Table of Contents[1]

[1] Online version of the proceedings is available at http://www.cs.ust.hk/~qyang/aips02/

Workshop Program

9:00- 9:05 Opening Remarks (M. Veloso and Q. Yang)

9:05-10:30 Session I (Web and Database Services) Chair: M. Veloso

- **Invited Talk** (40 mins) *Invited Talk*
- (15 min) Greg Barish <barish@ISI.EDU> and Craig Knoblock ISI/USC. An Expressive and Efficient Language for Information Gathering on the Web
- (15 min) Sheila McIlraith <sam@KSL.Stanford.EDU>, Stanford University and Tran Cao Son, New Mexico State University. Adapting Golog for Composition of Semantic Web Services
- (15 min) Laurie Hiyakumoto hyaku@cs.cmu.edu and Manuela Veloso, CMU. Towards Planning and Execution for Information Retrieval

10:30-11:00 Coffee Break

11:00-12:25 Session II (Novel Business Applications) Chair: B. Nebel

- **Invited Talk** (40 mins) *Invited Talk*: Craig Knoblock, ISI/University of Southern California. *Title: Planning for Information Integration on the Web*
- (15 min) Qiang Yang qyang@cs.ust.hk, Hong Kong University of Science and Technology. Towards Statistical Planning for Marketing Strategies
- (15 min) Biplav Srivastava <sbiplav@in.ibm.com>, IBM. Using Planning for Query Decomposition in Bioinformatics
- (15 min) Wei Zhang <wei.zhang@pss.Boeing.com>, The Boeing Company. Planning as Sequential Decision Making

12:25-14:00 Lunch Break

14:00-16:00 Session III (Robotic Planning) Chair: S. Koenig

- **Invited Talk** (40 mins) *Invited Talk*: Brian Williams, MIT

- (15 min) Michael Beetz beetzm@informatik.tu-muenchen.de, Munich University of Technology, Germany. Towards Comprehensive Computational Models for Plan-based Control of Autonomous Robots
- (15 min) L. Castillo, J. Fdez-Olivares <faro@decsai.ugr.es> and A. GonzálezJuan Fdez-Olivares, Universidad de Granada. A conditional planning approach for the autonomous design of reactive and robust sequential control

- **Panel I** (45 min) Robotic Planning. **Panel Chair: Manuela Veloso**

16:00-16:30 Coffee Break

16:30-18:00 Session IV (Novel Formulations) Chair: Craig Knoblock

- (15 min) Elly Zoe Winner <elly+@cs.cmu.edu> and Manuela Veloso, CMU. Automatically Acquiring Planning Templates from Example Plans
- (15 min) Keith Golden (kgolden@ptolemy.arc.nasa.gov) and Jeremy Frank, NASA Ames Research Center. Universal Quantified Plans
- (15 min) Wout van Wezel <w.m.c.van.wezel@bdk.rug.nl> and René Jorna, University of Groningen, The Netherlands Binding characteristics for planning diversity

- **Panel II** (45 min) Novel Planning Formulations for Web and Business Applications. **Panel Chair: Qiang Yang**

18:00-18:05: Closing Remarks

WORKSHOP CO-CHAIRS:

Manuela Veloso (CMU) veloso@cs.cmu.edu and Qiang Yang (Hong Kong University of Science and Technology, Hong Kong) qyang@cs.ust.hk

PROGRAM COMMITTEE:

Craig Knoblock (ISI/USC) knoblock@isi.edu
Sven Koenig (Georgia Institute of Technology) skoenig@cc.gatech.edu
Bernhard Nebel (Institut für Informatik, Germany) nebel@informatik.uni-freiburg.de
Wei Zhang (Boeing) wei.zhang@pss.Boeing.com

Preface

It is healthy for any scientific field to take critical self-examinations periodically. A review of recent publications in major conferences reveals that the majority of published work in planning has been the efficient sequencing of operators in a well-defined state-based world. The majority of this work assumes that a planning system is given clear goal, operator and state definitions in a logical form, and that the goal to be accomplished is to find sequences of operators to achieve these goals. While significant progress has been made in this direction, there has been a lack of widespread applications of this model as compared to, for example, machine learning, data mining and speech recognition. This raises the questions of whether these assumptions actually hold in real world applications, and if this direction of research is indeed highly relevant to practical applications of planning. The heart of this exploration is the question: what does real planning involve?

This AIPS'02 workshop on real-world planning is then devoted to the exploration of alternative views of planning. We wish to organize the workshop to explore novel issues underlying planning that are beyond operator sequencing. We encourage a bottom-up approach that start at high-impact application areas that might not be considered as planning traditionally. We further encourage a backwards analysis of the applications towards alternative views and definitions for planning. In our Call for Papers, we gave a list of potential high impact areas for planning, including

- Supply Chain Management Applications
- Real time Robot Planning
- Real time Multiagent and Multirobot Planning
- Travel Planning
- Planning for Information Gathering and Database Queries
- Planning and Data Mining
- Business Marketing Strategy Planning and Financial Planning
- Urban planning
- Planning in Software Engineering and Workflow Management

Furthermore, in each of these applications, we wish to make clear the nature of the problem, the issues of interest to planning researchers, exposition of applications domains, and criteria of success. The techniques involve

- Planning and execution
- Plan management
- Deliberative and reactive planning
- Plan recognition
- Plan quality
- Planning and resource allocations
- Planning and learning

The workshop takes place on April 23, 2002 in Toulouse, France before the AIPS 2002 conference. It features invited talks by Craig Knoblock and Brian Williams, two panels and paper presentations. The authors of the 13 papers in this collection made an important effort in the direction that we hoped to follow. In addition, the program committee members Sven Koenig, Craig Knoblock, Bernhard Nebel and Wei Zhang diligently reviewed papers and provided valuable comments to many. We wish to thank them all!

Manuela Veloso and Qiang Yang, April 2002

Invited Talk

Planning for Information Integration on the Web

Craig Knoblock
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
knoblock@isi.edu

There are many interesting planning problems in the context of information integration, ranging from general issues such as query planning and planning for information gathering to specific applications such as travel planning. Unfortunately, few of these problems fit the standard mold for Strips-style planning. In this talk I will describe several planning problems and how the characteristics of these problems lead us to work on very different approaches from what planning research has typically focused on. Based on this experience, I will present my views on how the planning community should broaden its definition of planning to encompass a wider variety of planning tasks.

Invited Talk

Model-based Programming of Cooperative Agile Vehicles

Brian Williams
MIT
williams@mit.edu

In the future, webs of unmanned vehicles will act together to robustly achieve elaborate missions within uncertain environments. This web may be a distributed satellite system forming an interferometer, or may be a heterogeneous set of rovers and blimps exploring Mars. We coordinate these systems by introducing a reactive model-based programming language (RMPL) that combines within a single unified representation the flexibility of embedded programming and reactive execution languages, and the deliberative reasoning power of temporal activity planners. To support fast mission planning as graph search, the KIRK planner compiles an RMPL program into a temporal plan network (TPN), similar to those used by temporal planners, but extended for symbolic constraints and decisions. To robustly coordinate agile air vehicles or rover maneuvers we combine the Kirk planning algorithm with algorithms for cooperative, kinodynamic path planning, randomized algorithms for kinodynamic path planning. This work will be described in the context of our cooperative vehicle test beds, including a set of ATRV rovers, a distributed sensor net, and a range of autonomous helicopters and air vehicles.

An Expressive and Efficient Language for Information Gathering on the Web

Greg Barish and Craig A. Knoblock

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
{barish, knoblock}isi.edu

Abstract

While network query engines make it possible to gather and combine data from multiple Web sources, these systems primarily focus on efficient query execution and do not solve some of the more complicated problems of online information gathering. Such problems require alternative types of control flow and better integration with the external world; the unique nature of the Web requires query plans be expressive enough to accommodate these demands. In this paper, we describe an information gathering plan language that is expressive and promotes efficient execution. Through its support for subplans, recursion, and a unique set of operators, the language allows plans that can interactively gather data over a series of pages, monitor remote sources, and asynchronously notify users of updates and results. We also present a execution system that efficiently implements the plan language using a dataflow-style executor capable of pipelining data between operators.

Introduction

Current research on network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001) has shown that it is possible to gather and combine data from multiple Web sources using plans similar to those found in traditional database systems. However, such research has focused primarily on the efficiency of plan execution and has tended to ignore the problems associated with more complicated types of Internet information gathering.

The unique nature of the Web is such that certain types of queries require a plan language more expressive than those capable of only basic integration. Consider collecting the results of a search engine query. Nearly all search engines display query results spread across multiple result pages. To collect all of the data, an automated system must be capable of interleaving the collection of partial results with navigation to additional results and must be able to eventually decide when to stop. The control flow required for such a task is not supported by the plan languages of existing network query engines.

Another unique aspect of the Web is that it is highly dynamic and there is considerable interest in being able to monitor sources. However, since the Web has no built-in trigger facility, one has to "discover" updates by querying the Web over a period of time that extends beyond that of a single interactive query. To track an integrated set of data requires a language capable of managing intermediate results and communicating important updates to users asynchronously (i.e., via e-mail) as necessary. Again, most network query engines do not support such capabilities.

While these examples demonstrate that better plan expressivity is desirable, so too is efficient query execution. Gathering data on the Web is an I/O-intensive process that renders CPUs idle for periods of *seconds* during plan execution. Thus, what is needed is a plan language that is not only expressive but also very efficient: specifically, one that supports highly concurrent execution.

A plan language can provide substantial degrees of parallelism in two ways. The first is to support a dataflow representation of plans. The partial ordering of operators enabled by a dataflow representation describes execution in maximally parallel terms – operators are only limited by their own data dependencies. A second language-level strategy is to support operators capable of processing pipelined data (i.e.,. tuple-oriented processing). Pipelining refers to the production and consumption of data as soon as possible – producers emit incremental results to consumers – enabling both to work in parallel on the same relation.

In this paper, we present an information gathering plan language that is both expressive and efficient. The proposed plan language is modular and supports the notion of *subplans* to encourage reusability and facilitate recursion. In addition, the language consists of operators that interact with the external world so that it is possible to monitor sources and asynchronously notify users of important updates. While providing better expressivity, plans in this language are efficient because they consist of a dataflow-style ordering of operators and because those operators support the pipelining of data during execution.

The rest of this paper is organized as follows. Section 2 establishes basic terminology, discusses the details of more difficult Web query tasks, and provides an example that we will use throughout the rest of the paper. In Section 3, we propose an information gathering plan language and describe how it enables us to solve the types of problems shown earlier. In Section 4, we discuss the efficient execution of plans generated in this language. Finally, in Section 5, we discuss the related work, both in terms of network query engines and intelligent agents.

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Gathering and Monitoring Web Data

In this section, we describe the problem of gathering and monitoring data on the Web. We first describe basic integration tasks and how existing information gathering plan languages allow these tasks to be completed. Next, we describe more complicated types of information gathering tasks and how they necessitate a more expressive plan language. Finally, we provide an example problem that will be the basis for discussion throughout the paper.

Basic information gathering tasks

Basic Web-based information consists of retrieving data from multiple sources, combining, and then filtering as necessary. For example, the plans described in (Friedman & Weld 1997), (Ives et al. 1999), and (Barish et al. 2000) query distinct Web sites, combine the data found in both (either by unioning or joining that data), and then either filter these results or use them to query other web sources. These plans have simple control flow and involve the same types of operators found in traditional database systems – Retrieve, relational operators like Select, Project, Join, and set-theoretic operators like Union, Minus, and Intersect.

Current technology for querying the Web in this manner exists in two forms. One is that provided by Web-based information mediators (Genesereth et al. 1996, Knoblock et al. 2001). These systems use high-level domain models to describe how logical entities are related Web sources. They utilize Web site *wrappers* to convert semi-structured HTML into structured relations and thus allow web sites to be queried as if they were databases. Mediators have largely focused on enabling multiple heterogeneous data sources to be integrated (through query reformulation). With these systems, it is possible to write queries that are answered through information gathering plans that combine and filter data from multiple sources.

A second, more recent technology for accomplishing these types of tasks comes in the form of network query engines. Although these systems enable the Web to be queried in the same way that mediators do, they have generally focused on the need for efficient execution and on the need to process online XML data. They have been mostly concerned with adaptive execution techniques to overcome the inherent latency of querying remote web sites.

In short, existing mediators and network query engines allow Web data to be queried in a manner similar to that found in traditional database systems. The control flow of the plan and types of operators involved are largely the same. In general, these systems have focused largely on the challenges of interoperability and efficiency.

More complicated tasks

The nature of the Web is such that the expressivity provided by traditional query plans is often insufficient for solving other types of common, yet more complicated online information gathering tasks. In particular, the Web is unique in at least two major respects: (a) it is primarily a visual medium and (b) its highly dynamic nature often invites the need for monitoring. Let us consider how each of these aspects independently impacts online querying.

As a visual medium, data on the Web is often organized in a way that makes sense for visual consumption. For example, querying a web source through an HTTP POST or GET often results in answers to that query being organized across multiple pages. For example, a query to a search engine can result in hundreds of web pages that each contain part of the answer. To collect the complete answer to such queries, it is necessary to navigate to each page, collect the results on that page, find the "next page" link, navigate to the next page, collect the results on that page, and so on. This manner of alternating retrieval with navigation is unique to the Internet does not have an equivalent in traditional database systems.

Secondly, Web sources can be highly dynamic and often need to be monitored. Unfortunately, the Web lacks a database-style trigger facility that notifies users when data has changed and does not provide any automated means for identifying differences between current results and those that existed prior to the update. Instead, updates to its data are only realized through a process of repetitive querying, collection of new results, and then comparison of these new results with prior results to discover the differences. Thus, periodic execution and some sort of mechanism for comparison between queries is necessary.

Example

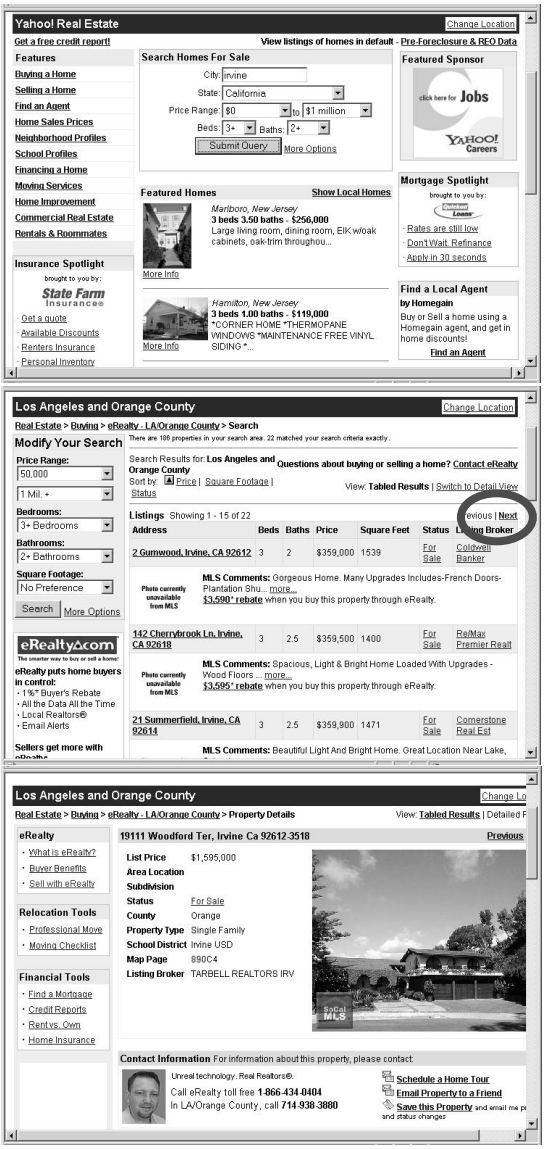
To demonstrate how more complicated types of information gathering problems require more expressive plans, it is useful to describe a detailed example. Consider using the Internet to locate a new home to buy. Suppose we wish to use a site like Yahoo Real Estate to periodically locate houses that meet our search criteria. For example, we wish to find houses that meet a certain set of price, location, and number of rooms constraints.

First, let us discuss how users perform this task manually. Figures 1a, 1b, and 1c show the interface and result pages for Yahoo Real Estate. To query for new homes, users first fill the criteria shown in Figure 1a – specifically, they enter information that includes city, state, maximum price, etc. Once they fill in this form, they submit their query to the site. The initial results are shown in Figure 1b. However, notice that this page only contains results 1 through 15 of 22. To get the remainder of the results, a "Next" link must be followed to the page containing results 16 through 22. Finally, to get the details of each house, users must follow the link associated with each listing. A sample detail screen is shown in Figure 1c.

In practice, performing this task requires manually repeating the above process over a period of days, weeks, or even months. The user must both query the site periodically and somehow keep track of new results. This latter aspect can require a great deal of work – users must note which houses in each result list are new entries.

It is possible to automate part of this process with current data integration technologies. For example, we can use mediators or network query engines to gather and extract data from web pages. But most of these systems do not provide any means for monitoring sources and none provide the ability to gather data spread across multiple pages. To accomplish both tasks, we need plans capable of expressing other types of control flow (such as looping) and operators that facilitate monitoring.

We can consider how such plans generally might look. Figure 2 shows an abstract plan for monitoring Yahoo Real Estate. As the figure shows, search criteria is used to generate houses from Yahoo Real Estate. Houses are separated from their "next page" link and compared against houses that already existed in a local database. Then, the resulting set of new houses are queried for their details and the results are e-mailed to the user. These new results are also appended to the database so that future queries can distinguish new results. Meanwhile, the "next page" link is followed and the resulting new houses go through the same process. Next page links are followed until no more pages are found (i.e., no more next link).



Figures 1a, 1b, & 1c: Querying Yahoo Real Estate

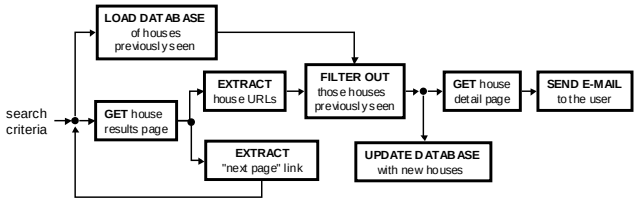


Figure 2: Abstract plan for Yahoo Real Estate

Expressive & Efficient Information Gathering

In this section, we describe an information gathering plan language that makes it possible to construct plans that can accomplish more complicated information gathering tasks, such as the type shown in the abstract plan Figure 2. There are several basic aspects of this plan language to consider – the dataflow representation of plans, the logical pipelining of data during execution, the typing and manipulations on data, the set of operators that are provided, support for modular design through the notion of subplans, and support for information gathering tasks that require looping through use of recursion.

Dataflow representation

All plans in the language we propose consist of a name, a set of input and output variables, and a set of unordered operators that represent the dataflow graph of the plan. A dataflow representation of a plan is desirable from an efficiency standpoint because it describes the maximally parallel mode of execution (Dennis 1974). In contrast to von Neumann models, which rely on an instruction counter to sequentially execute a list of instructions (or operators), a dataflow model allows execution to occur on any operator, whenever its data dependencies are fulfilled. This makes execution fully decentralized, independent for each operator. Thus, execution can be highly concurrent.

Figure 3 shows an abstract plan. As shown, a header part communicates the name of the plan (P1 in this example) and the list of input variables (a and b), and output variables (g). The body section of the plan contains the set of operators. The example below shows four operators – Op1, Op2, Op3, and Op4. Each operator instance consumes one or more inputs and produces zero or more outputs. As shown below, the set of inputs for each operator appears to the left of the colon delimiter and the set of outputs appears to the right of the delimiter.

Figure 4 illustrates how edges in the dataflow graph of operators are communicated through variable names. For example, as described by Figure 3, operator Op1 produces

```
PLAN P1 {
   INPUT: a, b
   OUTPUT: g

   BODY {
      Op1 (a, b : c)
      Op2 (b, c : d, e)
      Op3 (d : f)
      Op4 (e, f : g)
   }
}
```

Figure 3: Sample plan

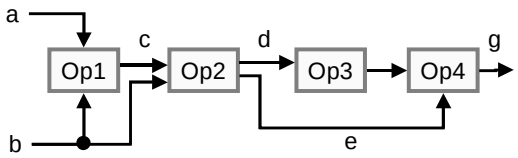


Figure 4: Plan represented as a dataflow graph

c, which is consumed by Op2, and Op2 produces d and e, which are consumed by Op3 and Op4, respectively. Figure 4 shows the corresponding edges that between the operators.

Although the body part of the plan language lists operators in a linear order, this ordering does not affect when they are actually executed. Per the dataflow model of processing, operators fire whenever their individual data dependencies are fulfilled. For example, Op2 can execute when any of its individual inputs b or c are present. Thus, Op2 executes once at the start of the plan (because b is available) and then shortly later on, when c becomes available. In summary, the only ordering of execution that exists at the plan level is that which is communicated by the producer/consumer relationship between operators

Logical data pipelining

Each of the variables in the plan above are logically relational data streams. A stream is a set of tuples in a relation, followed by an *end of stream* (EOS) marker. Operators in the plan logically execute when they receive a tuple for any of their input streams. The conditions that describe when operators can execute is also known in dataflow literature as the firing rule.

For example, a set-theoretic Union operator would take two input streams – *lhs* and *rhs* – and output a stream called *unioned_result* that consists of the unique set of tuples defined by the intersection of lhs and rhs This operator can fire whenever a lhs or rhs tuple is present and emit a unioned_result tuple for each firing. In part, this is due to the nature of the operator. A Minus operator, in contrast, would take two inputs named lhs and rhs and emit a *minus_result* stream that was based on the subtraction of rhs from lhs. However, even though the Minus operator can fire upon receiving a tuple, it cannot emit a *minus_result* tuple until the rhs stream EOS has been received. Both the Union and Minus operator must logically maintain state between invocations (both must not emit duplicates and Minus must keep all of the rhs in memory so that it can be applied to later lhs tuples).

Data types and common manipulations

Data in the system is communicated logically as relations and physically as tuples (i.e., through pipelining). Each tuple consists of a set of attribute/value pairs. Each attribute can be one of five types: char, number, date, relation (embedded), or document (i.e., a DOM object). Embedded relations are supported because they reduce the amount of data communicated during execution.

XML data is supported by the system and is associated with the Document attribute type. The language contains specific operators that allow XML to be converted to relations, for relations to converted to XML documents, and for attributes that are XML documents to be queried in their native form using XQuery. Since XML documents are encapsulated by tuples, they can be pipelined between operators; when/if it is desirable to pipeline the data contained in an XML document, the document is first converted to a relation, is streamed through the system, and can be put back together again as XML later, if desired.

In terms of common manipulations, operators vary on how they output their results with respect to the incoming data. In particular, there are two modes of interaction that merit discussion: the performing of dependent joins and the packing/unpacking of relations.

In data integration plans, it is common to use data collected from one source as a basis for querying additional sources. However, it is tedious (and sometimes impossible because of ordering constraints) to manually join the data input to an operator onto the output data it produces. Instead, many of the operators in this language perform a dependent join of input tuples onto the output tuples that they produce. For example, if the language supported an operator called Round that rounded a floating point value in a column to its nearest whole integer value, and if the input data consisted of the tuples ((Jack, 89.73), (Jill, 98.21)) then the result after the Round operator executes would be of ((Jack, 89.73, 90), (Jill, 98.21, 98)). Dependent joins simplify plans and solve problems related to the joining data when no unique key on the input relation exists.

A related mode of interaction to discuss involves the packing and unpacking of relations. Packing relations is useful when you want to associate a relation with an aggregate function, such as count. Instead of creating and managing two distinct results (which often need to be joined later), it is cleaner and more space-efficient to perform a dependent join on the packed version of an input relation with the result output by an aggregate-type operator. For example, if the language supported an aggregate operator called Average, then the result of processing the input described earlier would be (((Jack, 89.73), (Jill, 98.21)), 93.97). Unpacking a relation is necessary to get at the original data. Packing and unpacking is a common activity when a conditional operator needs to evaluate an aggregate measure of a relation and then route it to the proper set of consumer operators which then unpack the data.

Operators

To accomplish more complicated types of information gathering tasks, three basic types of operators are necessary; those that:

- **Gather and manipulate data:** These include the traditional relational operations as well as those capable of processing XML data.
- **Facilitate monitoring:** To effectively monitor data sources, the language includes operators that can access local databases, so that intermediate results can

be stored and then compared against later. In addition, other monitoring operators enable results to be communicated asynchronously – for example, through e-mail, fax, or cell phone.

- **Promote extensibility:** Generally, operating on data either involves operating on individual tuples (single-row functions) or operating on sets of tuples (aggregate functions). The language includes operators that allow users to extend the existing plan language to meet any kind of single-row or aggregate data manipulation necessary.

We now describe the operators in more detail and focus on those not found in other types of network query engines.

Operators for gathering and manipulating data. The language supports basic operators for gathering data from the Web and manipulating it (filtering, combining, etc.). These operators are shown in Table 1.

Of these, the **Wrapper** operator is the most interesting. Its purpose is to use values from an input relation as the basis for querying a specified Web source. In general, wrappers are mechanisms for querying a remote semi-structured Web site as if it were a local relational database. Calling a wrapper involves providing input constraints (if any), executing the wrapper, and then collecting its results. Correspondingly, our Wrapper operator uses values from each tuple of an input relation as the input constraints and queries the remote site accordingly. Results generated by each input tuple are combined with the input that generated them – this is referred to as a *dependent join*. The language also includes operators for manipulating XML data, including XQuery for querying XML and Xml2Rel and Rel2Xml for converting XML data to relational and vice versa. The bulk of the remaining manipulation operators are familiar and can be found in current network query engines and mediators.

Monitoring operators. There are two aspects to the monitoring process – the ability to keep track of past results and the ability to asynchronously notify users of updates. To accomplish these tasks, the plan language we propose supports a set of monitoring-related operators. These are shown in Table 2.

The DbQuery, DbAppend, DbExport, and DbUpdate operators allow plans to interact with local databases. This

Name	Purpose
wrapper	Extracts web page data as relation
xml2rel	Converts XML document into a relation
rel2xml	Converts a relation to an XML document
xquery	Manipulates attributes that are XML documents
select	Filters relation based on specified criteria
project	Extracts specified attributes from relation
join	Combines relations based on specified criteria
union	Performs set union of two relations
minus	Performs set minus of two relations
intersect	Performs set intersect of two relations
pack	Embeds relation in single attribute tuple
unpack	Expands embedded relation from single attribute tuple

Table1: Data manipulation operators

Name	Purpose
dbquery	Fetches relation from DB based on query
dbappend	Append to existing relation in DB
dbexport	Export relation to DB
dbupdate	Processes an update query (no results returned)
email	Emails data to specified e-mail address
fax	Faxes data to specified fax number
phone	Sends text message to specified cell phone number
null	Conditionally routes stream based on if another is empty

Table 2: Monitoring operators

makes it possible to robustly monitor data sources for long periods of time. The Email, Fax, and Phone operators allow data to be accumulated and sent to recipients asynchronously. By its very nature, monitoring is a non-interactive process between user and agent and thus some form of offline propagation of updates is needed.

Extensibility operators. To increase the expressive power of the language, two additional operators – Apply and Aggregate – are included. Both are shown in Table 3. Apply calls user-defined single-row functions on each tuple of relational data and performs a dependent join on the input tuple with its corresponding result. For example, a user-defined single-row function called SQRT might return a tuple consisting of two values: the input value and its square root. The Aggregate operator calls user-defined multi-row functions and performs a dependent join on the packed form of the input and its result. For example, a COUNT function might return a relation consisting of a single tuple with two values: the first being the packed form of the input and the second being the count of the number of distinct rows in that relation.

Name	Purpose
apply	Apply single row function to each relation tuple
aggregate	Apply multi-row function to relation

Table 3: Extensibility operators

Subplans

To promote language supports references to subplans reusability and to facilitate recursion (described later), the. Executing a subplan simply refers to the calling of one plan from another.

Recall that all plans are named and consist of a set of input and output streams. Thus, plans present the same interface as operators. It is thus a simple matter to refer to a plan as if it were an operator. For example, consider the example plan P1 introduced earlier. Figure 5 shows how another plan P2 can reference P1as a subplan.

Subplans encourage modularity and re-use. Once written, a plan can be used as an operator in any number of

```
PLAN P2 {
   INPUT: w, x
   OUTPUT: z

   BODY {
      Op5 (w : y)
      P1 (x, y : z)
   }
}
```

Figure 5: Calling a subplan

future plans. This effectively allows users to build whatever operators they need by combining the set of existing operators as necessary. At the same time, subplans can be easily scheduled as part of a dataflow-style plan and can benefit from data pipelining - just like any other typical plan operator does.

For example, one could develop a simple subplan called *Persistent_Diff*, shown in Figure 6, that uses the existing operators DbQuery, Minus, Null, and DbAppend to take any relation, compare it to a named relation stored in a local database. This plan determines if there was an update, appends the result, and returns the difference. Such a subplan could be as an operator in many types of other plans. Note that executing a subplan does not force us to sacrifice the efficiency of dataflow execution and data pipelining: the Null and DbAppend operators execute at the same time that result is returned to the higher level plan; they also execute on data as soon as it becomes available from the Minus operator.

Recursion

In addition to promoting modularity and re-use, subplans make another form of control flow possible: recursion. As described earlier, a number of online information gathering tasks require some sort of looping-style control flow.

For example, when processing results from a search engine query, an automated information gathering system needs to collect results from each page, follow the "next page" link, collect results from the next page, collect the "next page" link on that page, and so on – until it runs out of "next page" links. If we were to express this in von-Neumann style programming language, we might use a *Do...While* loop to manage this type of information gathering need. However, under a dataflow-model of execution, such an approach in practice requires a fair of synchronization and additional operators.

Instead, this problem can be solved quite elegantly with recursion. We can use subplan reference as a means by which to repeat the same body of functionality and we can use the Null operator as the basis for the exit condition.

As an example of how recursion is used, consider the abstract plan for processing the results of a search engine query. A higher level plan called *Query_Search_Engine*, shown in Figure 7, posts the initial query to the search engine and retrieves the initial results. It then processes the results with a subplan called *Gather_and_Follow*. The search results themselves go to a Union operator and the next link is eventually used to call *Gather_and_Follow* recursively. The results of this recursive call are combined at the Union operator with the first flow.

There are a few notable aspects to the plans shown in

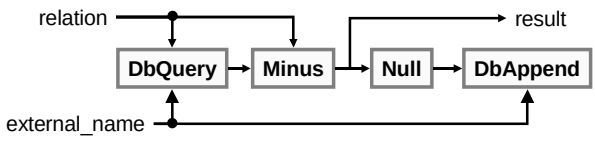


Figure 6: The *Persistent_Diff* subplan

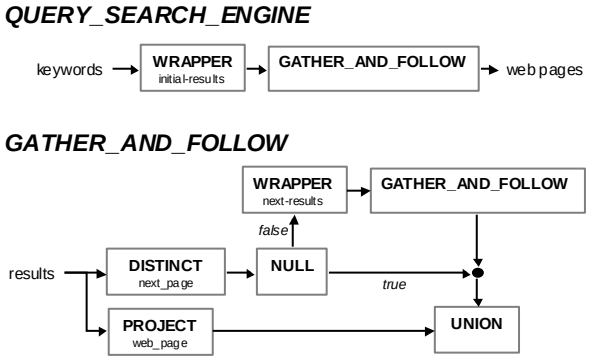


Figure 7: Example of recursion

Figure 7. First, a recursive approach requires very few operators: through the subplan facility, we are able to re-use the body of the gathering-and-following task. Second, data pipelining is exploited: even though recursive execution might go quite deep, results from higher levels are streamed out, back to the higher level *Query_Search_Engine* plan as soon as possible via the pipelined Union operator. Third, notice that we continue to merely require one type of conditional – the Null operator. When the last page is reached, Null routes the EOS to Union (and not to Wrapper, as it normally does). This ends the Union at the lowest level of recursion and this EOS trickles all the way back to the top of the plan, per standard tail-recursive execution.

Revisiting the example

Let us now revisit the earlier house search example and see how such a plan would be written with the proposed plan language. Figure 8 shows one of the two plans, *Get_Houses*, required to implement the abstract real estate plan in Figure 2. *Get_Houses* calls the subplan *Get_Urls*; this plan is nearly identical to the recursive subplan *Gather_And_Follow* in Figure 7, so it is omitted for the sake of brevity. The rest of *Get_Houses* works as follows:

a. A Wrapper operator fetches the initial set of houses and link to the next page (if any) and passes it off to the *Get_Urls* recursive subplan.
b. A Minus operator determines which houses are distinct from those previously seen; new houses are appended to the persistent store.
c. Another Wrapper operator investigates the detail link for each house so that the full set of criteria (including picture) can be returned.
d. Using these details, a Select operator filters out those that meet the specified search criteria.
e. The result is aggregated and e-mailed to the user.

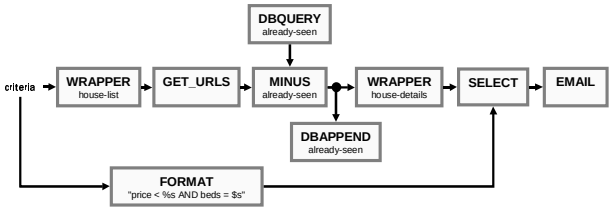


Figure 8: The *Get_Houses* plan

An Efficient Plan Execution Architecture

In this section, we describe an architecture that can efficiently execute the types of plans described in the last section. This architecture is composed of two parts: the language described in the previous section and a dataflow-style executor that efficiently processes these plans.

High-level design

The high-level design of the architecture is shown in Figure 9. The figure shows that the input to the executor is a plan; in addition, a schedule for execution (once, daily, hourly, etc) is input. During each execution, the plan may interface with a local database (e.g. to store tracking information). The figure also shows that it is possible for the plan to communicate updates through a variety of asynchronous communication mechanisms.

Once an input plan is received by the executor, it constructs an internal dataflow graph based on plan operators. At this time, any subplan and recursive relationships are resolved, merging in operators from those plans as appropriate. The system then feeds in input data and execution commences. The input data triggers a subset of plan operators to start firing; their execution and subsequent production trigger other operators that consume their output and so on. If the plan is interactive, output data will be immediately returned to the user as it is produced. Otherwise, it is assumed that the method of user notification (such as email) is already encoded in the plan.

Thus, in the Yahoo Real Estate example, a user can submit the main part of the plan, a set of input data shown, and the schedule of "daily" to the system. The plan will then be executed once (immediately) and an initial set of house search results will be e-mailed to the user. The plan will then be automatically run the next day.

Parallelism during execution

The executor uses threads to service operator execution, and thus functions similar to a threaded dataflow machine (Papadapoulous & Traub 1991). When a tuple becomes available (either via input or through operator production), a thread is assigned to execute a method on the consuming operator with that data. Threads are drawn from a fixed pool, to throttle excessive parallelism and prevent machine resources from being swamped.

The first time that input arrives for a particular operator, an initialization method for that operator is called. During this time, stateful data structures are initialized. All future firings may use this state data structure as is appropriate for that operator. For example, the Union operator uses the state to save all tuples that it has previously output so that it does not output duplicates. The Minus operator keeps this information as well as the entire set of rhs tuples in its state – thus, when new lhs tuples arrive, they are first compared to the rhs set and, if not in this set, output only if they have not been previously output. When EOS markers have been received on each of an operators' inputs, all accumulated state is deleted. State is maintained per level of iteration; thus re-entrant, recursive execution is guaranteed to be correct.

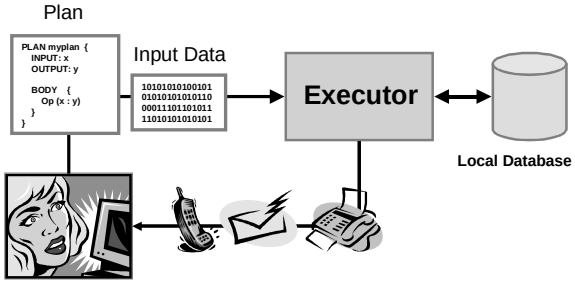


Figure 9: Executor design and interface

In summary, the executor we describe functions as a virtual threaded dataflow machine. By using threads to service operator firings, operator execution can be as horizontally parallel as the number of threads in the fixed pool. Furthermore, it is possible for a producer and consumer operator to fire concurrently on the same logical relation (the consumer operating on an earlier tuple while the producer operates on a later tuple in the stream) thus implementing a form of pipelined, or vertical parallelism.

Data coloring for re-entrancy. Recursion implies that plans are re-entrant and thus introduces an additional complexity – distinguishing data between recursive levels. To address this, the system assigns a color to all data at a particular logical level of execution. For example, during the execution of *Get_Houses*, the input data and any data produced at the same level as a result is assigned the same color. Whenever a subplan like *Get_Urls* is called (including when recursive calls are made), the tuples routed to that subplan are assigned a new color. This allows tuples at multiple levels of execution to be correctly managed by operators in the recursive subplan.

Related Work

In this section, we discuss two areas of related work: that of efficient Internet querying by network query engines and the set of more existing, more general, agent executors.

Network query engines. Recently, network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001), have been proposed as means for efficiently gathering information on the Internet. These systems are mostly concerned with the efficiency of query execution. and have proposed adaptive execution strategies to reduce I/O latencies. Like the work described here, these systems represent plans as dataflow graphs and pipeline data between operators. The major difference between existing network query engines and the work described here is in terms plan expressivity. While network query engine research has proposed new operators related to XML processing and adaptive execution, they do not support operators that facilitate monitoring. These systems do not support conditional execution and it is not possible to loop through query results spread across multiple Web pages. In contrast, the plan language here supports conditional

execution, allows plans themselves to be operators, and supports recursion as means for looping during execution.

General plan executors. It is also useful to compare the work here to existing and more general plan execution systems. These systems have proposed highly concurrent execution models similar in spirit to dataflow machines. For example, RAPS (Firby 1994) described execution as a set of concurrent processes while PRS-Lite (Myers 1996) supported concurrent task execution as well as more complex synchronization and control flow. Both projects focused on specifying an event-driven mechanism for the parallel execution of partially-ordered plans – similar to execution of a dataflow graph. The work described here differs from these more generic architectures by focusing, like network query engines, specifically on plans that not only require the enablement of operators, but the routing of information between them. as well. Thus, our work is more closer in spirit to the unified approach of (Williamson et al. 1996), yet it extends that work by specifying an actual plan language, adding support for recursion and subplan execution, and by proposing a dataflow execution architecture.

Conclusion and Future Work

In this paper, we have described an information gathering plan language that promotes better expressivity while retaining the efficiency of traditional plan representation. Support for subplans and recursive execution allow plans to loop through query results that are spread across multiple Web pages. Operators that are extensible and are better integrated with the external world facilitate plans that are capable of monitoring an integrated set of remote sources for an extended period of time. Though expressive, the plan language is dataflow in terms of representation and its operators support the pipelining of data during execution. Thus, such plans can be efficiently executed.

We are currently investigating a method for speculative execution for information gathering plans (Barish & Knoblock 2002) that uses machine learning techniques to analyze data occurring early during execution so that predictions can be made about data that will be needed later in execution. The result is a new form of dynamic execution parallelism that can lead to significant speedups. We are also currently working on an Agent Wizard, which allows the user to define agents for monitoring tasks simply by answering a set of questions about the task. The Wizard will work similar to the Microsoft Excel Chart Wizard, which builds sophisticated charts by asking the user a set of simple questions. The Wizard will generate information gathering plans using the language described in this paper and schedule them for periodic execution.

Acknowledgements

The research here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197, F30602-00-1-0504, F30602-98-2-0109, in part by the Air Force Office of Scientific Research under grant number F49620-01-1-0053, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, cooperative agreement number EEC-9529152. The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Barish, Greg; Chen, Yi-Shin; Knoblock, Craig A.; Minton, Steven; and Shahabi, Cyrus. The TheaterLoc Virtual Application. *Innovative Applications in Artificial Intelligence (IAAI)*. 2000

Barish, Greg and Knoblock, Craig A. Speculative Execution for Information Gathering Plans. To appear, *AIPS-2002*.

Dennis, Jack B. First version of a data-flow procedure language, *Lecture Notes in Computer Science 19, pp362-376*. 1974.

Firby, R.J. Task Networks for Controlling Continuous Processes. *AIPS-1994*.

Friedman, Marc and Weld, Daniel S. Efficient execution of information gathering plans. *IJCAI-1997*.

Genesereth, Michael R.; Keller, Arthur M.; and Duschka, Oliver M. Infomaster: An information integration system. *SIGMOD-97*.

Hellerstein, Joseph M.; Franklin, Michael J.; Chandrasekaran, Sirish; Deshpande, Amol; Hildrum, Kris; Madden, Sam; Raman, Vijayshankar; and Shah, Mehul A. Adaptive query processing: technology in evolution. *IEEE Data Eng Bulletin 23(2)*. 2000.

Ives, Zachary G.; Florescu, Daniela; Friedman, Marc; Levy, Alon Y.; and Weld, Daniel S. An adaptive query execution system for data integration. *SIGMOD-1999*.

Knoblock, Craig A.; Minton, Steven; Ambite, Jose Luis ; Ashish, Naveen; Muslea, Ion; Philpot, Andrew G.; and Tejada, Sheila. The Ariadne Approach to Web-based Information Integration. *International Journal on Cooperative Information Systems (IJCIS) Special Issue on Intelligent Information Agents: Theory and Applications. Vol 10 (1-2): 145-169*. 2001.

Myers, Karen. A procedural knowledge approach to task-level control. *AIPS-1996*.

Naughton, Jeffrey F.; DeWitt, David J.; Maier, David.; et al. The Niagara Internet query system. *IEEE Data Engineering Bulletin, 24(2)*. 2001

Papadopoulos, Gregory M. and Traub, Kenneth R. Multithreading: A revisionist view of dataflow architectures. In *Proc of the 18th Intl Symposium on Computer Architecture*. 1991

Williamson, Mike; Decker, Keith; and Sycara, Katia. Unified Information and Control Flow in Hierarchical Task Networks. *AAAI Workshop: Theories of Action, Planning, and Ctrl*. 1996.

Planning with Complex Actions

Sheila McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
sam@ksl.stanford.edu

Ronald Fadel
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
rfadel@ksl.stanford.edu

Abstract

In this paper we address the problem of planning with complex actions. We are motivated by the problem of automated Web service composition, in which planning *must* be performed using predefined complex actions or services as the building blocks of a plan. Planning with complex actions is also compelling in primitive action planning domains because it enables the exploitation of reusable subplans, potentially improving the efficiency of planning. This paper provides a formal, semantically-justified account of how to plan with complex actions using operator-based planning techniques. A key contribution of this work is the definition, characterization, and computation of preconditions and conditional effects for complex actions. While we use the situation calculus and Golog to formalize the task and our solution, the results in this paper are directly applicable to most action theories and planning systems. In particular, we have developed a PDDL-equivalent compiler that computes the preconditions and effects of complex actions, thus enabling wide-spread use of these results. Finally we provide an approach to planning that enables us to exploit deductive plan synthesis or alternatively ADL planners to plan with complex actions. Our approach to complex-action planning is sound and complete relative to the corresponding primitive action domain.

1 Introduction

Given a description of an initial state, a goal state, and a set of actions, the planning task is to generate a sequence of actions that, when performed starting in the initial state, will terminate in a goal state. Typically, actions are primitive and are described in terms of their precondition, and (conditional) effects. Our interest is in planning with complex actions as the building blocks for a plan. Complex actions are actions composed of primitive actions using typical programming language constructs. E.g.,complex actions move(obj,orig,dest) and goToAirpt(loc) are defined as:

move(obj,orig,dest) $\doteq$[1] pickup(obj,orig);putdown(obj,dest)
goToAirpt(loc) $\doteq$ **if** loc=Univ **then** shuttle(Univ,PA);
train(PA,MB);shuttle(MB,SFO) **else** taxi(loc,SFO)

Our primary motivation for investigating complex action planning is to automate *Web service composition* (e.g., [13]). Web services are self-contained Web-accessible computer programs, such as the airline ticket service at www.ual.com, or the weather service at www.weather.com. These services can be conceived as complex actions. Consider ual.com's buyAirTicket($\vec{x}$) service. This service can be described as the complex action locateFlight($\vec{x}$); **if** Available($\vec{x}$) $\wedge$ OKPrice($\vec{x}$) **then** buyAirTicket($\vec{x}$);...[2]. The task of automated Web service composition is to automatically sequence together Web services such as buyAirTicket($\vec{x}$) or getWeather($\vec{y}$) into a composition that achieves some user-defined objectives. The task of automated Web service composition is, by necessity, a problem of planning with complex actions. But how do we represent these complex actions (Web services) and how do we plan with them?

