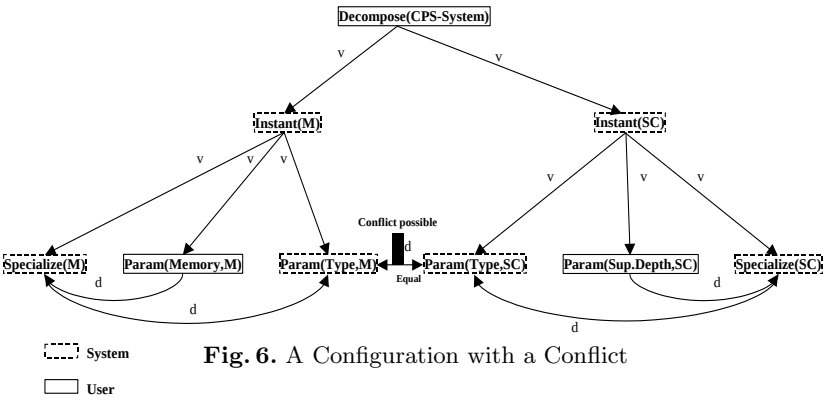

Fig. 6. A Configuration with a Conflict

4.2 Dependencies in Structure-based Configuration

Dependencies between configuration steps and constraints are listed and explained in detail in the following.

Between configuration steps: In this section for each type of configuration step the dependencies are listed. A configuration step type, the dependency type, and the dependent configuration step type is given.

1. Decomposing an aggregate is a prerequisite for existence of the parts as an instance.
2. Decomposing an aggregate determines the specialization of the aggregate.
3. Existence of an instance is a prerequisite for the parameterization of the parameters.
4. Existence of an instance is a prerequisite for the specialization of the instance.
5. Existence of an instance is a prerequisite for the decomposition of the instance.
6. Existence of an instance is a prerequisite for the integration of the instance.
7. Specialization of an instance to a concept type c is a prerequisite for further specializing the instance.
8. Specialization of an instance to a concept type c determines parameterization of all parameters which are new or changed in c.
9. Specialization of an instance to a concept type c determines decomposition of all relations which are new or changed in c.
10. Parameterizing a parameter p of an instance determines the specialization of the instance.
11. Integration of a part in an aggregate determines the decomposition of the aggregate.
12. Integration of a part in an aggregate determines the specialization of the aggregate.

Between constraints: At a first glance, dependencies between configuration steps and conceptual constraints could be considered by depending instantiation

steps with the patterns that match the related instance (Figure 7 upper part). But because only when all patterns of a conceptual constraint are fulfilled the related constraint relation is instantiated, one can summarize all patterns of a conceptual constraint to one node (as it is done in Figure 7 lower part). This node is a prerequisite for the constraint relation. Thus, dependencies between configuration steps and conceptual constraints can be computed by looking at all conceptual constraints. The rules between configuration steps and conceptual constraints are:

1. Instantiating a concept c is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c.
2. Parameterizing a parameter p of an instance of type c is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c and contains p.
3. Decomposing an aggregate via relation r is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c and contains r.

Between conceptual constraints and constraint relations the rule is:

1. All patterns of a conceptual constraints are prerequisites for all constraint relations specified in the constraint call!

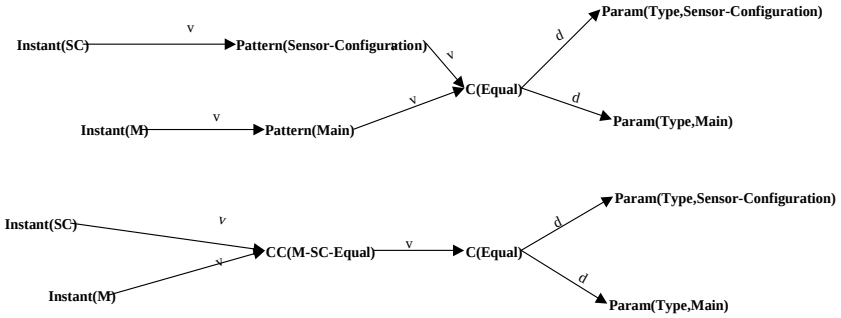

Fig. 7. Representation alternatives for dependencies between instantiation configuration steps and conceptual constraints, only the lower graph is actually necessary

Constraints always restrict parameters, or relations of diverse concepts, which is done by introducing constraint variables in the constraint net. Those variables can be identified with the appropriate configuration step, thus, leading from a situation given in Figure 8 lower part to the situation given in Figure 8 upper part.