What makes planning with complex actions difficult is that the traditional characterization of actions as operators with preconditions and effects does not apply, making operator-based planning techniques such as Blackbox, FF, GraphPlan, BDDPlan, etc., inapplicable, at least at face value. In this paper we provide a formal, semantically-justified account of how to characterize, represent and precompile the preconditions and effects of complex actions, such as buyAirTicket($\vec{x}$), under a frame assumption [16]. This enables us to treat complex actions such as buyAirTicket($\vec{x}$) as planning operators and to apply standard planning tech-

[1] Denotes "defined as."
[2] Example is simplified for illustration purposes.

niques to planning with complex actions. Planning results in a plan in terms of complex actions from which a plan in terms of primitive actions is easily expanded, if desired[3].

A secondary motivation for this work is to improve the efficiency of planning by representing useful (conditional) plan segments as complex actions. As we show, our approach to planning with complex actions can dramatically improve the efficiency of plan generation by reducing the search space size and the length of a plan.

The idea of planning with some form of abstraction or aggregation is not new, and there has been a variety of work in this area including ABStrips (e.g., [17]), planning with macro-operators (e.g., [11] and [6]), and most notably HTN planning (e.g., [5]). Our work is fundamentally different from these approaches, and in particular from HTN planning, both in terms of i) the representation of complex actions (aka HTN *non-primitive tasks*), and ii) the method of planning. In this paper we precompile complex actions into planning operators described in terms of preconditions and effects that embody all possible evolutions of the complex action. In contrast, HTN planners do not use a declarative representation of the preconditions and effects of tasks. Rather, methods are associated with tasks, and tasks are pre-arranged into a network of compositions, without the full programming constructs we use to describe complex actions [18]. Further, HTN planners operate by searching for plans that accomplish task networks using task decomposition and conflict resolution. In contrast, having precompiled our complex actions, we can apply standard operator-based planning techniques to generate a plan, followed by plan expansion.

Our work is somewhat similar in methodology to [2], which proposes to encode planning constraints by compiling the constraints together with the original planning problem into a new unconstrained problem. The resultant planning problem can be solved using classical planning methods, and the resultant plan *decompiled* to provide a solution in the original problem domain. The general methodology of compilation and subsequent expansion is similar to what we propose. Nevertheless, the general problem is different. We are compiling complex actions into new plan operators. These complex actions represent Web services that we wish to reason with as black-box components. The constraints used in [2] are constraints upon the domain, and thus capture different types of planning information than our more procedural complex actions. Further the formal treatment and results are different.

We also contrast our work to the use of Golog (e.g., [12]) in planning. In this paper we use Golog as the formal language to describe complex actions, however the role these actions play in planning is very different. Golog complex actions are traditionally used to specify non-deterministic programs. In combination with deductive plan synthesis [7], a Golog program expands to a situation calculus formula which constrains the search space for a plan. This is similar to the role of domain-specific knowledge, as exemplified by systems such TALPlanner [4], BDDPlan [10] and ASP [18]. In all these systems, complex actions constrain the search space, but are not used as operators in plan construction.

The research presented in this paper is of both theoretical and practical significance. From a theoretical standpoint, we provide a semantically-justified means of characterizing the preconditions, effects and successor situations of complex actions under a frame assumption, that embodies all possible trajectories of a complex action. This enables us to not only use operator-based planning methods to plan with complex actions, but also to prove formal properties of our approach. In particular, we prove that our approach to planning is sound and complete relative to corresponding primitive action domains. From a practical perspective, analysis shows a significant increase in the efficiency of planning with complex actions, relative to primitive action planning. We illustrate potential speedup with some experiments on the briefcase domain, using the FF planner ([9]). Finally, this paper provides a principled approach to automating Web service composition, that has far-reaching application to automated component-based software composition

2 Background: Situation Calculus & Golog

We use the situation calculus and Golog to formalize the task and our solution. The expressive power and formal semantics of the situation calculus provide the theoretical foundations for our work, and for the later translation to PDDL.

Briefly, the situation calculus is a logical language for specifying and reasoning about dynamical systems [16]. In the situation calculus, the state of the world is expressed in terms of functions and relations (fluents) relativized to a particular situation s, e.g., $F(\vec{x}, s)$. A situation s is a history of the primitive actions, e.g., a, performed from an initial, distinguished situation S_0. The function $do(a, s)$ maps a situation and an action into a new situation. A situation calculus theory $\mathcal{D}$ comprises the following sets of axioms:

- domain independent foundational axioms, Σ.
- successor state axioms, $\mathcal{D}_{SS}$, one for every fluent F.
- action precondition axioms, $\mathcal{D}_{ap}$, one for every action a in the domain,which define $Poss(a, s)$.
- axioms describing the initial situation, $\mathcal{D}_{S_0}$.

[3] For many Web service applications, expansion is not relevant.

• unique names axioms for actions, $\mathcal{D}_{una}$.

Successor state axioms, originally proposed [15] to address the frame problem, are created by compiling effect axioms into axioms of this form[4]: $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$ where $\Phi_F(\vec{x}, a, s) = \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))$. (See [16, pg.28-35] for details.)

Example: In the interest of simplicity, we illustrate concepts in this paper in terms of an action theory with three actions $pickup(x), putdown(x)$ & $drop(x)$, and three fluents $holding(x), broken(x)$ & $hot(x)$. (1)-(3) comprise $\mathcal{D}_{ap}$, and (4)-(6) comprise $\mathcal{D}_{SS}$[5].

$$Poss(pickup(x), s) \equiv \neg holding(x, s) \quad (1)$$
$$Poss(drop(x), s) \equiv holding(x, s) \quad (2)$$
$$Poss(putdown(x), s) \equiv holding(x, s) \quad (3)$$
$$holding(x, do(a, s)) \equiv a = pickup(x) \vee holding(x, s) \wedge a \neq putdown(x) \wedge a \neq drop(x) \quad (4)$$
$$broken(x, do(a, s)) \equiv a = drop(x) \vee broken(x, s) \quad (5)$$
$$hot(x, do(a, s)) \equiv hot(x, s) \quad (6)$$

Golog (e.g., [12, 16, 3]) is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing extralogical constructs for assembling primitive situation calculus actions, into complex actions δ. [3] shows how these complex actions can be considered to be first-class objects in the language. $Do(\delta, s, s')$ is an abbreviation that macro-expands into a situation calculus formula, as defined inductively below. The formula says that it is possible to reach s' from s by executing a sequence of actions specified by δ [16].

Prim. action: $Do(a, s, s') \doteq Poss(a, s) \wedge s' = do(a[s], s)$
Test: $Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$
Seq.: $Do(\delta_1; \delta_2, s, s') \doteq \exists s''. Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$
Nondet. act.: $Do(\delta_1 \mid \delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$
Nondet. arg.: $Do((\pi x)\delta(x), s, s') \doteq \exists x. Do(\delta(x), s, s')$

The construct, **if** ϕ **then** δ_1 **else** δ_2 **endIf** is defined as $[\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$. The Golog language also includes nondeterministic iteration, δ^*, which executes δ zero or more times. The while-loop construct,**while** ϕ **do** δ **endWhile** is defined in terms of nondeterministic iteration as $[\phi? : \delta]^*; \neg\phi?$. For now, we exclude nondeterministic iteration, and while-loops, whose macro-expansions are second order, and which may be non-terminating. Instead, we define a bounded notion of while, **while** $_k(\phi)$ δ, which is guaranteed to terminate, and is commonly used in Web services. **while** $_k(\phi)$ δ executes like the original while-loop except that it loops at most k times, even if ϕ still holds after the k^{th} iteration. Formally, **while** $_k(\phi)$ δ corresponds to k conditional branchings as follow:

$$\textbf{while}\ _1(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ \delta\ \textbf{endIf}\ [6] \quad (7)$$
$$\textbf{while}\ _k(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ [\delta; \textbf{while}\ _{k-1}(\phi)\delta]\ \textbf{endIf} \quad (8)$$

[4]For space, we will only consider relational fluents here.

[5]Notation: formulae are universally quantified with maximum scope unless noted. Action arguments suppressed.

A deterministic version of the choice construct (π') is defined in a longer paper. These constructs are used to specify complex actions such as buyAirTicket($\vec{x}$) or goToAirpt(loc). Traditional usage of Golog is to apply deductive plan synthesis to find a sequence of actions $\vec{a} = [a_1, \ldots, a_n]$ that realizes a Golog program, δ relative to domain theory, $\mathcal{D}$. I.e., $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\vec{a}, S_0)$, where $do(\vec{a}, S_0)$ is an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

3 Problem: Planning with Complex Actions

Given a set of primitive actions, $\mathcal{A}$ together with an associated set of complex actions, $\Delta_{\mathcal{A}}$, our objective is to use an operator-based planner to compose complex and primitive actions to achieve some goal. To do this, we must characterize the preconditions, effects, and the situation resulting from performing a complex action.

3.1 Preconditions, Effects, Resulting Situations

For analysis, our actions $\mathcal{A}$ are axiomatized in a situation calculus action theory $\mathcal{D}$, and our complex actions $\Delta_{\mathcal{A}}$ are described in Golog. For now, we restrict our focus to terminating complex actions described in Section 2.

Resulting Situation: We wish to characterize the situation resulting from performing the complex action δ. Observe that many complex actions are nondeterministic. They may have several different executions, each terminating in a different situation. As such, we can't define a function analogous to $do(a, s)$. Instead, we introduce the abbreviation $do_{ca}(\delta, s)$ to denote a situation resulting from performing complex action δ in s. $do_{ca}(\delta, s)$ ranges over the set of *executable* situations and corresponds to a so-called *ghost situation* [16, pg.52-53], when δ is not physically realizable. The interpretation of $do_{ca}(\delta, s)$ is constrained by the following axiom, which is added to $\mathcal{D}$ producing theory $\mathcal{D}_{ca}$.

For all complex actions δ and situations s:

$$Do(\delta, s, do_{ca}(\delta, s)) \vee (\neg\exists s''. Do(\delta, s, s'') \wedge \neg executable(do_{ca}(\delta, s))) \quad (9)$$

where $executable(s)$ denotes a situation, all of whose actions in the situation action history are $Poss$-ible [16]. I.e., $executable(s) \doteq (\forall a, s^*). do(a, s^*) \sqsubseteq [7] s \rightarrow Poss(a, s^*)$ It follows that:

$$\mathcal{D}_{ca} \models \forall s. executable(s) \wedge Do(\delta, s, do_{ca}(\delta, s)) \rightarrow executable(do_{ca}(\delta, s)) \quad (10)$$

[6]**if -then -endIf** is the obvious variant of **if -then -else -endIf**.

[7]The order relation on situations in the situation tree [16].

Preconditions: $Poss_{ca}(\delta, s)$ denotes the preconditions of complex action δ. Intuitively, the preconditions of a complex action are the preconditions of all the actions that make up the execution of δ. E.g., for $a_1; a_2$,

$$Poss_{ca}(a_1; a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, do(a_1, s)).$$

This is captured tidily in the inductive definition of Do. We define the precondition of complex action δ , $Poss_{ca}(\delta, s)$ as:

$$Poss_{ca}(\delta, s) \equiv \Pi^*_\delta(s) \tag{11}$$

where $\Pi^*_\delta(s) \equiv \exists s'.Do(\delta, s, s')$. These are *intermediate* action precondition axioms.

Proposition 1 (Properties of $Poss_{ca}(\delta, s)$)
These axioms follow from $\mathcal{D}_{ca} \cup (11)$.

$$\begin{aligned} Poss_{ca}(\delta, s) &\equiv Do(\delta, s, do_{ca}(\delta, s)) \\ executable(s) \wedge Poss_{ca}(\delta, s) &\equiv executable(do_{ca}(\delta, s)) \end{aligned}$$

Effects: Intuitively the effects of a complex action are the effects of each action in the execution of δ, modulo the effects of subsequent actions. We assume that fluents whose truth value is not changed by an action, persist. $F(\vec{x}, do_{ca}(\delta, s))$ denotes that fluent F is true in the situation resulting from performing complex action δ in s. We capture the effects of complex actions as successor state axioms. Since all but trivial complex actions involve multiple intermediate situations, strictly speaking, we cannot define successor state axioms for complex actions. Rather, we define the notion of a pseudo-successor state axiom. Here we define *intermediate* pseudo-successor state axioms, making them "Markovian" in the section to follow via regression.

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi^*_F(\vec{x}, \delta, s)], \text{ where,}$$
$$\Phi^*_F(\vec{x}, \delta, s) \equiv \exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge s' = do_{ca}(\delta, s). \tag{12}$$

We need the $s' = do_{ca}(\delta, s)$ since some complex actions are nondeterministic. This enables us to identify the particular sequence of actions in the instantiation of the complex action that leads to the truth/falsity of the fluent F.

3.2 Pseudo-Markovian Complex Actions

In order to plan with complex actions as operators, we must make our characterization *pseudo-markovian*. That is, we wish to characterize the preconditions strictly in terms of the situation in which the complex action execution is initiated, and the effects, strictly in terms of the initiating and terminating situations of the complex action. To do so we appeal to regression rewriting [19], regressing over the successor state axioms for the *primitive actions* in our domain theory $\mathcal{D}$. Unfortunately, the formulae over which we need to regress are not, by definition, regressable using $\mathcal{R}$ [16, pg.62], since we are not regressing to S_0, and since the macro-expansion of $Do(\delta, s, s')$ does not yield a nested representation of situations. Since regression is a syntactic rewriting, this is problematic. We define a suitable (small) variant of Reiter's regression operator, $\mathcal{R}^s$, that first rewrites the macro-expansion of Do so that situations are expressed as nested do's, and that enables regression to an arbitrary situation s, rather than to S_0. We define the preconditions and effects of δ in terms of a set of action precondition axioms, $\mathcal{D}_{caap}$, of the form of (13) and a set of pseudo-successor state axioms, $\mathcal{D}_{caSS}$, of the form of (15).

Preconditions:
Action Precondition Axioms, $\mathcal{D}_{caap}$, one for every $\delta \in \Delta$:

$$Poss_{ca}(\delta, s) \equiv \Pi_\delta(s) \tag{13}$$

where $\Pi_\delta(s) \equiv \mathcal{R}^s[\Pi^*_\delta(s)]$ from (11), i.e. $\mathcal{R}^s[\exists s'.Do(\delta, s, s')]$.

Example (continued): Consider the complex action $pickup(x)$; **if** $hot(x)$ **then** $drop(x)$ **else** $putdown(x)$ **endIf**, which we denote as δ_1 for parsimony. Its action precondition axiom is defined as follows.

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s''))] \end{aligned} \tag{14}$$

Following our regression, $Poss_{ca}(\delta_1, s) \equiv \neg holding(x, s)$.

Successor State Axioms: Observe that while a situation calculus axiomatization has one successor state axiom for every fluent, we currently define one pseudo-successor state axiom for every fluent-complex action pair.

Pseudo-Successor State Axioms, $\mathcal{D}_{caSS}$, one for every fluent-complex action pair:

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi_F(\vec{x}, \delta, s)] \tag{15}$$

where $\Phi_F(\vec{x}, \delta, s) \equiv \mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)]$, $\mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)] \equiv \mathcal{R}^s[\exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge do_{ca}(\delta, s) = s']$

Example (continued): The pseudo-successor state axiom for fluent $broken(x, do_{ca}(\delta_1, s))$ is:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow [broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s''))) \wedge \\ &\quad broken(x, s') \wedge do_{ca}(\delta_1, s) = s'] \end{aligned} \tag{16}$$

Applying our $\mathcal{R}^s$ regression operator, (16) becomes:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow (\ broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \neg holding(x, s) \wedge [hot(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(drop(x), do(pickup(x), s)) \\ &\quad \vee\ \neg hot(x, s) \wedge broken(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(putdown(x), do(pickup(x), s))] \ ) \end{aligned}$$

Though the computation looks complex, regression rewriting is a powerful tool and the final pseudo-successor state axiom is simple. Observe that a pseudo-successor state axiom not only defines the conditions under which fluent F is true after performing complex action δ, but it also defines the action trajectory upon which the truth of F is predicated. This is most valuable with nondeterministic actions.

Note that when the definition of $Poss_{ca}(\delta, s)$ and the intermediate pseudo-successor state axiom, ((11) and (12), respectively) are conjoined to $\mathcal{D}_{ca}$, they entail the complex action precondition axioms and the complex action pseudo-successor state axioms.

Proposition 2 $\mathcal{D}_{ca} \cup (11) \cup (12) \models \mathcal{D}_{caap} \cup \mathcal{D}_{caSS}$

Effect axioms: While we have encoded the effects of our complex actions, together with a solution to the frame problem in terms of pseudo-successor state axioms, many planners use effect axioms, rather than successor state axioms, solving the frame problem in the procedural code of their planner, rather than representationally. Hence, for completion we define effect axioms for complex actions, $\mathcal{D}_{caef}$.

Effect Axioms $\mathcal{D}_{caef}$, up to one positive effect axiom and one negative effect axiom for every fluent - complex action pair, where the execution of δ can potentially change the truth value of fluent $F \in \mathcal{F}$:

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^+(\vec{x}, s) \rightarrow F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (17)$$

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^-(\vec{x}, s) \rightarrow \neg F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (18)$$

Proposition 3 (Effect Axioms Entailment)

$$\mathcal{D} \cup \mathcal{D}_{caap} \cup \mathcal{D}_{caSS} \models \mathcal{D}_{caef}$$

I.e., the positive and negative complex action effect axioms are entailed by the pseudo-successor state axioms. Hence, we can easily extract effect axioms for complex actions from our pseudo-successor state axioms.

In this section we have provided a representation of the preconditions, successor state axioms and effects of complex actions under a frame assumption. They are characterized in terms of $\mathcal{D}_{caap}$, and $\mathcal{D}_{caSS}$, and follow from the semantically-justified account of actions in the situation calculus. In the section to follow, we show how these representations of complex actions lead to a simple approach to planning with complex actions.

4 Complex Actions Planning

Given our operator-based characterization of complex actions in terms of their preconditions and effects, we turn to the problem of operator-based planning with these complex actions. For now, we restrict our consideration to the subset of complex actions that are deterministic, I.e., primitive actions a, sequences $\delta_1; \delta_2$, conditional **if** ϕ **then** δ_1 **else** δ_2 **endIf**, and **while** $_k(\phi)$ δ, plus others described in a longer paper.

Following the problem statement in Section 3, our approach is to take as input [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] – an action theory $\mathcal{T}_\mathcal{A}$ and a set of complex actions $\Delta_\mathcal{A}$, both defined in terms of actions in $\mathcal{A}$. Following the results in the previous section, we **COMPILE** [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] into a new theory $\mathcal{T}_{\mathcal{A}'}$, in terms of actions $\mathcal{A}'$ (generally $\mathcal{A} \subseteq \mathcal{A}'$), where each complex action in $\Delta_\mathcal{A}$ corresponds to a new primitive action in $\mathcal{A}'$. Next, **PLAN**ning is performed in $\mathcal{T}_{\mathcal{A}'}$ to produce a plan in terms of $\mathcal{A}'$. To extract a plan in terms of the primitive actions, we **REWRITE** the theory, replacing primitive actions from $\mathcal{A}'$ by their corresponding complex actions, $\Delta_\mathcal{A}$. Finally, using $\mathcal{T}_\mathcal{A}$, the resulting sequence of primitive actions is **EXPAND**ed from the plan in $\mathcal{A}'$ into a plan in terms of $\mathcal{A}$.

Next, we show how this approach is realized, first using the situation calculus and deductive plan synthesis, and then using an arbitrary operator-based planning system that allows conditional effects of actions in PDDL.

4.1 Deductive Plan Synthesis and Expansion

The following is the theory with primitive actions $\mathcal{T}_\mathcal{A}$.

$\mathcal{T}_\mathcal{A}$= $\Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{SS} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$.

(1) COMPILE[$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] $\rightarrow$ $\mathcal{T}_{\mathcal{A}'}$:
- Define $\mathcal{D}_{caap}$ and $\mathcal{D}_{caSS}$ as described in Section 3.2.
- $\mathcal{D}'_{ap} \leftarrow \mathcal{D}_{caap} \cup \mathcal{D}_{ap}$. $\mathcal{D}'_{SS} \leftarrow \mathcal{D}_{caSS}$. $\mathcal{A}' \leftarrow \mathcal{A}$.
- $\forall \delta_i \in \Delta_A$: Create a primitive action a'_i. Substitute "a'_i" for "δ_i" in $\mathcal{D}'_{ap}$& $\mathcal{D}'_{SS}$. $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{a'_i\}$.
- $\mathcal{D}'_{SS} \leftarrow$ MERGE($\mathcal{D}'_{SS}, \mathcal{D}_{SS}$). Update $\mathcal{D}_{una}$ to $\mathcal{D}'_{una}$.

COMPILE produces a situation calculus theory in actions $\mathcal{A}'$, comprising all the original primitive actions $\mathcal{A}$ plus new primitive actions corresponding to each complex action in $\Delta_\mathcal{A}$. $\mathcal{T}_{\mathcal{A}'} = \Sigma \cup \mathcal{D}'_{ap} \cup \mathcal{D}'_{SS} \cup \mathcal{D}'_{una} \cup \mathcal{D}_{S_0}$

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal] $\rightarrow$ plan[$\mathcal{A}'$]: Given a goal formula, $Goal(s)$ in the language of $\mathcal{T}_\mathcal{A}$, planning can be achieved via deductive plan synthesis in $\mathcal{T}_{\mathcal{A}'}$. Following [7, 16], $\mathcal{T}_{\mathcal{A}'} \vdash \exists s.Goal(s)$. From the binding of s, we can read off a plan $[a'_1 \ldots, a'_n], a'_i \in \mathcal{A}'$, a plan in $\mathcal{A}'$. [16] describes a variety of situation calculus planners implemented in Prolog.

(3) REWRITE[plan[$\mathcal{A}'$]] $\rightarrow$ plan[$\mathcal{A}$,$\Delta_\mathcal{A}$]: Rewrite the plan $[a'_1 \ldots, a'_n], a'_i \in \mathcal{A}'$ as a plan $[\alpha_1, \ldots, \alpha_n]$ in ($\mathcal{A}$,$\Delta_\mathcal{A}$), where $\alpha_i = a_i$, for all $a'_i \in \mathcal{A}$, otherwise α_i equals the corresponding δ_i from the compilation in Step (1).

(4) EXPAND[plan[$\mathcal{A}$,$\Delta_\mathcal{A}$],$\mathcal{T}_\mathcal{A}$] $\rightarrow$ plan[$\mathcal{A}$]: Use our same deductive machinery to extract a final plan in $\mathcal{A}$ from our plan in ($\mathcal{A}$,$\Delta_\mathcal{A}$), by expanding the complex actions in $[\alpha_1, \ldots, \alpha_n]$. We do so by trivially rewriting our plan as a

sequence of complex actions in Golog $\delta_G = \alpha_1; \alpha_2; \ldots; \alpha_n$. A Golog interpreter, written in Prolog will return a binding for situation s' where $\mathcal{T}_\mathcal{A} \vdash (\exists s').Do(\delta_G, s, s') \wedge Goal(s')$. From the situation s' we can read off a plan $[a_1 \ldots, a_m], a_j \in \mathcal{A}$.

Note that every plan our approach finds is also a plan in the original primitive action theory, and vice-versa.

Theorem 1 *$\mathcal{T}_{\mathcal{A}'}$ and $\mathcal{T}_\mathcal{A}$ are defined as in Section 4.1. Let $Goal(s)$ be a formula uniform in s such that $Goal(s) \in \mathcal{L}(\mathcal{T}_\mathcal{A}) \cap \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, the intersection of the languages of $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_{\mathcal{A}'}$ respectively. For all ground situations σ' of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{T}_{\mathcal{A}'} \models executable(\sigma') \wedge Goal(\sigma')$ iff there exists a ground situation σ of $\mathcal{T}_\mathcal{A}$ such that $\mathcal{T}_\mathcal{A} \models executable(\sigma) \wedge Goal(\sigma)$ and EXPAND[REWRITE[seq(σ')],$\Delta_\mathcal{A}$] = seq(σ), where $seq(do(\vec{a}, s)) = \vec{a}$.*

Proof Sketch: First, by construction of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{L}(\mathcal{T}_\mathcal{A}) \subset \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, and, for any action a in $\mathcal{T}_\mathcal{A}$, $\mathcal{T}_{\mathcal{A}'}$ contains the successor state and action precondition axioms of a in $\mathcal{T}_\mathcal{A}$. It follows that, for any term s which denotes a situation in the language of $\mathcal{T}_\mathcal{A}$, $\mathcal{T}_\mathcal{A} \models Goal(s) \wedge executable(s)$ iff $\mathcal{T}_{\mathcal{A}'} \models Goal(s) \wedge executable(s)$. Second, since in any situation s, the expansion of an executable complex action is also executable and has the same effects, for any executable complex plan $[a'_1, a'_2, ..., a'_n]$ in $\mathcal{T}_{\mathcal{A}'}$, EXPAND[REWRITE$[a'_1, a'_2, ..., a'_n]$,$\Delta_\mathcal{A}$]$= [a_1, a_2, ..., a_m]$ is an executable plan in $\mathcal{T}_{\mathcal{A}'}$, and $do([a'_1, a'_2, ..., a'_n], S_0)$ and $do([a_1, a_2, ..., a_m], S_0)$ are the same *states* in $\mathcal{T}_{\mathcal{A}'}$(i.e. fluents has the same truth value in both situations). Finally, by definition of the REWRITE and EXPAND steps, $a_1, a_2, ..., a_m$ are actions in $\mathcal{T}_\mathcal{A}$. It follows that $do([a_1, a_2, ..., a_m], S_0)$ is a term in the language of $\mathcal{T}_\mathcal{A}$ which denotes a situation, and thus $\mathcal{T}_\mathcal{A} \models Goal(do([a_1 ... a_m], S_0)) \wedge executable(do([a_1 ... a_m], S_0))$ if and only if $\mathcal{T}_{\mathcal{A}'} \models Goal(do([a'_1 ... a'_n], S_0)) \wedge executable(do([a'_1 ... a'_n], S_0))$.

Planning in $\mathcal{T}_{\mathcal{A}'}$ is sound and complete with respect to planning in $\mathcal{T}_\mathcal{A}$. Thus our approach to complex action planning via transformation of the theory is well-founded.

4.2 Exploiting Existing Operator-Based Planners

Our approach is not limited to planners realized in the situation calculus. Most popular planners don't use a successor state axioms representation of the effects of actions. E.g., all of the planners that participate in the AIPS Planning competition use PDDL as an initial specification of the action theory. In this section we show how to exploit an arbitrary operator-based planner that accepts PDDL planning domains with conditional effects [14], in order to plan with complex actions.

(1) COMPILE[$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$]: Rather than employing successor state axioms, PDDL describes the effects of actions in terms of (conditional) effects without a solution to the frame problem. Section 3.2 provides a semantic justification for an intuitive algorithm that compiles a PDDL representation of the preconditions and effects of actions in $\mathcal{T}_\mathcal{A}$, together with complex actions $\Delta_\mathcal{A}$ into a new PDDL representation of preconditions and effects in $\mathcal{T}_{\mathcal{A}'}$, without going through the intermediate stage of creating successor state axioms. (We have such an algorithm, but space precluded its inclusion in this paper.) Intuitively, the effects of a complex action are the effects of each action in the execution of δ, modulo subsequent effects.

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal]: Given a compiled PDDL representation $\mathcal{T}_{\mathcal{A}'}$, we can generate a plan with any planner that accepts PDDL with conditional effects. (We used FF [9].)

(3) REWRITE & (4) EXPAND: We can use STEP (3)-(4) from Section 4.1. Alternatively, we can write a (fairly straightforward) algorithm to expand the final plan in $\mathcal{A}'$. For maximal efficiency, we would cache the conditions that uniquely determine the expansion of each complex action in a situation.

5 Elaborations on Complex Action Planning

In this section we examine elaborations on complex action planning. In particular, we examine the conditions under-which adding complex actions to a theory causes other actions to be redundant and thus removable. Removing redundant actions is desirable because it reduces the plan search space. In an extended version of this paper, we discuss concurrency in complex action planning.

5.1 Removing Weaker Actions

When a complex action δ_1 is compiled into a primitive action theory as a new primitive action a_1, another primitive action, a_2 may become redundant in the sense that in any situation s, if a_2 is possible, a_1 is also possible and has exactly the same effects as a_2. More generally, we define the notion that primitive action a_1 is *stronger* than primitive action a_2, $a_1 \succeq a_2$ (and conversely that a_2 is weaker than a_1) as follows:

$$a_1 \succeq a_2 \leftrightarrow [\, Poss(a_2, s) \rightarrow Poss(a_1, s) \wedge SS(do(a_1, s), do(a_2, s))\,] \quad (19)$$

where $SS(s, s')$ is an abbreviation for the first-order formulae that is true iff situations s and s' have the same state. The relation $\succeq$ is a preorder (it is reflexive and transitive). It follows that for any situation calculus theory T and goal formula $G(s)$, $T \models \exists s.G(s)$ iff $T' \models \exists s.G(s)$, where T' is T with all weaker actions removed.

Note that removal of weaker primitive actions may result in removal of the optimal plan. In particular, if $a_1 \succeq a_2$ and a_1

is a compiled complex action that can expand into multiple primitive actions, then by removing a_2, we may lose the optimal plan with respect to the number of primitive actions in our initial domain. Also note that the notion of stronger actions does not capture all the conditions underwhich an action is redundant. In particular, a_2 may be conditionally redundant, or it might be redundant relative to a_1 in some situation, and redundant relative to a_3 in others.

Example: Let a_1 and a_2 be primitive actions in $\mathcal{T}_\mathcal{A}$, let $a_2$achieve the preconditions for a_1, and let $Poss(a_1)$ be the situation suppressed expression [16, pg.112] for $Poss(a_1, s)$. Define complex action δ_3 as **if** $\neg Poss(a_1)$ **then** a_2 **endIf** ; a_1. If we compile a_1, a_2 and δ_3 in $\mathcal{T}_\mathcal{A}$ into primitive actions a_1', a_2' and a_3' in $\mathcal{T}_{\mathcal{A}'}$, following Section 4.1, then it follows that $a_3' \succeq a_1'$.

5.2 Irrelevant Actions with Respect to a Goal

Let $G(s)$ be a goal predicate that is true iff s satisifies the goal formula. If the direct effect of an action a can never make $G(s)$ true, and if a cannot directly achieve the preconditions of any of the actions, then a is irrelevant with respect to goal predicate $G(s)$ and can be removed. Formally, given a primitive action a_1 and goal predicate G, we consider a_1 as G-irrelevant in $\mathcal{T}$ if and only if, for a_2 ranging over all actions in $\mathcal{T}$ except a_1, it follows from T that:

$$executable(do(a_1, s)) \leftrightarrow [\,(G(do(a_1, s)) \rightarrow G(s)) \wedge (Poss(a_2, do(a_1, s)) \rightarrow Poss(a_2, s))\,] \quad (20)$$

If a_1 is G-irrelevant, then a_1 will not be in any optimal successful plan to achieve G, and can be removed from the set of actions when planning to achieve G.

Example (continued): In the previous example, we showed that a_1' could be removed from $\mathcal{T}_{\mathcal{A}'}$. It then follows that, a_2' will never be needed to make a_1' $Poss$-ible. If a_2' can never directly achieve the preconditions for any other actions in the theory, then for all goal predicate $G(s)$ which are not among the effects of a_2', a_2' is G-irrelevant.

6 Web Service Composition

The primary motivation for our work was to be able to compose Web services using operator-based planning techniques. With the results of Sections 3 and 4, we have addressed a fundamental barrier to automated Web service composition. Service providers such as Amazon or United Airlines will describe their Web services (Web-accessible programs) as processes. In our vision of the Semantic Web, this will be done using the DAML+OIL Web service ontology, DAML-S [1], whose process description constructs are similar to Golog. (The relationship between DAML-S and the situation calculus is well-defined and has been used to define the semantics of DAML-S.) To produce black-box or compiled representations of Web services for automated composition, we can exploit the compilation techniques described in this paper. Using them, we compile process-oriented program descriptions of services into black-box component descriptions. Once Web services process descriptions have been compiled, we can use standard operator-based planning techniques to automatically compose Web services.

7 Efficiency of Complex Action Planning

A secondary motivation for our work was to potentially improve the efficiency of planning (e.g., [11, 8]) through our operator-based approach to complex action planning. We restrict our attention to complex action planning with the deterministic actions listed in Sect. 4. Compiling a complex action δ is polynomial in the number of primitive action occurrences in its definition. Note that this step can be performed offline, and is amortized over multiple planning runs. The expansion step is itself linear in the length of the plan, and in the number of branchings in the complex action definition. Of no surprise, plan generation dominates the computational cost. In particular: i) complex action operators tend to have more complex preconditions and effects than primitive actions, and ii) the size of the search space will be changed. However, i) causes only a linear slowdown and thus, the crucial point is ii).

Although the following analysis can be adapted to almost any classical planner, for simplicity of the argument, let's consider a breadth-first search forward planner. Given n ground actions, if the shortest successful plan is of length l, the size of the primitive action domain search space is $O(n^l)$. Adding d ground complex actions yields $n' = n+d$ ground actions in the compiled domain. [8] claims that adding actions that correspond to compositions of other actions will yield a larger search space. We identify conditions under which this is false.

Suppose the use of complex actions results in successful plans of length l/k, $k \geq 1$. One way to ensure this is by requiring complex actions to correspond to non-overlapping subplans in the shortest plan. In this case, the number of states visited to find a plan of length l/k will be $O(n'^{l/k})$ and the difference between the search spaces will be $O(n^l - n'^{l/k})$. If $(n^k - n')$ has a strictly positive lower bound for any problem, the new search space will be exponentially smaller than the old, as problem complexity increases. Informally, complex action planning reduces the planning search space when the complex actions significantly shorten the smallest successful plan relative to the increase they cause in the breadth of the search space.

Finally, in addition to this potential search space reduction,

some complex actions remove conflicts between the goals. This results in less backtrackings and enables the use of very efficient hill-climbing techniques (e.g., [9]).

8 Experimental Results

The techniques of Section 4.2 were implemented using the operator-based breadth-first search forward planner, FF [9]. FF supports conditional effects and uses its "enforced hill-climbing" whenever possible. We tested our approach on the ADL BRIEFCASE domain (BCD) [8]. This domain moves objects between locations using a briefcase. Three experiments were run on multiple instances of the problem, varying numbers of locations (#l) and portables (#p)[9].

The first experiment was simply BCD alone. Note that FF struggles as we increase (#p) and (#l). The next experiments involved the addition of the complex action Move-object. Move-object MO(locInit, locObj, Obj, locFinal) takes as input the location of the briefcase locInit, an object Obj, its location locObj, and a destination locFinal. It moves the briefcase to locObj, puts the object in the briefcase, moves the briefcase to locFinal, and removes Obj. MO is not a subplan of the shortest plan, so we would not necessarily expect it to do well. Further, it does *not* reduce the search space as it does not shorten the successful plan enough to compensate for the number of ground complex actions ($(n^k - n')$ is not positive). Nevertheless, adding the complex action move-object (BCD+MO) turns on FF's hill-climbing techniques, which reduce the number of nodes considered.

Finally, we designed a complex action that does correspond to subplans of the shortest successful plan and thus reduces the search space. The complex action LOC(loc-bc, loc), takes as input the location of the briefcase loc-bc, moves the briefcase to location loc, removes all the objects in the briefcase that should be at loc, and puts all other objects at loc in the briefcase. The goal defines where an object should be. To encode this complex action in PDDL, the action must know the goal at the time it executes. Hence, we added a persisting predicate $ShouldBeAt(Obj, Loc)$ to the domain. This predicate always has the same values as the $At(Obj, Loc)$ predicate in the goal statement. This complex action reduces the search space ($k \simeq \#p/\#l, d \simeq \#l$) and allows the use of hill-climbing techniques. Of no surprise, (BCD+LOC) presented the best results of all three experiment runs.

	#l:5, #p:20	#l:6, #p:30	#l:7, #p:42
BCD	5549 (1.39)	201006 (2261)	? ($>$ 40h)
BCD+MO	859 (11.83)	2345 (201.47)	5195 (2211)
BCD+LOC	75 (.08)	139 (.27)	260 (.85)

Number of nodes (and time of run in seconds).

[8] http://rakaposhi.eas.asu.edu/domain-syntax.html

[9] Experiments run on Sun Sparc v9, 2×750GHz, 4GB of mem.

9 Discussion and Summary

The work in this paper was motivated by the problem of automating Web service composition. In particular, we posed the problem of composing Web services such as UAL's buyAirTicket($\vec{x}$) or CNN's getWeather($\vec{y}$) in order to achieve a user-defined goal. These Web services are describable as simple programs, using typical programming language constructs. We conceived this task as the problem of planning with complex actions, with the restriction that the complex actions *had* to be the primitive building blocks of a plan. Consequently, we posed the problem of how to represent and plan with complex actions, using operator-based planning techniques. To this end, we embarked upon a theoretical analysis of the problem of how to represent complex actions as operators. The situation calculus provided the formal foundation for our work, enabling us to provide a formal definition of the preconditions, successor state axioms, and effects of complex actions under a frame assumption.

With this representational problem addressed we turned to the practical matter of how to plan. We proposed a method of planning that produced sound and complete plans relative to a corresponding primitive action domain. We showed how to use our results to plan via deductive plan synthesis as well as using an arbitrary operator-based planning system that accepts ADL as input.

We are currently incorporating these representation and compilation results into DAML-S [1], an AI-inspired markup language ontology for Web services. We're also incorporating the results into ongoing Web service composition work [13].

Finally, the second motivation for this work was to potentially improve either the efficiency of planning or the quality of the plans generated, by exploiting complex actions that capture some preferred subplans. We have shown how, in some domains, using relevant complex actions will result in a dramatic speedup of the planning process. We discussed the impact of our approach on the planning search space and illustrated predicted speedup with experiments.

Acknowledgements

We would like to thank Srini Narayanan for conversations related to this work. We gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DARPA Agent Markup Language (DAML) Program #F30602-00-2-0579-P00001.

References

[1] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Semantic markup for web services. In *Proc. International Semantic Web Working Symposium (SWWS)*, 2001.

[2] M. Baioletti, S. Marcugini, and A. Milani. Encoding planning constraints into partial order planning domains. In *Proc. 6th Conference on Knowledge Representation and Reasoning*, pages 608 – 616, 1998.

[3] G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[4] P. Doherty and J. Kvarnstrom. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[5] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, 1994.

[6] R.E. Fikes, P.E Hart, and N.J Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[7] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.

[8] Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In *Artificial Intelligence Planning Systems*, pages 150–158, 2000.

[9] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[10] S. Hölldobler and H.-P. Störr. BDD-based reasoning in the fluent calculus – first results. In *8th. Intl. Workshop on Non-Monotonic Reasoning (NMR'2000)*, 2000.

[11] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[12] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

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

[14] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR'89*, pages 324–332, 1989.

[15] R. Reiter. The frame problem in the situation calculus: A soundness and completeness result, with an application to database updates. In *Proceedings First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[16] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[17] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[18] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge – an answer set programming approach. In *6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Programming Approach Proceedings, pages 226–239, 2001.

[19] R. J. Waldinger. Achieving several goals simultaneously. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8: Machine Representations of Knowledge*, pages 94–136. Ellis Horwood, Chichester, 1977.

Towards Planning and Execution for Information Retrieval

Laurie S. Hiyakumoto and **Manuela M. Veloso**
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA 15213
{hyaku, mmv}@cs.cmu.edu

Abstract

The problem of information gathering has received considerable attention from the planning community in recent years. However, research in this area has generally assumed a user's information goal is perfectly represented by the query, and typically adopts a relational database model for representing query operations and information sources. In this paper, we present a planning and execution framework intended to address the more general information retrieval (IR) problem in which queries are only approximate representations of the user's true information goals, and complete models of the content of information sources are not available. Our approach reformulates traditional IR techniques such as synonym expansion and relevance-feedback as domain operators, explicitly modeling the uncertainty of action outcomes and of satisfying the user's information needs, and taking into account resource constraints such as time. A forward-chaining planning and execution algorithm guides action selection, execution and replanning decisions using a user-defined utility function. We have implemented this approach in the INSPIRE (INtegrated System for Planning and Information REtrieval) architecture.

Introduction

Traditional ad-hoc approaches to information retrieval (IR) are becoming increasingly inadequate for today's large, dynamic, heterogeneous document collections, the most obvious example of which is the World Wide Web. Current IR systems place most of the burden on the users, relying on them to identify sources likely to contain relevant information, compose an appropriate query, and sift through retrieved documents to extract relevant information. Clearly, as document collections continue to grow, it will become impractical for users to perform these tasks for all but the simplest requests. Even today there are more sources than a person could possibly access in a reasonable amount of time, many of which contain redundant, irrelevant, outdated, or even erroneous information.

Ideally, a user should be able to state a high-level request to an IR system and it would take care of the rest. Although we are far from achieving this goal today, one promising approach is to use planning to automate more of the IR process. In addition to the obvious benefit of reducing user effort, a planning approach has two important advantages over traditional approaches. First, it provides a structured framework for learning which IR tools are best-suited for different types of retrieval tasks. Second, it frees system developers to try radically different IR techniques without worrying about their comprehensibility to novice users. However, before a planning and execution system can successfully apply to IR, it must address several challenges presented by the task:

- **Incompletely-specified goals:** A query is often a poor approximation of the user's true information goals, and different information goals may be expressed by identical queries.
- **Partial and incremental goal satisfaction:** For many information goals, retrieval is not an all-or-nothing proposition. Partial satisfaction of a user's request is common and may be achieved *incrementally* via multiple iterations.
- **Incomplete knowledge of information source contents:** It is virtually impossible to determine the full extent of information available from many large heterogeneous collections (e.g. the web pages indexed by Google).
- **Uncertain action outcomes:** The performance of IR techniques varies from one use to the next. For example, synonym expansion will produce better results on some queries than others, and retrieval times will vary with network and machine loads.
- **Resource constraints:** Most users are not willing to wait more than a few seconds for a response unless the information is extremely valuable to them.
- **User interaction:** Success is uncertain until we actually execute a plan and receive feedback from the user. An implementation capable of interleaving execution with planning is essential, as is support for interaction with, and possibly intervention by the user.

This paper presents an integrated planning and execution framework that addresses these issues. Our domain operators are drawn from traditional IR techniques such as synonym expansion and relevance feedback. We define an abstract representation of the user's goals and resource constraints using a set of real-valued *metrics* and a utility-function specifying their relative importance. A forward-chaining planning and execution algorithm searches through

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

```
:name ( expand-query-with-synonyms )
:parameters ((?q query) system-time-spent est-query-quality )
:preconditions (
     :symbolic ( (not (applied-op-expand-with-synonyms ?q)) )
     :metric   ( (< est-query-quality 0.8) ) )
:effects (
     0.70 (:symbolic ( (del (not (applied-op-expand-with-synonyms ?q)))
                       (add (applied-op-expand-with-synonyms ?q)))
          :metric   ( (est-query-quality += 0.3)
                      (system-time-spent += 0.4)))
     0.28 (:symbolic ( (del (not (applied-op-expand-with-synonyms ?q)))
                       (add (applied-op-expand-with-synonyms ?q)))
          :metric   ( (est-query-quality -= 0.2)
                      (system-time-spent += 0.4)))
     0.02 (:symbolic ( (del (not (applied-op-expand-with-synonyms ?q)))
                       (add (applied-op-expand-with-synonyms ?q)))
          :metric   ( (system-time-spent += 0.5))) )
:execute ( synonym-expansion ?q )
```

Figure 1: Operator specification for `expand-query-with-synonyms`.

the space of *belief states*, sets of possible states and their associated likelihood, using expected utility and belief state uncertainty to guide action selection, execution, and replanning decisions.

This work contributes to planning research in two ways. First, it identifies planning challenges posed by the general IR task and presents a model that addresses them. Second, it presents an approach to handling incompletely-specified goals using an integrated planning and execution process.

The remainder of the paper is organized as follows. First, we describe the domain and problem representations we use. Secondly, we present our planning and execution algorithm. We then briefly describe the INSPIRE architecture that implements the approach, and discuss ongoing work in learning parameters to estimate the efficacy of domain operators. We further discuss the relationship between our work and previous research in planning and execution. We conclude with a discussion of current and future research directions.

The IR Planning Domain

An IR planning problem consists of a domain definition and a problem statement. The domain defines the problem-independent knowledge of the IR task which is common to different retrieval requests. The problem statement defines a problem-specific search context, the user's information goals, and resource constraints. In this section, we describe our representational choices for each.

Domain

An IR domain consists of:

- A set of types defining classes of objects (e.g., a `QUERY`)
- A set of metrics declaring resources, costs (e.g., `(ELAPSED-TIME 5)`) and *quality estimates* that will be monitored within states
- A set of operators defining the available atomic actions

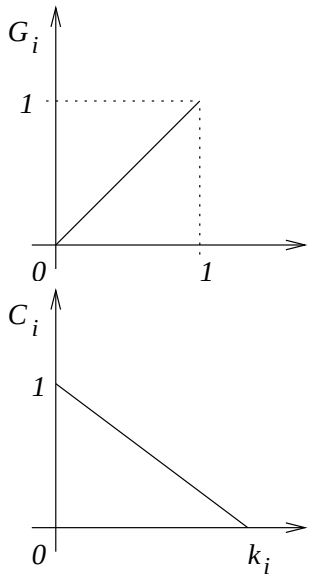
Metrics are declared as part of the domain description, and initialized within a problem statement. Like literals, they may be altered as either a side-effect or a primary effect of actions. In addition to representing consumable resources and execution costs, we also use metrics to define special *quality estimates* that are used to estimate how well the current state of knowledge will achieve a particular subgoal (e.g., `(est-query-quality 0.3)`). Representing knowledge goals in this way lets us compare the relative values of different states when the goals are only partially satisfied. (Discussion of how these quality estimates can be generated is deferred to a later section. For now, we will just assume the estimates are available to us.)

An operator is defined by its preconditions and effects, plus an *execution function* that declares the procedure that will actually be called during execution, along with the operator bindings to pass as arguments.

Preconditions are conjunctions of literal and metric expressions. Literal preconditions may contain typed variables, negation, and inequality constraints on numeric features. Metric preconditions specify inequality constraints on the metric values.

Effects consist of literals to be added to or deleted from the state, and update functions to apply to metrics. We can express uncertainty in these effects by declaring multiple effect sets that enumerate all possible sets of outcomes and the joint probability distribution for each.

A complete specification for a simple `expand-query-with-synonyms` operator is shown in Figure 1. This operator is used to revise a query by adding synonyms of individual query terms. For example, if applied to the single-word query *'car'*, we obtain a new query containing *'car, auto, automobile, machine, motorcar'*. It takes one variable of type `QUERY`, modifies two metrics, and has three possible outcomes: a state in which the query quality has improved, one in which it has remained unchanged (because no synonyms were available), and one in which the quality degrades. A metric precondition on `estimated-query-`




```
:objects ((Q1 query) (LYCOS search-engine) (GOOGLE
search-engine))

:util ((system-time-spent C1 (user-time-spent C2)
      (est-results-quality G1) (est-query-quality G2))

:init (:symbolic ((available-search-engine GOOGLE)
                  (available-search-engine LYCOS)
                  (query-confidence Q1 0.2)
                  (query-terms Q1 2))
      :metric ((user-time-spent 0)
             (system-time-spent 0)
             (est-query-quality 0.2)
             (est-results-quality 0)) )

:goal (:util-criteria (threshold 0.4)
      :succ-criteria (threshold 0.8)
      :utilfn ((1 C1) (10 C2) (100 G1) (20 G2)) )
```

Figure 2: Sample problem statement specifying the initial state, goal criteria and utility function to use.

`quality` limits applicability to queries whose quality is below the given threshold. We also specify a literal precondition and effect that restrict it to one-time application, as this technique is generally only effective the first time it is used on a particular query.

Problem Statement

A specific problem instance is defined by a set of available objects and their types, an initial state description, and a goal specification. The initial state declares the set of literal facts that are known to be true, and assigns initial values to each of the metrics. A goal specification defines the three components needed to evaluate the success or failure of the planning and execution session:

- A function $U(s)$ that estimates state utility from the current metric values in the state
- A minimum utility value for successful termination G_{thresh}
- A minimum satisfiability threshold S_{thresh}

The utility function defines the relative importance of the knowledge goals, costs, and resources. It is used to estimate progress towards our information goals, and guides the action selection process by comparing the expected utilities of the resulting states. Thus, to be useful, a utility function must accurately reflect the user's goals by providing a relative ordering on the states consistent with the user's preferences, correctly mapping goal states to high utility values and non-goal states to low values.

Our domain language currently limits utility functions to weighted combinations of functions for individual metrics m in the domain, each of which produces a normalized value between zero and one:

$$U(s) = \frac{\sum_m w_m U_m(s)}{\sum_m w_m}$$

The utility threshold G_{thresh} specifies the *minimum* utility value required for a satisficing solution. Any executed sequence with a utility value greater than this is assumed to have achieved the goals.

For example, the problem statement shown in Figure 2 defines a simple utility function comprised of two cost metrics and two quality metrics. Each metric has its own utility function (e.g., `system-time-spent` maps to C_1), and the goal statement declares the relative weights to assign to each. The diagrams at the left of the statement depict two very simple mappings; others are possible.

In addition to a utility threshold, a goal declares a satisfiability threshold S_{thresh} between zero and one, indicating how confident we must be in our success likelihood before we can declare planning "successful". A lower value indicates a greater tolerance for risking a lower utility outcome and wasted execution. We elaborate on the relationship between this and our planning and execution algorithm in the next section.

Interleaving Planning and Execution

As previously noted, the incompletely-specified knowledge goals of information-seeking tasks necessitate use of a forward-chaining algorithm. Moreover, although we can estimate the likelihood that our plan succeeds, until we receive feedback from the environment (via evaluation of retrieved documents and explicit user-feedback), we cannot be certain we have actually achieved the goals. Thus, execution and replanning must be an integral part of the planning process.

We use an integrated planning and execution algorithm that performs a best-first search across *belief states*. Our algorithm, presented in Table 1, is supplied with a domain model D, and a problem statement consisting of: an initial belief state I, a user-defined utility-function U, a utility success threshold G_{thresh}, and a value specifying a confidence threshold for termination S_{thresh}.

GenerateOrExecutePlan(domain D, belief state I, utility function U, goal threshold G_{thresh}, satisfiability threshold S_{thresh})

1. Initialize the current belief state and candidate successor belief states
   $B_{cur} \leftarrow I$
   $C \leftarrow$ Successors(D, B_{cur}, U)

2. If ProbabilisticallySatisfies(B_{cur}, U, G_{thresh}, S_{thresh})
   $P \leftarrow$UnexecutedActionSequence(B_{cur})
   If ($P \neq \emptyset$)
      $B_{new} \leftarrow$ Execute(PopFront(P))
      Update(B_{cur}, C, B_{new})
      If (ChooseReplanOrContinue(B_{cur}, C) == replan)
         GenerateOrExecutePlan(D, B_{new}, U, G_{thresh}, S_{thresh})
      Else goto 2.
   Else return success.

3. Else if (ReachedSearchLimit() or
   ($C == \emptyset$ and UnexecutedActionSequence(B_{cur}))== $\emptyset$)) return failure.

4. If (ChooseExecutionOrExpansion(B_{cur},C) == expand)
   $B_{cur} \leftarrow$ ChooseBestOne(C)
   $C \leftarrow (C - B_{cur}) \cup$ Successors(D, B_{cur}, U)
   Goto 2

5. Else
   $P \leftarrow$UnexecutedActionSequence(B_{cur})
   $B_{new} \leftarrow$ Execute(PopFront(P))
   Update(B_{cur}, C, B_{new})
   If ChooseReplanOrContinue(B_{cur}, C) == replan)
      GenerateOrExecutePlan(D, B_{new}, U, G_{thresh}, S_{thresh})
   Else goto 2.

Table 1: The planning and execution algorithm.

Beginning with the initial belief state as the root and an empty plan, the planner evaluates all successor belief states reachable from the current state by applying a single operator, and selects the one with the highest expected utility. The current belief state B_{cur} is updated to the projected belief state, and the selection process repeats. The actual plan, the sequence of operator applications required to transform the initial state into the projected current belief state B_{cur}, is implicitly maintained within each belief state by storing its generating operator and parent belief state.

At each step, the algorithm considers the tradeoff between executing the first unexecuted operator in the plan and continuing to plan with the uncertain outcomes of the projected belief states. If an execution step is carried out, it is followed by an assessment of the need for replanning.

The algorithm terminates when all steps in the plan have been executed and the confidence and utility thresholds for goal satisfaction are met, or there are no additional actions the planner can take. We now provide details on the belief state representation and supporting functions.

State Representation

We represent the state-space at two levels: the *individual state* level, at which individual operators are applied, and the *belief state* level, at which the planning and execution algorithm operates.

An individual state consists of grounded metrics and literals. (Note that we do not make a closed-world assumption; the absence of a fact does not imply its falsehood.) In turn, a belief state consists of a finite set of individual states representing all possible current states and their corresponding likelihoods. This gives us the ability to represent the degree of uncertainty in our state knowledge.

During the planning and execution process, new belief states are generated by the Successors() function shown in Table 2.

Successors(domain D, belief state B, utility function U)

1. Generate the set of all applicable operators A for the current belief state B.

$$A = \bigcap_{s \in states(B)} \{o \mid o \in operators(D), preconds(o) \subseteq s\}$$

2. For each operator $a \in A$, generate a successor belief state B_a,
   and calculate it's expected utility EU

$$B_a = \{s' \mid \forall s \in states(B), \forall e \in effects(a) : literals(s') = literals(s) \cup adds(e) - deletes(e); metrics(s') = apply(mfun(e), metrics(s)) P(s') = P(s)P(e)\}$$

$$B_a.parent \leftarrow B$$

$$B_a.action \leftarrow a$$

$$B_a.EU = \sum_{s' \in states(B_a)} P(s')U(s')$$

3. Return the set of successor belief states.

Table 2: The Successors algorithm.

Given an applicable operator, the function simulates the effects of applying it to the current belief state B_{cur}. The result is a new belief state containing the set of all possible outcome states and the probabilities of seeing each outcome. Note that a pair of outcome states within a belief state may contradict one another. However, once we actually *execute*

the plan steps leading to this belief state, only a single outcome state will remain.

An operator is considered to be *applicable* in the current belief state if all of its preconditions are satisfied in *every state* within the belief state. This ensures we have a valid plan regardless of which state we are actually in. Although it is not explicitly represented in Table 2, the generation of the successor belief states works with fully instantiated operators. Figure 3 shows a pictorial view of the generation process.

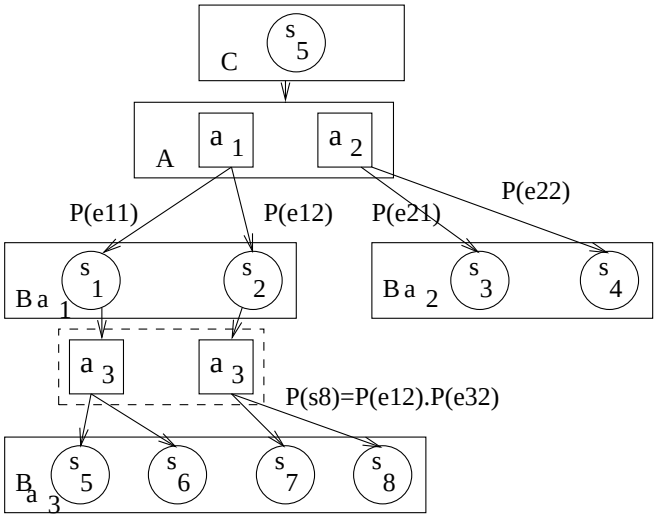


Figure 3: A pictorial view of the generation process.

When a new belief state is created, we also compute its expected utility EU. This value is equal to the weighted sum of the estimated utilities of each outcome state comprising the belief state. It is used for action selection by the function ChooseBestOne() that simply selects the belief state with the highest expected utility. B_{cur} is advanced to this new state, and the operator that generated the belief state becomes the next step in the plan.

Execution

The main advantage of executing in the IR domain is that it provides the planner with additional information that can be used to estimate how well the retrieval process is going, reducing the uncertainty and the number of possible states the planner must consider during forward projection. It may also allow us to terminate earlier if we are fortunate enough to discover our actual state was much better than our projections.

The main disadvantage of execution in this domain is that we cannot recover resources such as time once they are consumed, potentially leading to a worse (less optimal) result than if we had continued planning. In the worst case, where resources are severely limited, foolishly executing steps without sufficient lookahead may result in failure to find *any* solution because we no longer have the resources available to complete the task.

Our system supports three different execution strategies, the first two of which clearly represent the extremes of possible approaches:

- **Conservative (Deliberative)** - Always plan until we are forced to execute, i.e., no more planning is possible
- **Reactive** - Always execute as soon as an unexecuted plan step is available
- **Informed** - Base our execution decision on the features of our current belief state and alternate candidates.

We are currently exploring several different versions of informed strategies. Table 3 shows a simple one.

ChooseExecutionOrExpansion(belief state B, candidates C)

1. If ((C == $\emptyset$) or (**Var(Utilities(** $states(B)$)) > R_{thresh}))
   return execute
2. Else return expand

Table 3: Sample decision for execution vs. planning.

This function chooses execution when it is the only choice available, or when the variance of utilities in the current belief state exceeds a fixed threshold. The intuition here is that planning is probably not a very good option in belief states with bimodal utility distributions consisting of very high and very low utility outcomes. If executing puts us in a high utility state, we may be done without additional planning; if it puts us in a low utility state, then now we know that we are probably better off pursuing other options. Either way, we benefit.

In addition to considering the distribution of utilities in our current belief state and the number of alternatives we have, other strategies include consideration of the potential that execution has to change our view of the world (i.e., the number of alternatives execution introduces), and how costly the next step is to execute.

After executing an operator, an update function records the plan step as having been executed, removes candidates at branches of the search tree above the executed node, and recomputes our current belief state.

Replanning

Currently, for simplicity, we always choose to replan after executing a step. However, we eventually intend to replace this with a true decision point, taking into consideration:

- How consistent our new belief state is with respect to the original plan.
- Whether operators that were previously not applicable have now become applicable.
- The relative cost of updating the existing belief state tree versus the cost of replanning from scratch.

Termination

There are three different termination conditions that may occur during the planning and execution process: a solution may be found, the process may hit a pre-defined search limit, or it may fail because there are no additional actions available to take. We limit out discussion to just the success condition as both failure cases are straightforward.

The function ProbabilisticallySatisfies() (Tabel 4) uses the S_{thresh} satisfiability threshold to determine when the goal is considered "satisfied". The intuition behind this function is that if we have a plan that is already very likely to produce a successful outcome, additional planning may not be very useful. It is probably more productive to switch into execution mode to obtain feedback verifying or disproving the possibility that we are done.

ProbabilisticallySatisfies(belief state B_{cur}, utility function U, goal threshold G_{thresh}, satisfiability threshold S_{thresh})

$$\sum_{s \in B_{cur}} P(s)\delta(s) \geq S_{thresh}$$

where:

$$\delta(s) = \begin{cases} 1 & \text{if } U(s) \geq G_{thresh} \\ 0 & \text{otherwise} \end{cases}$$

Table 4: Probabilistic goal satisfaction.

As defined earlier, this user-specified threshold S_{thresh} is a value between zero and one that indicates how willing the user is to risk a suboptimal solution and wasted execution. A threshold of one indicates a user who is not willing to tolerate any risk (a strong satisfiability requirement); a small non-zero value indicates a user who is very risk tolerant (a weak satisfiability requirement). It is worth pointing out that although this threshold influences how suboptimal the result may be, it does not make any guarantees for optimality. Even in the risk-averse case, we are still only looking for satisficing solutions.

Implementation

We have implemented this approach as part of the INSPIRE (INtegrated System for Planning and Information REtrieval) architecture pictured in Figure 4. INSPIRE consists of three major components: the planner (IRplan), the execution manager (IRexecute), and the user interface. The user interface module is responsible for handling all the direct interactions with the user. It also takes care of translating the user input into an abstract representation that the planner can use. Upon receiving a new request in the form of a text query, the interface stores it in the data-repository, and generates a new problem statement containing the identifier by which the query was indexed, the query features and other state information, and a goal statement. This in turn initiates the planning and execution algorithm contained within the IRplan module. When an execution decision is made, the planner sends an execution request to the IRexecute module. The

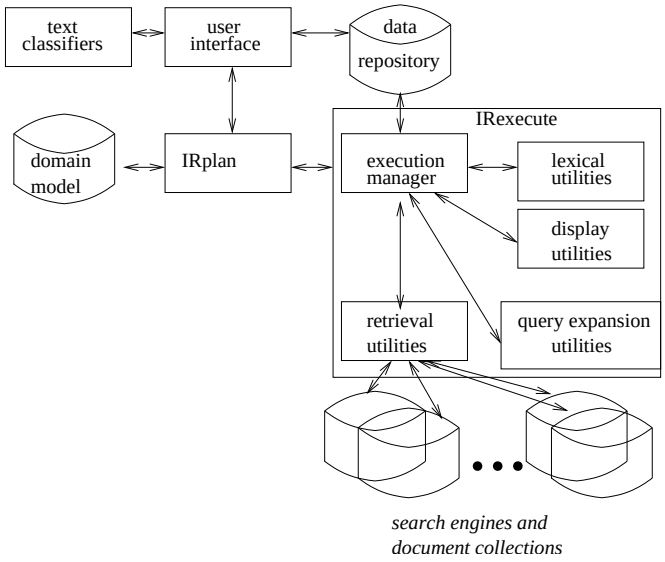


Figure 4: The INSPIRE Architecture.

execution manager retrieves any data it needs from the data repository and proceeds to carry out the requested task. After execution, it saves any new data to the repository, generates a results summary for the planner and returns control to the process.

At present, the display tools required to support post-processing operators are not implemented (as indicated by the dotted box). The results returned to the user consist of the first few lines from the 10 most-highly-ranked documents. We also do not provide mechanisms that allow the user to interrupt the system. User-control is returned only after the planning and execution process is finished or when a feedback request is executed.

Estimating Domain Parameters

One important aspect of our approach, which we have yet to address here, is the origin of the parameters used in the domain model. Currently, we use static prior values for the probabilities of operator outcomes and estimates of metric effects (including quality estimates). However, the development of better models and the addition of the infrastructure necessary to support learning is the major focus of our present research. We briefly outline two of the more interesting issues we are currently considering below.

- **Categorization of information needs**: Users of information retrieval systems have very diverse information needs ranging from requests for a specific document, to exploratory searches (Belkin *et al.* 1995). The ability to pre-classify incoming requests into a set of general goal categories could help us generate problem statements with more precise goal requirements.
- **Quality Estimates**: Research in information retrieval has long focused on defining useful measures of document 'relevance' (Rijsbergen 1979). However, for the purposes of planning, estimating the quality of intermediate

products and resources such as queries and sources are also important. Similar to (Kekäläinen & Järvelin 1998), we are currently evaluating methods for predicting query quality both in stand-alone requests and in the context of incremental search.

Related Work

The challenges presented by the IR task touch upon many different areas of planning research. In this section we describe a representative sample of this work.

The area that is perhaps most relevant to the current work is the problem of planning for information gathering. It has been the focus of considerable attention within the planning community in recent years (e.g.. (Golden 1998; Knoblock 1996; Levy, Rajaraman, & Ordille 1996; Kwok & Weld 1996; Barish *et al.* 2000)). As in the general IR task, the goal of information gathering is to obtain information that satisfies a user's query, often in the face of incomplete knowledge and resource constraints. Unlike the current work however, many of these systems are modeled after database paradigms: they are restricted to querying structured sources, require a model of source contents, and assume that the information goal is perfectly represented by the query. Information gathering systems such as Sage (Knoblock 1995), and XII (Golden, Etzioni, & Weld 1994) interleave execution to gather information, but adopt the simple policy of delaying execution for as long as possible (based on (Ambros-Ingerson & Steel 1988)). PUCCINI (Golden 1998) adopts a slightly more relaxed approach to execution. It allows execution to occur even when it is uncertain that the plan will be satisfied, as long as the outcome is verified afterwards.

In addition to information gathering systems that interleave planning and execution to support sensing, others have considered this problem in a more general planning context. Stone and Veloso (Stone & Veloso 1996) describe an extension to the Prodigy planner to support user-guided execution. They also suggest other execution policies based on abstraction hierarchies and learning from observing the user. Nourbakhsh (Nourbakhsh 1997) develops a set of three types of termination conditions under which the execution can start. They are based on abstraction, assumptive planning, and relative partial plan quality.

The problem of planning for uncertain outcomes has been addressed by several researchers within the context of conditional planning (e.g. (Peot & Smith 1992; Pryor & Collins 1996; Weld, Anderson, & Smith 1998)) and probabilistic planning (e.g. (Draper, Hanks, & Weld 1994; Kushmerick, Hanks, & Weld 1995; Blythe 1998; Blum & Langford 1998)). However, unlike the current work these systems generally work with completely specified goals and don't include execution support. Typically the probabilisitic systems declare success when a probability threshold value is exceeded.

Recently, there has been increased interest in heuristic search techniques for planning. One example is the GPT planner (Bonet & Geffner 2000), which uses forward chaining search heuristics, allows incomplete information, and has a belief-space representation similiar to the work here. We address this challenge within the context of information processing tasks.

Decision-theoretic approaches to planning (e.g. (Haddaway & Suwandi 1994; Haddawy & Hanks 1998; Williamson & Hanks 1994; Boutilier, Dean, & Hanks 1999)) are also similar to the current work in that they use a utility function to evaluate alternative plans and make it easy to take into account resource contraints.

Other relevant work from outside the planning community includes Microsoft's Lumiere project (Horvitz *et al.* 1998), which infers users' needs within the context of a software help system. Microsoft has also worked with developing utility-based models for user interfaces (Horvitz 1999).

Conclusions

In this paper, we have defined a set of challenges presented by the general real-world information retrieval task and described an integrated planning and execution system designed to address them. We have presented a represention that models the various sources of uncertainty in the domain, and uses the uncertainty in the state to guide our decisions for planning, execution, and replanning.

From a planning perspective, the IR domain and our approach provide an interesting framework in which to address a variety of difficult planning problems including: interleaving execution and planning, planning with incompletely specified goals, partial and incremental goal satisfaction, and considering the role of the user in the planning/execution loop. The INSPIRE architecture aims at addressing these issues that are of great importance in bringing automated planning tools to support user's goal decision making and goal achievement. From an IR perspective, the INSPIRE architecture presents an opportunity to provide users with better support in the information retrieval process, as well as to gain greater insight into the value of various IR techhniques.

Our approach has been implemented and thorough evaluation with real users and tasks is part of our ongoing and future work. We are in fact currently addressing question-answering tasks to further focus the general IR task. The INSPIRE architecture is the substrate for our work.

Although the questions of interleaving planning and execution and planning under uncertainty have been previously addressed, our research aims at grounding these questions within the task of information processing. Within this challenging domain that includes users, we envision learning the values of the parameters of our approach.

References

Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution, and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 83–88.

Barish, G.; DiPasquo, D.; Knoblock, C. A.; and Minton, S. 2000. A dataflow approach to agent-based information management. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI-2000)*, 138–139.

Belkin, N. J.; Cool, C.; Stein, A.; and Thiel, U. 1995. Cases, scripts, and information-seeking strategies: On the design of interactive information retrieval systems. *Expert Systems and Applications* 9:379–395.

Blum, A., and Langford, J. 1998. Probabilistic planning in the graphplan framework. In *AIPS98 Workshop on Planning as Combinatorial Search*, 8–12.

Blythe, J. 1998. *Planning Under Uncertainty in Dynamic Domains*. Ph.D. Dissertation, Carnegie Mellon University.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 31–36.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Efficient sensor management for planning. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, volume 2, 1048–1054. AAAI Press.

Golden, K. 1998. Leap before you look: Information gathering in the puccini planner. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 70–77.

Haddaway, P., and Suwandi, M. 1994. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*.

Haddawy, P., and Hanks, S. 1998. Utility models for goal-directed decision-theoretic planners. *Computational Intelligence* 14(3):392–429.

Horvitz, E.; Breese, J.; Heckerman, D.; Hovel, D.; and Rommelse, K. 1998. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, 256–265.

Horvitz, E. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of CHI'99, ACM SIGCHI Conference on Human Factors in Computing Systems*, 159–166.

Kekäläinen, J., and Järvelin, K. 1998. The impact of query structure and query expansion on retrieval performance. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 130–137. Melbourne, Austrailia: ACM.

Knoblock, C. A. 1995. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1686–1693.

Knoblock, C. A. 1996. Building a planner for information gathering: A report from the trenches. In *Artificial Intelligence Planning Systems: Proceedings of the Third International Conference (AIPS-96)*, 134–141.

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76:239–286.

Kwok, C. T., and Weld, D. 1996. Planning to gather information. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-96)*, volume 1, 32–39. AAAI Press.

Levy, A. Y.; Rajaraman, A.; and Ordille, J. J. 1996. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, 251–262. Bombay, India: VLDB Endowment, Saratoga, Calif.

Nourbakhsh, I. 1997. *Interleaving Planning and Execution*. Ph.D. Dissertation, Stanford University.

Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, 189–197.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research* 4:287–339.

Rijsbergen, C. V. 1979. *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow.

Stone, P., and Veloso, M. M. 1996. User-guided interleaving of planning and execution. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press. 103–112.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 897–904.

Williamson, M., and Hanks, S. 1994. Optimal planning with a goal-directed utility model. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 176–181.

Towards Statistical Planning for Marketing Strategies

Qiang Yang
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Kowloon, Hong Kong, China
qyang@cs.ust.hk

ABSTRACT

AI planning has traditionally dealt with developing sophisticated plans for achieving well-defined goals in well-defined domains. In this paper we consider the problem of building marketing plans for massive customers to achieve a company's financial goal in a business-planning domain. Corporations and institutions are often interested in strategic planning for marketing strategies to target their customers and outperform their competitors. For example, a stockbroker company may draft a marketing plan for retaining valuable customers or for switching a potential customer to a true customer. Planning in these applications consists of market segmentation, marketing-action selection and validation. For such problems, the traditional planning frameworks no longer apply. Instead, planning is done based on statistical reasoning of previous cases and patterns. In this paper, we present a novel framework that incorporates data mining, case based reasoning and planning to support marketing-strategy planning. In our approach, we discover case bases by data mining on the customer database and formulate plans based on the mined cases or "role-models". These plans are not guaranteed to work for each individual; however, based on previous experience, they have a high probability of succeeding. We explore the tradeoff among time, space and quality of computation in this framework. We demonstrate the effectiveness of the methods through empirical results.

1. Introduction

AI planning has traditionally focused on generating plans for a single user to achieve some well-defined goals. In this paper, we consider the problem of generating marketing plans to be acted on massive customers who achieve a company's financial goals, where the actions and goals are only implicitly defined. Our work is motivated from the realistic problems of developing marketing strategies to increase a company's overall profit, and is based on a statistical consideration of the given dataset by building actionable classification models using data mining algorithms. Compared to traditional data mining tasks, we take one step further than simple classification of data; we not only use statistical models to classify customers, but also produce marketing plans to be acted on the customers that make them to switch classes. As we shall see, the traditional planning framework where goals and actions are formally and logically specified no long applies. Instead, we propose a statistical planning framework for constructing these plans using data mining and case based reasoning.

Consider the example shown in Table 1. Suppose that we are given a customer database from a mobile-phone company. The last attribute records whether the customers signed on to a new service contract after some marketing actions, such as sending a free gift, has been applied to the customer. Based on these marketing data, we are interested in knowing what marketing plans would be the most effective in order to increase the chance of valuable customers to sign on to a new service contract. For example, for a customer Basil, we are interested in knowing whether we should give the customer a fee reduction, sending him a new gift or allowing him longer free airtime. We are also interested in how much fee reductions we should provide for Basil to achieve our purpose while keeping the marketing cost at a minimum.

Table 1. A Cell Phone Company's Marketing Planning Problem

		Mobile Phone Data			
Cellphone Users		Fee Reduction ($)	Gift	Free calls	Stayed?
	John	10	Y	80 min	Yes
	Beatrice	20	Y	100 min	Yes
	Mary	30	N	120 min	Yes
	Mathew	20	N	100 min	No
	Steve	15	Y	150 min	No
	Basil	?	?	?	Yes

This example introduced a number of interesting issues for planning. Traditionally, this type of problems were not considered as planning problems, because there are no logically formulated actions with preconditions and effects, no logically provably correct goals and initial states. However, on a close examination, there are many

aspects of the marketing plan problem that are of interest to planning researchers. First, although goals are not explicitly given, they can be discovered. For example, a possible goal is to identify a potential role model, say John, as a potential marketing goal for Basil, and formulate marketing actions to make Basil resemble John as much as possible. This may involve giving Basil a fee reduction of $10.00, sending him a gift and allowing for no less than 80 min of free airtime. Second, although there are no logically formulated actions such as the Strips representation, the actions are clearly present. For example, fee reduction for a customer is indeed a potential marketing action that can be taken. Third, similar to considerations in planning with uncertainty, the marketing plans are not guaranteed to work for any particular customer. **B**oth the costs and probability of success are taken into account. The goal here is to maximize the expected utility of the overall marketing plan for all customers.

The marketing plan problem also has a wide range of applications that are beyond business marketing. For example, it can be formulated as an advice generation for students who apply for graduate schools. Instead of rejecting a graduate school applicant with only a "no" answer, we suggest steps that might be taken by the applicant in the future to increase his/her chance of being admitted the next time around.

Table 2. An example customer database.

Customer	Salary	Cars	Mortgage	Loan Approved?
John	50K	2	None	Y
Mary	40K	1	300K	Y
…	…	…	…	…
Steve	40K	1	None	N

Table 3. Prescribed plans for Steve.

Advice for Steve	Salary	Cars	Mortgage
Plan 1	40K→50K	1→2	
Plan 2			0→300K

Similarly, plans can be constructed to give advice to customers who fall short of loan applications. As an example, consider a customer database shown in Table 2. Suppose that we are interested in providing an advice for Steve (the last row) who failed to apply for a bank loan. Obviously, there are many candidate actions that one can advise Steve to take in order to succeed in his next loan application. For Steve, we can advise him to find another job with a salary close to 80K and increase his car number from one to three; this will make him look more like John. Alternatively, we can advise Steve to take up a mortgage from the bank worth at least 300K. This will make Steve look more like Mary. In either situation, Steve might have a higher chance of succeeding than before, but the actions come with different costs. The prescribed actions for Steve are shown in Table 3.

The above-formulated problem can be stated as a combination of data mining problem and case-based reasoning problem [6, 14], where the key issue is to look for low-cost plans with high success probabilities for customers based on previous experience. These plans can be generated on a case-by-case basis as in the previous situation for Basil and Steve, or the plans can be a single strategy that is applicable for a subset of customers; for example, a decision might be to send gives to all customers whose income is over $50,000.00 a year. The actions are only given implicitly in the form of attributes and their combinations, and the effects of actions can only be discovered statistically.

In this paper, we present a novel formulation of the above marketing plan problems for AI planning. We explore a case-based planning solution to the problem, where the case bases are extracted from a large raw customer database using data mining techniques. We consider the overall utility of the marketing plans developed, and propose solutions that provide tradeoff between quality of solution and speed of computation.

2. Understanding the Problem

The marketing strategy-planning problem departs from traditional planning significantly. First, the goals are not clearly given in a logical manner, as is done in many other planning algorithms. The goal in marketing is to increase the overall profit of a company while keeping the cost low. This goal has to be translated to individual actions for each customer. A second difference from the traditional planning is that actions are not given in the traditional way. Instead of starting out with a well-defined set of actions schemata, in business marketing the actions are only implicitly given. These actions must be constructed as the marketing strategy takes shape. For example, in direct marketing in a cell phone company, the effects of actions such as reducing the customer fees can only be measured when all customers' responses are known in the end. Finally, the marketing strategic plans themselves are not necessarily partially ordered action sequences. Instead, they are a set of actions on a segment of customers or on all customers that change the attribute values of a database.

We formulate the problem as a combination of data mining and case-based planning problems. In this approach, we first identify typical positive cases from a large dataset to form a case base, and then use the case base formulate the actions that adapt each incoming problem by finding its nearest neighbor in the case base.

More specifically, we first classify the training data into two classes: the "good" data set contains data that belong to customers who have already been accepted into the good class and the bad set those who have not. Given this labeled dataset, our second step is to perform a clustering analysis to find out a number of representative good cases of customers that can be "role models" for the rest and that represent the centroids of the good class distribution. We also identify a subset of highly relevant and actionable attributes of the database table that can be used to generate actions. The relevant actions are derived based on a feature extraction algorithm. The actionable attributes help project both the good and the bad databases on this set of attributes.

The representative data points discovered by data mining comprise a case base. For each new customer in the testing data, we compute a nearest neighbor from the case base for each customer in bad class. The new customers are then given advise on what actions are needed to transform themselves to a good case.

There are two important issues in this approach. The first issue is how to construct a concise case base from a large database. In this paper, we consider three approaches. The most naïve one is to simply use the original database as the case base. While this model allows the creation of optimal plans from the past data, this approach is highly inefficient. The second approach constructs clusters from the database, and takes the centroids of the clusters as the potential cases for the case base. This approach can be very efficient, but the quality of the cases is still not optimized. This is because in creating role models for the positive class, it is more desirable to find cases that are "close" to the majority of the negative instances. These cases are often located near the "boundary" of the distributions of these classes. In order to find these boundary cases, our third approach is to apply a support vector machine-learning algorithm for extracting the support vectors as cases. These cases can give rise to more cost-effective plans.

A second important issue is that for each advisee, how to select the target role model for advice generation. Here we consider two approaches. The first is to apply a nearest neighbor algorithm, which computes the distance between cases from the cost of actions attached to the attributes. While this approach provides plans that minimize the total costs, it does not give advice on the success likelihood of the plans. In practice, it is not always guaranteed that a switching plan will work. The probability of success is highly dependent on the distribution of classes around a role model. Thus, our second approach is to consider the utility of each potential role model, taking into account both the cost of switching and the probability of success. We show that this approach provides a much better advice plan for the customers.

From an AI planning perspective, this paper raises several challenging issues. First, instead of considering a well-defined goal, we need to consider the problem of finding goals to achieve. These goals correspond to role models in our case base. Second, instead of finding plans for a single user, we need to find plans for massive users and consider their overall gain. Third, instead of given logical representations of actions ahead of time, we have to compose these actions statistically. Finally, instead of delivering the plans for a robotic system to execute, our plans are in the form of advises for users to follow. These advises have immediate applications in the financial and marketing applications.

Our approach is related to several existing areas of research. The first is data mining and machine learning. In this area, researchers are interested in building statistical models of the database for classification and data analysis [4, 5, 8]. A typical statistical model partitions the test data into different classes according the trained model learned from the training data. A large literature exists in this area, such as work on decision tree analysis [8] association analysis [11] and Bayesian network models.

A second area of research is case-based reasoning, in particular case-base maintenance [9, 10] and case-based planning [14]. In the case-based planning work of Veloso [14], a new problem is solved by consulting a case base of past solutions. A new solution is then formed by deriving the difference between a past solution and the new problem by applying case-adaptation methods. However, to apply case-based planning to our problem, a case base have to be known and the plans in the case base have to be logically well formed. We don't have this luxury in marketing strategy planning, because the case base have to be discovered from the customer data first. Related to this issue is the so-called case-base maintenance problem, which has received much attention lately. In case-base maintenance [10] the focus has been on how to update case bases from problem solution pairs, and not on least-cost case-base generation for class-transformation.

In AI Planning, the objective has been to find a sequence or sequences of actions to achieve a user specified goal or objective. In our situation, the goal to be achieved is only implicitly given; that is, the goal is to enable a customer to become eligible for credit loan. The plans are to convert all the bad customers into good

ones, using the case base as guide. This objective is related work in decision theory [12, 13], but the problem of scaling up using a small case base has not been addressed. The key issue again is to find a good case base from the database so that the overall cost of negative-case transformation is minimized.

3. Case-Base Mining By Clustering

We formulate the case-mining problem formally. Given a database of customer records, we assume that each customer record is labeled as either a positive or negative class. Multiple class generalization is possible but will be considered separately in future work. Each attribute is labeled either as actionable or non-actionable. For each actionable attribute A and values v1 and v2 of A, there is a cost function: *cost (A, v1, v2)* which is a real value.

The problem to be solved is to find a case base of *K* positive instances, such that the total cost of converting from all negative instances in the test set to their nearest neighbors in the case base is minimal. We will present two approaches to the problem, one requiring that the use specify the value for *K*, and the other does not.

In the extreme situation, the value of *K* is equal to the size of the population of all positive instances. In this case, the system is able to find the optimal solution, where each bad case is paired with its closest positive case. The drawback of this extreme situation is that the computational cost for planning for each individual negative instance is too large; in the realistic situation, the database may contain millions of customers. Finding the optimal solution for all negative instances is neither feasible nor necessary in practice.

Our first step is to find *K* near-optimal instances to populate the case base. When K is not known in advance, we can apply a second method discussed below to find the optimal cases. This case base will consist of *K* instances that are highly representative of the distribution of the customer information in the original database. In this work, we use all positive instances as training data for the case base model, and the negative instances as testing data set for evaluation. To see the interplay of efficiency with the size of case base, we show experimental results of quality of advice versus the time to come up with the switching plans of a certain quality. The case base quality is defined as the total cost to transform all negative instances into a positive counterpart in the testing set.

Our first case-base mining algorithm is described in more detail in Table 3. Given an input database, we divide the database into a training database and a testing database. The training database consists of the positive instances of the original database, whereas the testing data are the negative instances.

Table 3. **Algorithm** *Centroids-CBMine* (database DB, int K)

Steps	**Begin**
1	*casebase*= emptyset;
2	DB = *RemoveIrrelevantAttributes*(DB);
3	Separate the DB into DB+ and DB-;
4	Clusters+ = *ApplyKMeans*(*DB+*, *K*);
5	**for each** *cluster* in Clusters+, **do**
6	C = *findCentroid*(cluster);
7	*Insert*(C, *casebase*);
8	**end for**;
9	Return *casebase*;
	End

In the algorithm *Centroids-CBMine* in Table 3, the input database is DB. There are two classes in this database, where the positive class corresponds to population of desired cases and the negative class the unconverted cases. Step 2 of the algorithm performs feature extraction by applying a feature filter to the database to remove all attributes that are considered low in information content. For example, if two attributes *A1* and *A2* in the database are highly correlated, then one of them can be removed. Similarly, if an attribute *A* has very low classification power for the data, then it can be removed as well. In our implementation, we apply a C4.5 decision-learning algorithm to the database DB. After a decision tree is constructed, the attributes that are not contained in the tree are removed from the database; these are the irrelevant attributes.

Step 3 of the algorithm separates the training database into two partitions, a positive-class subset and a negative-class subset. Step 4 of the algorithm performs the K-means clustering on the positive-class sub-database [2]. K-means finds *K* locally optimal centroids by repeatedly applying the *EM* algorithm on a set of data. Other good clustering algorithms can also be used here in place of K-means. Step 6 of the algorithm finds centroids of the K clusters found in the previous step. These centroids are the bases of the case base constructed thus far, and are returned to the user. Finally, Step 9 returns the case base as the output.

Once the case base is built, it can then be applied to a set of testing negative-class cases to see what the total cost would be for converting all the negative cases to positive ones. For each negative class case C1 in the test data set, a one-nearest neighbor algorithm is applied to the case

base to find the most similar case C2. The difference between C1 and C2 are used to generate the switch plan.

A critical issue for this approach is the tradeoff among the size of the case base, the quality of the model built and the time taken to build the case-base model. The quality measure is defined in terms of the total cost of converting negative instances to positive ones based on the basic cost elements of performing a conversion from one value to another for the actionable attributes. Recall that these elementary cost functions are *cost(A, v1, v2)*, which is a real value denoting the cost of switching attribute A from value v1 to value v2. Then, the cost of the model on an entire population of test data is the sum of all costs for all actions on each datum in the testing set. Assuming that the *j*th attribute for an *i*th customer is A_{ij}, Equation (1) shows the cost formula.

$$Cost = \sum_{i=1}^{|DB|} \sum_{j=1}^{J} \cos t(A_{ij}, v_1, v_2) \quad (1)$$

The specification of the cost of switching an attribute A_{ij} from v1 to v2 can depend more than the attribute and its two values; it can in fact depend on the context of the switching. In this paper we simplify this consideration by only considering the attribute itself; but this restriction can be relaxed later.

4. Case-Base Mining by Support Vector Machines

The centroid-based case-mining method extracts cases from the positive-class cluster centroids and takes into account only the positive class distribution. By considering the distribution of both the positive and negative class clusters, we can do better.

The key issue then is to identify the positive cases on the *boundary* between the positive and negative cases, and select those cases as the final ones for the switching-plan generation. The cases along the boundary hyper-plane correspond to the support vectors found by an SVM classifier [3, 7]. These cases are the instances that are closest to the maximum margin hyper-plane in a hyperspace after an SVM system has discovered the classifier [3].

By exploiting the above idea, we have a different case-mining algorithm, SVM-*CBMine()*. In the first step, we perform SVM learning on the database to locate the support vectors. Then we find the support vectors and insert them into the case base. This algorithm is illustrated in Table 4.

Compared with the Centroids-CBMine algorithm, the SVM-CBMine algorithm has several advantages. First, because the cases are the support vectors themselves, there is no need to specify the input parameter *K* as in the Centroids-CBMine algorithm; the parameter *K* is used to determine the number of clusters to be generated in K-means. This corresponds to parameter-less case mining. Second, because the cases are themselves the boundary cases, they are naturally better examples for the entire negative-class members to switch to; the costs would be lower. However, the SVM based methods have also their drawbacks. A potential drawback is that SVM based classifiers are usually very costly to generate and are highly dependent on the data distribution. As we will see below in the experimental section, there are many datasets for which the SVM classification fails to produce any result within a limited amount of time, resulting in no case bases at all. As we will also point out, such situations occur when there is no clear boundary between the two classes.

Table 4. **Algorithm** *SVM-CBMine* (database *DB*, int *K*)

Steps	**Begin**
1	*casebase* = Emptyset;
2	*Vectors* = *SVM*(*DB*);
3	**for each** positive support vector C in *Vectors* **do**
4	Insert(C, *casebase*);
5	**end for**
6	Return casebase;
	End

5. Experimental Results

Our experiments are aimed at finding out the tradeoff among the system execution time, which is the model-building time plus the model-application time on test cases, the size of the model (the number of cases) and the total cost of switching plans for converting all negative examples into positive ones. We are also interested in the effect of distribution of the training data on the model quality. In our experiments we tested on both artificially generated data and some real data sets. The comparison made here did not use any attribute filtering algorithm to remove the irrelevant attributes. The experiments are performed on an Intel PC with one Gigahertz CPU.

We first tested the algorithms on a artificial data set generated on a two-dimensional space (x, y), using a Gaussian distribution with different means and co-variance matrix for the + and the – classes. When the means of the two distributions are separated, we expected the class boundaries are easy to identify by the

SVM-based method (see Figure 1). On the other hand, when the two means are very close to each other, there will not be an easy-to-find boundary; in this case the centroid-based method will perform better. The cost of switching a negative case to a positive one is defined as the Euclidean distance on the (x, y) plane.

In this distribution, the mean for the positive case distribution is *mean1*=(7, 8), with a co-variance matrix [(0.6, 0.3), (0.3, 1.8)]. For the negative class, the location of the mean *mean2* moves from being far away from the *mean1* to being close to it. The co-variance matrix for the positive class is defined as [(0.8, -0.5) (-0.5, 3.2)]. Table 5 shows the test results. In this table, SVM stands for the *SVM-CBMine* result. SV=3 indicates that three support vectors were found to populate the case base. Parameter *K* indicates the number of clusters generated by K-means algorithm for the *Centroids-CBMine()* system. In the table, the time, size and cost values are all indicated. Finally, Optimal (last row) indicates the cost and time for the model using all positive examples as the cases in the case base. Similarly, a second distribution is listed in Table 6, corresponding to the situation when the centers for the negative distribution are moved closer to that of the positive distributions. As can be seen from the progression of the data distribution, as the two classes are distributed farther apart from each other, the SVM-based method is a clear winner. This is because it uses far less time than the optimal method, and yet its total cost is nearly the same as that of the optimal method. As can be seen from the K-means based method, as the number *K* of clusters increases, the cost of switching plans also decreases. However, the time it takes to build and execute the model also increases with *K*. On the other hand, as the two distributions move close to each other such that there are no clear boundaries, as in the case of Table 6, the SVM method selects nearly all the positive examples as cases in the case base, rendering it useless. Thus, its time expense is also very high. In this case, the K-means based method is recommended.