Between configuration steps and constraint relations:

1. Parameterization of a parameter p constraint call contains variable referring to p.
2. Decomposition of a relation r constraint call contains variable referring to r.

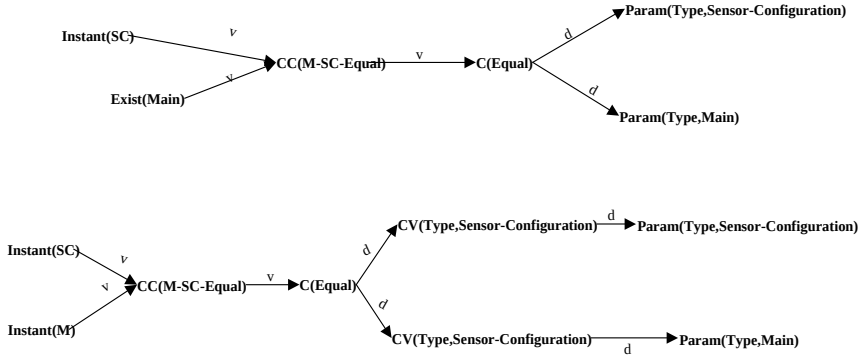

Fig. 8. Example: Identifying Constraints and Configuration Steps

3. Specialization of an instance constraint call contains variable referring to $instance - of$.

Constraints themselves can be uni-directed or multi-directed. Therefore, the determination dependencies between the constraint variables are defined accordingly:

1. Uni-directed Constraint of the form: $e_1 \dots e_n \rightarrow e_m \dots e_z$:
   $e_1 \dots e_n$ determines $e_m \dots e_z$.
2. Multi-directed Constraint of the form: $e_1 \dots e_n \leftrightarrow e_m \dots e_z$:
   e_1 determines $e_2 \dots e_z$
   e_2 determines $e_1, e_3, \dots e_z$
   etc.

In Figure 9 the complete dependency graph for the example introduced above is given.

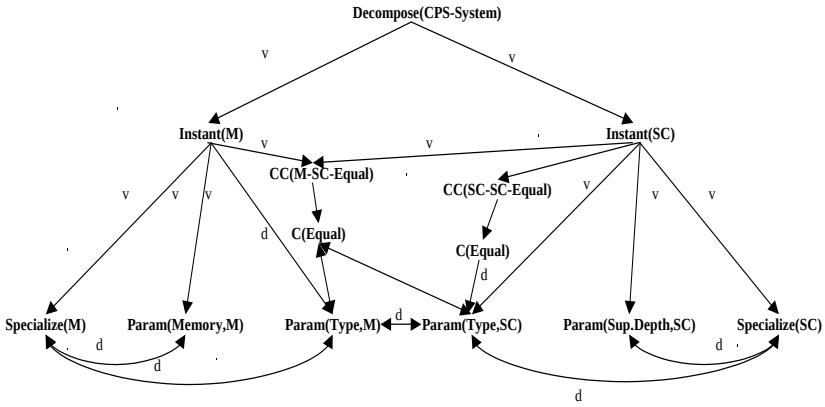

Fig. 9. Dependency graph for the example

Value-Related Dependencies: Up to now, only dependencies without considering values are taken into account. But also value-related dependencies like optional, alternative, multiple parts, dependencies related to a specific parameter value, number restricting constraints, and alternatives for integration steps (or-operator in part-of) have to be taken into account. For each part which is mentioned in any decomposition a graph is generated containing all dependencies related to that part - i.e. a sub graph for that part is created. In the dependency graph the *v-edge* to this graph is indicated with the number restriction of the decomposition, e.g. $[0..1], [1..1], [0..n]$.