Table 5. Result for *mean2* = (3, 4) (See Figure 3)

	Cost	Time (s)
K = 10	1934.2	1.3
K = 50	1464.6	6.6
K = 100	1420.8	13.8
SVM SV = 3	1225.6	0.9
Optimal	1220.0	7.03

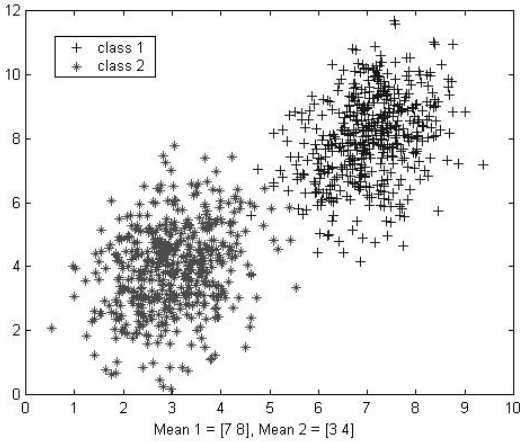


Figure 1. Distribution of the class 1=positive (+) and class 2=negative (*) data.

Table 6. Result for ***mean2*** = [7 8]

	Cost	Time (s)
K = 10	259.3	1.3
K = 50	133.4	6.5
K = 100	101.6	12.0
SVM SV = 487	58.5	25.6
Optimal	56.7	8.5

We next compared the three models on the some UCI data sets [1]. In Table 7, which is the data from German credit approval data, the SVM based method is unable to find any clear boundaries between the two classes; thus in this case the K-means based method wins. SVM cannot produce any case base either for other UCI data sets including the adult Database, the Liver-disorders Database, etc.

Table 7. German Credit dataset: 20 attributes, 2 classes 1000 instances (700 +, 300 -)

	Cost	Time (s)
K = 10	1211.2	3.3
K = 50	1108.9	7.6
K = 100	1060.8	14.9
SV = 700	934.6	18.8
Optimal	934.6	17.5

6. Utility Guided Plan Generation

The previous sections solved the plan generation problem using a nearest neighbor approach. The plan used to advice a customer is one that is associated with the least cost. While this is guaranteed to generate a cost effective plan, it is not guaranteed to generate a plan that will achieve its intended target all the time. In reality, the positive and negative cases are often distributed in a mixed manner. Several negative cases may surround a positive case. When executing a customer-switching plan, it is likely that the customer following the plan will land on a wrong target; it is wrong because it corresponds to a "unreliable" positive case whose neighborhood is dominated by negative instances, rendering the switching low probability of success. A more sensible method will consider not only the cost of switching, but also the probability of success of each switching.

We can estimate the probability of success of switching to a certain target to be the probability density of positive instances around a target. More formally, let $p(+|t)$ be the probability density of an instance t, $cost(x, t)$ be the cost of switch from x to target case t, and $maxCost$ be the maximum value among the different costs of switching from x to every possible case y in the case base. The utility function we use for ranking cases in a case base is defined in Equation (2) below. The target case t with the maximum rank is chosen as the role model for switching-plan generation for customer x.

$$rank(x,t) = p(+|t) - \frac{\cos t(x,t)}{\max Cost} \qquad (2)$$

Finally, we performed a scale-up test using the by IBM QUEST synthetic data generator. We generated the training dataset with nine attributes, 50% positive class and 50% negative class. An excerpt of the database is shown in Table 8. Our results are shown in Table 9. It is clear from the table that with large data, the centroid-based method is able to scale up much better than the SVM based method.

Table 8. An excerpt from the synthetic dataset.

Salary	Commission	Education	Car	…
65498	49400	1	2	…
24523	0	2	3	…
78848	0	2	6	…
74340	29463	0	3	…
42724	0	1	4	…

Table 9. CPU-time comparison of Centroid-based Method and SVM-based method.

***Log*$_{10}$*(N),* *N*=Database size**	**CPU Time (Seconds)**	
	Centroid-based	**SVM**
2	0.660	0.650
2.5	1.640	1.870
3	6.480	14.330
3.5	23.120	319.180
4	95.850	3,834.900
4.5	312.970	No result in 5 hrs
5	1,938.430	No result in 7 hrs

7. Conclusions and Future Work

In this paper, we proposed a case-based solution for the switching-plan generation, a problem that arises in business planning. The central issue of the problem lies in the discovery of high-quality case bases from a large data set. This problem corresponds to finding goals to achieve for each customer. We proposed two solutions for the problem. For the data distribution where the two classes are clearly separated, the SVM-CBMine algorithm, which is an SVM-based method, should be used. When the data distributions are not separated well by a boundary, the cluster-centroids based method is recommended. Furthermore, we compared the solutions where plan generation is done based on distance alone and the solution where the probability of success is also taken into account. It was shown that the solution with utility consideration is superior. In addition, the centroid-based method is shown to scale much better than the SVM-based method, demonstrating a quality-speed tradeoff.

In the future, we will continue to explore other forms of case mining and the related problem of switch-plan generation. Other cost functions will be considered. Attributes weights will be taken into account as well.

Acknowledgment

We thank Hong Cheng and Ke Wang discussions and assistance with the experiments. The research is supported by a grant from Hong Kong University of Science and Technology.

8. References

[1] C. L. Blake, C.J. Merz (1998). *UCI Repository of machine learning databases* Irvine, CA: University of California, Department of Information and Computer Science. http://www.ics.uci.edu/~mlearn/MLRepository.html

[2] P. S. Bradley and U. M. Fayyad. *Refining initial points for **k-means** clustering*. In Proceedings of the Fifteenth International Conference on Machine Learning (ICML '98), pages 91--99, San Francisco, CA, 1998. Morgan Kaufmann.

[3] G. C. Cowley. *MATLAB Support Vector Machine Toolbox. v0.54B* University of East Anglia, School of Information Systems, Norwich, Norfolk, U.K. NR4 7TJ, 2000. http://theoval.sys.uea.ac.uk/~gcc/svm/toolbox

[4] C. X. Ling and C. Li. *Data mining for direct marketing: Problems and solutions*. In Proceedings 4th International Conference on Knowledge Discovery in Databases (KDD-98), New York, 1998.

[5] P. Domingos and M. Richardson. Mining the Network Value of Customers. Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. August 2001. ACM. N.Y. N.Y. USA

[6] D. Leake. *Case-based Reasoning -- Experiences*, Lessons and Future Directions. AAAI Press/ The MIT Press, 1996.

[7] J. C. Platt, *Fast training of support vector machines using sequential minimal optimization*, in Advances in Kernel Methods - Support Vector Learning, (Eds) B. Scholkopf, C. Burges, and A. J. Smola, MIT Press, Cambridge, Massachusetts, chapter 12, pp 185-208, 1999.

[8] J. Quinlan C4.5: *Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

[9] B. Smyth and M. T. Keane. *Remembering to forget: A competence--preserving deletion policy for case--based reasoning systems*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp 377--382, 1995

[10] *Computational Intelligence Journal, Special Issue on Case-base Maintenance*. Blackwell Publishers, Boston MA UK. Vol. 17, No. 2, May 2001. Editors: D. Leake, B. Smyth, D. Wilson and Q. Yang.

[11] R. Agrawal and R. Srikant. *Fast algorithm for mining association rules*. Proceedings of the Twentieth International Conference on Very Large Databases. 1994. pp 487-499

[12] F. Bacchus and A. Grove, *Graphical Models for Preference and Utility*, Proceedings of the Uncertainties in AI 1995.

[13] R. L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Trade-offs*, Wiley, New York, 1976

[14] M. Veloso *Planning and learning by analogical reasoning*. Number 886 in Lecture Notes in Artificial Intelligence. Springer Verlag. 1994

[15] Q. Yang. *Intelligent Planning*. Springer Verlag. 1997

Using Planning for Query Decomposition in Bioinformatics

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

Abstract

Domains like bioinformatics are *complex* data integration domains because data from remote sources *and specialized applications* need to be combined to answer queries. An important characteristic of such a domain is that actions may be mutually exclusive or causally related. Moreover, there is (partially complete) domain specific knowledge about how queries should be answered since an average user is a sophisticated domain expert. Query plans in these domains should not only answer the query but also respect any user intent or domain guidance provided to improve the perceived quality of the result and query execution time. Previously, methods like rule inversion and view unfolding have been found to be more effective than AI planning in obtaining access sequences for sources without interactions, a case when sources are just repositories of data.

We present a solution in SHQPlanner, a hierarchical temporal planner for query planning and execution monitoring in complex domains based on previous theoretical work on HTN planning with partial domain knowledge on one hand and temporal reasoning for query cost on the other. SHQPlanner is a sound, complete, and efficient domain-independent query planner that can incorporate partial query decomposition, source preferences, data and application interaction, and temporal constraints.

Introduction

The amount of genomic data available for analysis online is vast and ever growing. Yet, a biologist wishing to gain insight from them is lost in data model, data formats, and interfaces of particular data sources. Many sources have data as formatted file with specialized Graphical User Interfaces (GUIs) and retrieval packages. The design choices made by the autonomous data sources considers the complexity of data, efficiency of analytical tools, multi-platform support and cost of implementation, but not integration issues which would have lead to more adoption of database management systems. The heterogeneity of the data sources and the multitude of non-standard implementations makes providing uniform access for such data sources an integration nightmare.

The computer science field of data integration (also known as information integration or information gathering(Lambrecht & Kambhampati1996)), which lies at the cross-roads of Artificial Intelligence and Databases, studies how to provide access to multiple autonomous heterogeous data sources in a uniform fashion(Levy1998). Given a global world model, a set of information sources, a mapping of contents of information sources to the world model, and a query on the world model, the objective is to return information contained in the information sources that answers the query on the world model. An agent integrating the different heterogeneous sources must return only the actual information that satisfies the user's query and no more.

It turns out that biology[1] is both complex and large compared to previously considered domains in data integration. The specific characteristics are:

- In bioinformatics, data from sources and specialized applications (either located at the sources or implemented by the mediator) must be combined to answer queries. For example, gene expression data from biochips is clustered by a suitable application or a search for proteins should be fed to the BLAST application for protein similarity search and only similar proteins, which is the result of the application, should be used for pathway analysis.
- The user, a biologist, is a specialist of the domain and has (partially complete) domain specific knowledge about how queries could be efficiently answered for meaningful insights. A form of domain knowledge, the search control knowledge, may dictate that subgoals of actions may be mutually exclusive or causally related.
- Additionally, since there is considerable choice for sources and applications (pre-processing, post-processing or analytical) of a particular type(Baxevanis2002) (e.g., protein, pathway, publication, gene expression data), the user usually has strong preferences about which sources and applications are used to answer queries.
- Query decomposition in bioinformatics has a mixed-initiative planning flavour. The reason is that queries in biology can take very long time due to extreme range of data sizes that may be retrieved. Hence, biologists quickly want to decide if a line of biological exploration is worthwhile before investing more time in it looking for refined results.

[1] We are particularly interested in Bioinformatics, which is the application of information sciences (mathematics, statistics and computer science) to increase our understanding of biology.

Conventional data integration approaches consider the data sources as repositories of data but not as applications (which may in turn embody complex interactions with other sources), and they do not provide any mechanism for leveraging domain-specific user guidance. *We argue that data integration in domains with these characteristics are best addressed by using AI planning for query decomposition.*

AI Planning has been considered in conventional data integration to determine the best way to integrate data from different sources(Knoblock1995; Knoblock & Ambite1998; Kwok & Weld1996), and monitor the actual execution of source requests. Planning tackles the problem of composing a sequence of actions so that an agent can go from the initial state to the goal state given the set of legal executable actions(Kambhampati & Srivastava1995). However, planning for query decomposition seems to have lost support in favour of cheaper methods (Levy1998) like rule inversion(Duschka1996) and view unfolding(Qian1996). The main reason is that since the search space is made up of *information* states, there is no subgoal interaction among actions as they can always be executed on the sources to gain the information needed. Hence, the conclusion drawn was that using planning for just sequencing source-call actions is an over kill. But when sources can also be applications, they may encode physics of their interactions[2] which may prevent an action from being always applicable. For example, if a user's authentication request is denied (by a trusted third-party), her already available credentials (information like password or certificate) to access a source may become invalid. Hence, we need to revisit planning for data integration to address action interactions.

Hierarchical Task Network (HTN) Planning (Erol1995) is a paradigm in planning to capture user intent about desirable solutions and what actions are used in them. However, duration (temporal properties) of an action has not been widely modeled in HTN. In this paper, we extend a recent forward chaining (also called forward state-space) temporal planner, *Sapa*(Do & Kambhampati2001) to handle task hierarchies (also called *schemas* in HTN planning) and other features useful for biology. Specifically, we describe *SHQPlanner*, a hierarchical temporal planner, for query planning and execution monitoring in biology and other such domains where data from sources and applications have to be integrated, and user has background knowledge about what kind of plans are acceptable. The advantage of our approach is that the hierarchy can embody the domain knowledge while the temporal specification can help reasoning with query cost models. Important features of *SHQPlanner* are: (a) it has the ability to incorporate partial query decomposition and source preferences, data and application interaction, and incremental updates (b) it is a domain-independent query planning algorithm (c) it reasons with temporal constraints for cost estimates and (d) it is sound, complete, and efficient.

Our work is in the context of an end-to-end XML-based Bioinformatics application called *e2e*(Adak et al2002) that facilitates analysis of gene expression data by providing a uniform access to diverse online data sources (i.e., proteins, literature, pathways and gene expression data) and representing the annotations on the intermediate data in a XML format, eXpressML(Adak et al2001). For example, text summarization can be performed on the result from a literature source and the top few keywords are represented in eXpressML. In *e2e*, eXpressML can be queried by a XML language or processed by advanced statistical tools. We are in the process of integrating *SHQPlanner* into *e2e*.

Here is the outline of the paper: we describe the bioinformatics domain in the next section and survey the existing methods here for data integration. Next, we pose query decomposition as a planning problem and show how *Sapa*, a temporal planner, can be extended to support schemas. We then describe the working of the new planner, *SHQPlanner*, and present initial results. Finally, we conclude with pointers to future work.

Bioinformatics: A complex data integration domain

We are interested in data integration in the large and complex domain of bioinformatics to facilitate data analysis. Here, according to a recent survey(Baxevanis2002), there are atleast 335 data sources in early 2002[3] with at least 6-10 sources of similar type (for example, protein, pathway, publication, gene expression data, etc.). In contrast, conventional data integration solutions have dealt with very few sources with little overlap in content. Moreover, there is rich domain information on how results for queries should be obtained and strong user preference for sources (example, one biologist may prefer SWISS-PROT to PIR for protein information due to its affiliation). The completeness (but not correctness) of the results is negotiable in favor of performance and timeliness. The data size can be large (in megabytes or gigabytes). Finally, the user may want to employ the unique native data analysis capabilities of the data sources.

Data analysis in bioinformatics is done in two phases:

1. Identify genes which constitute an interesting regulatory pattern by applying a set of statistical analysis/ clustering methods like hierarchical, k-means, fuzzy and self-organizing feature maps.
2. Identify functional relationships among selected genes based on maximum number of information/ data sources to develop and verify hypotheses. In a sense, the clusters from the first step are characterized by a set of meaningful features from the databases. Additionally, relationship among the genes is predicted and validated/understood.

Karp (Karp1996) has identified the issues in data integration in bioinformatics and alluded to the need of a planning module through what he calls a Relevance Module. An example of a query that the biologist may want to ask in *e2e* is (modified slightly from (Karp1996)):

Q: *Find examples of co-expressed genes that code for enzymes in a single metabolic pathway.*

[2] The fact that some predicates are not true as the result of an action but not others, is governed by the physical knowledge about the world being modeled.

[3] Up from 281 in 2001.

Query Q can be more precisely described as - find a set of genes G from a pathway P where:

- There exists R, the set of all reactions in P
- There exists E, the set of all enzymes that catalyze a reaction in R
- G is the set of genes that encode an enzyme in E, and
- Genes in G are similar in their expression level according to some algorithm.

To answer the query, the system needs to access a protein pathway database like KEGG, the user's gene expression data and a pathway similarity algorithm. We generate the necessary query plan using *SHQPlanner* at the end.

Existing Approaches

A data integration system can be characterized by the amount of transparency it provides to the user. The different types of transparency, in increasing degree of abstraction are:

1. Format transparency or the user not having to know about the data formats supported by a source and the individual ways to access them.
2. Location Transparency or the user not having to know about the location of a piece of information on the source. This gives the impression of a single location for all information while the access details to the source is handled by the system.
3. Schema Transparency or the user not having to know the schemas of individual sources to reconcile the final result. This allows the user to query data across sources in an integrated manner.
4. Source Transparency or the user not having to know if a particular source exists. For example, the user will know that protein information can be obtained from the system but she will not have to know the source from which such an information can be obtained. Source transparency necessitates a domain ontology(Benjamin et al1998; Critchlow et al1998) so that the user interacts with the system using domain concepts and the concepts are in turn reconciled with available sources.

A variety of approaches have been developed for integrated access to heterogeneous data sources in genomics. In the *link-driven federation* approach (e.g., SRS(Etzold & Argos1993), LinkDB(Fujibuchi et al2001)), the user can browse the content of a source and also switch sources using system-provided hyperlinks. These systems provide limited format transparency to the user. In *view integration* (e.g., DiscoveryLink(Haas et al2001), K2/Kleisli(Davidson et al2001)), a virtual global schema in a common data model is created using the source description. Queries on the common data model are then automatically decomposed to source level queries. A variation of view integration is the *warehousing approach* (e.g., GUS(Davidson et al2001), (Widom1995)) where instantiation of the global schema is created, i.e., all data of interest is locally stored and maintained for integrated access. Both view and data integration provide format, location and schema transparency to the user. The holy-grail in data integration is to provide source transparency as well, which leads to *semantic integration*. In the TAMBIS(Goble et al2001) system, a common ontology of molecular biology describes the concepts of interest and data source characteristics are directly mapped to the common concepts. The user interacts in the ontological realm while the system deals with the sources.

While transparency is a desirable property to have in genomics, users may selectively want to lose it to gain control over how the queries are answered and utilize source specific features. The drawback of the existing systems is that while they handle transparency for the user, they do not model the relationships between the applications and the data sources in a formal and general framework. Consequently, even *if available*, they fail to leverage a biologist's (user's) background knowledge about how a query should be answered for her purpose or respect user's intent in the final plan. Since a typical data integration system is interactive, it makes sense to employ additional inputs from the user about query optimizations or preferences of data sources.

A Hierarchical Temporal Planning Solution

The requirements for gathering information in bioinformatics domain lead us to look back at AI Planning for query decomposition[4]. A planning problem P is a 3-tuple $\langle I, G, A\rangle$ where I is the complete description of the initial state, G is the partial description of the goal state, and A is the set of executable (primitive) actions. An action sequence S is a solution to P if S can be executed from I and the resulting state of the world contains G.

A HTN planning problem(Kambhampati et al1998) can be seen as a planning problem where in addition to the primitive actions, the domain contains schemas which represent non-primitive (abstract and non-executable) actions and acceptable rules to reduce non-primitive actions to primitive and other non-primitive actions (hence an hierarchy of actions). All non-primitive actions are eventually reduced to primitive actions so that the resultant plan is executable. The acceptable solutions to a HTN problem not only achieve the top-level goals but can also be parsed in terms of the non-primitive actions that are provided for the top-level goals(Barrett & Weld1994).

Sapa(Do & Kambhampati2001) is a heuristic forward chaining (also known as forward state space) planner that can handle actions with durations, metric resource constraints and temporal deadlines. It starts from the initial state and applies actions as they become applicable taking into account their duration and when (either start or end of the duration) each effect becomes valid. In order to guide its search, Sapa builds a temporal relaxed planning graph, and uses action and resource measures to calculate heuristic distance to the goal.

We present *SHQPlanner* in which *Sapa* is extended by introducing non-primitive actions into the domain actions, A, and providing reduction schemas for them. Moreover, we

[4] An altogether different reason to consider planning in data integration is execution monitoring(Knoblock & Ambite1998) but we do not focus on this here.

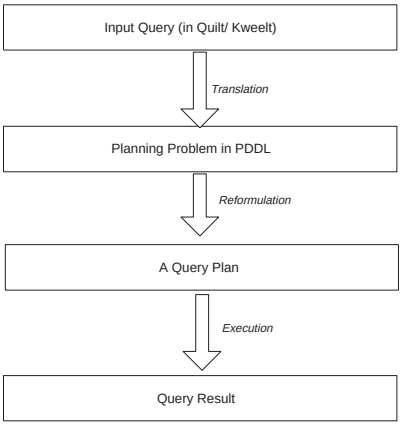


Figure 1: *Stages in processing the XML Kweelt/Quilt query of e2e with SHQPlanner.*

allow users to give preferences over parameter values of actions and schemas. We leverage the duration modeling of actions so that we can reason about query costs. We also modify the heuristic estimates to account for merged actions in the partial plan.

An XML Kweelt(Sahuguet2001) query in *e2e* goes through the stages described in Figure 1. Each query posed by the user corresponds to a new query planning problem, P_i where the initial states I and G states are different but it uses the same domain actions, A. The solution to P_i is executed by the query executor to obtain the final result. In case of failure, the query executor will pose a different planning problem $P_i^{'}$ based on failure and any partial result it already has accumulated.

Schema Specification

The specification of schema in *SHQPlanner* is allowed through the $:schema$ construct which is described in Figure 2. In this, the $:duration$ field records the minimum and maximum duration of each sequence of actions which are permissible while reducing this schema. The $:precondition$ field is a place holder to specify additional preconditions beyond those of the constituent primitive actions which should be true to apply the merged action. The $:effect$ field records the *primary effects*(Kambhampati et al1998) of the schema for which the merged action should be introduced into the plan. This takes care of basic concern in HTN planning that a very complex plan is considered because a non-primitive action is used to achieve secondary effects. The $:method$ field specifies the sequence of actions that are to be used for reductions. If there is more than one sequence, a *choice* delimiter is used.

As an example, in Figure 3, a schema is given which encodes that the primary reason to make use of this schema is to prepare the data. In order to achieve the effect, there is a choice between three sequences of actions corresponding to the alternative reductions available.

```
(:schema ⟨schema-name⟩
    :parameters
        ({⟨?var⟩ - ⟨var-type⟩})
    ;; Static duration
    :duration
        (⟨min-dur⟩, ⟨max-dur⟩)
    ;; Preconditions true at the start
    :precondition
        [(and] {⟨predicate⟩ - (⟨start-time⟩, ⟨end-time⟩)}[)]
    :effect
        [(and] ⟨predicate⟩) - ⟨effect-time⟩ [)]
    :method
        [(choice] [{(sequence] { ⟨action⟩ - ⟨duration⟩ } [)] } [)] )
```

Figure 2: *The format for schema specification. Fields in [ ] are optional while fields in { } are one or more.*

```
(:schema RESULT-PREPARE-SCHEMA
    :parameters
        (?d - data ?q - query)
    ;; Static duration
    :duration
        (st, + st 4)
    ;; Preconditions true at the start
    :precondition
        ()
    :effect
        (prepared_data ?d) - et
    :method
        (choice
            (sequence
                (CLUSTER-DATA ?d)
                (PREPARE-DATA ?q ?d))
            (sequence
                (ALIGN-DATA ?d)
                (PREPARE-DATA ?q ?d))
            (sequence
                (SUMMARIZE-PUBLICATION ?d)
                (PREPARE-DATA ?q ?d))))
```

Figure 3: *An example schema.*

Algorithm: Generate-Merged-Actions
Input: Schema s

Output: M = [] ; set of primitive actions
1. For each action sequence L_i in s, create action M_i
2. M_i.name = Make-unique-name(s)
3. M_i.parameters = s.parameters
4. M_i.precondition = s.precondition
5. M_i.duration = 0
6. M_i.primary = s.effect
7. For each action a_j in L_i
8. If a_j is non-primitive, iterate over Generate-Merged-Actions(a_j)
9. M_i.duration = M_i.duration + a_j.duration
10. If(M_i.duration $\ni$ [s.min_dur, s.max_dur]) continue
11. M_i.parameter = M_i.parameter $\bigcup$ a_j.parameter
12. M_i.precondition= M_i.precondition $\bigcup$ a_j.precondition - M_i.effect
13. M_i.effect = M_i.effect $\bigcup$ a_j.effect
14. End-for
15. M = M $\bigcup$ M_i
16. End-for

Figure 4: *Procedure to produce primitive (merged) actions based on reductions in a schema.*

(:prefers ($\langle Type_1 \rangle$ $\langle value_{11} \rangle$ $\langle value_{12} \rangle$ ...)
($\langle Type_2 \rangle$ $\langle value_{21} \rangle$ $\langle value_{22} \rangle$ ...))

Figure 5: *Specification of value preferences.*

Algorithm: Heuristic-Adjust-Minimal
Input: Partial plan, P
Output: Adjusted heuristic value of the plan

1. $length$ = Sapa-heuristic(P)
2. $solution$ = Sapa-relaxed-plan(P)
3. Foreach merged action, M_i in the $solution$ of relaxed problem
4. Foreach action, a_j constituting the merged action
5. If $a_j \in solution$
6. numMergeRedundantActions ++
7. End-if
8. End-for
9. End-for
10. $length$ = $length$ + numMergeRedundantActions

Figure 6: *Heuristic adjustment for potential non-minimal plans.*

Reduction of a Schema

A HTN planner can transform any non-primitive action into executable actions by recursively applying the available reduction information from the schemas. Since *Sapa* is a forward chaining planner, one can interpret the reduction of a schema eventually into a *sequence* of primitive actions. We use this insight to pre-process (specifically, top-down parse) the schema into a set of *merged actions* that correspond to the sequential execution of the primitive actions in the final reduction. Figure 4 describes a procedure to create such actions in a top-down manner. The merged actions along with the original primitive actions are fed to Sapa for its normal execution.

The merged actions are like primitive actions except that they also record primary effects which will be used during search. Also, if the duration of a sequence lies outside the range of permissible durations, no corresponding merged action is created. The merged actions can be now used by *Sapa* to plan in the regular way.

Specification of Value Preferences

A $:prefers$ construct is introduced in the domain description to allow a user to specify her preference for a variable's values. This information is used while instantiating variables of a particular type in an action. In Figure 5 for example, while considering variables of $Type_1$, $value_{11}$ is the most preferred value followed by $value_{12}$, and so on. Using this mechanism, the user can provide information to the planner like query PIR only if a query on data source SWISS-PROT has failed.

Heuristic Adjustments

Sapa uses heuristics based on actions and resource usage to guide its search. With merged actions also added to the set of original primitive actions, we have to account for the fact that the merged actions signify user intent. We ensure this by increasing the heuristic estimate of a plan by some ϵ for each effect supported by a primitive action in place of an available merged action.

We also want to discourage plans where primitive as well as the merged actions are present to provide the same effect because the plan could possibly be non-minimal. In Figure 6, a procedure is described to capture this requirement by increasing the heuristic value of such a plan by the number of potentially redundant primitive actions.

Soundness and Completeness

A planner is sound if it generates executable plans and it is complete if the planner will find a solution whenever one exists. Since *Sapa* is both sound and complete(Do & Kambhampati2001), the only complication is introduced by the merged actions that are produced as a result of schema reductions.

In *SHQPlanner*, as part of the schema elicitation process, a verification procedure ensures that (a) all the primitive actions mentioned in the schema are declared, and (b) each action sequence in the reductions lead to some executable sequence of primitive actions. Therefore, the merged actions are ensured to be executable and consequently, any plan produced by *SHQPlanner* is sound.

Completeness is guaranteed in *SHQPlanner* as long as no plan without the merged actions are pruned away. We never

⟨A0⟩. 0.0 – RETRIEVE-DATA (stanford geneexp)
:duration 2.0

⟨A1⟩. 0.0 – RETRIEVE-DATA (kegg pathway)
:duration 2.0

⟨A2⟩. 2.0 – EXTRACT-GENES-ENCODING-DATA (pathway)
:duration 1.0

⟨A3⟩. 2.0 – CLUSTER-DATA (geneexp)
:duration 1.0

⟨A4⟩. 3.0 – FIND-COXPRESS-GENES (pathway geneexp)
:duration 3.0

⟨A5⟩. 6.0 – PREPARE-DATA (q1 pathway)
:duration 3.0

Figure 7: *Query plan for the example query Q.*

⟨A0⟩. 0.0 – RETRIEVE-DATA (pir protein)
:duration 2.0

⟨A1⟩. 0.0 – RETRIEVE-DATA (swiss-prot protein)
:duration 2.0

⟨A2⟩. 2.0 – ALIGN-DATA (protein)
:duration 1.0

⟨A3⟩. 3.0 – PREPARE-DATA (q1 protein)
:duration 3.0

⟨A4⟩. 6.0 – VISUALIZE-RESULT (q1 protein)
:duration 1.0

Figure 8: *A query plan without schemas.*

prune such a plan but only penalize it through heuristic adjustments to be further down in the search queue.

Hence, *SHQPlanner* is both sound and complete.

Initial Results

We have incorporated the discussed extensions in *Sapa* (and a few search pruning tricks known to work well in forward chaining algorithms) to build the *SHQPlanner.* Additionally, we have built a small domain describing the actions to query gene expression, protein, publication and pathway sources, and run clustering, sequence alignment and text summarization applications for bioinformatics.

Running *SHQPlanner* as a traditional query planner, we show the query plan returned for the example query, Q. $Stanford$ refers to a public gene expression database while $KEGG$ is a pathway database. Two data sources and a standard clustering application are needed to complete the query. Thus, the query planner does not have to depend on user input for generating a valid query plan.

⟨A0⟩. 0.0 – MERGED:DATA-FETCH-SCHEMA:
%RETRIEVE-DATA%ALIGN-DATA
(pir protein)
:duration 3.0

⟨A1⟩. 0.0 – MERGED:DATA-FETCH-SCHEMA:
%RETRIEVE-DATA%ALIGN-DATA
(swiss-prot protein)
:duration 3.0

⟨A2⟩. 3.0 – PREPARE-DATA (q1 protein)
:duration 3.0

⟨A3⟩. 6.0 – VISUALIZE-RESULT (q1 protein)
:duration 1.0

Figure 9: *A query plan with schemas. The merged actions will be replaced with the corresponding action sequences in a post-processing step.*

Now, suppose the biologist wants to retrieve aligned protein data and see the matched sequences in a viewer. In Figure 8, a query plan is shown for the problem where data is retrieved from the two protein sources and then they are aligned with each other. But if the user only wants to align proteins of each source respectively, it is much simpler to realize the requirement with a suitable schema(*DATA-FETCH-SCHEMA* here). Figure 9 has the resultant plan.

Running the query planner in the bioinformatics domain for a number of user queries has shown that the end users are generally favorable to specifying schemas in order to control the resulting query plan. The main challenge currently is to provide a suitable user interface so that the biologist is free from syntactic hassles. The query plan is generated in a few milliseconds.

Query decomposition as mixed-initiative planning: Queries in biology can take very long time due to sheer size of data to be retrieved. The domain is also characterized by high variability in data sizes – some attributes are in a few hundred bytes while others are in megabytes. The ability of *SHQPlanner* to reason with action durations is very useful for the biologist in exploring alternative query plans based on changing deadlines. Specifically, they want to quickly decide if a line of biological exploration is worthwhile before investing more time in it looking for refined results. In future, we propose to use runtime statistics from the query executor to update duration information.

Conclusion and Future Directions

In this paper, we considered data integration in a complex domain (bioinformatics) where sources can be data repositories as well as applications, and presented a hierarchical temporal planner to perform query. Another characteristics of the domain - bioinformatics - is that the user is a biologist who has specialized knowledge about the domain. Specifically, the user has (partial) schemas or domain knowledge about how a query should be resolved, e.g., a solution for

protein search may be to fetch data, merge results, run a particular application, and show the result. Prior work in hierarchical task network (HTN) planning has approached such problems with partial schemas and actions. The approach considered non-primitive actions and their reduction schemas as part of the domain specification (i.e., the set of available actions) and generalized the usual refinements to handle non-primitive actions. We used similar techniques in *Sapa* to capture user intent. Additionally, we used Sapa's temporal engine to model query costs and prune long plans.

In future, we plan to extend the work in many directions. First, the role of planning can be extended to query execution by allowing for plan monitoring and replanning in the event of failure(Knoblock & Ambite1998). Second, we have to make the bioinformatics domain richer with more primitive actions for additional types of sources. Third, a convenient user interface must be developed to make schema elicitation from the biologist easier. Finally, since *SHQPlanner* itself is a domain independent planner, we want to investigate its usage in more conventional temporal planning domains.

Acknowledgements

I wish to thank Minh Binh Do for providing the *Sapa* code, Sudeshna Adak for useful discussions about the bioinformatics domain, and Subbarao Kambhampati for pointers on planning approaches for data integration and comments on an earlier draft of this paper.

References

Adak, S., Srivastava, B., Kankar, P., and Kurhekar, M. 2001. A Common Data Representation for Organizing and Managing Annotations of Biochip Expression Data. *Unpublished Technical Report.*

Adak, S., Batra, V., Bhardwaj, D., Kamesam, P., Kankar, P., Kurhekar, M., and Srivastava, B. 2002. Bioinformatics for Microarrays. *Submitted for publication.*

Barrett, A., and Weld, D. 1994. Task Decomposition via Plan Parsing. *Proc. AAAI.*

Baxevanis, A. 2001. The Molecular Biology Database Collection: 2002 update. *Numcleic Acids Research, Vol. 30, No. 1*.

Benjamin. V.R., Fensel, D., and Perze, A.G. 1998. Knowledge Management through Ontologies. *Proc. 2nd Int. Conf. PAKM.*

Critchlow, T., Ganesh, M., and Musnick, R. 1998. Automatic Generation of Warehouse Mediators Using an Ontology Engine. *Proc. 5th KRDB Workshop.*

Davidson, S., Crabtree, J., Brunk, B., Schug, J., Tannen, V., Overton, C., and Stoeckert, C. 2001. *K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources.* IBM Systems Journal, March.

Do, B., and Kambhampati, S. 2001. *Sapa: A Domain-Independent Heuristic Metric Temporal Planner.* Proc. European Conference on Planning.

Duschka, O. 1997. Query Optimization Using Local Completeness. *Proc. AAAI*.

Erol, K. 1995. *Hierarchical task network planning: Formalization, Analysis, and Implementation.* Ph.D. thesis, Dept. of Computer Science, Univ. of Maryland, College Park, USA.

Etzold, T., and Argos, P. 1993. *SRS: An Indexing and Retrieval Tool for Flat File Data Libraries.* Computer Application of Biosciences, 9:49-57.

Fujibuchi, W., Goto, S., Migimatsu, H., Uchiyama, I., Ogiwara, A., Akiyama, Y., and Kanehisa, M. 1998. *DBGET/LinkDB: an Integrated Database Retrieval System.* Pacific Sym. Biocomputing, pp 683-694.

Goble, C., Stevens, R., Ng, G., Bechhofer, S., Paton, N., Baker, P., Peim, M., and Brass, A. 2001. *Transparent Access to Multiple Bioinformatics Information Sources.* IBM Systems Journal, Vol. 40, No.2, pp 532-551.

Haas, L., Schwarz, P., Kodali, P., Kotlar, E., Rice, J., and Swope, W. *DiscoveryLink: A system for integrated access to life sciences data sources.* IBM Systems Journal, Volume 40, Number 2.

Kambhampati, S., Mali, A., and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. *Proc. AAAI*.

Kambhampati, S., and Srivastava, B. 1995. Universal Classical Planner: An algorithm for unifying state space and plan space approaches. *In New Trend in AI Planning: EWSP 95, IOS Press*.

Karp, P. 1996. A Strategy for Database Interoperation. *Journal of Computational Biology*.

Knoblock, C. 1995. Planning, Executing, Sensing and Replanning for Information Integration. *Proc. IJCAI-95.*

Levy, A. 1998. Combining Artificial Intelligence and Databases for Data Integration. *At http://citeseer.nj.nec.com*

Knoblock, C., and Ambite, J. 1997. Agents for Information Gathering. *Software Agents, J. Bradshaw (ed), AAAI/MIT Press, Menlo Park, CA; Also at http://citeseer.nj.nec.com/169819.html.*

Kwok, C., and Weld, D. 1996. Planning to Gather Information. *13th AAAI National Conf. on Artificial Intelligence, AAAI/ MIT Press, Portland, Oregon, 32–39.*

Lambrecht, E., and Kambhampati, S. 1996. Planning for Information Integration: A Tutorial Survey. *ASU-CSE-TR 96-017.*

Pottinger, R., and Levy, A. 2000. A Scalable Algorithm for Answering Queries using Views. *Proc. 26th VLDB.*

Qian, X. 1996. Query Folding. *12th Int. Conference on Data Engineering, New Orleans, Louisiana, 48–55.*

Sahuguet, A. 2001. Kweelt. *http://db.cis.upenn.edu/Kweelt/.*

Widom, J. 1995. Research Problems in Data Warehousing. *Proc. 4th CIKM.*

PRUDENT: A Sequential-Decision-Making Framework for Solving Industrial Planning Problems

Wei Zhang

Boeing Phantom Works
P.O. Box 3707, MS 7L-66
Seattle, WA 98124-2207
wei.zhang@boeing.com

Abstract

Planning and control are critical problems in industry. In this paper, we propose a planning framework called PRUDENT to address many common issues and challenges we are facing in industrial applications, including incompletely known world models, uncertainty, and very large problem spaces. This framework considers planning as sequential decision-making and applies integrated planning and learning to develop policies as reactive plans in an MDP-like progressive problem space. Deliberative planning methods are also proposed under this framework. This paper describes the concepts, approach, and methods of the framework.

Introduction

Planning and control are critical problems in industry. At Boeing, we are facing a wide range of problems where effective planning and control are crucial and the key to business success. In manufacturing, we are dealing with highly challenging problems that require integration of the functions from plan generation to execution from low-level, largely automated factory control to high-level enterprise resource planning (ERP) and supply-chain management (SCM). In the autonomous-vehicles business sector, we face challenges in solving a variety of planning and control problems in designing unmanned vehicles for us in air, space, ground, and underwater. In enterprise computing network protection and security, our business must deal with the challenges of building effective intrusion-detection and system-monitoring *policies*—like *universal plans* (Schoppers 1987)—that can ensure the security of a computing environment as well as accurate, timely response to unpredictable events and novel patterns.

While *operator sequencing* is an important family of techniques that can be applied for building plans in these task domains, it does not necessarily address all of the important technical challenges. In a completely deterministic world, it is possible to build a plan perfectly before execution, thus when the plan is executed—following the pre-planned or scheduled sequence of actions—the desired outcome will result. In the real world, however, incompletely foreseen events are often considered normal, thus a useful planning system must be able to know what to do when things go unexpected and for many circumstances must consider such uncertainty as a *regular structure* as opposed to the exception.

This nature is shared by all the problems listed above. To provide more competitive solutions, we take a broader view of planning where the tasks of a planner go inside all the stages of problem solving, including initial planning and possibly many iterations of replanning (interleaved with plan execution) to build and continuously improve plans and problem-solving policies. With this view, we turn our attention to the contents of planning, or plans, as opposed to the activity itself (a one-step task-arrangement activity), thus, the focus of planning can be perfectly described as *sequential decision making*—the process of determining sequences of actions for achieving goals. We refer to this view of planning as *process-based planning* so as to emphasize continuous policy (plan) improvement throughout a whole problem-solving process.

This paper presents a framework for dealing with real-world planning problems from this point of view. The framework is called PRUDENT, short for **P**lanning in **R**egular **U**ncertain **D**ecision-making **EN**vionment**T**, designed for addressing problems with regular uncertain structures across their whole problem space. A major contribution of the PRUDENT framework, from the technical point of view, is the introduction of sequential-decision-making techniques—specifically, *partial-policy reinforcement learning* techniques—to perform both *reactive* planning and *deliberative* planning in a process parallel to plan execution. From the practical standpoint, with integrated planning and learning, PRUDENT provides a promising tool for solving the problems described above. While reactive plans—plans reacting to a sensed environment—are the primary means to act in non-deterministic environments, adding deliberative plans may improve problem-solving capability significantly, particularly when facing problems requiring *timely* response to unpredictable events. A purely reactive plan lacking carefully pre-planned sequences of actions is slow and often fail to proceed when problems occur during sensing and data processing.

The paper is organized as follows. The following section first provides some necessary background for the PRUDENT

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

framework and then describes basic PRUDENT concepts. The main body of the paper describes an approach proposed using partial-policy reinforcement learning for developing reactive plans, world models, and deliberative plans. The paper concludes with a brief summary.

Planning as Sequential Decision Making and PRUDENT

Background

Markov Decision Processes (MDPs), originated in the study of stochastic control (Bellman 1957), is a widely applied, basic model for describing sequential decision making under uncertainty. In general, an MDP can be considered as an extension of the deterministic state-space search model for general problem solving. This extension allows modelling of non-deterministic state transitions, which are described as stochastic processes with *static* probabilistic state transition functions. The model comprises five components: (1) a finite state space $S = \{s_i | i = 1, 2, ..., n\}$, (2) an action space that defines actions that may be taken over a state space: $A = \{A(s) | \forall s \in S\}$ where $A(s)$ is a finite set of actions defined on state s, (3) a probabilistic state transition function $P(s_i, a, s_j)$ describes the probability of making a state transition from any one arbitrary state s_i to a state s_j (which maybe the same) when an action a defined on $A(s_i)$ is taken, (4) a reward function (or cost function) $R(s_i, a, s_j)$ over the problem space that specifies an instant reward that the agent will receive after an action is performed (under a corresponding state transition) , and (5) a discrete time space $T = \{0, 1, 2, ...\}$. Note the form of state-transition functions above says that the possible next states depend and only depend on the current state, independent of the previous ones. This characteristic is called the *Markov property*.

The task for an *agent* in an MDP environment is to determine, for a given future *time horizon* $H \in T$ (where H may be finite or infinite), a *policy* to apply over time that results in the maximal expected total future reward. This policy is referred to as an *optimal policy*. Specifically, a policy specifies an action to be taken in each state. At any state s, taking the action provided by an optimal policy, specified with respect to its time horizon H, guarantees maximizing the expected total future reward in the time frame.

While MDPs provide a powerful way to allow modeling state-space search under uncertainty, they also possess mathematical beauties to allow structured, efficient policy computation. With limited space, we summarize these algorithmic aspects as follows.

- Value function: A value function V of a policy π defines the *value*—the expected total future reward with respect to a time horizon H—of a state $s \in S$ using this policy over all states: $V_H^\pi(s) = E[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1})]$. V_H^π can computed recursively from V_{H-1}^π: $V_H^\pi(s) = \sum_{s' \in S | a = \pi(s)} P(s, a, s')[R(s, a, s') + \gamma V_{H-1}^\pi(s')]$. Here γ, $0 \leq \gamma \leq 1$, is the discounting factor controlling the influence of rewards in the past with a degree of *exponential decay*.
- Value iteration: The value iteration algorithm, or dynamic programming, for computing an optimal policy π^* is developed using *Bellman update* of *optimal value function* V^* (Bellman 1957): $V_H^*(s) := \max_{a \in A(s)} \left(\sum_{s' \in S} P(s, a, s')[R(s, a, s') + \gamma V_{H-1}^*(s')] \right)$. Once the optimal value function (with respect to a time horizon) is computed, an optimal policy can be obtained by executing a *one-step greedy lookahead* search using the optimal value function. This means knowing the V^* is equivalent to knowing a π^* (Note V^* is unique but it may correspond to multiple π^*s).
- Infinite time horizon with discounted rewards: Under the infinite time horizon, $0 \leq \gamma < 1$ should be applied. The optimal value function for $H \to \infty$ converges by value iteration under various conditions. While there are many interesting theoretical convergence results, our interest lies in real-world problems where limited time space is concerned.
- Policy iteration: When H is large, it may be more efficient to use the *policy iteration* algorithm. Policy iteration starts with an arbitrary policy π and then repeats the following policy evaluation-improvement steps: (1) evaluation: compute V^π, and (2) improvement: obtain greedy policy $\pi(s) := \arg\max_{a \in A(s)} \sum_{s' \in S} P(s, a, s')[R(s, a, s') + \gamma V_{H-1}^\pi(s')]$.

In the last decade, the MDP framework has been heavily revisited and studied in AI and machine learning communities, leading to the advances in reinforcement learning (Barto *et al.* 1995, Kaelbling *et al.* 1996, Sutton and Barto 1998) and decision-theoretic planning (Dean *et al.* 1995, Boutilier and Puterman 1995). The PRUDENT framework is developed based on these advances.

PRUDENT Concepts

PRUDENT is designed to address real-world problems that share the following common properties and challenges.