Multiple instances of one concept can occur when conceptual constraints with patterns filtering the same concept. In such a case:

- Also only one partial graph for each concept is generated, e.g. only one for Sensor-Configuration!
- Constraints between different parameters of those concepts are indicated with further *d-edges*
- Constraints between the same parameters (e.g. Type) are indicated with numbers on *d-nodes*

Dependencies for constraints that contain number restrictions can be identified with decomposition steps. This can be done because with those constraints the number of parts of an aggregate is determined and the control module instantiates the parts according to those numbers. Thus, the behavior in this case is like a decomposition step.

4.3 Computing Dependencies

Given a configuration model in a declarative syntax (e.g. in CKML or in a tool-specific language like EngCon's XML notation [1] or BHIBS of KONWERK [3]), the dependencies are already present in the configuration model and can be transformed to an appropriate representation, e.g. a graph data type. This can be done by parsing the configuration model by taking the declarative syntax into account and using the previously explained rules.

Transforming the declarative model into a dependency graph In the previous section for each situation (i.e. configuration step, conceptual constraint and constraint relation) the dependencies are identified. By using those rules and one of the following methods, a complete dependency graph can be computed.

A further opportunity is to compute the dependencies by configuration, thus, the dependency graph is "configured" e.g. by using KONWERK. This can be done by once defining a model which describes the dependency rules in terms of concepts, i.e. defining a dependency meta model. While configuring with such a dependency model the concepts of the configuration model (e.g. of the configuration model of Figure 1) are inspected with inspection methods. Inspection methods typically compute information on objects themselves, here on concepts themselves, like all parameters of concept A, all patterns of conceptual constraint

C etc. Configuration with such a meta model would lead to a dependency graph which represents the dependencies of the configuration model. The dependency graph can again be used as a configuration model which is activated during configuration with the configuration model activating a task-specific dependency graph.

Because procedural knowledge defines known dependencies between configuration steps, this knowledge can be incorporated when the dependency graph is computed. Thus, when computing the dependency graph procedural knowledge can be used for more restricting the resulting graph. This topic has not yet been taken into account.

5 Using Dependencies for Evolution

In the previous sections we have discussed how the dependency analysis is realized. In this section we show how the known dependencies can be used to build complex change operations. The CPS example from Figure 1 in Section 2 will be extended to show how modifications to the configuration model can lead to inconsistencies. Therefore, complex change operations are build to prevent such situations.

We assume that exactly one Sensor-Configuration of type One-Sensor exists – this one is an Ultrasonic Sensor (see Figure 10).

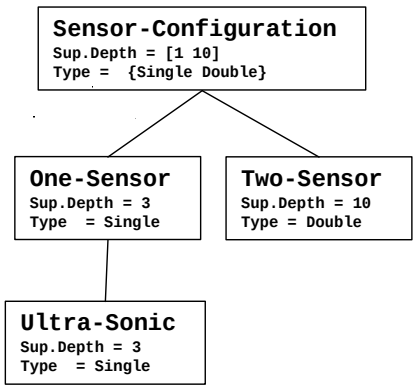

Fig. 10. The CPS Example with Ultrasonic Sensor

In our example, a new type of sensor is created – a Short-Range Radar. There also exists a Sensor-Configuration with exactly one of these sensors. Therefore we introduce SR Radar as a new Specialization of One-Sensor. The short range radar has a supervision depth of 5 meters (which is not a subset of the corresponding parameter value in One-Sensor). Thus, the value for the parameter Sup.Depth of the One-Sensor has to be corrected accordingly. Figure 11 shows the correct new version of the configuration model.

To avoid the inconsistency mentioned above, complex change operations are built from the basic change operations *add concept* and *enlarge parameter value*. This results

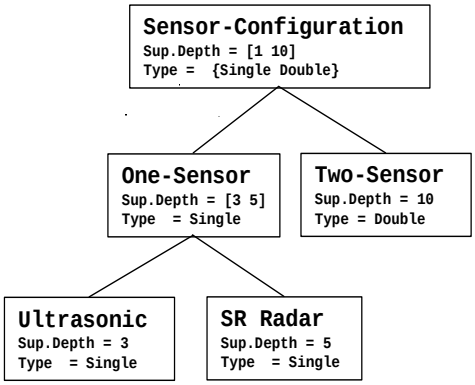

Fig. 11. The CPS Example with Ultrasonic and SR Radar Sensors

in the following complex change operations:

add concept(Ultrasonic) → *add concept(Ultrasonic), apply superconcept(Ultrasonic, One-Sensor)*

add concept(SR Radar) → *add concept(SR Radar), apply superconcept(SR Radar, One-Sensor), enlarge parameter value (One-Sensor Sup.Depth [3 5])*