- **Incompletely known world model:** PRUDENT considers real problems where the world model is not completely known but underlying structures of the model exist and these structures may be explored and learned.
- **Uncertainty:** PRUDENT deals with problems where uncertainty is considered normal, possibly appearing throughout a problem space. This makes an MDP-like state-space model a favorable choice. In a quite *static* environment, knowledge of environment can be gained relatively easily by executing a process. This knowledge normally results in a reduced level of uncertainty for a learned model by eliminating unlikely transitions and making other transitions more certain. In a rather *dynamic* environment, however, new problems may occur during execution. This could introduce additional uncertainty into a model.
- **Non-Markov problems:** Problems are not Markovian under a natural view. For example, in a manufacturing process we collect sensor data every second. In the natural representation that uses the original state configurations (based on sensing and other conditions) and a regular time scale (by second), we find it clear that dependencies exist between future states and historical conditions.

The problems we face under a natural view normally are not Markovian.

- **Very large problem space:** Problems are complicated and require use of massive states to describe all the details. Such a large state space makes it impossible to build a complete universal plan. Standard dynamic programming and policy iteration for computing policies are not feasible.
- **Progressive state space:** A *progressive* state space is not an *ergodic* space where any state in the space can be reached from any other with finite steps. States are largely partially ordered. Making moves in a progressive space without a purposeful plan (say following a random walk) is likely to lead an agent from one end of a space to the other (the finish) end. In general, a progressive state space allows inclusion of a relatively small number of loops for modeling often occurred UNDOs and REDOs of a task or a sequence of tasks.

Accordingly, the PRUDENT design is baded on the following key concepts.

- **Planning:** The PRUDENT architecture is built on the MDP-like state-space structure. This makes PRUDENT a reactive planner. A partial-policy reinforcement learning approach is developed for this architecture to incorporate deliberative planning into this reactive-planning based framework. This paper argues that such a design is a natural choice for addressing the type of the problems discussed above.
- **Sensing:** Sensing is a basic requirement for reactive planning. PRUDENT utilizes sensing for three purposes: getting environment state information for a reactive plan, providing possibly useful information for a deliberative plan, and learning to better describe world models.
- **Learning:** The data received from sensing enable learning. Learning can be performed either during real-time or off-line. The task of learning is two-fold: (1) learning to better describe world models under various degrees of world dynamics, from quite static to more dynamic, and (2) coordinating with planners to learn to build and improve plans to act properly and more optimally in an environment.
- **Problem solving:** As a generalized planning system, PRUDENT supports *iterative problem solving*. We refer to a *goal-oriented* task from a start state to a goal (finish) state as a single problem-solving process. This process in PRUDENT supports interleaved planning (including replanning) and execution with incorporated learning functions. Such a process may continue for many iterations, possibly with different start points and different goals and change of conditions.
- **Problem formulation and transformation:** PRUDENT also provides functions for transforming original planning and control tasks into a state-space model, facilitating formulation of an MDP-like problem. A well-formulated problem can avoid many difficulties for planning and learning algorithms. It is important to notice that a non-Markov state space often may be transformed into a Markovian one by using a different state representation. Dependencies between future states and historical conditions may be removed by grouping temporally-dependent states and restructuring a state space using generalized states.

Approach and Methods

PRUDENT planning and learning follow the partial-policy reinforcement learning paradigm. This section first presents some important preparation issues, followed by the major elements of the PRUDENT approach: (1) learning partial policies as reactive planning, (2) learning world models, (3) real-time learning, and (4) planning sequences of actions.

Preparation Considerations

Applying PRUDENT planning first requires formulating an MDP-like problem space, describing states, actions, state-transition relations, and problem objectives in the form of reward function, time scale, and search horizon. We say the PRUDENT problem-space structure is MDP-like because it adopts the same fundamental elements as MDPs.

One major difference between PRUDENT and MDPs is that PRUDENT does not assume it has a complete knowledge of state transitions and its model is learned and updated during execution. Therefore, there is no need to carefully study and hand-engineer the state-transition probabilities at the beginning. An initial state-transition model can be quite rough.

Another difference is that a PRUDENT problem space does not require satisfying the Markov property. However, as an important principle, PRUDENT encourages use of more MDP-like structures whenever possible, maximally removing the dependency between future states and the history. A more MDP-like problem space can make planning and learning much easier.

For many problems it may be quite straightforward to come up with an MDP-like problem space for PRUDENT. But in other cases, various difficulties may be encountered, making it hard to completely remove historical dependencies for a transformed model. Typical problems causing these difficulties include historical dependencies across long-time periods, historical dependencies in variable time scale, incomplete sensing (the world may be partially observable), and incorrect sensing (errors and noise in sensing).

Learning Partial Policies

This function learns a *partial policy* as a reactive plan offline under a fixed state-transition function.

When building a plan for a task involving in a very large problem space, one basic strategy is *divide-and-conquer*. Set a number of sub-goals in an order (a partial order) and accomplish these sub-goals in the defined order. PRUDENT learns partial policies using the same strategy. Table 1 shows the procedure.

The algorithm is a modified value iteration procedure, which learns a partial value function to obtain a partial policy—the policy greedy to this partial value function. It is

Table 1: Partial Policy Learning Algorithm

procedure PARTIALPOLICYLEARNER(S,A,P,R,G,s_0,σ)
inputs:
$S=\{s_i|i=1,2,...,n\}$ // *a finite state space*
$A=\{A(s)|\forall s\in S\}$ // *an action space*
$P=\{P(s,a,s')|\forall s\in S\&\forall a\in A(s)\}$ // *a state-transition function*
$R=\{R(s,a,s')|\forall s\in S\&\forall a\in A(s)\}$ // *a reward function*
$G=\{S_g,O_g\}$ // $S_g=\{g_i|i=1,2,...,m\}\subseteq S$ *is a set of goals* // *and* O_g *is a partial order of the goals*
s_0 // *a start state*
σ // *a set of scope rules*

INITVALUE() // *initialize value function* $V(s):=0,\forall s\in S$
repeat until (STOPPINGRULES()) // *repeat until stopping rules are satisfied*
for all $g\in S_g$ // *select g backward according to* O_g
BACKWARDUPDATE(g,S,A,P,R,σ) // *perform backward updates from g*
FORWARDUPDATE(s_0,S,A,P,R,σ) // *perform forward updates from* s_0
for all $g\in S_g$ // *select g forward according to* O_g
FORWARDUPDATE(g,S,A,P,R,σ) // *perform forward updates from g*
end repeat

end procedure

designed for goal-oriented problems with progressive problem spaces. For problems with this structure, rewards (or major rewards) are typically received when a goal or sub-goal is achieved. In real applications, Tesauro's backgammon programs applied zero rewards on all states until they reach the end of a game when the agent receives reward 1 if it wins or -1 if it loses (Tesauro 1992). In reinforcement learning applications for space shuttle processing for NASA, the program presented in (Zhang and Dietterich 1995) applies a measure of the quality of a schedule as a reward when a final feasible solution (a sub-goal) is obtained, while for other states, all operations (repairing steps for modifying and improving a current schedule) are assessed with a constant small penalty to encourage developing feasible solutions with the smallest number of repairs.

The procedure works as follows. Initially, the value for each state is set to 0. The main procedure updates values following a backward-forward update process iteratively until a stopping rule encoded in the function STOPPINGRULES is satisfied. Ideally, the procedure stops when the value function converges. Other rules may be included to allow a process to stop at other conditions, such as running out of time.

Each iteration first updates values backward. The backward update process starts with a final goal and then works successively backward on the rest of the goals according to the provided order of the goals O_g. When the BACKWARDUPDATE subroutine is called for selected goal state g, it starts with state $s:=g$ and updates its value,

$$V(s):=\max_{a\in A(s)}\left(\sum_{s'\in S}P(s,a,s')[R(s,a,s')+\gamma V(s')]\right).$$

Then it selects all state $s^{\#}$ that can directly lead to s with a single action ($P(s^{\#},a,s)>0$) and for all $s:=s^{\#}$ updates $v(s)$ using the same formula. This backward-update step proceeds until a scope rule in σ is satisfied. σ works as a set of heuristic rules. If it is possible to estimate the pariwise distance between all successive goals as well as the distance between s_0 and the first set of sub-goals, one possibly good σ rule is "set update steps to half of the largest distance". This rule expects that for any pair of successive goals, $V(s)$ can be computed by a backward process in the second half of the space and for the first half the values can be computed by a forward process.

After a backward update process is finished, in the same iteration, a forward update process starts. Forward update starts with s_0 and works forward through the goal states. Each FORWARDUPDATE call starts from the first state s (s_0 or a sub-goal g) and finds all possible next states s' of s and puts s' into a pool. Then for all s in the pool, it pops s and repeats the same step, putting all possible next states of s into the pool. This state-space growth process continues until a scope rule in σ is satisfied. All processed states are selected. After the state space is determined, FORWARDUPDATE sorts the states according to their current values, from the largest to the smallest, then updates values for all states using this order. This allows efficient use of updated values on the states that have been connected to goal states, because only states connected to certain goals can receive large values.

Learning World Models

This function learns state-transition functions in the form of world models. Learning utilizes existing incomplete models to try to improve them to better describe the world.

Learning state-transition functions is based on observations of state transitions made during system execution. PRUDENT employs the following three sets of learning parameters for learning and improving world models.

- Degree of environment dynamics. In a quite static environment, historical observations over a long period of time can be employed. In a quite dynamic environment, however, only data collected in a short history is used. This set of parameters determines the time period when data is selected for learning.
- Degree of observation reliability. This addresses real problems with incomplete sensing and incorrect sensing. Observations are carefully reviewed and selected. This set of parameters controls selection of observations individually.
- Conditions of variations. Variations of state transitions and their conditions are carefully studied. This attempts to find conditions for non-deterministic state transitions. If possible, states may be redefined by adding more conditions to existing specified states and splitting them. This may effectively remove many uncertain state transitions. This set of parameters determines if states need to be restructured.

Once correct, relevant data are selected, updating state transition probabilities is straightforward. PRUDENT applies

the standard maximum-likelihood method:

$$p(x,a,y) = \frac{n^a_{xy}}{n^a_x},$$

where n^a_{xy} is the number of cases that the environment switches to state y after action a is taken at state x and n^a_x is the number of cases in the selected observations where action a is taken at state x.

Restructuring states is a difficult task. Presenting techniques for accomplishing this task exceeds the scope of this paper.

Real-Time Learning

This function is developed for applications with fairly poor understanding of environments or quite dynamic environments. In such environments, since the current policy and world model are not reliable and often fail, adjusting them in real time by making use of current experience immediately is considered as a wise choice.

PRUDENT applies the *real-time dynamic programming* paradigm, or RTDP, developed by Barto et al (Barto *et al*. 1995). This employs trajectory-based reinforcement learning to learn partial policies. For learning world models in real time, PRUDENT applies the maximum-likelihood method described above as well as the *adaptive real-time dynamic programming* method developed by Barto et al as well.

Planning Sequences of Actions

This function attempts to develop deliberative plans based on the state-space based reactive planning paradigm. It is developed for applications with quite static environments where there are various deterministic sub-problems or substructures or knowledge can be learned to allow removal of various uncertain structures in a world model.

There are basically two conditions preventing making a deliberative plan from the MDPs based reactive planning framework: uncertainty and the needs for sensing. These two conditions are related. When state transitions are not deterministic, sensing becomes necessary in execution because of the need for determining states. And this is true vice versa.

While deliberative planning for an MDP-like environment may not be applicable in general, special problems in such an environment often exist that make building such a plan important. Here are three families of such problems.

- Planning for worst possibilities. For example, playing chess is a non-deterministic process. For quick response, it is important for an agent to have a deliberative plan to play against opponent's best moves. Planning for the worst possibility with a single worst case is a deterministic problem. In this case, the sequence of actions can be pre-determined without sensing.
- Planning for situations where the same sequence of actions is often applied. This involves part of a space where state transitions are quite deterministic. Making a deliberative plan can help fast execution by possibly avoiding most expensive step-by-step sensing and data processing activities.
- Planning for situations where sensing often fails. In this case, a deliberative plan can provide a backup plan that doesn't depend on sensing, replacing reactive plans.

PRUDENT considers developing deliberative plans for these three families of problems. Additional steps are added to the partial-policy reinforcement learning methods presented above to allow learning sequences of actions to come up with a deliberative plan. PRUDENT basically provides two methods. The first method returns sequences of actions for dealing with worst possibilities. The second method returns all sequences of actions corresponding to all possible trajectories for a quite deterministic sub-space. Each returned sequence of actions employs the greedy policy to the learned values of the states along a trajectory. If too many trajectories are generated, a useful parameter for controlling the number is selecting only the k most-likely trajectories (e.g., for planning a chess game, consider the trajectories that your opponent is most likely to adopt).

Selecting the k most-likely trajectories for PRUDENT is straightforward, because state transition probabilities are available. PRUDENT employs a lookahead parameter κ (usually $\kappa \leq 5$) to deal with possible combinatorial explosions. In the lookahead region, it performs a κ-step exhaustive search and computes the joint probability of state transitions for each of the returned trajectories. After κ-step trajectories are generated, the method extends each of them by performing 1-step lookahead greedy search, returning a single "most-likely" trajectory (in terms of the greedy heuristic) for each trajectory length, $\kappa+1, \kappa+2, ..., N$ (N is a given limit for the length). Finally, the k most-likely trajectories are returned from the generated pool. Since longer trajectories result in smaller joint probabilities, we compare trajectories by grouping them by the length. When comparing trajectories of different lengths, we use a simple normalization method that multiplies the likelihood value for an n-step trajectory by 2^n.

Summary

In summary, this paper proposed the PRUDENT planning framework to address many common issues and challenges we are facing in industrial applications, including incompletely known world models, uncertainty, and very large problem spaces. This framework considers planning as sequential decision-making and applies integrated planning and learning to develop policies as reactive plans in an MDP-like progressive problem space. Deliberative planning methods are also proposed under this framework.

Application of this framework to real-world problems is in practice. Our practices are conducted mainly for the problems present in three domains: manufacturing, autonomous systems, and security and network management. With increased capability of collecting massive data from domain processes, opportunities for applying integrated planning and learning are increasingly large.

This paper is a work-in-progress. We expect to release part of our application results in public in a near future.

References

A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

C. Boutilier and M. L. Puterman. Process-oriented planning and average-reward optimality. In *IJCAI-95*, pages 1096–1103, 1995.

T. L. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 1-2(76):35–74, 1995.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of AI Research*, 4, 1996.

M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

R. S. Sutton and A. G. Barto. *Introdution to Reinforcement Learning*. MIT Press, 1998.

G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–277, 1992.

W. Zhang and T. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI-95*, pages 1114–1120, 1995.

Towards Comprehensive Computational Models for Plan-based Control of Autonomous Robots

Michael Beetz
Munich University of Technology
Department of Computer Science IX
Orleanstr. 34
D-81667 Munich, Germany

Abstract

In this paper we present an overview of recent developments in the plan-based control of autonomous robots. We identify computational principles that enable autonomous robots to accomplish complex, diverse, and dynamically changing tasks in challenging environments. These principles include plan-based high-level control, probabilistic reasoning, plan transformation, and context and resource-adaptive reasoning. We will argue that the development of comprehensive and integrated computational models of plan-based control requires us to consider different aspects of plan-based control — plan representation, reasoning, execution, and learning — together and not in isolation. This integrated approach enables us to exploit synergies between the different aspects and thereby come up with simpler and more powerful computational models.

In the second part of the paper we describe *Structured Reactive Controllers (SRCs)*, our own approach to the development of a comprehensive computational model for the plan-based control of robotic agents. We show how the principles, described in the first part of the paper, are incorporated into the SRCs and summarize results of several long-term experiments that demonstrate the practicality of SRCs.

Introduction

In recent years, autonomous robots, including XAVIER, MARTHA (AFH+98), RHINO (BCF+00; BAB+01), MINERVA, and REMOTE AGENT, have shown impressive performance in longterm demonstrations. In NASA's Deep Space program, for example, an autonomous spacecraft controller, called the *Remote Agent* (MNPW98), has autonomously performed a scientific experiment in space. At Carnegie Mellon University XAVIER (SGH+97), another autonomous mobile robot, has navigated through an office environment for more than a year, allowing people to issue navigation commands and monitor their execution via the Internet. In 1998, MINERVA (TBB+00) acted for thirteen days as a museum tourguide in the Smithsonian Museum, and led several thousand people through an exhibition.

These autonomous robots have in common that they perform plan-based control in order to achieve better problem-solving competence. In the plan-based approach robots generate control actions by maintaining and executing a plan that is effective and has a high expected utility with respect to the robots' current goals and beliefs. Plans are robot control programs that a robot cannot only execute but also reason about and manipulate (McD92a). Thus a plan-based controller is able to manage and adapt the robot's intended course of action — the plan — while executing it and can thereby better achieve complex and changing tasks. The plans used for autonomous robot control are often reactive plans, that is they specify how the robots are to respond in terms of low-level control actions to continually arriving sensory data in order to accomplish their objectives. The use of plans enables these robots to flexibly interleave complex and interacting tasks, exploit opportunities, quickly plan their courses of action, and, if necessary, revise their intended activities.

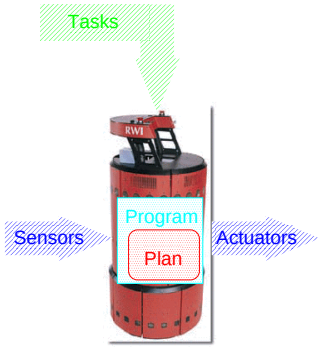


Figure 1: Plan-based control of robotic agents. The control program specifies how the robot is to respond to sensory input to accomplish its task. The plan is the part of the control program that the robot explicitly reasons about and manipulates.

To be reliable and efficient, autonomous robots must flexibly interleave their tasks and quickly adapt their courses of action to changing circumstances. Recomputing the best possible course of action whenever some aspect of the robot's situation changes is not feasible but can often be made so if the robots' controllers explicitly manage the robots' beliefs and current goals and revise their plans accordingly. The use of plans helps to mitigate this situation in at least two ways. First, it decouples computationally intensive control decisions from the time pressure that dom-

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

inates the feedback loops. Precomputed control decisions need to be reconsidered only if the conditions that justify the decisons change. Second, plans can be used to focus the search for appropriate control decisions. The can neglect control decisions that are incompatible with its intended plan of action.

In the remainder of this paper we proceed as follows. In the first part, we describe principles and building blocks of computational models for plan-based control. In the second part, we will then outline our initial steps towards such a comprehensive computational model that contains the building blocks introduced in the first part.

Principles of Plan-based Control

Plans in plan-based control have two roles. They are both executable prescriptions that can be interpreted by the robot to accomplish its jobs and syntactic objects that can be synthesized and revised by the robot to meet the robot's criterion of utility. Besides having means for representing plans, plan-based controllers must also be equipped with tools that enable planning processes to (1) project what might happen when a robot controller gets executed and return the result as an execution scenario; (2) infer what might be wrong with a robot controller given an execution scenario; and (3) perform complex revisions on robot controllers.

Let us now consider some of the key issues in the plan-based control of robotic agents: *dynamic system perspective, probabilistic reasoning, symbol grounding*, and *context and resource-adaptive reasoning.*

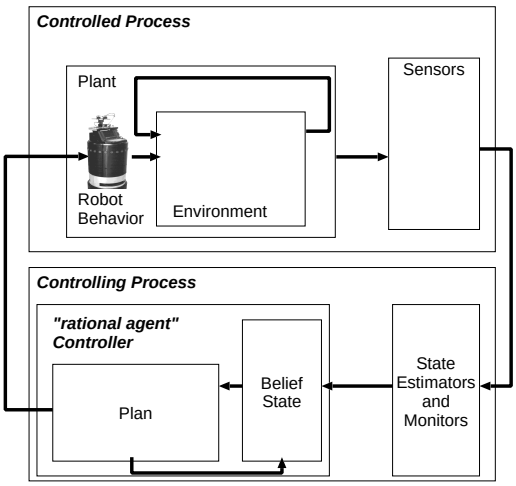


Figure 2: Block diagram of our dynamic system model for autonomous robot control. Processes are depicted as boxes and interactions as arcs.

Dynamic System Perspective. Since flexibility and responsiveness to changing situations are important characteristics of the robot behavior, we use ***dynamic systems*** as the primary abstract model for programming the operation of the integrated plan-based controller (see figure 2). In this model, the state of the world evolves through the interaction of two processes: the controlling process – the robot's control system – and the controlled process, which comprises events in the environment, physical movements of the robot and sensing operations. For complex dynamic systems, it is often useful to further decompose the controlled process into an environment and a sensing process. The environment process changes the world state and the sensing process maps the world state into the sensor data received by the robot. This suggests making a similar decomposition of the controlling process into state estimation and action generation processes. State estimation processes compute the robot's beliefs about the state of the controlled system. Auxiliary monitoring processes signal system states for which the controlling process is waiting. An action generation process specifies the control signals supplied to the controlled process as a response to the estimated system state.

The main consequence of this model is that robot action plans must control concurrent and continuous processes both flexibly and reliably.

Probabilistic Reasoning. Probabilistic reasoning is a key technique employed in the control of autonomous robots. Probabilistic reasoning is employed in a number of different ways.

First, plan generation and revision methods compute plans that have a probability of achieving a given goal with a probability higher than a specified threshold or they compute plans with the best expected cost benefit trade-off (BDH98; KHW95; DHW94). To employ such probabilistic planning techniques actions are represented through probabilistic effect models and the planning techniques consider probability distributions over world states rather than the states themselves. The advantage of these techniques is that they can properly handle the uncertainty caused by incomplete knowledge and inaccurate and unreliable sensors and the uncertainty resulting from non-deterministic action effects. The main problem associated with these techniques are the computational cost associated with the application of these techniques.

The second area in plan-based control where probabilistic reasoning techniques are heavily used is the interpretation of sensor data acquired by the robots' sensors (Thr00). The plan-based high-level control of robotic agents is founded on abstract perceptions of the current state of objects, the robot, and its environment. In order to derive such abstract perceptions from local and inaccurate sensors robustly, plan-based controllers often employ probabilistic state estimation techniques (SB01). The state estimators maintain the probability densities for the states of objects over time. Whenever state information is requested by the planning component, they provide the most likely state of the objects.

The probability density of an object's state conditioned on the sensor measurements received so far contains all the information which is available about the object. Based on this density, one is not only able to determine the most likely state of the object, but one can also derive even more meaningful statistics like the variance and entropy of the current estimate. In this way, the high-level system is able to reason about the reliability of an estimate.

A third application field of probabilistic reasoning is learning. Probabilistic reasoning techniques enable robots

to learn symbolic actions, probabilistic action models, and competent action selection strategies from experience.

Symbol Grounding. One of the key difficulties in the application of plan-based control techniques to object manipulation tasks is the symbol grounding or anchoring problem. In most plan representations constants used in the instantiations of plan steps denote objects in the world. This is a crude oversimplification because robots often do not have direct physical access to the objects themselves. Rather the control systems must use object descriptions that are computed from sensor data in order to manipulate objects. The use of object descriptions rather than objects to instantiate manipulation actions yields problems such as ambiguous, inaccurate, and invalid object descriptions. Powerful computational models of plan-based control must therefore have much more expressive representational means to make these problems transparent to the planning techniques. Several researchers (Fir89; McD90; CS00) have developed techniques to incorporate object descriptions into plan-based control.

Plan Transformation. Another key issue in the plan-based control of robots, in particular for those robots that are to act in dynamic and complex environments, is the fast formation and adaptation of plans. A very common idea for achieving fast plan formation is the idea of a *plan library*, a collection of canned plans for achieving standard tasks in standard situations (McD92b). However, such plans cannot be assumed to execute optimally. In a situation where an unexpected opportunity presents itself during the execution of the robot's tasks, for example, a canned plan will have trouble testing for the subtle consequences that might be implied by an alteration to its current plan. The decision criteria to take or ignore such opportunities must typically be hardwired into the canned plans when the plan library is built.

An alternative is to equip a robot with *self-adapting plans*, which carry out plans with the constraint that, whenever a specific belief of the robot changes, a runtime plan adaptation process is triggered. Upon being triggered, the adaptors decide whether plan revisions are necessary and, if so, perform them. Plan adaptation processes are specified explicitly, modularly, and transparently and are implemented using declarative plan transformation rules.

Context and Resource-adaptive Operation. To make its control decisions in a timely manner the plan-based controller applies various resource-adaptive inference methods (Zil96). These enable the controller to trade off accuracy and the risk of making wrong decisions against the computational resources consumed to arrive at those decisions. Moreover, the results of the resource-adaptive reasoning are employed to adapt the execution modes of the process in response to the robot's context (BACM98).

Building Blocks of Plan-based Control

The building blocks of plan-based control are the representation of plans, the execution of plans, various forms of automatic learning, and reasoning about plans, including plan generation and transformation, and teleological, causal, and temporal reasoning.

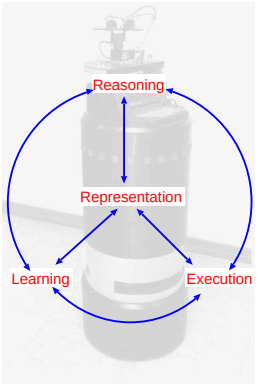


Figure 3: The main components of plan-based control are plan representation, execution, learning, and reasoning and their interactions.

But before we dive in and discuss the building blocks of modern plan-based control models let us first get an intuition of how traditional robot planning techniques function. Most of these techniques are based on the problem-space hypothesis (New90): they assume problems can be adequately stated using a state space and a set of discrete and atomic actions that transform states to successor states. A solution is an action sequence that transforms any situation satisfying a given initial state description into another state that satisfies the given goal. Plan generation is the key inferential task in this problem-solving paradigm.

As a consequence, representational means are primarily designed to simplify plan generation from first principles. Problem space plans are typically used in layered architectures (BFG+97), which run planning and execution at different levels of abstraction and time scales. In these approaches planning processes use models that are too abstract for predicting all consequences of the decisions they make and planning processes cannot exploit the control structures provided by the lower layer. Therefore they lack appropriate means for specifying flexible and reliable behavior and plans can only provide guidelines for task achievement.

Contrary to the plan space approach, plan-based control of robotic agents takes the stand that there is a number of inference tasks necessary for the control of an autonomous robot that are equally important. These inference tasks include ones that enable the competent execution of given plans, ones that allow for learning plans and other aspects of plan-based control, and various reasoning tasks, which comprise the generation and assessment of alternative plans, monitoring the execution of a plan, and failure recovery.

These different inference tasks are performed on a common data structure: the plan. Consequently, the key design issues of plan-based control techniques are representational and inferential adequacy and inferential and acquisitional efficiency as key criteria for designing domain knowledge representations (RK91). Transferring these notions to plan-based control, we consider the representational adequacy of plan representations to be their ability to specify the neces-

sary control patterns and the intentions of the robots. Inferential adequacy is the ability to infer information necessary for dynamically managing, adjusting, and adapting the intended plan during its execution. Inferential efficiency is concerned with the time resources that are required for plan management. Finally, acquisitional efficiency systems is the degree to which they support the acquisition of new plan schemata and planning knowledge.

To perform the necessary reasoning tasks the plan management mechanisms must be equipped with inference techniques to infer the purpose of subplans, find subplans with a particular purpose, automatically generate a plan that can achieve some goal, determine flaws in the behavior that is caused by subplans, and estimate how good the behavior caused by a subplan is with respect to the robot's utility model. Pollack and Horty (PH99) stress the point that maintaining an appropriate and working plan requires the robot to perform various kinds of plan management operations including plan generation, plan elaboration, commitment management, environment monitoring, model- and diagnosis-based plan repair, and plan failure prediction.

It does not suffice that plan management mechanisms can merely perform these inference techniques but they have to perform them fast. The generation of effective goal-directed behavior in settings where the robots lack perfect knowledge about the environment and the outcomes of actions and environments are complex and dynamic, requires robots to maintain appropriate plans during their activity. They cannot afford to entirely replan their intended course of action every time their beliefs change.

To specify competent problem-solving behavior the plans that are reasoned about and manipulated must have the expressiveness of reactive plan languages. In addition to being capable of producing flexible and reliable behavior, the syntactic structure of plans should mirror the control patterns that cause the robot's behavior — they should be realistic models of how the robot achieves its intentions. Plans cannot abstract away from the fact that they generate concurrent, event-driven control processes without the robot losing the capability to predict and forestall many kinds of plan execution failures. A representationally adequate plan representation for robotic agents must also support the control and proper use of the robot's different mechanisms for perception, deliberation, action, and communication. The full exploitation of the robot's different mechanisms requires mechanism-specific control patterns. Control patterns that allow for effective image processing differ from those needed for flexible communication, which in turn differ from those that enable reliable and fast navigation. To fully exploit the robot's different mechanisms, their control must be transparently and explicitly represented as part of the robot's plans. The explicit representation of mechanism control enables the robot to apply the same kinds of planning and learning techniques to all mechanisms and their interaction.

The defining characteristic of plan-based control is that these issues are considered together: plan representation and the different inference tasks are not studied in isolation but in conjunction with the other inference tasks. The advantage of this approach is that we can exploit synergies between the different aspects of plan-based control.

Plan management capabilities simplify the plan execution problem because programmers do not have to design plans that deal with all contingencies. Rather plans can be automatically adapted at execution time when the particular circumstances under which the plan has to work are known. Plan execution mechanisms can also employ reasoning mechanisms in order to get a broader coverage of problem-solving situations. The REMOTE AGENT, for example, employs propositional reasoning to derive the most appropriate actions to achieve the respective immediate goals (WN97; NW97). On the other side, competent plan execution capabilities free the plan management mechanism from reasoning through all details. Reasoning techniques such as diagnostic and teleological reasoning are employed in transformational learning techniques in order to perform better informed learning decisions and thereby speed up the learning process (BB00). Skill learning mechanisms have also been applied to the problem of learning effective plan revision methods (Sus77). There is also a strong interaction between the learning and execution mechanisms in plan-based control. Learning mechanisms are used to adapt execution plans in order to increase their performance . Competent execution mechanisms enable the learning mechanisms to focus on strategical aspects of problem-solving tasks.

Structured Reactive Controllers: a Computational Model of Plan-based Control

After having described the general components of computational models of plan-based control I want to give you now a brief overview of our own approach to the development of such integrated computational models. The robot controllers that realize this computational model are called ***Structured Reactive Controllers (SRCs)*** (Bee01). Structured Reactive Controllers are self-adapting plans that specify concurrent reactive behavior. They revise themselves during the execution of specified user commands in order to exploit opportunities and avoid predictable problems. They are also capable of experience-based learning.

Structured Reactive Controllers use a very expressive plan language, called RPL (McD91), and a number of software tools for predicting the effects of executing plans, for teleological and causal reasoning about plans, for revising plans during their execution, and for automatically learning routine plans.

Given a set of jobs, an SRC concurrently executes the default routines for each individual job. These routine activities are general and flexible and work well in standard situations. They can cope well with partly unknown and changing environments, run concurrently, handle interrupts, and control robots without assistance over extended periods. For standard situations, the execution of these routine activities causes the robot to exhibit an appropriate behaviour while achieving its purpose. While it executes routine activities, the SRC also tries to determine whether its routines might interfere with each other and monitors robot operation for non-standard situations. If one is found, it will try to antici-

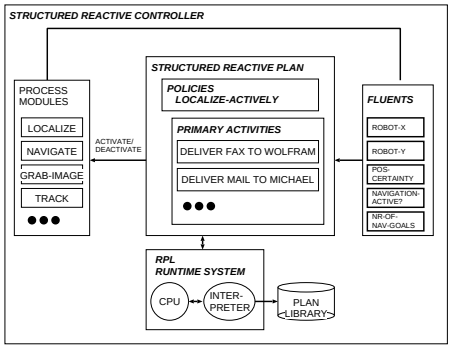


Figure 4: Components of a structured reactive controller. The structured reactive plan specifies how the robot responds to changes of its fluents, registers that are asynchronously set by the sensing processes. The interpretation of the structured reactive plan results in the activation, parameterization, and deactivation of process modules that execute and monitor the physical continuous control processes.

pate behaviour flaws by predicting how its routine activities might work in these non-standard situations. If necessary, it revises its routines to make them robust for this kind of situation. Finally, it integrates the proposed revisions into the activities it is pursuing.

Transformational Planning of Concurrent Reactive Plans. Consider the following plan adaptor, which illustrates the planning techniques employed by SRCs.

With plan adaptor Whenever the robot detects an open door that was assumed to be closed
if this situation ***is an opportunity***
then it ***changes its course of action*** to make use of the opportunity

Concurrent reactive plan

The plan adaptor is triggered by a change of its belief about an door being open or closed. Upon being triggered the adaptor decides whether a change in the intended course of activity is suitable and if so performs it. The process of plan adaptation is realized through transformational planning (McD92b; Bee00).

Transformational planning is implemented as a search in plan space. A node in the space is a proposed plan; the initial node is the default plan created using the plan library. A step in the space requires three phases. First, a plan adaptor *projects* a plan to generate sample execution scenarios for it. Then, in the *criticism* phase, a plan adaptor examines these execution scenarios to estimate how good the plan is and to predict possible plan failures. It diagnoses the projected plan failures by classifying them in a taxonomy of failure models. The failure models serve as indices into a set of transformation rules that are applied in the third phase, *revision*, to produce new versions of the plan that are, we hope, improvements.

Prediction in Structured Reactive Controllers Temporal projection, the process of predicting what will happen when a robot executes its plan, is essential for many robots to successfully plan courses of action. To be able to project their plans, robots must have causal models that represent the effects of their actions. These causal models should be sufficiently realistic to predict the behavior generated by modern autonomous robot controllers accurately enough to foresee a significant range of real execution problems. This can be achieved if action models reflect the facts that physical robot actions cause continuous change; that controllers are reactive systems; that the robot is executing multiple physical and sensing actions; and that the robot is uncertain about the effects of its actions and the state of the environment.

The problem of using such realistic action models is obvious. Nontrivial concurrent plans for controlling robots reliably are very complex. There are usually several control processes active. Many more are dormant, waiting for conditions that trigger their execution. The branching factors for possible future states — not to mention the distribution of execution scenarios that they might generate — are immense. The accurate computation of this probability distribution is prohibitively expensive in terms of computational resources.

Learning Symbolic Robot Plans. We have already stressed the importance of representing the plans that the robot has committed to execute explicitly as a means of economically using the limited computational resources for flexible task execution and effective action planning. However, this raises the question of how such plans can be obtained. Many autonomous mobile robots consider navigation as a Markov decision problem. They model the navigation behavior as a finite state automaton in which navigation actions cause stochastic state transitions. The robot is rewarded for reaching its destination quickly and reliably. A solution for such problems is a mapping from states to actions that maximises the accumulated reward. Such state-action mappings are inappropriate for teleological and diagnostic reasoning, which are necessary to adapt quickly to changing circumstances and quickly respond to exceptional situations.

We have therefore developed XFRMLEARN (BB00), a learning component that builds up explicit symbolic navigation plans automatically. Given a navigation task, XFRMLEARN learns to structure continuous navigation behaviour and represents the learned structure as compact and transparent plans. The structured plans are obtained by starting with monolithic default plans that are optimized for average performance and adding subplans to improve the navigation performance for the given task.

XFRMLEARN's learning algorithm works as follows. XFRMLEARN starts with a default plan that transforms a navigation problem into an MDP problem and passes the MDP problem to RHINO's navigation system. After RHINO's path planner has determined the navigation policy the navigation system activates the collision avoidance module for the execution of the resulting policy. XFRMLEARN records the resulting navigation behaviour and looks for stretches of behaviour that could be possibly improved. XFRMLEARN

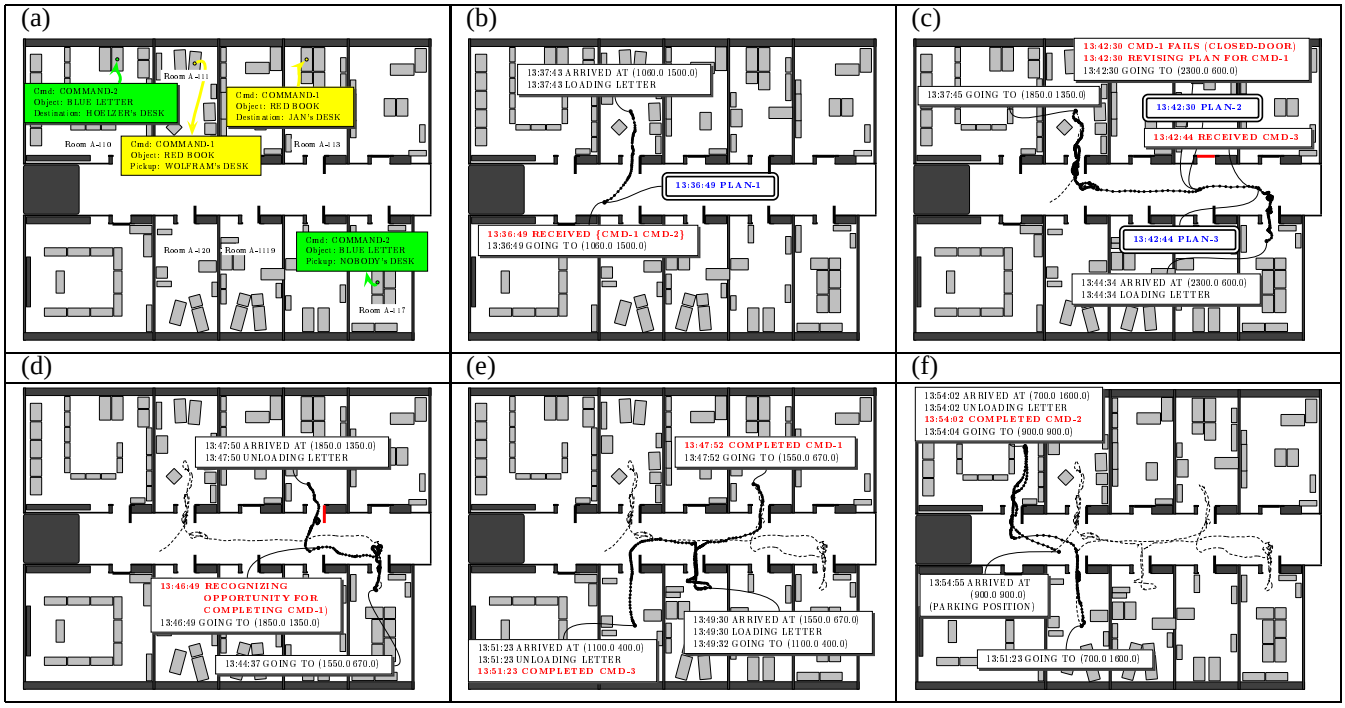


Figure 6: Execution trace for a delivery tour. RHINO receives two commands 6(a). Upon receiving the two commands the SRC puts plans for the commands into the plan, computes an appropriate schedule, and installs it. It also adds a control process that monitors that the rooms it must enter are open. The order of the delivery steps are that RHINO starts with picking up the book (Fig. 6(b)) and delivering it in A-113. After RHINO has left room A-111, it notices that room A-113 is closed (Fig. 6(c)). Because RHINO cannot complete the delivery of the book the SRC revises the plan by transforming the completion of the delivery into an opportunity. RHINO receives a third command which is integrated into the current schedule (Fig. 6(d)). As it passes room A-113 on its way to A-119 it notices that the door is now open and takes the opportunity to complete the first command (Fig. 6(d)). After that it completes the remaining steps as planned (Fig. 6(e-f).

then tries to explain the improvable behaviour stretches using causal knowledge and its knowledge about the environment. These explanations are then used to index promising plan revision methods that introduce and modify subplans. The revisions are subsequently tested in a series of experiments to decide whether they are likely to improve the navigation behaviour. Successful subplans are incorporated into the symbolic plan. An learning session is shown in figure 5.

Using this algorithm can autonomously learn compact and well-structured symbolic navigation plans by using MDP navigation policies as default plans and repeatedly inserting subplans into the plans that significantly improve the navigation performance. The plans learned by XFRMLEARN support action planning and opportunistic task execution by providing plan-based controllers with subplans such as traverse a particular narrow passage or an open area. More specifically, navigation plans (1) can generate qualitative events from continuous behaviour, such as entering a narrow passage; (2) support online adaptation of the navigation behaviour (drive more carefully while traversing a particular narrow passage) (Bee99), and (3) allow for compact and realistic symbolic predictions of continuous, sensor-driven behaviour (BG00).

Long-term Demonstrations

This section describes several experiments (figure 7) that evaluate the reliability and flexibility of the RHINO system and the possible performance gains that it can achieve.

The flexibility and reliability of runtime plan management and plan transformation has been extensively tested in a museum tourguide application. The robot's purpose was to guide people through a museum, explaining the exhibits to be seen along the robot's route. MINERVA (figure 7(a)) operated in the "Smithsonian Museum" in Washington for a period of thirteen days (TBB$^+$99). It employed an SRC as its high-level controller. During its period of operation, it was in service for more than 94 hours, completed 620 tours, showed 2668 exhibits, and travelled over a distance of more than 44 kilometers. The SRC directed MINERVA's course of action in a feedback loop that was carried out more than three times a second. MINERVA used plan adaptors for the installment of new commands, the deletion of completed plans, and for tour scheduling. MINERVA made about 3200 execution time plan transformations while performing its tourguide job. MINERVA's plan-based controller differs from RHINO's only with respect to its top-level plans in plan library and some of the plan adaptors that are used.

In another experiment we have evaluated the capabilities

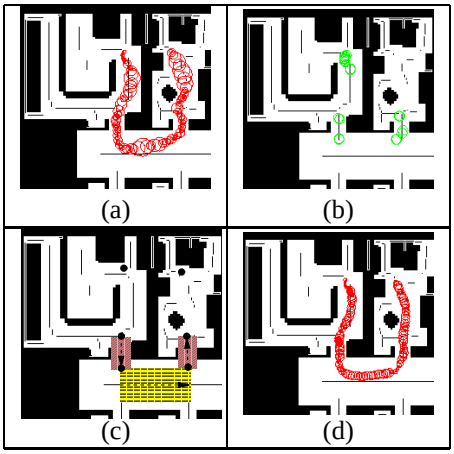


Figure 5: The figure visualizes a summary of a learning session: A behaviour trace of the default plan (a); behavior stretches where the robot moves conspiciously slowly (b); the added subplans in the learned navigation plan (c); and a behaviour trace of the learned plan, which is on average 29% faster than the default plan (d).

Figure 7: The mobile robots MINERVA (a) and the RWI B21 robot RHINO (c) that are used in the experiments.

of plan-based controllers to perform predictive plan management. This experiment has shown that predictive plan transformation can improve the performance by outperforming controllers without predictive capabilities in situations which require foresight while at the same time retaining their performance in situations that require no foresight. Figure 6 pictures an execution trace for a sample problem-solving scenario.

Conclusions

Our longterm research goal is to understand and build autonomous robot controllers that can carry out daily jobs in offices and factories with a reliability and efficiency comparable to people. We believe that many behavior patterns, such as exploiting opportunities, making appropriate assumptions, and acting reliably while making assumptions, that make everyday activity efficient and reliable require plan-based control and the specification of concurrent, reactive plans.

In this paper we have presented an overview of recent developments in the area of plan-based control of autonomous robots. Computational principles including plan-based high-level control, probabilistic reasoning, symbol anchoring, plan transformation, and context and resource-adaptive reasoning are incorporated in a number of state-of-the-art systems.

We believe that a necessary step towards more powerful plan-based robot controllers is the development of comprehensive and integrated computational models that address issues plan representation, reasoning, execution, and learning at the same time. A key component of such a computation model is the design of the plan representation language such that it allows for flexible and reliable behavior specifications, computationally feasible inference, stability in the case of runtime plan revisions, and automatic learning of symbolic plans for robot control.

Comprehensive computational models will enable us to tackle new application areas, such as the plan-based control of robot soccer teams, and longterm application challenges, for example, the robotic assistance of elderly people and the plan-based control of robotic rescue teams after disasters such as earthquakes.

References

R. Alami, S. Fleury, M. Herb, F. Ingrand, and F. Robert. Multi robot cooperation in the Martha project. *IEEE Robotics and Automation Magazine*, 5(1), 1998.

M. Beetz, T. Arbuckle, M. Bennewitz, W. Burgard, A. Cremers, D. Fox, H. Grosskreutz, D. Hähnel, and D. Schulz. Integrated plan-based control of autonomous service robots in human environments. *IEEE Intelligent Systems*, 16(5):56–65, 2001.

M. Beetz, T. Arbuckle, A. Cremers, and M. Mann. Transparent, flexible, and resource-adaptive image processing for autonomous service robots. In H. Prade, editor, *Procs. of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 632–636, 1998.

M. Beetz and T. Belker. Environment and task adaptation for robotic agents. In W. Horn, editor, *Procs. of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, 2000.

W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.

C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of AI research*, 1998.

M. Beetz. Structured reactive controllers — a computational model of everyday activity. In O. Etzioni, J. Müller, and J. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents*, pages 228–235, 1999.

M. Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers, 2000.

M. Beetz. Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:25–55, March/June 2001.

P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.

M. Beetz and H. Grosskreutz. Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. In *Pro-*

ceedings of the Fifth International Conference on AI Planning Systems, Breckenridge, CO, 2000. AAAI Press.

S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: preliminary report. In *Proc. of the 17th AAAI Conf.*, pages 129–135, Menlo Park, CA, 2000. AAAI Press.

D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In K. Hammond, editor, *Proc. 2nd. Int. Conf. on AI Planning Systems*. Morgan Kaufmann, 1994.

J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. Technical report 672, Yale University, Department of Computer Science, 1989.

N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

D. McDermott. Planning reactive behavior: A progress report. In K. Sycara, editor, *Innovative Approaches to Planning, Scheduling and Control*, pages 450–458, San Mateo, CA, 1990. Kaufmann.

D. McDermott. A reactive plan language. Research Report YALEU/DCS/RR-864, Yale University, 1991.

D. McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.

D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, 1992.

N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote agent: to go boldly where no AI system has gone before. *Artificial Intelligence*, 103(1–2):5–47, 1998.

A. Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, Massachusetts, 1990.

P. Nayak and B. Williams. Fast context switching in real-time propositional reasoning. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 50–56, Menlo Park, 1997. AAAI Press.

M. Pollack and J. Horty. There's more to life than making plans: Plan management in dynamic, multi-agent environments. *AI Magazine*, 20(4):71–84, 1999.

E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, New York, 1991.

D. Schulz and W. Burgard. Probabilistic state estimation of dynamic objects with a moving mobile robot. *Robotics and Autonomous Systems*, 34(2-3):107–115, 2001.

R. Simmons, R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, and M. Veloso. Xavier: Experience with a layered robot architecture. *ACM magazine* Intelligence, 1997.

G. Sussman. *A Computer Model of Skill Acquisition*, volume 1 of *Aritficial Intelligence Series*. American Elsevier, New York, NY, 1977.

S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Minerva: A second generation mobile tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99)*, 1999.

S. Thrun, M. Beetz, M. Bennewitz, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic algorithms and the interactive museum tour-guide robot minerva. *International Journal of Robotics Research*, 2000. to appear.

S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

B. Williams and P. Nayak. A reactive planner for a model-based executive. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1178–1185, San Francisco, 1997. Morgan Kaufmann Publishers.

S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

A conditional planning approach for the autonomous design of reactive and robust sequential control programs*

L. Castillo, J. Fdez-Olivares, A. González
Departamento de Ciencias de la Computación e Inteligencia Artificial
E.T.S. Ingeniería Informática. Universidad de Granada
18071 Granada. {L.Castillo,faro,A.Gonzalez}@decsai.ugr.es

Abstract

In this work we present a conditional planning approach based on new semantical notions which allow the generation of correct conditional plans in real and uncertain domains. The planning algorithm which embodies these original semantics is based on POP techniques and it has a clear practical application: it can be seen as an autonomous design process able to obtain conditional plans which can be interpreted as *closed-loop* control programs.

Introduction

It is well known that current work on planning under uncertainty is mainly focused on the establishment of sound planning models (Bonet & Geffner 2000; Cimatti & Roveri 1999; Geffner 1998; Rintanen 1999; Son & Baral 2001) and fast planning algorithms (Weld, Anderson, & Smith 1998), however it is also known that they lack of real applicability (Wilkins 2001). Thus, a main conclusion of the last planning conference was concerned with the necessity of developing new approaches of planning under uncertainty with practical application.

This call for practical planning approaches, in the concrete field of conditional planning, involves at least three foundamental issues: (1) the development of planners which deal with actions that do not always have certain outcomes, and which assume that the state of the world will not always be completely known, (2) the development of planners which embody a model of actions expressive enough for real applications, and (3) the development of semantical concepts in order to support a planning algorithm which guarantees the quality of the results.

In this sense, this paper tackles the problem of practical planning in real and uncertain domains where several agents exhibit a behavior which is affected by the existence of sources of uncertainty, that is, logical or physical entities which supply information about the environment. Concretely, we are interested in the field of software engineering of sequential control programs for manufacturing systems, one of the main topics in which planning community is interested.

*This work has been supported by the spanish government CICYT under project TAP99-0535-C02-01.

A manufacturing system is composed of a set of devices which may be seen as a set of agents acting in an uncertain environment, whose behavior is conditioned by sensors, and which must be globally coordinated in order to achieve a common goal. That goal is a specification of a process on products which has to be carried out by the agents, which are coordinated by a *sequential control program* that guides the operation of the system. Figure 1 shows an example of a manufacturing system where we can find the following agents:

A pump, P1, which transports water from the tank T1 to the tank T2, and which requires that the valve V1 be open. Additionally, there is a sensor, S_{level}, used to inform about the level of water at T1. The sensor can be in two possible states: on (T1 contains water) and off (T1 is empty). The behavior of P1 is restricted by the sensor S_{level}: when S_{level} is in the state on, P1 turns on, and when S_{level} is in the state off, it turns off.

A pump, P3, which transports chlorine from the tank TC to T2, and which requires that the valve V3 be open.

Two pumps, P21 and P22, which transport soda from the tank TS to T2. Both pumps require that the valve V2 be open, but only one of them may be active. The sensor S_{avble} can be in two states (s_1 or s_2),and it is used to decide what pump must be turned on.

The goal of this system is to obtain neutral pure water in T2 (initially contained in T1). Additionally, the sensor S_{pH} (which can be in three possible states $\{n, a, b\}$) informs whether the pH of the water contained in T2 is neutral (S_{pH} is in state n), acidic (S_{pH} is in state a) or basic (S_{pH} is in state b). Thus, the operation of the system must take into account that if the pH of the water contained in T2 is acidic then it will be necessary to add soda to T2, and if the pH is basic then it will necessary to add chlorine.

In order to obtain a control program for this manufacturing system, an expert follows a design process which receives as input the description of the system and a specification of its operation. The result of this process is a *closed-loop* control program, that is, a sequence of actions to be performed by the agents of the system, which take into account the information supplied by sensors, and which must incorporate conditional structures (possibly nested) in order to adequately describe the complex operation of the agents

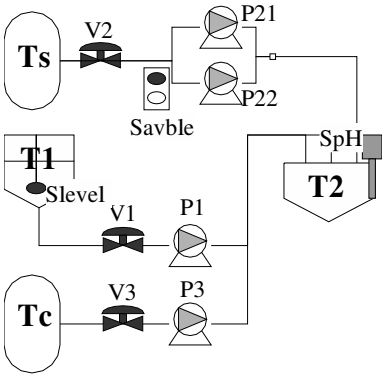


Figure 1: A manufacturing system for the neutralization of pure water.

of the system.

In this sense, the design process of a reactive and robust sequential control program is a problem that may be tackled within a planning with incomplete knowledge framework, taking into account that the application of planning techinques will require to obtain a complete solution by incorporating all the foreesable contingencies at running time. However, we have to rule out some possible alternative approaches as, for example, interleaving planning an execution (Koenig & Simmons 1998). The reason is that, in the context of manufacturing systems, the use of an execution process to improve a previously obtained incomplete plan might be extremely harmful for the system operation. Therefore, the best choose is a pure conditional planning approach.

In this context, previous work on planning applied to manufacturing operation (Aylett *et al.* 1998; PLANET 2001; Castillo, Fdez-Olivares, & González 2001a; 2001b) has shown that classical partial order planning may be applied as a successful technique for the autonomous design of *open-loop* control sequences. However none of these approaches assumes that the knowledge managed by a planner in a real domain may be incomplete. This is a shortcoming that limits the expressiveness of these approaches (that is, many real problems cannot be represented), and reduces the quality of their results (that is, the plans obtained cannot be conceived as "real" control programs because they lack of conditional structures).

Thus, the rest of this paper will be devoted to introduce a conditional planning approach, based on POP techniques, for real-world problem solving, with application to the autonomous design of closed-loop sequential control programs for manufacturing systems, and taking into account the following needs:

- The agents of a domain must be able to rapidly react to detected changes produced in the environment (For example, the pump P1 must be turned off when the sensor S_{level} has been detected to be in the state off).
- Agents must show a robust behavior, that is, they have to reach the proposed goal no matter what contingencies might be detected. (For example, the pH of the water must be neutral, independently of the state which the sensor S_{pH} is in).
- Plans obtained must incorporate actions capable of obtaining information from the environment, and actions capable of making decisions in run-time. In addition, plans must include (possibly nested) conditional structures in order to be accepted by human experts.

The paper is organized as follows: first we will introduce the knowledge representation (for domains, conditional plans and problems) used in this approach, next we will define a semantics for conditional plans and, finally, we will describe a conditional planning algorithm based in this semantics able to obtain correct conditional plans.

Knowledge Representation

In this section we will firstly describe how to represent and manage the incomplete knowledge originated by the existence of discrete sources of uncertainty in a domain that, in our application example, corresponds to the layout of a manufacturing system. Next we will introduce the knowledge representation used to describe domains, actions, problems, and plans.

Representing Incompleteness

In our approach we represent a source of uncertainty by means of a *sensor*. A *sensor* σ represents a discrete source of uncertainty, and it is associated with a finite set of *possible states* which are represented as symbols ($\mathcal{D}(\sigma) = \{u_1, \ldots, u_n\}$). In real domains, sensors affect the behavior of agents and, in the field of sequential control programming, this influence is represented by means of discrete variables which can take different values at run-time. During the design step of a control program, these variables are used by experts to describe the reactive or conditional behavior that, at running time, will be exhibited by the agents in a manufacturing system. This kind of variables may be represented in a planning model as an special type of planning variables called *run-time variables*, which have been used as a way for dealing with incompleteness in previous approaches of planning under uncertainty (Etzioni *et al.* 1992; Olawsky & Gini 1990; Olawsky, Krebsbach, & Gini 1995).

As other approaches do, we use run-time variables to represent that a source of uncertainty may affect the knowledge of the planner, although in a slightly different way. Concretely, a run-time variable *!x* is a variable which can be instantiated with constant symbols (belonging to a finite and discrete domain $\mathcal{D}(!x)$), and which is associated with a single sensor $Sensor(!x)$, in such a way that every possible value of *!x* is associated with a single possible state of σ.

The utility of a run-time variable is closely related with its use in the knowledge representation based on *literals*. In the planning model we are introducing, a literal l is represented as a tuple $(Atom(l)\ .\ Rc(l))$, where $Atom(l)$ is a predicate (called the atom of l) which may include run-time variables in its terms , and $Rc(l)$ is a set of *knowledge restrictions*. A knowledge restriction is represented as a special literal $(\text{KNOWN}\ \ \sigma\ u)$, where σ stands for a sensor and u is one of its possible states. This extended syntax involves

some issues about the interpretation of a literal that we have to clarify:

- First, the truth-value of a literal l is the truth-value of *Atom*(l), which may be $true$, $false$ or $unknown$.
- Second, the value that a run-time variable *!x* may take (at running time) is unknown at planning time, and it always depends on a possible state of *Sensor*$(!x)$. Therefore, for a given literal l, if *Atom*(l) contains a run-time variable *!x*, then the truth-value of a complete instantiation of *Atom*(l) is restricted by the sensor *Sensor*$(!x)$, in such a way that it depends on a *knowledge production* caused by the detection of a state of *Sensor*$(!x)$.
- Third, taking into account the previous point, the set of knowledge restrictions of a literal allows to represent, at planning time, what sources of uncertainty (sensors) affect its truth-value, that is, knowledge restrictions are used to represent a *context* where a literal is known to be true or false. For example $l = (p\ .\ ((\text{KNOWN}\ \sigma\ u)(\text{KNOWN}\ \sigma'\ u')))$ is interpreted as "l is true when it has been detected that, simultaneously, σ is in the possible state u, and σ' is in the possible state u'", or in other words, "l is true in a context where σ takes the state u and σ' takes the state u'". Additionally, for a given literal l, *Rc*(l) may take the value T, meaning that there is no sensor which restricts the truth-value of l[1] , or the value NIL, meaning that the truth-value of l cannot be determined, representing an inconsistency.
- Finally, we have to say that not every set of restrictions is a valid one. Concretely, T is a valid set of knowledge restrictions, NIL is not a valid set, and a set of knowledge restrictions which contains two restrictions which refer to the same sensor is not a valid set (for example, the set $((\text{KNOWN}\ \sigma\ u_1)(\text{KNOWN}\ \sigma\ u_2))$ is not a valid one). Finally, a set of knowledge restrictions is valid when none of these rules applies.

The representation of knowledge restrictions is inspired on the concepts of *conditional context* and *context labels* used in (Peot & Smith 1992) and (Pryor & Collins 1996), and the validity of a set of knowledge restrictions is similar to the notion of *contexts compatibility* introduced in (Peot & Smith 1992). However, the model of actions of these known conditional planning approaches is basicaly the classical model of STRIPS, extended with sensing (or observe) actions. So, though these approaches manage uncertainty, their action representation is not expressive enough to face with problems in manufacturing domains. What is more, these models lack of a clear semantics that justify their planning algorithms.

Thus, the representation of knowledge restrictions will allow the introduction of new semantical concepts which will be the basis of a new conditional planning algorithm based on POP techniques, able to obtain ready-to-use sequential control programs. These new concepts will be based on an unification algorithm for literals with knowledge restrictions, which extends the classical unification algorithm, and which we describe next.

Definition 1 *Two literals, l_e and l, unify when their atoms unify and the set union of their knowledge restrictions is valid. In this case, the resulting literal l_u = **Unify**(l_e, l) is a literal defined as follows:*

Atom(l_u) *is the result of the classical unification of Atom*(l_e) *and Atom*(l)*, extended for run-time variables (a run-time variable may unify with a constant, or with a normal variable which contains a single constant in its codesignation constraints).*

Rc(l_u) *is a set of knowledge restrictions which includes Rc*(l_e)*, Rc*(l)*, and a newly generated set r_u of knowledge restrictions. Every knowledge restriction* $(\text{KNOWN}\ \sigma\ u)$ *of r_u is generated as result of the unification of a run-time variable !x, such that Sensor*$(!x) = \sigma$*, with a constant $k \in \mathcal{D}(!x)$, such that u is a state of σ associated with the value k.* □

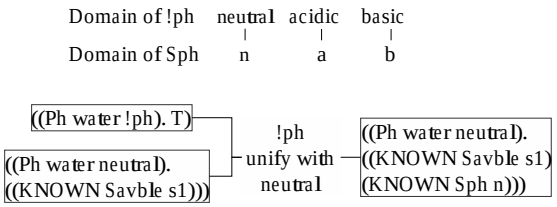


Figure 2: Unification of literals with knowledge restrictions.

Figure 2 shows an example of unification and, as can be seen, when a run-time variable is instantiated by a constant, the unification may lead to add new knowledge restrictions in the resulting literal. This means that, in order to determine the truth-value of the resulting literal, it is required to produce more knowledge than the one required to determine the truth-value of the unified literals.

Domain and Plans representation

In our model a domain is represented as a set of *agents* and a set of *sensors*. Every sensor σ is associated with a set of *sensing actions*. A sensing action is associated with a single state u of a sensor σ, and it is represented with a name (denoted as ***When***$(\sigma\ u)$) and with a set of effects. The effects of a sensing action ***When***$(\sigma\ u)$ represent facts of the world which are changed when a sensor σ is in the state u (See Figure 3). So, sensing actions are used to have access to these facts.

On the other hand, every agent g is represented by means of its properties (name, normal variables and run-time variables) and its behavior. The behavior is modeled as a finite automaton in which every *causal action* a which is executed by g contains a set *Req*(a) of literals, called *requirements*, which must be solved in order to achieve a correct execution of the action, and a set of effects, *Efs*(a), which include the change of state produced by a over the agent g. In order to achieve an adequate expressiveness for real-world planning,

[1]For operational purposes, we assume that T is included in any set of restrictions

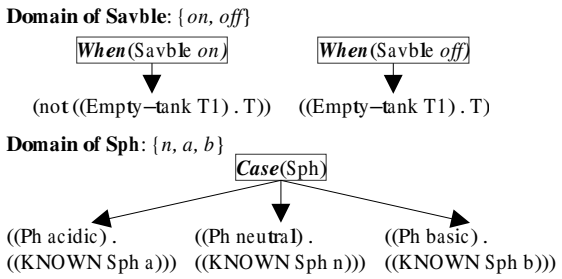


Figure 3: Sensing and decide-actions.

the representation of causal actions embodies the following features:

- Actions are considered as intervals, that is, every causal action a of an agent g executes over an interval, $[a, \mathit{End}(a)]$, defined from a until the next change of state of g, produced by another action, $\mathit{End}(a)$, of g.
- There are four kind of requirements: *previous* (they must be true before the action), *simultaneous* (they must be true during the interval of an action), *query* (used to find out facts before the action), and procedural (they must be true after the action) (See (Castillo, Fdez-Olivares, & González 2000; 2001a) for more details). Previous, simultaneous, and procedural requirements can only be solved by causal actions, but query requirements may be solved by causal or sensing actions.
- Every action a contains a set of knowledge restrictions, $\mathit{Rc}(a)$, which represent the context which a can be executed in (that is, the execution of an action may be affected by some states of the sensors in the domain).
- The effects of a causal action may contain literals with run-time variables. This means that, as a literal with run-time variables cannot be known at planning time, every action which contains run-time variables in its effects is a non-deterministic action. Additionally, in this work we assume, for simplicity purposes, that requirements do not contain run-time variables, which only one literal with run-time variables is allowed in the effects of a causal action, and which this literal can only contain one run-time variable.

The action model also includes another kind of actions called *decide-actions*. A decide action is associated with a single sensor σ, and it is represented with a name (denoted as $\boldsymbol{Case}(\sigma)$) and with a set of *possible effects* $\mathit{Efs}(\boldsymbol{Case}(\sigma)) = \{\mathit{PEf}_{u_1}, \ldots, \mathit{PEf}_{u_n}\}$ ($\{u_1, \ldots, u_n\}$ are the possible states of σ). Every possible effect PEf_{u_i} is a set of literals with knowledge restrictions, representing a causal transformation of the world in case of σ is in the possible state u_i (See Figure 3). Decide-actions are used at running time to make decisions about which course of actions must be followed, and, at planning time, they are automatically generated to represent the different outcomes of the execution of a non-deterministic action. This process will be described later, next we will describe how problems and plans are represented.

Definition 2 *A problem is represented as a tuple $\mathcal{P} = \langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$ where $\mathcal{D}$ is a domain,$\mathcal{O}$ is a set of (possibly ordered) literals which represent the high level goal, and $\mathcal{I}$ is a set of literals which represent an incomplete initial state. Every literal l in the initial state can be initially true, false or unknown, in such a way that:*

- *l is initially true if $l \in \mathcal{I}$ and l does not contain run-time variables.*
- *l is initially unknown if some of the following conditions holds:*
  - *i) l or $(not\ l)$ belongs to the effects of some sensing-action of the domain.*
  - *ii) l or $(not\ l) \in \mathcal{I}$ and it contains run-time variables.*
- *l is initially false if some of the following conditions holds:*
  - *i) $(not\ l) \in \mathcal{I}$ and l does not contain run-time variables.*
  - *ii) $l \notin \mathcal{I}$ and l is not initially unknown.* □

Definition 3 *A conditional plan is represented as a tuple $\Pi = \langle A_c, A_{when}, A_{case}, < \rangle$ where $A_c(\Pi), A_{when}(\Pi), A_{case}(\Pi)$ stand for a set of causal, sensing and decide actions, respectively, and $<$ stands for a partial order relation between them. A conditional plan contains two dummy causal actions: a_0, the first action of a conditional plan, which is a non-deterministic action whose effects encode the incomplete initial state[2], and a_∞, the last action of any conditional plan and whose requirements encode the goal.* □

Semantics

This section is centered on the study of which conditions have to be accomplished for a conditional plan to be correct. Intuitively, a conditional plan it said to be correct if every action in the plan is executable, the execution of the plan allows to reach the goal of the problem, and there are no conflicts between the actions of the plan.

Firstly, we have to say that sensing actions do not contain requirements because their execution depends on a sensor to be in a possible state, which is a non-foreseeable event. However, when a sensing action $\boldsymbol{When}(\sigma\ u)$ executes, its effects are known to be true, and it is also known that the sensor σ is in the state u. Thus, sensing actions will be used to satisfy knowledge needs, which will be represented in the query requirements of causal actions.

On the other hand the execution of a decide-action $\boldsymbol{Case}(\sigma)$ can be interpreted has the simultaneous execution of all the sensing actions associated with σ, which means that the uncertainty about the current state of σ is eliminated after a decide-action. The utility of decide-actions will be detailed later.

Thus, in this approach the executability conditions of sensing and decide actions are not a subject to be studied, so, next section will be centered on the study of the executability conditions of a causal action in a conditional plan, and how can we interpret the execution of deterministic and

[2]It is allowed for the effects of the special action a_0 to contain more than one literal with run-time variables

non-deterministic actions. Afterwards, we will introduce the semantical concepts which will allow to accomplish these conditions by means of literal satisfaction, and we will describe the causal structure of a conditional plan. This will finally lead to a definition of correct conditional plan.

Executability conditions

Definition 4 *We will say that a causal action a, such that $Rc(a) = r$, is executable when the following conditions hold:*

i) Their requirements are true.

ii) $\forall l \in Req(a), Rc(l) = r \vee Rc(l) = \mathrm{T}$. □

This definition establishes that every requirement of an executable action may either be true in the same context in which the action can be executed, or be true in all possible contexts (that is, when its set of knowledge restrictions is equal to T). Additionally, we have to define what we consider to be a correct execution of an action.

Definition 5 *We will say that the execution of an executable and deterministic action is correct when* $\forall l_e \in Efs(a), Rc(l_e) = Rc(a)$. □

That is, if a deterministic action executes in a context $Rc(a)$, then its effects are true in the same context.

However, this definition does not apply for non-deterministic actions. The effects of this kind of actions contain run-time variables, meaning that some literals of the effects of a non-deterministic action will be true in different contexts. Therefore, the truth-value of these literals can only be determined at running time but, at planning time, we can make use of decide-actions in order to represent the possible transformations produced by the execution of a non-deterministic action. This will be done by associating decide-actions to non-deterministic actions, in such a way that the effects of a decide-action will be used to represent the different outcomes of a non-deterministic action.

In this sense, for every literal in the effects of a non-deterministic action a, containing a run-time variable *!x* such that $Sensor(!x) = \sigma$, will be associated a decide-action $\textbf{\textit{Case}}(\sigma)$ to a (See Figure 4). This leads to define a process which allows to automatically generate decide-actions within a conditional plan.

Definition 6 *Let a be a non-deterministic action, such that $Efs(a)$ contains a literal l with a run-time variable !x. When a is included in a conditional plan, a decide-action is generated in that plan according to the following function:*

$\textbf{\textit{GenDec}}(a, !x)$
Let l be the literal of $Efs(a)$ which contains !x
Let $\sigma = Sensor(!x)$
FOR EACH $k_i \in \mathcal{D}(!x)$
  Let u_i be the associated state of σ with k_i
  Let p_i represent the atom of l where !x is unified with k_i
  Let $PEf_{u_i}(\textbf{\textit{Case}}(\sigma)) = \{(p_i \,.\, Rc(l) \cup ((\textsc{Known}\ \sigma\ u_i)))\}$
Let $Efs(\textbf{\textit{Case}}(\sigma)) = \bigcup_{u_i} PEf_{u_i}(\textbf{\textit{Case}}(\sigma))$
RETUN $\textbf{\textit{Case}}(\sigma)$

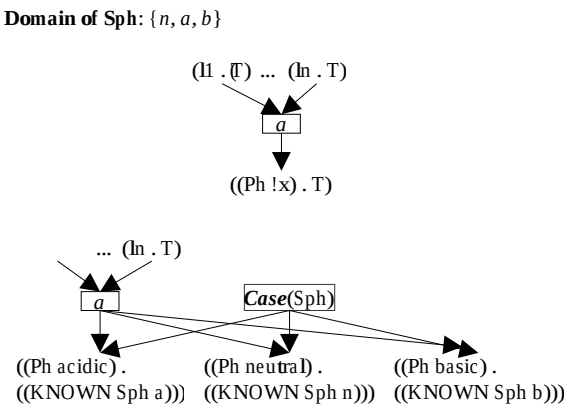


Figure 4: A decide action associated to a causal action.

Additionally, the possible effects of $\textbf{\textit{GenDec}}(a, !x)$ *become effects of a.* □

The generation of decide-actions will support the representation, in a single conditional plan, of different sequences of actions executed in different contexts, which can be seen as conditional branches. However, the existence of a decide-action in a conditional plan is not sufficient to create different conditional branches. The creation of conditional branches will be discussed in the next section, where we will introduce how to accomplish the executability conditions of causal actions (deterministic or not) when different execution contexts are represented in a conditional plan.

Different modes of satisfaction

In classical POP, a literal l in the requirements of an action a, included in a plan, may become true if there is another previous action b which satisfies l, that is, if this action contains a literal l_e in its effects which unifies with l. The process followed by our approach to determine the truth-value of the requirements of an action is also based on literal satisfaction. However, the new features of the unification algorithm described above lead to distinguish between different modes of satisfaction (See Figure 5), taking into account whether the unification generates new knowledge restrictions or not.

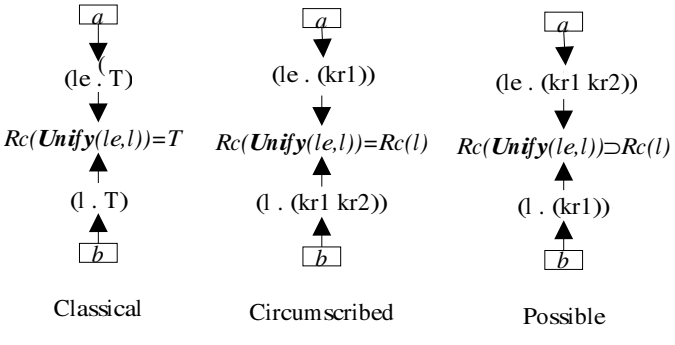


Figure 5: Different modes of satisfaction.

Definition 7 *Let b be a causal action with a literal l in its requirements, and let a be an action (of any kind) which contains a literal l_e in its effects which unifies with l, such that $a < b$ (if the requirement is procedural we consider $b < a$). The unification of l_e and l may lead to the following cases:*

- *Rc(**Unify**$(l_e, l)) = \mathrm{T}$: this is the case of classical satisfaction (noted as $a \xrightarrow{Sat} l$).*
- *Rc(**Unify**$(l_e, l)) \neq \mathrm{T}$ and Rc(**Unify**$(l_e, l)) =$ Rc(l): this case will be referred to as circumscribed satisfaction (noted as $a \xrightarrow{\mathbf{CSat}} l$).*
- *Rc(**Unify**$(l_e, l) \supset$ Rc(l): this case will be referred to as possible satisfaction supplying a set of knowledge restrictions defined as $r_a =$ Rc(**Unify**$(l_e, l) -$ Rc(l). (noted as $a \xrightarrow{\mathbf{PSat}} [l, r_a]$).* □

In the following, we will describe how these modes of satisfaction affect to the accomplishment of the executability conditions of a causal action.

When $a \xrightarrow{\mathbf{CSat}} l$ holds, for some action a and some literal l of the requirements of an action b, we have to take into account two possible cases: Rc$(a) =$ Rc(l) or Rc$(a) \in$ Rc(l). The first case is consistent with Definitions 4 and 5, but the second one violates the definition of correct execution for the action a , because b requires a to be executable in the context Rc(l), and a is known to be executable in a different context. So, in order to guarantee the accomplishment of Definition 5, it will be necessary to *propagate backwards* (backto the action a, its requirements and effects) a set of knowledge restrictions, following a process defined as follows:

Definition 8 *Let a and b be two causal actions such that $a \xrightarrow{\mathbf{CSat}} l$ holds, for some requirement l of b. The backward propagation of knowledge restrictions from b to a is defined by the following rule:*

BckProp(a, l)
IF Rc$(a) \subset$ Rc(l)
THEN FOR EACH $l \in$ Req$(a) \cup$ Efs(a)
ASSIGN Rc$(l) =$ Rc$(l) \cup$ Rc$(b) -$ Rc(a)
ASSIGN Rc$(a) =$ Rc$(a) \cup$ (Rc$(b) -$ Rc(a))
RETURN a

□

With respect to the case of a possible satisfaction, it is important to note that when $a \xrightarrow{\mathbf{PSat}} [l, r_a]$ holds, for some action a and for some literal l of the requirements of an action b, the Definition 4 is violated (because a requirement of b would be true in a context different from Rc(b)). In this case, in order to guarantee the executability conditions of b, there are two alternatives: to assume that l can either be true in the context Rc$(l) \cup r_a$, or be necessarily true in the context Rc(l).

In the first case, the executability conditions of b can be accomplished by means of a process which *propagate forwards* (toward the action b, its requirements and effects) the set of restrictions r_a. However, this propagation cannot be applied in any case. Concretely, if the literal satisfied is contained in the main goal, or b is a non-deterministic action whose effects contain a literal l_j, such that Rc$(l_j) \cup r_a$ is not valid, then forward propagation does not apply. This is defined in the following rule.

Definition 9 *Let a and b be two causal actions such that $a \xrightarrow{\mathbf{PSat}} [l, r_a]$ holds, for some requirement l of b. The forward propagation of the knowledge restrictions set r_a, from a towards b, is defined by the following rule:*

FwProp(a, l, b, r_a)
IF l *is a goal literal or*
b *is a non-deterministic action such that*
$\exists l_k \in$ Efs(b), Rc$(l_k) \cup r_a$ *is not valid*
THEN RETURN *FAIL*
ELSE FOR EACH$l \in$ Req$(b) \cup$ Efs(b)
ASSIGN Rc$(l) =$ Rc$(l) \cup r_a$
ASSIGN Rc$(b) =$ Rc$(b) \cup r_a$
RETURN a

□

It is necessary to remark that, indeed, forward propagation can be recursively applied through the causal structure of a conditional plan. This recursive process will be described in the next section.

In the case of $a \xrightarrow{\mathbf{PSat}} [l, r_a]$ holds, for a literal $l = (p\ .\ r)$, and forward propagation cannot be applied, it is interpreted that l must be necessarily true in the context r. This leads to define a new mode of literal satisfaction which will be referred to as *necessary satisfaction*. This concept will be defined just after we have described how to represent that a literal l must be necessarily true in a context r when $a \xrightarrow{\mathbf{PSat}} [l, r_a]$ holds (See Figure 6), by means of the concept of *conditional expansion*.

Definition 10 *Let a and b be two causal actions such that $a \xrightarrow{\mathbf{PSat}} [l, r_a]$ holds, for some literal l of the requirements of b. The conditional expansion of l with respect to the knowledge restrictions set r_a, is a set of literals defined by de following function:*

C-exp(l, r_a)
IF $r_a = \{\emptyset\}$ THEN RETURN $\{l\}$
IF $r_a = ((\mathrm{KNOWN}\ \sigma\ u)))$
THEN *Let* $C = \{\emptyset\}$
FOR EACH $u_i \in \mathcal{D}(\sigma)$
IF Rc$(l) \cup ((\mathrm{KNOWN}\ \sigma\ u_i))$ *is valid*
THEN *Let* $C = C \cup \{($Atom$(l)\ .$ Rc$(l) \cup$
$((\mathrm{KNOWN}\ \sigma\ u_i)))\}$
RETURN C
ELSE *Let* $r = (\mathrm{KNOWN}\ \sigma\ u))$ *(extracted from* r_a*)*
Let $C = \{\emptyset\}$
FOR EACH $u_i \in E(\sigma)$
IF Rc$(l) \cup ((\mathrm{KNOWN}\ \sigma\ u_i))$ *is valid*
THEN *Let* $C = C \cup$ ***C-exp***$(($Atom$(l)\ .$ Rc$(l) \cup$
$((\mathrm{KNOWN}\ \sigma\ u_i))), r_a - r)$
RETURN C

□

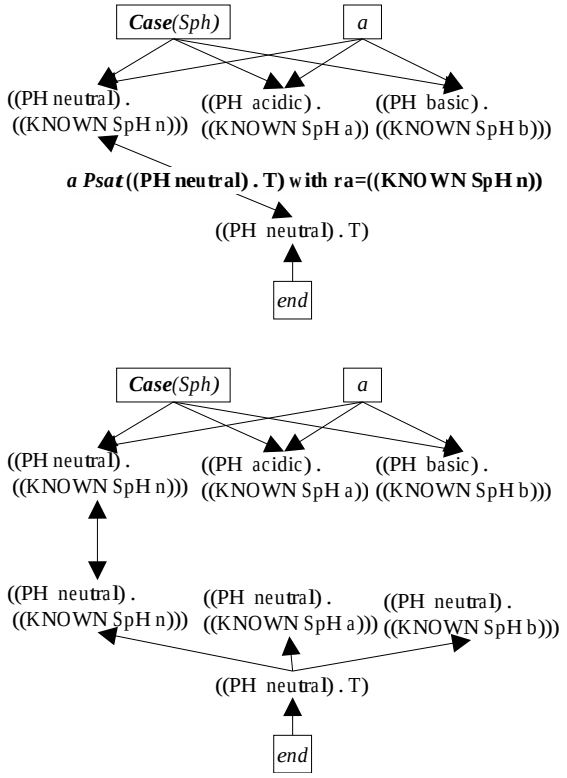


Figure 6: Conditional expansion of a literal with respect to r_a.

That is, the conditional expansion of a literal l, with respect to r_a is a set of literals that allow to represent that l must be true in the context $Rc(l) \cup r_a$, and in any other possible context $Rc(l) \cup r_i$, where r_i stands for a valid combination of the possible states of the sensors included in r_a (See Figure 6). The conditional expansion of a literal is the basis of the concept of necessary satisfaction which is defined as follows:

Definition 11 *Let a and b be two causal actions such that $a \xrightarrow{PSat} [l, r_a]$ holds for some literal in the requirements of b. We will say which l is necessarily satisfied in a conditional plan Π (noted as $\Pi \xrightarrow{NSat} [l, a]$) when $\forall l_c \in$ **C-exp**(l, r_a), $\exists a_c \in A_c(\Pi)$, such that $a_c \xrightarrow{CSat} l_c$ holds.* □

That is, a literal is necessarily satisfied in a conditional plan when the literals of its conditional expansion are satisfied in a circumscribed mode. In addition, it is worth noting that this mode of satisfaction will allow to generate, in a conditional plan, by means of the regression of the literals of a conditional expansion, sequences of actions which will satisfy (in a circumscribed mode) every literal of **C-exp**(l, a), for some literal l and some action a which satisfies l in a possible mode. The actions of these sequences will be executed in different contexts and, so, we can conclude that the conditional expansion of a literal l will allow to create conditional branches within a conditional plan.

Finally, we can redefine the conditions under a causal action is considered to be executable, in terms of the modes of satisfaction previously introduced.

Definition 12 *A causal action is executable if its previous, simultaneous, and procedural requirements are satisfied in a circumscribed or necessary mode, and its query requirements are satisfied in a circumscribed mode.*

Different kinds of causal links

In the previous section we have shown that the requirements of a causal action may be satisfied in different modes by different kinds of actions. This means that, in addition to causal links, the causal structure of a conditional plan embodies different kinds of causal dependencies. These kinds are the following ones:

- *Causal link*, noted as $[a \xrightarrow{l} b, c]$, which represents that a requirement l of an action b has been satisfied in a circumscribed or classical mode by a causal action a, and which has to be protected during the actions interval $[a, c]$ (if l is a previous or query requirement $c = b$, and if l is simultaneous $c = End(b)$).

- *Detection link*, noted as $\mathbf{D}[a \xrightarrow{l} b]$, which represents that a query requirement l of b has been satisfied in a circumscribed or classical mode by a sensing action a, and which has to be protected during the actions interval $[a, b]$.

- *Possible link*, noted as $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$. which represents that literal l_n which belongs to the conditional expansion of a previous, simultaneous, or procedural requirement l_n of an action b, has been satisfied in a circumscribed mode, and which has to be protected during the actions interval $[a, c]$.

These kinds of causal dependencies are mainly used to detect threats between actions in a conditional plan. The threats management is similar to the one followed in classical POP, although it is necessary to note that it is extended to incorporate the notion of actions interval and the representation of literals here presented. In order to define the concept of threat, it is necessary to know that an action a' (of any kind) negates the literal l when there is a literal $l' \in Efs(a)$ such that $Atom(l')$ negates $Atom(l)$ and $Rc(l') \cup Rc(l)$ is valid.

Thus, we can define the following types of threats:

- A causal action a' threatens a causal link $[a \xrightarrow{l} b, c]$ (or a possible link $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$) when a' negates l and the interval $[a, c]$ is unordered with respect to the interval $[a', End(a')]$.

- A sensing action a' threatens a detection link $\mathbf{D}[a \xrightarrow{l} b]$ when a' negates l and the interval $[a, b]$ is unordered with respect to a'.

Detection links, and the threats management for these links, allow to introduce order constraints between sensing actions, meaning that it is possible to describe a correct reactive behaviour for the agents of a domain.

On the other hand, causal and possible links allow to support a recursive forward propagation of knowledge restrictions, based on the forward propagation rule above defined (Definition 9). This recursive process starts when $a \xrightarrow{\textbf{\textit{PSat}}} [l, r_a]$ holds for some action a and some requirement l of another action b, and r_a are propagated towards b.

In this case, the causal dependencies (causal or possible links) which are produced by b must be revised, because the context of b has changed, and it is possible that the executabilty conditions of every action supported by b would be violated. Therefore, in order to guarantee the correct execution of every action a_b supported by b, it will be necessary to test whether these conditions have to be updated or not, that is, whether r_a must be propagated towards a_b. This lead to a "test-and-propagate" recursive process, through the causal structure of a conditional plan, which ends when the supported action is a_∞ or r_a cannot be propagated forward. This process is described in the following definition.

Definition 13 *Let a and b be two causal actions such that $a \xrightarrow{\textbf{\textit{PSat}}} [l, r_a]$ holds, for some literal l in the requirements of b. Let cl a causal or possible link which represents a causal dependence between a and b. The recursive propagation of r_a, from vc and through the causal structure of a conditional plan, is defined by the following function:*

R-FwProp(vc, r_a)
Let a, b and l be theproducer action, consumer action and the literal of vc, respectively
IF vc *is not a possible link*
THEN IF $b = a_\infty$ THEN RETURN $\{vc\}$
ELSE IF ***FwProp***(a, l, b, r_a) *does not apply*
THEN IF $r_a \cup Rc(b)$ *is not valid*
THEN RETURN *FAIL*
ELSE RETURN $\{vc\}$
ELSE *Let* $VC = \{vc\}$ *be a set of links*
FOR EACH *link* vc_b *such that* b *is its producer*
stop-prop = ***FwProp***(vc_b, r_a)
IF *stop-prop = FAIL* RETURN *FAIL*
ELSE $VC = VC \cup$ *stop-prop*
RETURN VC
ELSE RETURN $\{vc\}$□

The function **R-FwProp**(vc, r_a) returns a set of causal or possible links which contains those causal dependencies where forward propagation cannot be applied. Thus, every link returned can be seen as the representation of a possible satisfaction where it is not possible to propagate forwards. If during the recursive propagation a not valid set of knowledge restrictions is found, the function will return a general fail, which means that the forward propagation through the causal structure will not lead to obtain a correct plan. This function will be very useful in the conditional planning algorithm wich will be introduced in the next section. However, previously we have to establish the conditions under a conditional plan is considered to be correct.

Definition 14 *A conditional plan* Π*, constructed to solve a planning problem $\mathcal{P} = \langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$ is correct when every causal action in* Π *is executable, every literal in $\mathcal{O}$ is satisfied in a circumscribed or necessary mode by any executable action of* Π*, and there are no threats between the actions intervals of* Π*.* □

SolveSubGoal(l, a, Π)
Let b be the action such that $l \in$ *Req*(b).
Let c be the action *End*(b), if l is a simultaneous requirement, or the action b, if l is a previous requirement
IF a is a new action
THEN Insert a in Π
IF a is a non-deterministic action
THEN Insert **GenDec**$(a, !x)$ in Π
CASE $a \xrightarrow{\textbf{\textit{CSat}}} l$
IF l is a query requirement and a is a sensing-action
THEN Insert $\mathbf{D}[a \xrightarrow{l} b]$
ELSEIF l belongs to the conditional expansion of some literal l_n
THEN Insert $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$ in Π and **BckProp**(a, l)
ELSE Insert $[a \xrightarrow{l} b, c]$ in Π and **BckProp**(a, l)
CASE $a \xrightarrow{\textbf{\textit{PSat}}} [l, r_a]$
IF l belongs to the conditional expansion of some literal l_n
THEN Insert $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$ in Π
stop-prop =$\{\mathbf{P}[a \xrightarrow{l} l_n, b, c]\}$
ELSE Insert $[a \xrightarrow{l} b, c]$ in Π
stop-prop = **R-FwProp**$([a \xrightarrow{l} b, c], r_a)$
IF stop-prop = FAIL RETURN FAIL
FOR EACH vc in stop-prop
Insert the refinement task "Conditional Branching vc, r_a"

Figure 7: Algorithm for solving an unsolved goal flaw.

In the next section we will describe an algorithm able to obtain correct conditional plans according to this definition.

Planning algortithm

The previously defined notions have been incorporated into the partial order algorithm described in (Castillo, Fdez-Olivares, & González 2001a; 2000) resulting in a new conditional planning algorithm (called ADVICE), capable of constructing correct conditional plans, in the terms defined above, which can be interpreted as *closed-loop* control programs. Thus, ADVICE is based on a planning algorithm which generates a conditional plan by the successive application of refinement operations on a partially constructed conditional plan. These operations are carried out when, in the partially constructed conditional plan, it appears any of the following flaws:

- *Threats*. ADVICE solves the different types of threats defined above by demotion or promotion of actions intervals (See (Castillo, Fdez-Olivares, & González 2001a) for more details).
- *Unsolved subgoals*. Subgoals are solved following the algorithm shown in Figure 7. A literal which represents an unsolved subgoal may be satisfied by a non instatiated causal action or by an action included in the conditional plan in construction (if the literal represents a query requirement, the action may be a sensing action). In the first

MakeBranch(vc, r_a, Π)
IF vc is a causal link $[a \xrightarrow{l} b, c]$
THEN Insert the literals of **C-exp**(l, r_a) as new subgoals in Π
IF vc is a possible link $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$
THEN Non deterministally chosse a literal l' from the set $\{l, l_n\}$
Insert the literals of **C-exp**(l', r_a) as new subgoals in Π

Figure 8: Algorithm for conditional branch creation.

case, ADVICE inserts the action, associating a decide-action in case of the action is non-deterministic. If the literal is satisfied in a circumscribed mode, a backward propagation of knowledge constraints is performed. Then, if the literal belongs to the conditional exansion of another requirement, a possible link is inserted, otherwise, a causal link is inserted (if the literal is a query requirement satisfied by a sensing action, a detection link is inserted). On the other hand, if the literal is satisfied in a possible mode (we assume that query requirements cannot be satisfied in a possible mode), causal and possible links are inserted as in the previous case , and a recursive forward propagation process is performed (we assume that knowledge restrictions cannot be propagated towards the literals of a conditional expansion). If this process returns a fail, it is interpreted that the subgoal cannot be solved by the action being used, otherwise, every causal or possible link returned by this process will be considered as a new flaw of type "Conditional-Branching".

- *Conditional branching*. This new type of flaw raises when the forward propagation process cannot be applied on a causal or possible link. We have introduced above that every link returned by this process represents a possible satisfaction where it is not possible to propagate forwards. Therefore, according to Definitions 10 and 11, this flaw can be solved by means of a conditional expansion (See Figure 8). Thus, the introduction of the literal of the a conditional expansion as unsolved subgoals, will allow to guarantee that a literal can be satisfied in a necessary mode by the creation of different conditional branches. In addition, as can be seen in Figure 8, it is not possible to determine, in a possible link $\mathbf{P}[a \xrightarrow{l} l_n, b, c]$, which literal must be conditionally expanded (l or l_n). In this case, the flaw can be solved by means of two alternatives.