This is of course a very simple example. But it should make clear how the dependency analysis can be used to guarantee consistency for evolving configuration models. The properties (Type and Sup.Depth) e.g. are defined in the basic concept Sensor-Configuration and therefore can be automatically processed. The three types of consistency declared in Section 2 have to hold and can be processed separately. Furthermore, the dependency analysis can be used to inspect other relations and restrictions (that have been added to more specific concept definitions). This means, when changes are given, the computed dependent parts of the model have to be considered for corrections and / or remodeling.

6 Summary

We have shown how dependencies can be computed from the declarative model in structure-based configuration. Rules according to the interpretation of the configuration model with the inference machine have been defined. They can be computed with simple graph algorithms and the computed dependencies can be used for computing impacts of changes of the configuration model. Short implementation experiences show that the dependency analysis can be easily implemented. However, a full implementation and integration in the configuration process is not yet available and will be done in future work.

Given a dependency graph for a configuration model, this graph can be used for diverse operations and presentations. If, for example, this information is made accessible during the configuration process it could help in reducing configuration effort. Furthermore, dependency graphs can be used for training purposes, e.g. by clarifying the relations and dependencies in the model.

Further topics are conceivable as potential application areas for the dependency analysis shown in this paper. Knowing and therefore being able to predict the impacts of changes, innovative configuration[6] can be realized. [5] Another application area is reconfiguration: knowing the modifications to a configuration model and their temporal order and knowing with which version of the configuration model a product has been configured with, it is possible to reconfigure it – e.g. for adapting new development or bug fixes. These topics have not yet been addressed by our research group and therefore present future work.

Acknowledgments

This research has been supported by the European Community under the grant IST-2001-34438, ConIPF - Configuration in Industrial Product Families.

References

1. V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz, 'EngCon - Engineering & Configuration', in *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, (July 19 1999).
2. A. Günter, *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995.
3. A. Günter and L. Hotz, 'KONWERK - A Domain Independent Configuration Tool', *Configuration Papers from the AAAI Workshop*, 10–19, (July 19 1999).
4. A. Hein, J. MacGregor, and S. Thiel, 'Configuring Software Product Line Features', in *Proc. of ECOOP 2001 - Workshop on Feature Interaction in Composed systems*, Budapest, Hungary, (June, 18 2001).
5. L. Hotz and T. Vietze, 'Innovatives Konfigurieren in technischen Domänen', in *Proceedings: S. Biundo und W. Tank (Hrsg.): PuK-95 - Beiträge zum 9. Workshop Planen und Konfigurieren*, Kaiserslautern, Germany, (February 28 - March 1 1995). DFKI Saarbrücken.
6. M. Klein and N.F. Noy, 'A Component-based Framework for Ontology Evolution', in *Proceedings of the Workshop on Ontologies and Distributed Systems, IJCAI-03*, Acapulco, Mexico, (2003).
7. T. Krebs, L. Hotz, C. Ranze, and G. Vehring, 'Towards Evolving Configuration Models', in *Proc. of 17. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2003) – KI 2003 Workshop*, pp. 123–134, Hamburg, Germany, (September, 15-18 2003).
8. S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick, 'A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems', in *Proceedings of In-Vehicle Software 2001 (SP-1587)*, pp. 43–55, Detroit, Michigan, USA, (March, 5-8 2001).

[6] A configuration process is called innovative, when a solution is computed that has not previously been covered by the configuration model

An Extensible Synthesis Framework

Theodor Lettmann lettmann@upb.de
Benno Stein stein@upb.de

Paderborn University
Department of Computer Science
D-33095 Paderborn, Germany

Abstract This paper describes the background and ideas of an on-going development at our institute: The creation of a framework that comprises state-of-the-art configuration and synthesis technology.

Key words: configuration framework, design problem solving, synthesis tasks

1 Introduction

Starting point of a design problem is a space $\mathcal{S}$ of possible design solutions along with a set D of demands. Solving a design problem means to determine a system $S^* \in \mathcal{S}$ that fulfills D. Typically, S^* is not found by experimenting in the real world but by operationalizing a search process after having mapped the system space, $\mathcal{S}$, onto a model space $\mathcal{M}$. $\mathcal{M}$ comprises all models M that could be visited during the design process.[1] Figure 1 illustrates these connections.

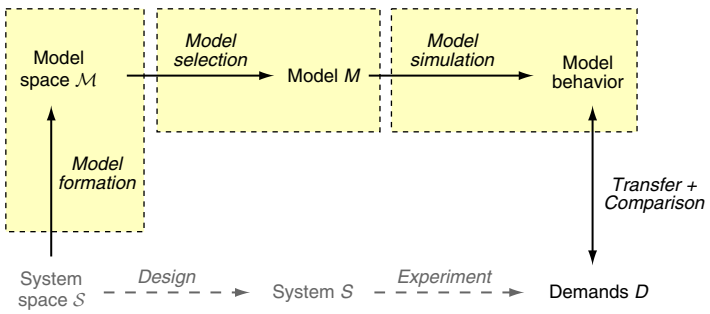

Figure 1. A generic scheme of design problem solving according to [16]: Given is a space $\mathcal{S}$ of possible design solutions and a set of demands D. On a computer, $\mathcal{S}$ is represented as a model space, $\mathcal{M}$, wherein a model M^* is searched whose behavior fulfills D.

It is the job of a design algorithm to efficiently find a model $M^* \in \mathcal{M}$ whose simulation produces a behavior that complies with D and which optimizes a possible goal criterion. A synthesis framework can support this job at several places:

[1] Following Minsky we call M a model of a system S, if M can be used to answer questions about S [9]. In particular in connection with design problems M may establish a structural, a functional, an associative, or a behavioral model.

1. *Model Formation.* Given a system space $\mathcal{S}$ there exist several paradigms to define a suited model space $\mathcal{M}$ for synthesis purposes. They include differential equations [6], taxonomies (is-a relations), compositional hierarchies [3], design graph grammars [1, 12], resource descriptions [5], case-based reasoning [7, 11], or propositional logic [13, 14].
2. *Model Selection.* Usually model selection does not happen by chance but is a guided process that defines in which way the search is organized. In fact, most of the model formation paradigms mentioned before advise a particular search strategy.
3. *Model Simulation.* Within this step the "behavior" of the selected model is analyzed—a process which can vary largely in its complexity: E. g., when a behavior-based design problem is to be solved, complex differential-algebraic equations need to be processed, while in a simple compositional design problem merely the existence of components is checked [4].

There is a strong research background for solving synthesis problems at our institute, and, in the past, we have developed several specialized configuration solutions for real world tasks [6, 12, 15]. Based on this experience we launched in 2003 the development of an extensible synthesis framework that shall comprise a wide range of technologies related to defining model spaces on the one hand, and, on the other hand, to select, search, and simulate models from a given model space.

Note that we are not aiming at a *generic* synthesis engine since we know that many synthesis problems require specialized and tailored solutions. Instead, we take existing configuration and synthesis technology and put much emphasis on software engineering aspects: extendibility, transparent coupling of technologies, state-of-the-art interfaces, or Web-based access.

Related Work

There has been considerable research effort related to configuration and design in the last 20 years. The starting point for developing tailored software techniques to tackle synthesis problems was the R1 system, which emerged from a joint project between CMU and DEC in the early eighties [8]. R1 was no generic platform but specifically designed to realize the configuration of Vax computers.

The PLAKON system developed from the TEX-K project (1986 - 1990) whose objective was the creation of a knowledge-based kernel for planning and configuration tasks, independent from a concrete domain. Configuration technology in PLAKON was centered around the skeletal configuration paradigm. Aside from powerful compositional and taxonomic descriptions PLAKON enabled the modeling of basic functional and associative constraints as well as the specification of control knowledge to guide the search.