Figure 9 shows the conditional plan generated by ADVICE which solves the manufacturing problem described in the introduction of this paper. The plan represents correctly the required behaviour:

- The pump P1 and the valve V1 *react* to the information supplied by the sensor S_{level}, in such a way that they turn off when S_{level} is in the state off.
- On the other hand, the actions of the agents V2, V3, P21, P22, and P3, are conditionally structured in such a way that these agents show a robust behavior which allows to obtain a neutral pH. In addition, the plan contains nested conditional branches (which may be discover by means of the recursive propagation process defined above) to represent the alternative operation of the agents P21 and P22.

As can be seen in this example, ADVICE satisfies the needs formerly established in the introduction of this work.

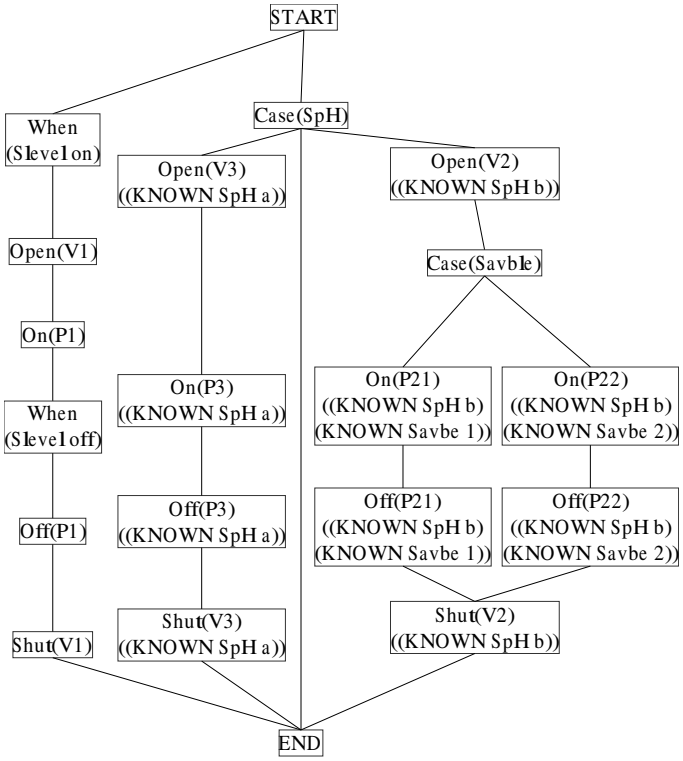


Figure 9: A conditional plan for the problem formerly introduced.

Conclusions

In this work we have presented a conditional planning approach based on an expressive model of actions, and on new semantical notions which allow to generate correct conditional plans, in real and uncertain domains. The planning algorithm which embodies these original semantics can be seen as an autonomous design process able to obtain conditional plans, which can be interpreted as ready-to-use *closed-loop* control programs.

Taking into account the planning process and the plans obtained, ADVICE can be seen as a step forward in the field of planning applied to software engineering, overwhelming the main shortcomings (concerned with incomplete knowledge management and plan quality) of current planning approaches to manufacturing systems operation. It must also be said that ADVICE has been extensively used for the automatic synthesis of closed-loop control programs. This experimentation is being carried out in close collaboration with experts on industrial domains within a research project, and it will appear in other paper in preparation.

On the other hand, appart from the semantical concepts presented, one of the main advantages of this approach is

that it embodies a more expressive model of actions than the one used in other conditional planning approaches, allowing to obtain conditional plans with nested conditional structures. That is, as opposite to other POP approaches which assume incomplete knowledge (Pryor & Collins 1996; Etzioni *et al.* 1992), the representation of a plan obtained by ADVICE is a DAG and not a tree (Weld, Anderson, & Smith 1998), a characteristic that needs to be incorporated into any plan intended to be useful and understandable for human experts.

Our current research is currently focused on the integration of this conditional planning approach with the hybrid planning model (which mixes hierarchical planning and POP, but which assumes complete knowledge) described in (Castillo, Fdez-Olivares, & González 2001b)). This will lead to develop a planning system able to obtain hierarchical and conditional plans which could be interpreted as hierarchical and conditional control programs. We think that such as system will be extremely helpful to the experts on industrial automation.

References

Aylett, R.; Soutter, J.; Petley, G.; and Chung, P. 1998. AI planning in a chemical plant domain. In *European conference on artificial intelligence*.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 52–61.

Castillo, L.; Fdez-Olivares, J.; and González, A. 2000. A three-level knowledge-based system for the generation of live and safe petri nets for manufacturing systems. *Journal of Intelligent Manufacturing* 11(6):559–572.

Castillo, L.; Fdez-Olivares, J.; and González, A. 2001a. Mixing expresiveness and efficiency in a manufacturing planner. *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)* 13:141–162.

Castillo, L.; Fdez-Olivares, J.; and González, A. 2001b. On the Adequacy of Hierarchical planning characteristics for Real-World Problem Solving. In *Proceedings of the Sixth European Conference on Planning ECP-01)*.

Cimatti, A., and Roveri, M. 1999. Conformant planning via model checking. In *Proceedings of the Fifth European Conference on Planning (ECP-99)*, 21–33.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. Third. Int. Conf. on Principles of KRR-92*, 115–125.

Geffner, H. 1998. Modelling intelligent behaviour: The Markov decision process approach. *Lecture Notes in Computer Science* 1484:1–12.

Koenig, S., and Simmons, R. 1998. Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In *Proceedings of the International Conference on Artificial Intelligence Planning System*, 154–163.

Olawsky, D., and Gini, M. 1990. Deferred planning and sensor use. In *Innovative Approaches to Planning, Scheduling and Control*, 166–174. Morgan Kauffman, San Mateo.

Olawsky, D.; Krebsbach, K.; and Gini, M. 1995. An analysis of sensor-based task planning. Technical Report TR95-51, Dep. Comp. Science. University of Minneapolis.

Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proc. First Int. Conf. of AIPS*, 189–197.

PLANET. 2001. The PLANET roadmap on AI planning and scheduling. http://www.planet-noe.org.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision based approach. *Journal of Artificial Intelligence Research* 4:287–339.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research* 10:323–352.

Son, T., and Baral, C. 2001. Formalizing sensing actions. a transition function based approach. *Artificial Intelligence* 19–91.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI-98*.

Wilkins, D. E. 2001. A call for knowledge-based planning. *Artificial Intelligence Magazine* 22.

Automatically Acquiring Planning Templates from Example Plans

Elly Winner and Manuela Veloso
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{elly,veloso}@cs.cmu.edu
fax: (412) 268-4801

Abstract

General-purpose planning can solve problems in a variety of domains but can be quite inefficient. Domain-specific planners are more efficient but are difficult to create. In this paper, we introduce template-based planning, a novel paradigm for automatically generating domain-specific programs, or *templates*. We present the DISTILL algorithm for learning templates automatically from example plans and explain how templates are used to solve planning problems. DISTILL converts a plan into a template and then merges it with previously learned templates. Our results show that the templates automatically learned by DISTILL compactly represent its domain-specific planning experience. Furthermore, the templates situationally generalize the given example plans, thus allowing them to efficiently solve problems that have not previously been encountered.

Introduction

Planning is a powerful tool for action selection, since it offers a guarantee that a proposed plan achieves an agent's goals. If efficient, agents could re-plan to deal with unexpected situations. However, general-purpose planning is too slow to use in most real-time systems and does not scale to large problems. In order for planning to be feasible in these situations, some knowledge about the domain being solved must be used in the planning process, either by using a domain-specific planner or by using domain-specific knowledge to narrow the search.

Many researchers have focused on learning domain-specific control knowledge for planning automatically, usually in the forms of control rules, macro operators, and plan case libraries. There have also been several efforts focusing on writing domain-specific planners to quickly solve planning problems in particular domains without resorting to generative planning. These programs are currently hand-written, but this process is tedious and often quite difficult.

In this paper, we introduce the DISTILL algorithm, which automatically extracts these domain-specific planning programs (which we call *templates*) from example plans, and show how to use them to solve planning problems. We call these domain-specific planning programs *templates*. Table 1 shows a simple example template that solves all problems in the gripper domain that involve moving balls from one room to another.

```
while (in_goal_state (at(?1:ball ?2:room)) and
       in_current_state (at(?1:ball ?3:room)) and
       not same (?2:room ?3:room) and
       in_current_state (at-robby(?5:room))) do
   if (not same (?3:room ?5:room)) then
      move(?4 ?5 ?3)
   pick(?1 ?4 ?3)
   move(?4 ?3 ?2)
   drop(?1 ?4 ?2)
```

Table 1: A simple template that solves all gripper-domain problems involving moving balls from one room to another.

In some domains, finding optimal solutions is NP-complete. Therefore, templates learned automatically from a finite number of example plans cannot be guaranteed to find optimal plans. Our goal is to extend the *solvability horizon* for planning by reducing planning times and allowing much larger problem instances to be solved. We believe that post-processing plans can help improve plan quality.

Our work on the DISTILL algorithm for learning templates focuses on converting new example plans into templates in if-statement form and merging them, where possible. Our results show that merging templates produces a dramatic reduction in space usage compared to case-based or analogical plan libraries. We also show that by constructing and combining the if statements appropriately, we can achieve automatic *situational generalization*, which allows templates to solve problems that have not been encountered before without resorting to generative planning or requiring adaptation.

We first formalize the concept of templates. Next, we present our novel DISTILL algorithm for learning templates from example plans and present our results. We then discuss how to use templates to solve planning problems. Finally, we explore future work and present our conclusions.

Related Work

Control rules (Minton 1988a; Katukam & Kambhampati 1994; Etzioni 1993) act as a search heuristic during the planning process by "recommending" at certain points which branch of the planning tree the planner should explore first. They do not reduce the complexity of the planning task,

since they cannot *eliminate* branches of the search tree. They also capture only very local information (preference choices at specific branches of the planning search tree), ignoring common sequences of actions or repeated structures in example plans. It is difficult for people to write good control rules, in part because one must know the problem-solving architecture of the planner in order to provide useful advice about how it should make choices (Minton 1988b), and computer-learned control rules are often ineffective (Minton 1988b). Also, using control rules introduces a new problem for planners: when to create and save a new rule. Unrestricted learning creates a *utility* problem, in which learning more information can actually be counterproductive: it can take longer to search through a library of rules to find the ones that would help to solve a planning problem than to find the solution to the problem by planning from scratch (Minton 1988b).

Macro operators (Fikes, Hart, & Nilsson 1972; Korf 1985) combine frequently-occurring sequences of operations into combined operators. A macro can then be applied by the planner in one step, thus eliminating the search required to find the entire sequence again. Each new macro operator adds a new branch to the planning tree at every search node. Although they can decrease the search depth, the added breadth can make planning searches slower, so, as with control rules, it is difficult to determine when to add a new macro operator. Some research has studied the problem of how to learn only the most useful macros (Minton 1985), but the efficacy of macros has, in general, been limited to hierarchically decomposable domains.

Another approach to learning planning knowledge, *case-based reasoning*, attempts to avoid generative planning entirely for many problems (Hammond 1996; Kambhampati & Hendler 1992; Leake 1996). Entire plans are stored and indexed as *cases* for later retrieval. When a new problem is presented, the case-based reasoner searches through its case library for similar problems. If an exact match is found, the previous plan may be returned with no changes. Otherwise, the reasoner must either try to modify a previous case to solve the new problem or to plan from scratch. Utility is also a problem for case-based planners; many handle libraries of tens of thousands of cases (Veloso 1994a), but, as with control rules, as the libraries get larger, the search times for relevant cases can exceed the time required to plan from scratch for a new case.

A variant of case-based reasoning that deserves mention is *analogical reasoning*, which also stores case libraries and attempts to modify previous cases to solve new problems (Veloso 1994a; 1994b). However, in addition to storing the problem and the plan, analogical reasoners also store the problem-solving rationale behind each plan step. This makes it easier to modify previous cases to solve new problems. However, deciding when to abandon modification and plan from scratch is still a problem, as are retrieving cases from the library and determining whether to save new cases.

Some work has addressed learning programs for planning, but this has been limited to learning *iterative* (Shell & Carbonell 1989) and *recursive* (Schmid 2001) macros: macro operators that contain repeated sequences of steps.

Defining Templates

A template is a domain-specific planning program that, given a planning problem (initial and goal states) as input, either returns a plan that solves the problem or returns failure, if it cannot do so. Templates are composed of the following programming constructs and planning-specific operators:

- **while** loops;
- **if** , **then** , **else** statements;
- logical structures (**and** , **or** , **not**);
- **in_goal_state** , **in_current_state** , **in_initial_state** operators;
- **same** operator;
- plan predicates; and
- plan operators.

In order for templates to capture repeated sequences in while loops and to determine that the same sequence of operators in two different plans has the same conditions, they must update a current state as they execute by simulating the effects of the operators they add to the plan. Without this capability, we would be unable to use such statements as: **while** (condition holds) **do** (body). Therefore, in order to use a template, it must be possible to simulate the execution of the plan. However, since template learning requires full models of the planning operators, this is not an additional problem.

Table 1 shows a template that solves all gripper-domain (Long 2000) problems involving moving balls between rooms. The template is composed of one while loop: while there is an ball that is not at its goal location, move to the ball (if necessary), pick up the ball, move to goal location of the object, and drop the ball.

Learning Templates: the DISTILL Algorithm

The DISTILL algorithm, shown in Table 2, learns templates from sequences of example plans, incrementally adapting the template with each new plan. One benefit of online learning is that it allows a learner with access to a planner to acquire templates on the fly in the course of its regular activity. And because templates are learned from example plans, they reflect the *style* of those plans, thus making them suitable not only for planning, but also for agent modeling.

DISTILL can handle domains with conditional effects, but we assume that it has access to a complete model of the operators and to a minimal annotated partial ordering of the observed total order plan. Previous work has shown that operator models are learnable through examples and experimentation (Carbonell & Gil 1990; Wang 1994) and has shown how to find minimal annotated partial orderings of totally-ordered plans given a model of the operators (Winner & Veloso 2002).

The DISTILL algorithm converts observed plans into templates (see "Converting Plans into Templates") and merges them by finding templates with overlapping solutions and combining them (see "Merging Templates"). In essence, this builds a highly compressed case library. However, another key benefit comes from merging templates with overlapping

Input: Minimal annotated consistent partial order $\mathcal{P}$,
current template T_i.
Output: New template T_{i+1}, updated with $\mathcal{P}$

procedure DISTILL ($\mathcal{P}$, T_i):
  $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\emptyset$)
  until match **or** can't match **do**
    if $\mathcal{A} = \emptyset$ **then**
      can't match
    else
      $\mathcal{N} \leftarrow$ Make_New_If_Statement(Assign($\mathcal{P}$, $\mathcal{A}$))
      match $\leftarrow$ Is_A_Match($\mathcal{N}$, T_i)
    if not can't match **and not** match **then**
      $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\mathcal{A}$)
  if can't match **then**
    $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\emptyset$)
    $\mathcal{N} \leftarrow$ Make_New_If_Statement(Assign($\mathcal{P}$, $\mathcal{A}$))
  $T_{i+1} \leftarrow$ Add_To_Template($\mathcal{N}$, T_i)

procedure Make_New_If_Statement($\mathcal{P}_\mathcal{A}$):
  $N \leftarrow$ empty if statement
  for all terms t_m in initial state of $\mathcal{P}_\mathcal{A}$ **do**
    if exists a step s_n in plan body of $\mathcal{P}_\mathcal{A}$ **such that**
        s_n needs t_m **or** goal state of $\mathcal{P}_\mathcal{A}$ needs t_m **then**
      Add_To_Conditions(N, **in_current_state** (t_m))
  for all terms t_m in goal state of $\mathcal{P}_\mathcal{A}$ **do**
    if exists a step s_n in plan body of $\mathcal{P}_\mathcal{A}$ **such that**
        t_m relies on s_n **then**
      Add_To_Conditions(N, **in_goal_state** (t_m))
  for all steps s_n in plan body of $\mathcal{P}_\mathcal{A}$ **do**
    Add_To_Body(N, s_n)
  return N

procedure Is_A_Match($\mathcal{N}$, T_i):
  for all if-statements I_n in T_i **do**
    if $\mathcal{N}$ matches of I_n **then**
      return true

procedure Add_To_Template($\mathcal{N}$, T_i):
  for all if-statements I_n in T_i **do**
    if $\mathcal{N}$ matches I_n **then**
      $I_n \leftarrow$ Combine(I_n, $\mathcal{N}$)
      return
  if $\mathcal{N}$ is unmatched **then**
    Add_To_End($\mathcal{N}$, T_i)

Table 2: The DISTILL algorithm: updating a template with a new observed plan.

solutions: this allows the template to find *situational generalizations* (Harris 1995) for individual sections of the plan, thus allowing it to reuse those sections when the same situation is encountered again, even in a completely different planning problem.

Generalizing Situations

We make several assumptions about what makes one planning *situation* different than another, and about how the observed planner will solve problems. We assume that two objects of the same type will be treated the same by the planner. Thus, two situations are equivalent if they contain the same number and types of objects in the same relationships. We assume that the planner will respond to equivalent situations with the same plan. This allows the DISTILL algorithm to identify common situations that occur in the solutions of several planning problems, and to extract their solutions for independent use in other problems.

Converting Plans into Templates

The first step of incorporating an example plan into the template is converting it into a parameterized if statement. First, the entire plan is parameterized. DISTILL chooses the first parameterization that allows part of the solution plan to match that of a previously-saved template. If no such parameterization exists, it randomly assigns variable names to the objects in the problem. [1]

Next, the parameterized plan is converted into a template, as formalized in the procedure Make_New_If_Statement in Table 2. The conditions on the new if statement are the initial- and goal-state terms that are *relevant* to the plan. Relevant initial-state terms are those which are needed for the plan to run correctly and achieve the goals (Veloso 1994a). Relevant goal-state terms are those which the plan accomplishes. We use a minimal annotated partial ordering (Winner & Veloso 2002) of the observed plan to compute which initial- and goal-state terms are relevant. The steps of the example plan compose the body of the new if statement. We store the minimal annotated partial ordering information for use in merging the template into the previously-acquired knowledge base.

Figure 1 shows an example minimal annotated partially ordered plan with conditional effects. Table 3 shows the template DISTILL creates to represent that plan. Note that the conditions on the generated if statement do not include all terms in the initial and goal states of the plan. For example, the template does not require that e(z) be in the initial and goal states of the example plan. This is because the plan steps do not generate e(z), nor do they need it to achieve the goals. Similarly, b(x) and the conditional effects that could generate the term c(x) or prevent its generation are also ignored, since it is not relevant to achieving the goals.

if (**in_current_state** (f(?0:type1)) **and**
  in_current_state (g(?1:type2)) **and**
  in_goal_state (a(?0:type1)) **and**
  in_goal_state (d(?1:type2))) **then**
  op1
  op2

Table 3: The template DISTILL would create to represent the plan shown in Figure 1.

Merging Templates

The merging process is formalized in the procedure Add_To_Template in Table 2. The templates learned by

[1] Two discrete objects in a plan are never allowed to map onto the same variable. As discussed in (Fikes, Hart, & Nilsson 1972), this can lead to invalid plans.

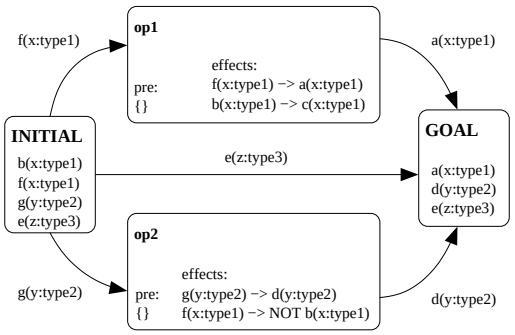


Figure 1: An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOTb$.

the DISTILL algorithm are sequences of non-nested if statements. To merge a new template into its knowledge base, DISTILL searches through each of the if statements already in the template to find one whose body (the solution plan for that problem) matches that of the new problem. We consider two plans to match if:

- one is a sub-plan of the other, or
- they overlap: the steps that end one begin the other.

If such a match is found, the two if statements are combined. If no match is found, the new if statement is simply added to the end of the template.

We will now describe how to combine two if statement templates, *if*$_1$= **if** x **then** abc and *if*$_2$= **if** y **then** b, when the body of *if*$_2$ is a sub-plan of that of *if*$_1$. This process is illustrated in Figure 2. [2]. For any set of conditions C and any step s applicable in the situation C, we define C_s to be the set of conditions that hold after step s is executed in the situation C. We also define a new function, $Relevant(C, s)$, which, for any set of conditions C and any plan step s, returns the conditions in C that are relevant to the step s.

As shown in Figure 2, merging *if*$_1$ and *if*$_2$ will result in three new if statements. We will label them *if*$_3$, *if*$_4$, and *if*$_5$. The body of *if*$_3$ is set to a and its conditions are $Relevant(x, a)$. The body of *if*$_4$ is b and its conditions are $Relevant(x_a, b)$ **or** $Relevant(y, b)$. [3] Finally, the body of *if*$_5$ is c and its conditions are $Relevant(x_a b, c)$. Whichever of *if*$_1$ or *if*$_2$ is already a member of the template is removed and replaced by the three new if statements.

Illustrative Results

Table 4 shows a template learned by the DISTILL algorithm that solves all problems in a blocks-world domain with two blocks. There are 555 such problems [4], but the template needs to store only two plan steps, and DISTILL is able to learn it from only 6 example plans.

Table 5 shows a template learned by the DISTILL algorithm to solve all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm. Although there are 1932 such problems, [5] the DISTILL algorithm is able to learn the template from only five example plans. It successfully generalizes situations within individual plans for use in other plans. Also note that only five plan steps (the length of the longest plan) are stored in the template.

Our results show that templates achieve a significant reduction in space usage compared to case-based or analogical plan libraries. In addition, templates are also able to situationally generalize known problems to solve problems that have not been seen, but are composed of previously-seen situations.

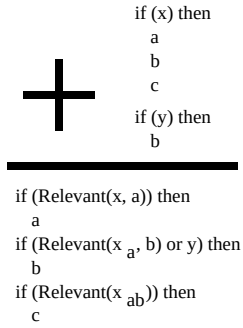


Figure 2: Combining two if statements when the body of one is a sub-plan of the body of the other.

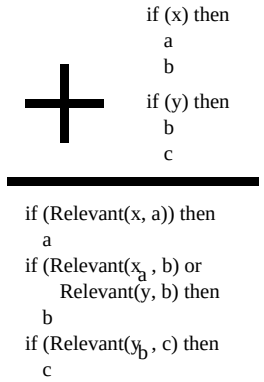


Figure 3: Combining two if statements when their bodies are overlapping.

[2] Combining two if statements with overlapping bodies is similar. It is illustrated in Figure 3

[3] Note that, though $Relevant(x, a) \subseteq x$, $Relevant(y, b) = y$.

[4] Though the initial state must be fully-specified in a problem, the goal state need only be partially specified. There are only three possible fully specified states in the blocksworld domain with two blocks, but there are 185 valid partially specified states.

[5] As previously mentioned, each problem consists of one fully-specified initial state (in this case, there are 6 valid fully-specified initial states), and one partially-specified goal state (in this case, there are 322).

if (**in_current_state** (clear(?1:block)) **and**
        in_current_state (on(?1:block ?2:block)) **and**
        (**in_goal_state** (on(?2:block ?1:block)) **or**
            in_goal_state (on-table(?1:block)) **or**
            in_goal_state (clear(?2:block)) **or**
            in_goal_state (¬on(?1:block ?2:block)) **or**
            in_goal_state (¬clear(?1:block)) **or**
            in_goal_state (¬on-table(?2:block))
        )) **then**
    move-from-block-to-table(?1 ?2)
if (**in_current_state** (clear(?1:block)) **and**
        in_current_state (clear(?2:block)) **and**
        in_current_state (on-table(?2:block)) **and**
        (**in_goal_state** (on(?2:block ?1:block)) **or**
            in_goal_state (¬clear(?1:block)) **or**
            in_goal_state (¬on-table(?2:block))
        )) **then**
    move-from-table-to-block(?2 ?1)

Table 4: A template learned by the DISTILL algorithm that solves all two-block blocks-world problems.

if (**in_current_state** (at(?3:ball ?2:room)) **and**
        in_current_state (at-robby(?1:room)) **and**
        (**in_goal_state** (at(?3:ball ?1:room)) **or**
            in_goal_state (¬at(?3:ball ?2:room)) **or**
            in_goal_state (holding(?3:ball))
        )) **then**
    Move(?1 ?2)
if (**in_current_state** (at(?3:ball ?2:room)) **and**
        in_current_state (at-robby(?2:room)) **and**
        (**in_goal_state** (at(?3:ball ?1:room)) **or**
            in_goal_state (¬at(?3:ball ?2:room)) **or**
            in_goal_state (holding(?3:ball))
        )) **then**
    Pick(?3 ?2)
if (**in_current_state** (holding(?3:ball)) **and**
        in_current_state (at-robby(?2:room)) **and**
        (**in_goal_state** (at(?3:ball ?1:room)) **or**
            (**in_goal_state** (¬at(?3:ball ?2:room)) **and**
                in_goal_state (¬holding(?3:ball)))
        )) **then**
    Move(?2 ?1)
if (**in_current_state** (holding(?3:ball)) **and**
        in_current_state (at-robby(?1:room)) **and**
        (**in_goal_state** (at(?3:ball ?1:room)) **or**
            in_goal_state (¬holding(?3:ball))
        )) **then**
    Drop(?3 ?1)
if (**in_current_state** (at-robby(?1:room)) **and**
        in_goal_state (at-robby(?2:room))
) **then**
    Move(?1 ?2)

Table 5: A template learned by the DISTILL algorithm that solves all gripper-domain problems involving one ball two rooms, and one robot with one gripper.

Planning with Templates

Our algorithm for generating plans from templates is shown in Table 6. As previously mentioned, while executing the template, we must keep track of a current state and of the current solution plan. The current state is initialized to the initial state, and the solution plan is initialized to the empty plan. Executing the template consists of applying each of the statements to the current state. Each statement in the template is either an plan step, an if statement, or a while loop. If the current statement is a plan step, make sure it is applicable, then append it to the solution plan and apply it to the current state. If the current statement is an if statement, check to see whether it applies to the current state. If it does, apply each of the statements in its body; if not, go on to the next statement. If the current statement is a while loop, check to see whether it applies to the current state. If it does, apply each of the statements in its body until the conditions of the loop no longer apply. Then go on to the next statement.

Sometimes there may be many ways to apply an if statement or a while loop to the current state. For example, if we have a statement like, "**if** (**in_current_state** (not-eaten(?a:apple))) **then** eat(?a)", and there are several uneaten apples in the current state, it is unclear which apple should be eaten. However, one of our primary assumptions is that all objects that match the conditions may be treated the same, so, in this case, it doesn't matter which apple is eaten.

Detecting and Handling Failures

There are three ways a template may fail to generate the correct solution plan. It may have run through the whole template and found no solution steps at all, though the initial state is not the same as the goal state. Or, it may have found some plan steps to execute, but, by the end of the template, did not reach the goal state. Finally, it may have found some plan steps to execute, but found that they were not applicable to the current state. A failure is detected when we attempt to execute steps that are not applicable in the current state or when the template finishes executing and its final state does not match the goal state. The way we currently handle failures is by handing the problem off to a generative planner, and then to add that new solution to the template.

Future Work

We are actively pursuing several research directions in template-based planning: enriching the process of converting an example plan into a template, refining the process of merging templates, and better handling failures by allowing the partial solutions generated by the template to be used to help guide the general-purpose planner's search.

The DISTILL algorithm's process of converting an example plan into a template can be extended to identify and extract loops in the example plan. To do this, it must be able to define the running conditions and stopping conditions for the loop and to determine when two loops can be merged. Merging two loops is more difficult than merging if statements; for example, two loops cannot be merged unless they

Input: Template T, initial state $\mathcal{I}$, current state $\mathcal{C}$ (initialized to $\mathcal{I}$), and goal state $\mathcal{G}$.
Output: Plan P that solves the given problem.

procedure Apply_Template(T, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$):
  $P \leftarrow \emptyset$
  for each statement S_n in T **do**
    $P \leftarrow P +$ Apply_Statement(S_n, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$)
  if $\mathcal{G}$ is satisfied by $\mathcal{C}$ **then**
    return P
  else
    FAIL

procedure Apply_Statement(S, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$):
  $P \leftarrow \emptyset$
  if S is an if statement **then**
    if Applies_Now(S, $\mathcal{C}$, $\mathcal{G}$) **then**
      for each statement S_i in the body of S **do**
        $P \leftarrow P +$ Apply_Statement(S_i, $\mathcal{C}$, $\mathcal{G}$)
  if S is a while statement **then**
    while Applies_Now(S, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$) **do**
      for each statement S_i in the body of S **do**
        $P \leftarrow P +$ Apply_Statement(S_i, $\mathcal{C}$, $\mathcal{G}$)
  if S is a plan step **then**
    if not Applicable(S, $\mathcal{C}$) **then**
      FAIL
    $\mathcal{C} \leftarrow$ Apply_Step(S, $\mathcal{C}$)
    $P \leftarrow S$
  return P

Table 6: Template-based plan generation.

have the same stopping conditions, even if they contain the same steps.

The process of merging templates can be refined by extending the matching capabilities of the DISTILL algorithm. Currently, DISTILL finds only the first match between a new template and previously constructed templates. We could also search for the longest match available. This would involve searching over different variable bindings for the new plan, since different bindings could result in different matches. Also, we currently allow a new template to match only one previously constructed template. We may find that merging a new template with as many other templates as possible results in better situational generalization.

Conclusions

In this paper, we contribute a formalism for automatically-generated domain-specific planning programs (templates) and the novel DISTILL algorithm, which automatically learns templates from example plans. The DISTILL algorithm first converts an observed plan into a template and then combines it with previously-generated templates. Our results show that templates learned by the DISTILL algorithm require much less space than do case libraries. Templates learned by DISTILL also support situational generalization, extracting commonly-solved situations and their solutions from stored templates so they can be reused in different problems.

Acknowledgements

This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number No. F30602-00-2-0549. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of DARPA or AFRL.

References

Carbonell, J. G., and Gil, Y. 1990. Learning by experimentation: The operator refinement method. In Michalski, R. S., and Kodratoff, Y., eds., *Machine Learning: An Artificial Intelligence Approach, Volume III*. Palo Alto, CA: Morgan Kaufmann. 191–213.

Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62(2):255–302.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):251–288.

Hammond, K. J. 1996. Chef: A model of case-based planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 261–271. American Association for Artificial Intelligence.

Harris, J. R. 1995. Where is the child's environment? a group socialization theory of development. *Psychological Review* 102(3):458–489.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55(2-3):193–258.

Katukam, S., and Kambhampati, S. 1994. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94)*, volume 1, 582–587.

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–78.

Leake, D. B., ed. 1996. *Case-Based Reasoning: experiences, lessons, and future directions*. AAAI Press/The MIT Press.

Long, D. 2000. The AIPS-98 planning competition. *AI Magazine* 21(2):13–34.

Minton, S. 1985. Selectively generalizing plans for problem-solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, 596–599. Los Angeles, CA: Morgan Kaufmann.

Minton, S. 1988a. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Boston, MA: Kluwer Academic Publishers.

Minton, S. 1988b. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Schmid, U. 2001. *Inductive Synthesis of Functional Programs*. Ph.D. Dissertation, Technische Universität Berlin, Berlin, Germany.

Shell, P., and Carbonell, J. 1989. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*.

Veloso, M. M. 1994a. *Planning and Learning by Analogical Reasoning*. Springer Verlag.

Veloso, M. M. 1994b. Prodigy/analogy: Analogical reasoning in general problem solving. In Wess, S.; Althoff, K.-D.; and Richter, M., eds., *Topics on Case-Based Reasoning*. Springer Verlag. 33–50.

Wang, X. 1994. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, 335–340.

Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*. Forthcoming.

Universal Quantification in a Constraint-Based Planner

Keith Golden and Jeremy Frank
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, CA 94035
{kgolden,frank}@ptolemy.arc.nasa.gov

Abstract

We present a general approach to planning with a restricted class of universally quantified constraints. These constraints stem from expressive action descriptions, coupled with large or infinite universes and incomplete information. The approach essentially consists of checking that the quantified constraint is satisfied for all members of the universe. We present a general algorithm for proving that quantified constraints are satisfied when the domains of all of the variables are finite. We then describe a class of quantified constraints for which we can efficiently prove satisfiability even when the domains are infinite. These form the basis of constraint reasoning systems that can be used by a variety of planners.

1 Introduction

Softbots (**soft**ware ro**bots**) are intelligent software agents that sense and act in an environment, such as a computer operating system. Since software environments are so rich, there is almost no limit to the kinds of tasks that softbots can perform, including on-line comparison shopping, managing email, scheduling meetings, and processing data. Planner-based softbots (Etzioni & Weld 1994; Golden 1997) accept goals from users and invoke a planner to find a sequence of actions (e.g., commands or program invocations) that will achieve the goal.

We are working on softbots for data processing, including image processing, managing file archives, and running scientific models. Due to the richness of softbot problem domains in general, and data processing domains in particular, the planner must handle a rich action representation. In particular, it must support:

- **Universal quantification**: Many commands and programs operate on sets of things, where membership in the set can be defined in terms of necessary and sufficient conditions. For example,
  - The Unix `ls` (or DOS `dir`) command lists all files in a given directory.
  - The "`tar x`" (or `unzip`) command extracts all files in a given archive.
  - The `grep` command returns all lines of text in a file matching a given regular expression.
  - Most image processing commands operate on all pixels in an image or in a given region of an image.
- **Incomplete information**: It is common for softbots to have only incomplete information about their environment. For example, a softbot is unlikely to know about all the files on the local file system, much less all the files available over the Internet.
- **Large or infinite universes**: The size of the universe is generally very large or infinite. For example, there are hundreds of thousands of files accessible on a typical file system and billions of web pages publicly available over the Internet. The number of *possible* files, file path names, *etc.*, is effectively infinite. Given the presence of incomplete information and the ability to create new files, it is necessary to reason about these infinite sets.
- **Constraints**: As noted in (Chien *et al.* 1997; Lansky & Philpot 1993), data processing domains typically involve a rich set of constraints. By constraints, we mean any relations whose truth values can be computed.

The intersection of these features poses some interesting challenges. For example, the intersection of universal quantification and incomplete information means that standard approaches to dealing with universal quantification in planning (Penberthy & Weld 1992) don't work, and other approaches are needed (Golden 1998; Etzioni, Golden, & Weld 1997; Babaian & Schmolze 2000). This paper discusses the effect of universal quantification and large or infinite universes on constraint reasoning and proposes a way to accommodate universally quantified constraints into a constraint-based planner.

1.1 Universally quantified constraints

Universally quantified constraints can be exceedingly useful when representing image processing domains. For example, to represent an image-processing command that performs a horizontal flip of the pixels in a rectangular region of an image between (MINX, MINY) and (MAXX, MAXY), we might write something like:

$\forall x,y$ **when**(MINX$\leq x \leq$MAXX && MINY$\leq y \leq$MAXY)
output.value(x,y) := *input*.value(MAXX+MINX-x,y)

where *output*.value(x,y) is the pixel value of the image *output* at coordinates x,y, and similarly for *input*.value. The keyword **when** indicates a conditional effect. We might also

want to specify spatial transforms of an image, such as scaling or projections, or changes to color values. All of these are convenient to represent using numeric constraints, quantified over the pixels in the image or in the specified region.

In describing commands that act on text files, it is useful to quantify over lines or characters of text. For example, the `grep` command outputs all lines of text contained in the input that match a given regular expression:

$\forall$ *line* **when** (*input*.containsLine(*line*)
&& *input*.matches(*regexp*))
output.containsLine(*line*)

Similarly, many commands operate on sets of files, which can often be expressed in terms of a regular expression satisfied by their path names. For example, the files recursively contained in directory "/foo/bar" all have a path name matching "/foo/bar/.+", where ".+" means "any string at least one character long."

In these examples, we see that it is necessary to reason about constraints on variables with either infinite or very large domains.

1.2 Road map

In the remainder of the paper, we discuss how universally quantified constraints arise in the planning process and how they are solved. In Section 2 we describe how universally quantified constraints arise as subgoals in the planning process. In Section 3 we present a general approach to solving universally quantified constraints in a constraint network and an algorithm for implementing this approach, and we prove that the algorithm is both sound and complete. The general approach is not always possible to instantiate when there are infinite domains. In Section 4 we describe how to efficiently handle constraints with infinite domains under certain restrictions. In Section 5, we discuss how these techniques apply to an Earth Science domain that we are working on, and in Section 6 we present a detailed example covering both planning and constraint reasoning. In Section 7 we describe related work, and in Section 8 we conclude and describe future work.

2 Planning with universal quantification

The traditional approach to planning with universal quantification, used by UCPOP (Penberthy & Weld 1992) and other planners works as follows:

1. Universally quantified goals are replaced with the equivalent universally ground conjunctive goal, which is called the *universal base*.
2. Universally quantified effects are *peeled* as needed; that is, given an effect

   $\forall x$ **when**(P(x)) Q(x)

   and a goal, Q(a), a new ground effect is "peeled off" the forall effect to satisfy the goal:

   when(P(a)) Q(a)

   The result is the subgoal P(a).

Replacing goals with their universal base depends on the Closed World Assumption (all objects must be known) and on the number of objects in the universe being relatively small. In softbot domains, neither assumption is likely to be valid. For example, not all files accessible to the softbot will be known, and the number of available files can easily be thousands or millions. To address the problem that not all files are known, the softbot can first achieve a subgoal of knowing all the relevant files and then proceed as above (Etzioni, Golden, & Weld 1997), but that still leaves the problem that the number of files may be large. For example, suppose the softbot has the goal of making all of the files in the user's home directory group readable. This goal could be achieved by identifying all the files (recursively) contained in the home directory "~user" and then ensuring that each one is group readable, but it would take some time just to identify all the files. It is much simpler and faster to handle them all at once with a single Unix command, which recursively makes all files in the directory group readable:

```
chmod -R g+r ~user
```

Such an approach is supported in the PUCCINI planner (Golden 1998) by directly linking from universally quantified goals to universally quantified effects. The approach used by PUCCINI presupposes that the goals and effects are all expressed in terms of predicates, like group-readable, for which entailment can be determined using simple unification. When conditions include constraints as well as predicates, determining entailment requires additional mechanisms, as we discuss in Section 2.2.

2.1 Restrictions on universally quantified expressions

Given the requirement to support universally quantified goals directly with universally quantified effects, it is important to specify exactly what kinds of expressions the language will allow, since the unrestricted case would require first-order theorem proving, which is undecidable. In a goal, the use of the keyword **when** indicates that the antecedent and consequent refer to different times. For example, the goal **when**($\Phi(\vec{x})$) $\Psi(\vec{x})$ means that for all $\vec{x}$ that satisfy $\Phi(\vec{x})$ *when the goal is given* (*i.e.*, in the initial state), we want $\Psi(\vec{x})$ to be true *when the goal is achieved* (*i.e.*, in the final state). Thus, we can specify goals like "paint all the blue chairs green" without contradiction:

$\forall c$: chair **when** (*c*.color = `blue`) *c*.color = `green`

The planner has no control of what is true in the initial state, so it will never try to achieve the goal by falsifying the antecedent. To borrow a term from contingency planning, the antecedent specifies the *context* in which the consequent should be achieved.

Effects All universally quantified effects are conditional effects, in which the antecedent specifies restrictions on the universe(s) of the quantified variable(s) and the consequent specifies what will become true for members of the specified universes. These effects are of the form

$\forall \vec{x}, \vec{y}$ (**when**($\Phi(\vec{x}, \vec{y}, \vec{w})$) $\Psi(\vec{x}, \vec{w})$).

where Φ and Ψ are conjunctive expressions and variables in $\vec{w}$ are *action parameters*, variables in action schemas that need to be instantiated in order to obtain concrete actions. Limiting Φ to a conjunction is not a real limitation, since an expression of the form

when $(\Phi_1 \vee \Phi_2)\ \Psi$

can be rewritten as the conjunction of "**when**$(\Phi_1)\ \Psi$" and "**when**$(\Phi_2)\ \Psi$."

Effects cannot contain existential quantifiers,[1] or anything equivalent to existentials, such as universal quantifiers nested within an antecedent or negation. Allowing existentials or disjunctive consequents in effects would make them non-deterministic. Given the lack of nesting and existentials, all universals can be treated as free variables. All quantified variables appearing in Ψ must also appear in Φ. This is just a sanity check, since the domain of any quantified variable that does not appear in Φ is completely unrestricted. Φ may contain additional quantified variables, $\vec{y}$, that don't appear in Ψ.

Goals and preconditions The syntax of universally quantified goals and action preconditions is the same as that of effects, except that existential quantifiers nested within the universal quantifiers are allowed in Ψ:

$$\forall \vec{x}, \vec{y}, \exists \vec{z}\ (\textbf{when}(\Phi(\vec{x},\vec{y},\vec{w}))\ \Psi(\vec{x},\vec{z},\vec{w})).$$

All universal quantifiers precede all existential quantifiers; this is simply the negation of Skolem Normal Form. Goals can also explicitly refer to time. For example, we can ask for data on last Tuesday's rainfall. Whereas effects are not really restricted compared to the commonly supported subset of ADL (Pednault 1989), the limitations on universally quantified goals are more restrictive. This particular set of restrictions was chosen to support the class of goals required for the softbot domains that interest us, while simplifying the inference procedures.

2.2 Goal regression with quantified variables

The subgoaling, or goal regression, procedure we use is similar to that used by PUCCINI. We use the peeling technique outlined above, with the addition that quantified variables in the effect can be replaced by quantified variables in the goal. Suppose we have a goal **when**$(\Phi_g)\Psi_g$ that we want to satisfy using an effect **when**$(\Phi_e)\Psi_e$. If the right-hand side (RHS) of a goal Ψ_g contains multiple conjuncts, they are solved independently, so subgoals are all of the form **when**$(\Phi_g)\psi_g$, where ψ_g is a single literal. We rely on a unification function MGU(ψ_e, ψ_g), which returns the most general unifier between the effect literal ψ_e and the goal literal ψ_g. If the literals don't unify, MGU returns $\perp$. Otherwise, it returns a set of pairs $\{\langle v_e, v_g\rangle\}$, whose interpretation is that ψ_e unifies with ψ_g if all the constraints $v_e = v_g$ are satisfied.

The Goal Regression Algorithm To determine the conditions required for {**when**$(\Phi_e)\Psi_e$} to satisfy the goal {**when**$(\Phi_g)\psi_g$}, ψ_g is matched against each of the literals $\psi_e \in \Psi_e$, using the following procedure.

```
1.  regress({when(Φe) ψe}, {when(Φg) ψg})
2.    β = MGU(ψe, ψg)
3.    C = {}
4.    Φn := copy(Φe)
5.    if β = ⊥ then return failure
6.    for each ⟨ve, vg⟩ ∈ β
7.      if ve is quantified ∀
8.        then replace ve in Φn with vg.
9.      else if vg is quantified ∀
10.       then return failure.
11.     else C := C ∧ (ve = vg).
12.   end for
13.   for each ve ∉ β
14.     replace ∀ve in Φn with ∃ve'
15.   end for
15.   return {when(Φg) Φn} ∧ C
```

The reason that unmatched universally quantified variables can be replaced with existentials (line 14) is as follows: since the effect occurs for all v that satisfy Φ, and v isn't mentioned in the goal, it is only necessary to find *some* value of v that satisfies Φ. Any new $\exists$ variables are written inside the scope of all $\forall$ variables from the goal.[2]

Examples of Goal Regression We will now present some examples of goal regression. Suppose that we have an action to give a Mothers' Day card to all new mothers:

$$\forall p_1, p_2 : \text{person}\ \ \textbf{when}\,(p_1 = \text{parent}(p_2)\ \ \&\&\ \text{sex}(p_1) = \texttt{F}\ \ \&\&\ \text{age}(p_2) < 1)\ \text{has-card}(p_1)$$

and our goal is to give a card to Mary (*i.e.*, has-card(Mary)). Applying this action to satisfy the goal will result in the subgoal

$$\exists p_2' : \text{person}\ \ (\texttt{Mary} = \text{parent}(p_2')\ \ \&\&\ \text{sex}(\texttt{Mary}) = \texttt{F}\ \ \&\&\ \text{age}(p_2') < 1)$$

That is, the action will achieve the goal if Mary is female and has a child less than one year old.