Based on the experiences gathered in TEX-K the successor of PLAKON—the system KONWERK was created. Among others, KONWERK extended the modeling capabilities of PLAKON and came along with a clear modular architecture. Both PLAKON and KONWERK were implemented in Lisp.

The system ENCON can be considered as a partial relaunch of the KONWERK platform, having a strong emphasis on commercial requirements [2]. ENCON is implemented in Java, and, since its configuration kernel is encapsulated by an abstraction layer, the tool is qualified to bring configuration technology to the Web.

2 The ESF Synthesis Framework

The configuration and design projects that we conducted in the last fifteen years came from very different domains, such as logistic systems, telecommunication systems, fluidic engineering, computer networks, testing equipments, or software systems. Our endeavors to find adequate solutions taught us several lessons, among others the following:

- Finding the "right" level of abstraction when defining the model space $\mathcal{M}$ is both the most crucial and difficult job.
- Given a real-world configuration task, then no modeling paradigm will do it all, say, modeling concepts have do be adapted, modified, or even created.
- No configuration strategy can provide that degree of flexibility that it leads to an acceptable search performance for all projects mentioned above.

Nevertheless, every configuration task aims at the construction of a system (to be precise: model of a system) whose function fulfills a set of given demands D. Function, in turn, can be regarded as a consequence of the interplay between structure and behavior of a system [17]. Put another way, a configuration tool must provide concepts for modeling and processing both structure and behavior. [2]

These considerations form the ground for our implementation efforts that flow into the extensible synthesis framework ESF (cf. Figure 2). It provides modeling paradigms and processing methods related to the following three knowledge sources:

(a) *Strategy Knowledge.* Design or configuration strategies can be easily defined by means of a scripting language. Strategies are objects that can be created, stored, combined, triggered, or modified.

(c) *Component Knowledge.* This knowledge source relates to the definition of component classes and instances. Component behavior can be described at different levels of granularity, which includes resource descriptions or simple functional constraints, and will allow complex behavior descriptions based on the Modelica™ language in the near future [10].

(c) *Structural Knowledge.* Instead allowing only compositional and taxonomic hierarchies we use the rather new concept of design graph grammars [12, 15]. Design graph grammars have been developed as a powerful means to manipulate arbitrary structure models of technical systems.

ESF provides several algorithms for manipulating the aforementioned object types. In particular, objects can be shared among different algorithms using a blackboard; the blackboard is also used to store a certain state of the search or to remember alternative choice points among which a strategy may chooses the most promising one. New

[2] PLAKON, for instance, addressed the question of structure with concepts for modeling skeletal plans; likewise, it addressed the question of behavior by a constraint processing mechanism.

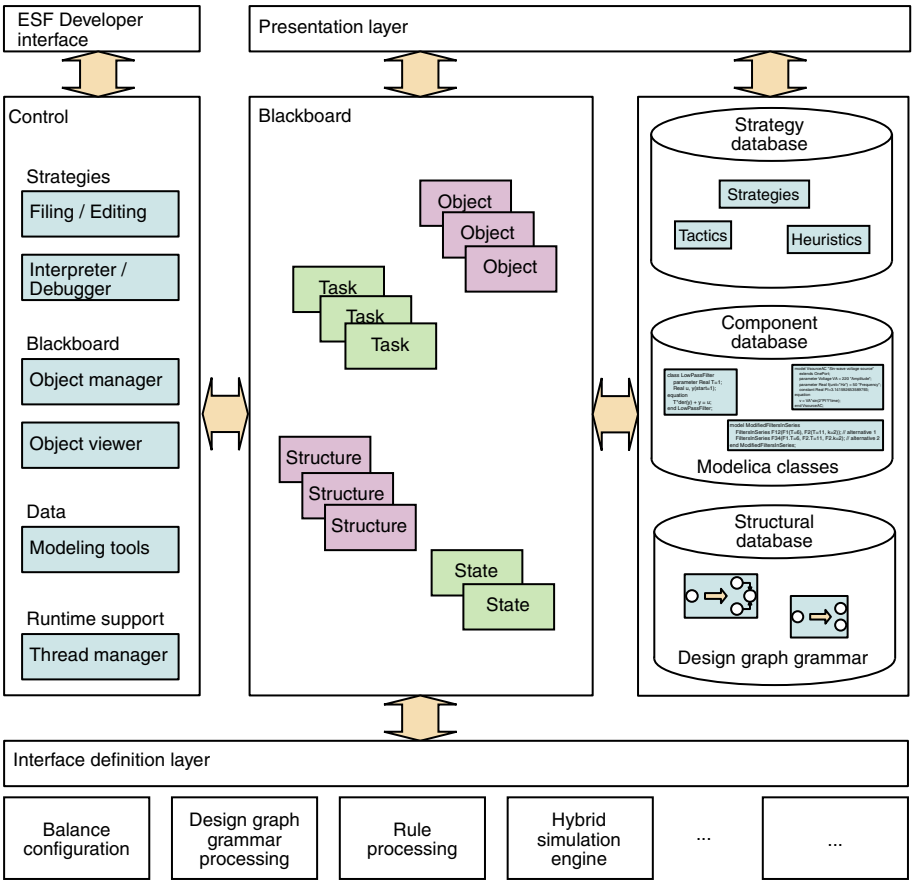

Figure 2. Conceptual view onto ESF. A blackboard serves as central communication turntable (middle); the three different knowledge sources are shown on the right-hand side, the respective algorithms are shown below.

modeling paradigms or component types can integrated straightforwardly: An interface definition layer prescribes a set of accessors and manipulation functions that each object must implement to be shareable via the blackboard.

ESF is being implemented in Java. Note that there are various interesting software-technical aspects that deserve an in-depth discussion; they will be described in a forthcoming white paper.

3 Application: Task-Oriented Plant Layout

This section outlines a particular job-shop manufacturing task that has been realized with ESF. The task combines several aspects of planning, configuration, and simulation: For the production of small metal parts such as screws or gear-wheels the necessary machines have to be selected, a suited processing sequence has to be found, and, a layout of the manufacturing plant has to be generated (see Figure 3).

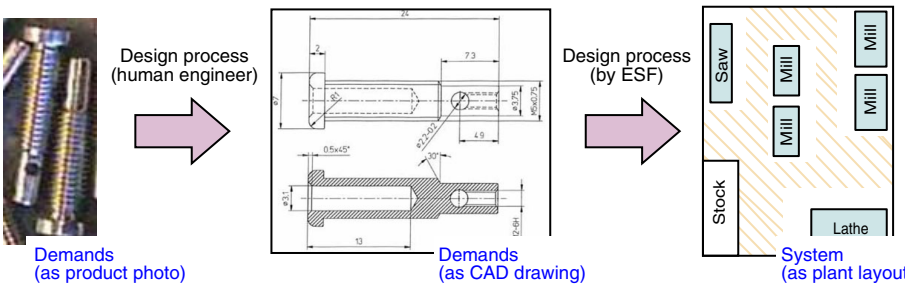

Figure 3. The figure shows the desired product (left), a technical drawing from which the manufacturing constraints are derived (middle), and the plant layout (right), which is part of the solution of the design process.

The solution of this design task happens in a cycle of the following steps:

1. Creation of a process flow graph by means of a design graph grammar.
2. Selection and parameterization of the machines used in the process flow graph.
3. Simulation of the work-flow. In case of a positive evaluation, the solution is compared to alternative realizations already derived.

The machines that are used to realize the production process provide certain functionalities: sawing, drilling, milling, nibbling, or turning. Their capacities and performances, as well as their constraints are modeled by means of the resource-based paradigm. For the simulation of the process flow along with the configured machines tailored algorithms were implemented and plugged into ESF.

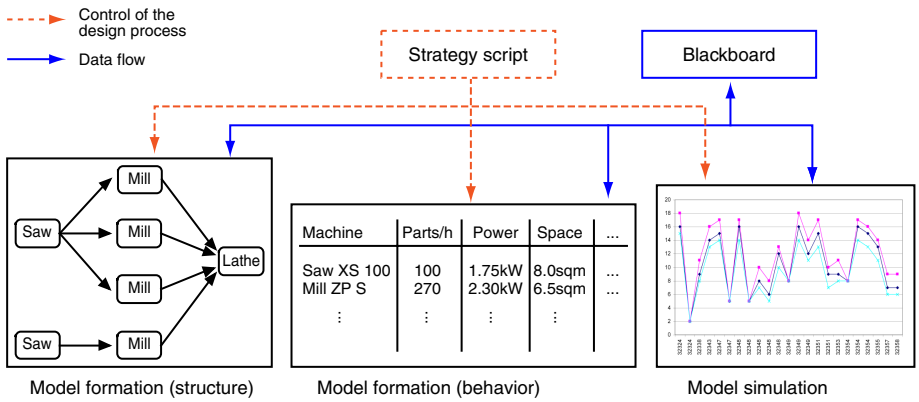

Figure 4. The employed modeling paradigms: Design graph grammars for plan generation (left) and resource-based configuration for material properties, production constraints, and space requirements (middle). The simulation (right) is necessary to determine an optimum solution.

A detailed description of the system, the design graph grammar rules, and the resource model can be found in [18].

References

1. Erik K. Antonsson and Jonathan Cagan. *Formal Engineering Design Synthesis*. Cambridge University Press, 2001. ISBN 0-521-79247-9.
2. V. Arlt, A. Günter, O. Hollmann, and T. Wagner. Engineering & Configuration—A Knowledge-Based Software Tool for Complex Configuration Tasks. In *Proceedings of the AAAI 99 Workshop on Configuration*, 1999.
3. R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode. PLAKON—An Approach to Domain-independent Construction. Technical Report 21, BMFT Verbundsprojekt, Universität Hamburg, FB Informatik, March 1989.
4. John S. Gero. Design Prototypes: A Knowledge Representation Scheme for Design. *AI Magazine*, 11:26–36, 1990.
5. M. Heinrich and E. W. Jüngst. A Resource-based Paradigm for the Configuring of Technical Systems for Modular Components. In *Proc. CAIA '91*, pages 257–264, 1991.
6. Marcus Hoffmann. *Zur Automatisierung des Designprozesses fluidischer Systeme*. Dissertation, University of Paderborn, Department of Mathematics and Computer Science, 1999.
7. Mary Lou Maher and Pearl Pu, editors. *Issues and Applications of Case-Based Reasoning in Design*. Lawrence Erlbaum Associates, Mahwah, New Jersey, 1997.
8. John McDermott. R1: A Rule-based Configurer of Computer Systems. *Artificial Intelligence*, 19:39–88, 1982.
9. Marvin Minsky. Models, Minds, Machines. In *Proceedings of the IFIP Congress*, pages 45–49, 1965.
10. Modelica Association. *Modelica™—A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial*. Modelica Association, Linköping, Sweden, 2000.
11. Michael M. Richter. Introduction to CBR. In Mario Lenz, Brigitte Bartsch-Spörl, Hans-Dieter Burkhard, and Stefan Weß, editors, *Case-Based Reasoning Technology. From Foundations to Applications*, Lecture Notes in Artificial Intelligence 1400, pages 1–15. Berlin: Springer-Verlag, 1998.
12. André Schulz. *On the Automatic Design of Technical Systems*. Dissertation, University of Paderborn, Department of Mathematics and Computer Science, 2001.
13. David B. Searls and Lewis M. Norton. Logic-based Configuration with a Semantic Network. *Journal of Logic Programming*, 8:53–73, 1990.
14. Benno Stein. *Functional Models in Configuration Systems*. Dissertation, University of Paderborn, Institute of Computer Science, 1995.
15. Benno Stein. *Model Construction in Analysis and Synthesis Tasks*. Habilitation, University of Paderborn, Institute of Computer Science, June 2001.
16. Benno Stein. Engineers Don't Search. In Wolfgang Lenski, editor, *Proceedings of a Symposium on Logic versus Approximation, Schloss Dagstuhl, Germany*, volume LNAI of *Lecture Notes in Artificial Intelligence*, Berlin Heidelberg New York, October 2003. Springer.
17. Nam Oyo Suh. *Axiomatic Design*. Oxford University Press, 2001. ISBN 0-19-513466-4.
18. Achim Wullenkort. Eine Entwicklungsumgebung für Design-Aufgaben. Diploma thesis, University of Paderborn, Institute of Computer Science, 2004.