Now suppose our goal is to give a card to all mothers of newborn boys:

$$\forall m, s : \text{person}\ \ \textbf{when}\,(m = \text{parent}(s)\ \ \&\&\ \text{sex}(m) = \texttt{F}\ \ \&\&\ \text{sex}(s) = \texttt{M}\ \ \&\&\ \text{age}(s) = 0)\ \text{has-card}(m)$$

If we use the action to give a card to all new mothers, the subgoal then becomes

$$\forall m, s : \text{person}\ \ \textbf{when}\,(m = \text{parent}(s)\ \ \&\&\ \text{sex}(m) = \texttt{F}\ \ \&\&\ \text{sex}(s) = \texttt{M}\ \ \&\&\ \text{age}(s) = 0)\ \{m = \text{parent}(s);\ \text{sex}(m) = \texttt{F};\ \text{age}(s) < 1\}$$

[1] Effects *can* introduce the creation of new objects, through the **new** keyword, which is similar in some respects to an existential quantifier, but that is outside the scope this paper.

[2] For completeness, it is also necessary to determine whether two or more effects combine to achieve a universally quantified goal. A technique called goal partitioning (Golden 1997), provides this ability, but at a high computational cost. We are investigating a way to lower this cost, but that is outside the scope of this paper.

Note that the left hand side of this expression is just the left-hand side of the original goal, and the right hand side is the "peeled" left hand side (LHS) of the effect. All subgoals from conditional effects are generated the same way, so the same LHS expression is carried back through successive goal regressions.

The right-hand side (RHS) literals m =parent(s) and sex(m) = F are clearly entailed by the LHS, which we can determine by unification, using a slight variation on the regression procedure above. When the LHS entails a literal on the RHS, we say that the goal literal is *trivially satisfied*, and remove it without further subgoaling.

The remaining goal condition, a constraint, is not so straightforward. Although age(s) = 0 clearly entails age(s) < 1, the two do not unify. As we discuss below, the purpose of reasoning about universally quantified constraints is to answer the entailment question for constraints.

The Form of Subgoals Subgoals are just goals, and obey the same restrictions. However, since subgoals are generated through a specific process, outlined above, it is worth showing that the process maintains the restriction on the form of subgoals.

- Since the subgoaling process always copies the LHS of the goal to the LHS of the subgoal, all restrictions obeyed by the former are obeyed by the latter. In particular, the LHS is conjunctive and it must not contain existentials.
- The RHS of the subgoal comes from the (peeled) LHS of the effect. Since the latter is conjunctive, so is the former.
- Quantified variables appearing in the RHS but not in the LHS are existential. To see why, consider that every quantified variable that appears in the RHS either originated in the goal or is a copy of a variable from the effect.
  1. If the variable appeared in the goal, then it cannot have been in the LHS of goal, since otherwise it would be in the LHS of the subgoal, contradicting our assumption. Since it was not in the LHS of the goal, it must be an existential.
  2. If the variable came from the effect, then it must be an existential, since, as indicated in line 14 of the regression algorithm, all universals in the effect that aren't replaced by variables from the goal are replaced by existentials.

2.3 From planning to constraints

In the remainder of the paper, we discuss how to tell if the LHS of a universally quantified subgoal entails the RHS when both sides contain constraints. We will not concern ourselves further with the details of the planning algorithm. We can convert the whole planning problem into a constraint problem, but it would also be possible to use a causal-link planner like PUCCINI (Golden 1998), and perform constraint reasoning to answer questions about whether certain subgoals are trivially satisfied (*i.e.*, the LHS entails the RHS). In either case, we can separate the problem of solving constraints to check subgoal satisfaction from the rest of the planning problem.

We assume that the planner produces candidate plans that are complete except for the instantiation of some action parameters and are correct subject to a list of subgoals being "trivially" satisfied (*i.e.*, no more actions need to be inserted into the plan). The planner sends the constraint reasoner this list of subgoals, which are of the form

$$\forall \vec{x}, \vec{y}, \exists \vec{z}\, (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w}))$$

along with some additional constraints on the parameters. The job of the constraint network is to either return an assignment to all of the unspecified parameters ($\vec{w}$) such that all of the subgoals are trivially satisfied, or return failure in case there is no such assignment. If the constraint network returns failure then the candidate plan is invalid, so the planner should continue searching. Otherwise, the candidate plan, instantiated with the values for $\vec{w}$ returned by the constraint network, is a valid plan.

3 Solving Quantified Constraints

In order to determine whether the subgoals are trivially satisfied, it is necessary to reason about the solutions to the CSPs induced by Φ and Ψ. Before proceeding, we review some standard CSP notation. Let X be a set of variables. Denote the domain of $x \in X$ as $d(x)$. Let D be the set of domains. Let $k = (x_1 \ldots x_i \ldots x_n; R)$ be a constraint; $x_i \in X$ and $R \subseteq d(x_1) \times \ldots \times d(x_n)$ is a relation defining the permitted assignments to the variables. Let K be the set of constraints. Then $C(X) = (X, D, K)$ is a CSP. A *solution* to the CSP is an assignment of values to the variables such that all constraints are satisfied. Let $S(C)$ be the set of solutions to C. Let L be a relation on a set of variables U, and let $\pi_V(L)$ be the projection of the relation L onto the set $V \subseteq U$. A CSP is *k-consistent* if any consistent assignment to k-1 variables can be extended to an assignment to k variables (k=2 is arc consistency.) A CSP is *strongly k-consistent* if it is j-consistent for all j$\leq$k.

Having reviewed these definitions, we now formally define quantified constraints:

Definition 1 *Let Φ, Ψ be CSPs. We then refer to a subgoal $\forall \vec{x}, \vec{y} \exists \vec{z} (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w}))$ as a* quantified constraint, *and refer to the constraints comprising Φ, Ψ as* primitive constraints. *A quantified constraint is* satisfied *for $\vec{w} = \vec{\theta}$ iff $\pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y}, \vec{\theta})) \subseteq \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}, \vec{\theta}))$.*

The general approach to solving quantified implications is straightforward. Given an expression of the form "all things that satisfy Φ also satisfy Ψ," we identify the set of things that satisfy Φ and check whether they also satisfy Ψ. We can think of this as an empirical proof technique: we're doing nothing more than checking the validity of the expression for all members of the universe.

Given a quantified constraint

$$\forall \vec{x}, \vec{y} \exists \vec{z} (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})),$$

the variables in $\vec{w}$ must be assigned values by a search procedure. As mentioned in Section 2, these variables represent the parameters of actions; the search over these values is a search over candidate plans. During this search, we can propagate the domains of the variables in $\vec{x}, \vec{y}$ based on Φ, but

do not assign these variables. We do not propagate based on the constraints in Ψ, because these constraints do not hold if the domains of the variables in Φ are empty. Once all of these variables are assigned, we are left with the constraint

$$\forall \vec{x}, \vec{y} \exists \vec{z} (\Phi(\vec{x}, \vec{y}) \Rightarrow \Psi(\vec{x}, \vec{z})),$$

where $\vec{x}$ represents one or more universally quantified variables common to Φ and Ψ. Again, as described above, the desired semantics of this implication is that everything satisfying Φ also satisfies Ψ. Thus, we must identify the set of tuples corresponding to the assignments to $\vec{x}$ that satisfy $\Phi(\vec{x}, \vec{y})$, and check that each tuple also satisfies $\Psi(\vec{x}, \vec{z})$. To do this, we solve both $\Phi(\vec{x}, \vec{y})$ and $\Psi(\vec{x}, \vec{z})$ for $\vec{x}$. We then check to see if $\pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y})) \subseteq \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}))$. Because the quantified constraint takes the form of an implication, if the set of solutions to Φ is empty, then the implication is satisfied vacuously, and there are no constraints on the values of the variables in $\vec{x}$. If there are solutions to Φ but $\pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y})) \not\subseteq \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}))$, then the quantified constraint is not satisfied, and some other assignment to the variables in $\vec{w}$ must be generated. Otherwise, the constraint is satisfied, and the domains of $\vec{x}$ are defined by the the restrictions imposed by Φ.

If the set of tuples satisfying Φ is finite, then enumerating them and checking that each one of them satisfies Ψ is relatively straightforward, though possibly time consuming. But what if the set is infinite? In the general case, there is nothing that can be done. However, as we will see, there are some useful classes of problems where it is possible to identify the infinite set of tuples satisfying $\Phi(\vec{x}, \vec{y})$ and check that they all satisfy $\Psi(\vec{x}, \vec{z})$ using efficient constraint propagation techniques.

It should be noted that the steps presented above can be done in a variety of ways. There is no need to assign all variables in $\vec{w}$ before beginning the process of identifying the domain of $\vec{x}$. It is also possible to fix the domains of $\vec{x}$ after solving Φ before solving Ψ and only check to see if any elements of these domains are eliminated during the solving of Ψ. These refinements are left as future work.

We present an algorithm for proving that quantified constraints are satisfied. The only assumptions are that there is a way of enumerating the variables in $\vec{w}$, and that there is some way of representing the values satisfying $\Phi(\vec{x}, \vec{y})$ and $\Psi(\vec{x}, \vec{y})$. In the following sections, we discuss specific techniques for performing these operations.

```
1. isSatisfied(γ)
2.  choose assignments for all variables w⃗.
3.  for (each (∀x⃗,y⃗,∃z⃗.Φ(x⃗,y⃗) ⇒ Ψ(x⃗,z⃗)) ∈ γ)
4.    if (S(Φ(x⃗,y⃗)) ≠ ∅)
5.       for (each α⃗ ∈ π_{x⃗} S(Φ(x⃗,y⃗)))
6.           if (α⃗ ∉ π_{x⃗} S(Ψ(x⃗,y⃗)))
7.              return failure.
8.       end for
9.  end for
10. return success.
```

We now prove that the algorithm is both sound and complete:

Theorem 1 *The algorithm for checking the satisfiability of quantified constraints is sound: it will not return success if, for any quantified constraint,* $\forall \vec{x}, \vec{y}, \exists \vec{z}.\Phi(x, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})$*, there is some assignment* $\vec{\alpha}$ *to* $\vec{x}$ *such that* $\exists \vec{y}, \forall \vec{z}.\Phi(\vec{\alpha}, \vec{y}, \vec{w}) \wedge \neg\Psi(\vec{\alpha}, \vec{z}, \vec{w})$.

Proof: Suppose otherwise. Then there is some some $\vec{\alpha}$ such that $\exists \vec{y}, \forall \vec{z}.\Phi(\vec{\alpha}, \vec{y}, \vec{w}) \wedge \neg\Psi(\vec{\alpha}, \vec{z}, \vec{w})$. The algorithm will only return success if each $w_i \in \vec{w}$ is singleton, and line 7 is not reached. This happens if

1. There are no quantified constraints (line 3). This contradicts the assumption that there is such a constraint.
2. $S(\Phi(\vec{x}, \vec{y}, \vec{w})) = \emptyset$ (line 4). This is equivalent to saying Φ is false for all $\vec{x}$, contradicting our assumption that there was some $\vec{\alpha}$ for which Φ was true.
3. $S(\Phi(\vec{x}, \vec{y}, \vec{w})) \neq \emptyset$ and there is no $\vec{\alpha}$ such that $\vec{\alpha} \in \pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y}, \vec{w}))$ and $\vec{\alpha} \notin \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{y}, \vec{w}))$ (lines 5,6). That is, there is no $\vec{\alpha}$ such that $\exists \vec{y}.\Phi(\vec{\alpha}, \vec{y}, \vec{w})$ and $\forall \vec{z}.(\neg\Psi(\vec{\alpha}, \vec{z}, \vec{w}))$, contradicting the assumption that $\exists \vec{y}, \forall \vec{z}.\Phi(\vec{\alpha}, \vec{y}, \vec{w}) \wedge \neg\Psi(\vec{\alpha}, \vec{z}, \vec{w})$.

Theorem 2 *The algorithm for checking the satisfiability of quantified constraints is complete: If, for all quantified constraints,* $\forall \vec{x}, \vec{y}, \exists \vec{z}.\Phi(x, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})$*, then the algorithm returns success.*

Proof: Suppose the algorithm returns failure, but for all quantified constraints, $\forall \vec{x}, \vec{y}, \exists \vec{z}.\Phi(x, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})$. The algorithm will return failure if there is some quantified constraint for which $S(\Phi(\vec{x}, \vec{y}, \vec{w})) \neq \emptyset$ and $\vec{\alpha} \in \pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y}, \vec{w}))$ but $\vec{\alpha} \notin \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}, \vec{w}))$ (line 6). But then $\pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y}, \vec{w})) \not\subseteq \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}, \vec{w}))$, which in turn violates the assumption that for all quantified constraints, $\forall \vec{x}, \vec{y}, \exists \vec{z}.\Phi(x, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})$.

Complexity Let n_Φ be the number of variables in Φ and let d_Φ be the size of the largest domain of any variable in Φ. Denote n_Ψ and d_Ψ similarly. The complexity of the algorithm is $O((d_\Phi)^{n_\Phi} + (d_\Psi)^{n_\Psi})$, because checking the satisfiability of the constraints potentially requires enumerating the solution space for both CSPs Φ, Ψ.

4 Handling infinite universes

The general approach discussed in Section 3 works for relatively small, finite domains. To handle large or infinite domains efficiently, we need to employ special-case constraint propagation techniques. We describe one such technique in detail in this section. The technique depends on being able to represent infinite domains concisely. In Sections 4.1 and 4.2, we discuss concise representations of infinite domains for numbers and strings, and describe classes of constraints for which these concise representations can store the valid domains exactly. In Section 4.3, we describe further restrictions on the form of the quantified constraints that allow us to check the satisfiability of these quantified constraints efficiently, even if the variable domains are infinite.

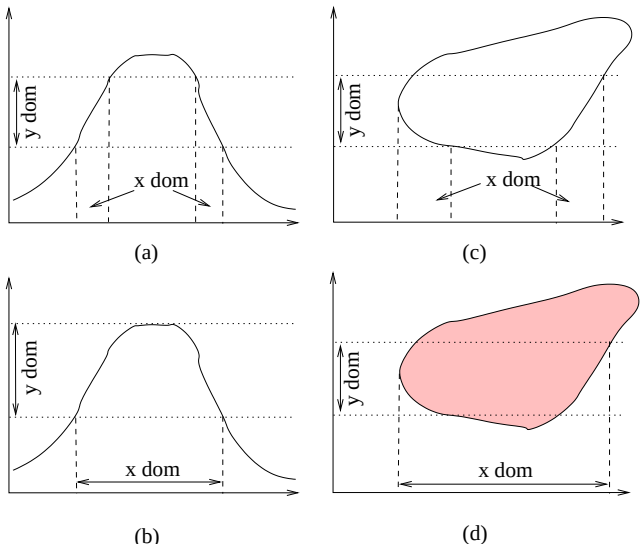


Figure 1: Reasoning about numeric functions and relations

4.1 Numeric domains

Large or infinite sets of numbers can be represented concisely using intervals. Additionally, we can determine whether one interval contains another efficiently. If we assume that all infinite numeric domains are represented as single intervals, the question of whether the domain of a numeric variable can represent exactly the possible values allowed by a constraint reduces to the question of whether the values for that variable allowed by the constraint can be represented as an interval. Assuming that the domains of the other variables in the constraint are also represented as intervals, the question then becomes whether the projection of an interval on one variable is an interval on another. We will consider both continuous (real) and discrete (integer) domains.

Continuous If the domain of x is continuous, then for every continuous function $y = f(x)$, if the domain of x is an interval, the domain of y will also be an interval. The converse is not necessarily true. However, the converse is true if f is either non-decreasing or non-increasing. If $f(x)$ increases and decreases in x, then there will be some y interval that corresponds to multiple x intervals (Figure 1a). However, if the y interval obeys certain restrictions, then the domain of x will still be an interval. In particular,

- neither of the horizontal lines representing the bounds of the y interval may cross f more than twice. Crossing twice corresponds to passing through one peak or trough in f.
- if one of the lines passes through a peak, the other line must be above the peak (Figure 1b), and if one line passes through a trough, then the other line must be below the trough.

We can apply the same sort of reasoning to relations (Figure 1c); however a special class of relations is worth noting. If any relation defines a convex region (Figure 1d), such that the relation is true for all points inside the region and false for all points outside it, then the projection of any interval on y will be an interval on x (or vice versa). Examples of convex regions are: $x < 10$, $y > 2x+1$, $x^2+y^2 \leq r^2$.

Continuous to discrete A function from a continuous (real) variable to a discrete (integer) variable is by definition not a continuous function. However, it may be regarded as a continuous function whose range is projected onto the integer number line. If such a description is valid, then the projection of any continuous interval on x will be a discrete interval on y. Going the other direction, intervals on y will map to intervals on x under the same circumstances as in the fully continuous case: non-decreasing functions, non-increasing functions, and relations defining convex regions.

Discrete A function whose domain is discrete will not, in general, project an interval onto another interval. For example, consider the simple case of $y = 2x$, where x and y are integers. The domain of y is the set of even numbers, which cannot be represented as an interval. However, when we consider relations defining convex regions, we again find that the projection of an interval is an interval. So although $y = 2x$ does not give an interval, $y \leq 2x$ does.

Other domain representations The decision to represent a numeric domain using a single interval has had a profound impact on the class of constraints that we can "solve" for particular variables. Another representation, such as a finite set of intervals, would allow additional constraints to be handled, though at the cost of additional complexity in constraint execution.

4.2 String domains

Just as infinite sets of numbers can be represented by intervals, infinite sets of strings can be represented by regular expressions. Regular expressions are a much more flexible representation than intervals, in that the set of regular expressions is closed under intersection, union and negation, whereas the set of intervals is only closed under intersection. Regular expressions (regexps) are equivalent to finite automata (FAs) in expressive power, and in fact we represent regexps as FAs, since the latter are easier to compute with. For example, deciding whether two FAs accept the same language can be done efficiently.

Concatenation The concatenation of two strings, x and y, yields another string, z. This constraint is represented as $z = x+y$. If the domains of x and y are regexps, the domain of z will simply be the regexp resulting from concatenating the regexps for x and y.

Less obviously, if the domains of x and z are regexps, the domain of y is a regexp. To construct an FA for y given FAs for x and z, we in effect traverse the FAs for z and x in parallel, exploring the cross-product of the nodes from the two FAs, starting with the pair of initial states and adding a transition $\{s_n, t_m\} \stackrel{lab}{\rightarrow} \{s_p, t_q\}$ from every node $\{s_n, t_m\}$ and every label lab such that the transitions $s_n \stackrel{lab}{\rightarrow} s_p$ and $t_m \stackrel{lab}{\rightarrow} t_q$ appear in the original FAs (see Figure 2). This is simply the operation that is performed when intersecting two FAs. Whenever

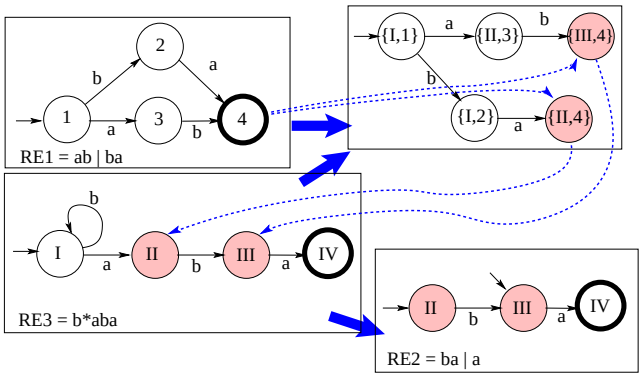


Figure 2: Given FAs for RE1 and RE3, find an FA for RE2 such that RE3 is concatenation of RE1 and RE2. First, traverse FAs for RE3 and RE1 in parallel, constructing cross-product FA (upper right). Then, identify states that are accept states for RE1 and mark the corresponding states in the FA for RE3 (shaded circles). Construct a new NFA (bottom) for RE2 by copying FA for RE3 and making marked nodes start nodes.

we reach a node $\{s,t\}$, such that node s is an accept state in the FA for x, we mark node t. After the traversal is complete, the marked nodes in the FA for z represent all of the states that can be reached by reading a string accepted by x.

A new nondeterministic FA (NFA) for y is constructed by copying the FA for z, making the start node a non-start node and making all the marked nodes new start nodes. The complexity of the whole operation is dominated by generating the cross-product FA (O(mn), where m and n are the number of nodes in the FAs for x and z, respectively). A similar procedure can be used to construct an NFA for x, given FAs for y and z.

Note that, in Figure 2, the FA for RE3 does not yet reflect the concatenation constraint. That is, RE3 accepts strings, such as bbaba, for which RE1 is not a prefix. When the constraint is enforced for all three variables, RE3 = aba | baba. It doesn't matter what order the variables are considered.

Containment The relation contains(a, b) means that string b is a substring of a. If the domain of b is a regexp r, then the domain of a is simply the regexp ".*r.*", where "." means "accept any character," so ".*" means "accept any string of zero or more characters." Less obviously, if the domain of a is a regexp, then so is the domain of b. Given an FA for a, we can construct an NFA for b by eliminating any dead-end nodes from a (that is, nodes from which it is impossible to reach an accept node), and then making all nodes in a both start and accept nodes.

4.3 Tractable Reasoning

In the previous sections we established that we can enforce consistency on a variety of constraints, even when the domains are infinite. We now show how to use these results to demonstrate that a quantified constraint is satisfied. In order to do this, we need some additional definitions. Let $C(X)$ be a CSP. Consider the hypergraph G_C, where the vertices of G_C are the variables of C and the hyperedges are the constraints. Assume we have imposed a total order o on the variables X. Freuder (Freuder 1982) defines the *width* of a variable $x \in X$ induced by ordering o as the number of variables earlier in the ordering that are in the scope of a constraint on x. The width of an ordering o is the maximum width of any variable induced by the ordering o, and the width of a CSP is the minimum width over all orderings.

We restate the following theorem from (Freuder 1982) without proof:

Theorem 3 *Let C be a CSP. If C is strongly k-consistent and the width of C is $< k$, then there is a variable order that will result in a backtrack-free search for a solution to C.*

We can now prove the following:

Corollary 1 *Let C be a CSP and assume C is strongly k-consistent and the width of C is $w < k$. Let x be the first variable in a search order inducing a width of $w < k$. Then $d(x) = \pi_x(S(C))$.*

Proof: We will show that each element of $d(x)$ can be extended to a solution to C. For each $\alpha \in d(x)$, make the assignment $x = \alpha$. Consider the assignment of any variable y. Now, since the width of C is $w < k$, we know that when we use a variable ordering that induces a width $w < k$, fewer than k variables sharing constraints with y are assigned before assigning y. Further, since we also know that C is strongly k-consistent, any consistent assignment of fewer than k variables can always be extended by one assignment. Thus, we can continue assigning variables without failure until all variables are assigned, regardless of the initial assignment to x.

Theorem 4 *Let $\forall\vec{x},\vec{y},\exists\vec{z}.\Phi(x,\vec{y},\vec{w}) \Rightarrow \Psi(x,\vec{z},\vec{w})$ be a quantified constraint such that:*

1. *Φ and Ψ share one universally quantified variable x whose domain is infinite, and x and any other infinite domain variables are only involved in constraints for which strong k-consistency can be enforced.*
2. *Φ and Ψ are strongly k-consistent.*
3. *There exists an ordering o_1 such that Φ has width $w < k$ induced by o_1 and x is the first variable in the order.*
4. *There exists an ordering o_2 such that Ψ has width $w < k$ induced by o_2 and x is the first variable in the order.*

Then the quantified constraint is satisfied if and only if $d_\Phi(x) \subseteq d_\Psi(x)$.

Proof: Since x is the only universally quantified variable shared between Φ and Ψ, we only need to check that $\pi_{\{x\}}S(\Phi(x,\vec{y},\vec{w})) \subseteq \pi_{\{x\}}S(\Psi(x,\vec{z},\vec{w}))$. Since we have assumed Φ and Ψ are k-consistent, and that each has an ordering that induces width less than k, the previous theorem allows us to conclude that all values of the first variable in the ordering are part of the solution space. But we have also assumed that, for both orderings, that variable is x. Thus, $\pi_{\{x\}}S(\Phi(x,\vec{y},\vec{w})) = d_\Phi(x)$ and $\pi_{\{x\}}S(\Psi(x,\vec{z},\vec{w})) = d_\Psi(x)$, and we are done.

We are now confronted with the problem of establishing strong k-consistency. For CSPs with variables with infinite

domains, arc-consistency can be enforced on tree-structured (width 1) CSPs in polynomial time, but no stronger result is known. In the case of finite domains, Freuder (Freuder 1990) has shown that, for certain families of CSPs called k-trees, strong k-consistency can be established in polynomial time in the number of variables. Our current implementation maintains strong k-consistency for primitive k-ary constraints over infinite numeric or string domains but only maintains arc consistency globally. Thus, we limit our attention to tree-structured CSPs.

Universally quantified constraints with infinite domains can be solved in time polynomial in the number of variables, but it is also necessary to consider the cost of computing the domain for each variable. In the case of numeric constraints, this cost is generally trivial, consisting of a few arithmetic operations. In the case of string domains, the cost depends on the size of the regular expressions representing the domains. Given two domains represented by FAs of size m and n, intersection of the two domains is O(mn), union is O($m+n$), negation is O(m), and enforcement of the constraints discussed in Section 4.2 is at worst O(mn). However, some of these operations produce NFAs as outputs, and others require deterministic FAs (DFAs) as inputs. Converting from an NFA to a DFA can result in an exponential increase in the size of the FA.

5 Applicability

We have implemented this approach in a constraint-based planner and are applying it to an Earth Science data processing domain that involves a mixture of image processing, text processing and other operations. Preliminary results indicate that the assumptions we make in this paper are valid for this domain. There are two main assumptions that potentially limit the applicability of our approach.

1. Constraints can be fully captured by the domain representation. This is really only a limitation for numeric constraints, since every string constraint in the domain can be captured fully using regexps. Most numeric constraints that appear in universally quantified expressions represent either convex regions of images or functions from real-valued measurements to integral pixel values. These constraints all obey this restriction.
2. The width of the constraint network defined by quantified constraints must less than the level of consistency enforced, and the left and right hand sides must share at most one quantified variable. This is a more serious limitation. Since the nature of the quantified constraints is dictated by quantified goals, it is possible to formulate goals that violate this restriction. Since the set of goals is open, we can't draw any conclusions about which goals are common without extensive user tests. On the other hand, in most goals we have looked at, quantified constraints result in tree-structured CSPs that trivially obey our assumptions.

6 An Image Processing Example

In this section, we illustrate the entire planning process, including generating subgoals through regression, determining entailment through unification and computing entailment for universally quantified constraints with infinite domains.

Suppose we have a grayscale image corresponding to the elevation over some region:

```
plot.xSize = XMAX;
plot.ySize = YMAX;
∀x,y: unsigned, el: real.
    when(x < XMAX && y < YMAX &&
         el=elevation(xProj(x),yProj(y)))
      plot.value(x, y) = hProj(el)
```

where words in ALL CAPS are constants, xProj and yProj are linear functions mapping the x, y coordinates of the image to the corresponding longitude, latitude that they represent, hProj is a linear function mapping elevation to pixel values in the image, with lower (blacker) values correspond to lower elevations, and elevation(x, y) is the elevation at longitude x, latitude y. The notation plot.xSize denotes the horizontal size of the image plot, and plot.value(x, y) means the pixel value at the coordinates x,y in the image plot.

Say we would like to produce a color image showing the same elevations, but highlighting particular ranges of elevation using different colors. For example, pixels corresponding to points below sea level should be blue and points above the snow line should be shades of gray.

One way to accomplish this would be by creating bitmaps or monochrome images corresponding to the the pixels of interest (*i.e.*, pixels above or below a particular value), and using these bitmaps to select the pixels on which particular operations, like coloring the pixels blue, will be performed. Suppose we have a threshold command, which takes an image, *in*, as input and has an argument specifying a threshold value, and outputs an image, *out*, the same size as the input, with a value of 255 for every pixel in the input whose value is above the threshold and a value of zero for every pixel below the threshold:

```
∀x,y: unsigned, v: pixelValue
  when ( x < in.xSize && y < in.ySize &&
         v = in.value(x, y) )
    when (v≤thresh) out.value(x,y) := 0;
    when (v>thresh) out.value(x,y) := 255;
```

where *thresh* is an action parameter of type pixelValue (*i.e.*, a variable from $\vec{w}$) denoting the threshold value, and a pixelValue is an integer in the range $[0,255]$. The use of nested **when** statements is merely a shorthand, where "**when** (Φ_1) {**when** (Φ_2) Ψ}" is equivalent to "**when** ($\Phi_1 \wedge \Phi_2$) Ψ}." Here, we focus on a single subgoal that arises during planning: to generate a threshold map, *sea*, based on elevation at sea level:

```
∀x′,y′: unsigned, elev: real.
  when(x′<XMAX && y′<YMAX &&
       elev=elevation(xProj(x′),yProj(y′)))
    when (elev > 0) sea.value(x,y) = 255;
    when (elev ≤ 0) sea.value(x,y) = 0;
```

Regressing this subgoal through the threshold action, we get:

$\forall x', y'$: unsigned, *elev*: real, $\exists v'$:unsigned
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
$x' <$ *in*.xSize;
$y' <$ *in*.ySize;
$v' =$ *in*.value(x, y);
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$ *thresh*;

We try to satisfy this goal using the initial state; specifically, letting the image *in* be plot.

$\forall x', y'$: unsigned, *elev*: real $\exists v'$: unsigned $\exists el'$:real
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
$x' <$ XMAX;
$y' <$ YMAX;
$v' =$ hProj(el');
$el' =$elevation(xProj(x'),yProj(y')));
in =plot;
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$ *thresh*;

The subgoal $el' =$elevation(xProj(x'),yProj(y'))) is trivially satisfied by unification if $el' =$*elev*. The subgoals $x' <$ XMAX and $y' <$ YMAX are also trivially satisfied. This can be determined easily by quantified constraint reasoning: The domain of x' established by the LHS is [0,XMAX-1], and the same domain is established by the RHS. Removing the satisfied terms, we get:

$\forall x', y'$: unsigned, *elev*: real $\exists v'$: unsigned $\exists el'$:real
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
$v' =$ hProj(el');
el'=elev;
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$ *thresh*;

which, simplified to it essence, gives us the following two quantified constraints.

$\forall e_1$: real. $(e_1 > 0) \Rightarrow$(hProj(e_1)>*thresh)*
$\forall e_2$: real. $(e_2 \leq 0) \Rightarrow$(hProj(e_2)$\leq$*thresh)*

Recall that hProj is an increasing linear function. Assume hProj(e)=$0.05e+42$. Note that although the domain of hProj is unbounded, the range is $[0,255]$, so all values of e below -840 map to 0, and all values above 4260 map to 255. Since we map real values onto integers, we will always round up.

These constraints share the parameter *thresh*, which needs to be assigned a value. As discussed above, there are a number of possible variable ordering strategies we could employ, the default being to choose a value for *thresh* and then see if the quantified constraints are satisfied. Say we pick the value 43. Let's tackle the constraint on e_1 first. Enforcing the LHS constraint sets the domain of e_1 to the interval $(0,\infty)$. On the RHS, propagating the value of *thresh* sets the domain of hProj(e_1) to $[44,255]$. The domain of e_1 then becomes $(20,\infty)$. Since the domain of e_1 is not the same as it was according to the LHS, the constraint is violated, so 43 is not a valid assignment to *thresh*.

Now say we pick 42. Once again, the domain of e_1 is $(0,\infty)$. This time, propagating *thresh* in the RHS makes the domain of hProj(e_1) $[43,255]$, resulting in a domain for e_1 of $(0,\infty)$, which is consistent with the LHS, so we proceed to the other forall constraint. Enforcing the LHS sets the domain of e_2 to the interval $(-\infty,0]$. Propagating the value of *thresh* in the RHS sets the domain of hProj(e_2) to $[0,42]$, resulting in a domain of $(-\infty,0]$ for e_2. Both forall constraints are consistent.

An alternative to branching on values of *thresh* would be to leave it unassigned and see if we can narrow down the choices through propagation. Working on the constraint on e_1 first, we enforce the LHS constraint, setting the domain of e_1 to the interval $(0,\infty)$. Propagating the value of e_1, the domain of hProj(e_1) is then $[43,255]$ and the domain of *thresh* is $[42,255]$. Since enforcing the RHS constraints did not shrink the domain of e_1, the first implication is valid so far. Enforcing the LHS of the second constraint sets the domain of e_2 to the interval $(-\infty,0]$. Enforcing the RHS sets the domain of hProj(e_2) to $[0,42]$ and restricts the domain of *thresh* to the singleton 42. The domain of e_2 did not shrink, and the reduction of the domain of *thresh* did not shrink the domain of e_1, so both implications hold, and the only valid parameter choice is 42, which is hProj(0), the pixel value corresponding to sea level.

7 Previous Work

Other planners, including (Golden, Etzioni, & Weld 1994; Golden 1998; Babaian & Schmolze 2000) also support universal quantification. The universally quantified statements in PSIPLAN (Babaian & Schmolze 2000) can include inequality constraints, which are used to exclude individuals from the universe of discourse. However, no prior planning systems support the ability to determine the validity of universally quantified constraints that we discuss here.

The Amphion system (Stickel *et al.* 1994) was designed to construct programs consisting of calls to elements of a software library. Amphion is supported by a first-order theorem prover. The task of assembling a sequence of image processing commands is similar to the task Amphion was designed to solve. However, the underlying representation we present here is a subset of first-order logic, enabling the use of less powerful reasoning systems. The planning problem we address is considerably easier than general program synthesis in that action descriptions are not expressive enough to describe arbitrary program elements, and the plans themselves do not contain loops or conditionals.

Ginsberg and Parkes (Ginsberg & Parkes 2000) point out that the satisfiability encoding of many STRIPS planning problems requires creating multiple grounded instances for axioms of the form $\forall xyz.(a(x,y) \wedge b(y,z) \Rightarrow c(x,z)$, then performing search over the truth values for all of the grounded instances. They propose a formulation in which $a(x,y)), b(y,z)$ and $c(x,z)$ are constraints on variables x,y,z and use this formulation to either search for units or find good variables to flip in local search. This is a different restriction on first-order logic from that we use, and furthermore, the domains of x,y,z are implicitly assumed to be finite.

L'Homme (L'Homme 1993) and Marriott and Stuckey (Marriott & Stuckey 1998) both describe methods of pre-

serving an interval representation of variables involved in arithmetic constraints while eliminating infeasible values. However, they explicitly assume that the interval representation is an unsound approximation to the domain of feasible values. Benhamou and Goualard (Benhamou & Goualard 2000) describe a method of sound but incomplete approximate propagation of infinite domains. Since we require both soundness and completeness in cases where that set may be infinite, we have made stronger restrictions on the types of reasoning performed.

8 Conclusions and Future Work

We have described a planning methodology for softbots that supports universal quantification, incomplete information, and constraints on variables with very large or infinite domains. We restrict the form of both goals and effects, while preserving the ability to express conditional effects and reason about incomplete information. Our approach uses a combination of unification and constraint reasoning to demonstrate entailment. We described an algorithm for proving or disproving entailment for constraints over finite domains, and identified a subclass of constraints for which the same algorithm can prove or disprove entailment for variables with infinite domains. This class of constraints has proven useful in the domains of planning for image processing and managing file archives.

When describing the algorithm to validate quantified constraints, we assumed that all parameters of the actions were assigned before validation occurs. As described in Section 6, there are times when it is worth deferring the decision about parameters to actions, because propagation will limit the possibilities. Exploiting these possibilities is the subject of future work.

We can potentially weaken the conditions on quantified constraints required to reason about variables with infinite domains. The condition that Φ and Ψ share only one variable can be relaxed when there is a procedure for checking the validity of the constraint without checking infinitely many values. One case is when all of the constraints describe linear equations or inequalities. In addition, it may be possible to generalize the conditions under which consistency enforcement allows us to conclude that all the values of a variable participate in solutions to a CSP. Finally, we can try to find more constraints on which we can enforce consistency when domains are infinite.

Acknowledgments We would like to thank Tania Bedrax-Weisss, Ari Jónsson, Wanlin Pang, Robert Morris and Ellen Spertus for their helpful comments and contributions to this work. This work was supported by the NASA Intelligent Systems program.

References

Babaian, T., and Schmolze, J. 2000. Psiplan: Open world planning with ψ-forms. In *Proceedings of the 5th Conference on Artificial Intelligence Planning and Scheduling*.

Benhamou, F., and Goualard, F. 2000. Universally quantified interval constraints. In *Proceedings of the 6th International Conference on the Principles and Practices of Constraint Programming*, 67–82.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

Etzioni, O., and Weld, D. 1994. A softbot-based interface to the Internet. *C. ACM* 37(7):72–6.

Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence* 89(1–2):113–148.

Freuder, E. 1982. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery* 29(1):24–32.

Freuder, E. 1990. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, 4–9.

Ginsberg, M., and Parkes, A. 2000. Satisfiability algorithms and finite quantification. In *Proceedings of the 7th Conference on Knowledge Representation*.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. AI*, 1048–1054.

Golden, K. 1997. *Planning and Knowledge Representation for Softbots*. Ph.D. Dissertation, University of Washington. Available as UW CSE Tech Report 97-11-05.

Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*.

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*.

L'Homme, O. 1993. Consistency techniques for numeric csps. In *Proceedings of the 13th International Conference on Artificial Intelligence*.

Marriott, K., and Stuckey, P. 1998. *Programming with Constraints: An Introduction*. The MIT Press.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. Principles of Knowledge Representation and Reasoning*, 324–332.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.

Binding characteristics for planning diversity

Wout van Wezel & René Jorna
Faculty of Management and Organization
University of Groningen, The Netherlands

w.m.c.van.wezel@bdk.rug.nl
r.j.j.m.jorna@bdk.rug.nl

Abstract
Planning is a field of interest in many scientific disciplines. The ambition of this theoretical paper is to offer a conceptual structure underlying the various planning approaches. Although we value diversity, we believe that a general perspective on planning should contain the following distinctions. Planning implies modeling a plan and making a plan. Furthermore, planning can be done for yourself or for others. Finally, the planning entity can be a natural, artificial, or organizational agent (actor). In this paper we explain these distinctions in greater detail. The basic inspiration is our belief that planning in the real world deserves the use of various research fields in planning and vice versa.

1. Introduction

In the past decades, planning has been subject of research in several scientific areas. These scientific areas cover a multitude of planning approaches that at first sight do not have much in common: psycho-physiological analysis, organizational science, linguistics, cognitive science, operations research, and spatial science, to name just a view. Still, no matter the research field, planning always concerns anticipating on the future and determining courses of action, so at some level, there must also be similarities between the various approaches that deal with planning. In this paper, we will show that planning is always essentially similar, and that apparent different approaches are not as different as they seem to be.

First, in section 2, we discuss what planning is. Section 3 describes how planning can be modeled generically. Section 4 describes a number of characteristics of planning. These characteristics can be used as a first starting point to compare planning approaches. This is taken up in section 5 where we provide a short overview of some planning approaches that are predominant in literature. Section 6 summarizes our perspective that planning approaches are more similar than that they are different.

2. Definition of planning

Where will we go and how do we get there? This question is an inherent part of the functioning of humans and organizations. The ability to anticipate and plan is usually seen as a required and perhaps even essential feature of intelligent systems. It is the fundament of goal directed behavior; systems that pursue goals need to take the future into account.

Intelligent systems use predictive models to behave anticipatory rather than reactively. An anticipatory system is "a system containing a predictive model of itself and/or its environment, which allows it to change a state at an instant in accordance with the model's predictions pertaining to a later instant" (Rosen, 1985). Our definition of planning will be built around this definition of anticipatory systems, by distinguishing four elements.

First, it is important to acknowledge that some entity must make the plan. Note that all kinds of entities can make plans, for example, humans, robots, computer programs, animals, organizations, etc. *Second*, someone or something must execute the plan, i.e., the intended future must somehow be attained. Again, this can be done by all kinds of entities, and the planning entities need not necessarily be involved in plan execution themselves. *Third*, the planning entity needs some kind of model of the future, since the future is essentially non-existent. This model should include states, possible actions of the executing entities and the effect of actions on the state they reside in, constraints, and goals. Planning and anticipation presume that such a predictive model is available, otherwise the chance that a plan can be executed as intended becomes a shot in the dark. The *fourth* element of planning is the plan itself. The plan signifies the belief that the planning entity has in the model of the future: the implicit or explicit actions in the plan will lead to the desired or intended future state. It can never be a full specification of the future itself because it can never be specified more precisely than the model of the future allows. It can, of course, be specified with less detail than the model of the future. Two kinds of plans are possible. First, the plan can specify the intended future state. The executing entity itself must determine how to get there. Second, the plan can specify the actions that the executing entity must perform. Although the desired future state is then not specified in the plan as such, it will, ceteris paribus, be reached by performing all specified actions.

3. Planning complexity and hierarchies

The four factors of planning entity, executing entity, model of the future, and plan are not only complex in their mutual relations, they are each also complex in themselves. Simon (1985) notes that complex systems are usually somehow ordered hierarchically in order to manage complexity. He uses the term hierarchy in the sense that a system is "composed of interrelated subsystems, each of the latter

being in turn hierarchic in structure until we reach some lowest level of elementary subsystems" (op. cit., p. 196). Note that this not necessarily means hierarchic in the sense of an authority relation; it means an ordering of parts in wholes, and these wholes are in turn parts of other wholes. We argue that planning is no exception; set aside trivial planning problems, planning always takes place hierarchically for two reasons: uncertainty and complexity (Starr, 1979). We argue that much of the differences between planning approaches can be contributed to the way in which these approaches partition the planning problem in independently solvable sub-problems. We will discuss this by describing both the structure of planning decisions and the structure of the decision domain, after which the definition of planning roughly described in Section 2 will be elaborated with the generic hierarchical view.

Making the plan: decision of the planning entity. A planning decision is a decision that determines the future. Although this sounds straightforward, there is a catch. This feature is common to all decisions, and common sense tells us that not all decisions are planning decisions. For now, we will overcome this confusion by defining a planning decision as a decision that is part of a collection of planning decisions. This is of course not a satisfactory definition due to its recursive nature. Therefore, we will get back to this issue shortly, but for now it will help us to understand and model planning decisions.

Due to the hierarchical nature of planning decisions, there are two kinds of such decisions. The first kind of decision is: *setting constraints*. A constraint is a rule that restricts the possible plan alternatives. Constraints can be determined beforehand as inherent part of the model of the future, but they can also be set during the process of plan creation. In fact, this latter type of constraint is what defines hierarchical relations between decisions; a decision at the higher hierarchical level constrains the search space for decisions at the lower level. In this way, the plan can be made stepwise. The second kind of decision is: *plan determination*. These are decisions that are not specified in greater detail by the planning entity. Note that the plan can need more detail during the execution of the plan. There are two reasons to separate constraint posing from making assignments. First, an assignment will not bring more detail in the planning. Therefore, if a planning problem is disposed of all intermediate levels, the assignments remain. Second, a constraint is a decision that should be able to handle feedback (in the planning process), if the constriction is too severe and if at a lower level a solution can not be found. Assignments should also be able to handle feedback, but only for events that arise during the execution of the plan and not during the process of plan creation.

In Figure 1 three decisions with their relations and characteristics are shown:

- There are two kinds of decisions in a hierarchy: imposing constraints (decision A in Figure 1), and making assignments (could be decision B and/or decision C in Figure 1, although these decisions could also pose constraints for further sub-decisions).
- A hierarchical planning decision is defined as a decision that constraints another decision (arrow 1). Therefore, the hierarchical relation between two decisions is based on the fact that a decision's solution space is restricted by the other decision.
- It might be difficult or impossible to make a decision within the imposed restrictions. Then, somehow this must be fed back to the decision that imposed the constraint (arrow 2).
- Decisions that share constraints must somehow be co-ordinated because their combined decisions determine whether the constraint is violated or not (arrows 3 and 4).

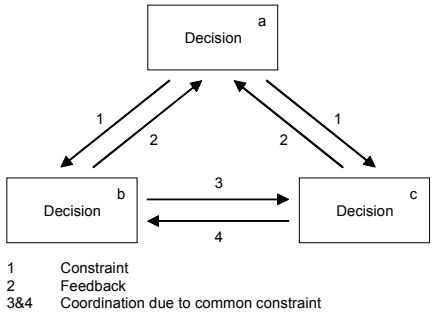


Figure 1. Basic hierarchic decision model

The significance of Figure 1 is that decision B and decision C can be decomposed in hierarchical structures themselves, and that decision A can be a sub-decision in a higher hierarchical structure. A collection of decisions with their hierarchical relations constitutes the way that a planning problem is tackled. The model in Figure 1 provides a structure that can be applied for all planning decisions on all planning levels. This view on planning implies that decisions are always structurally the same regardless of the decision level (Van Wezel, 2001).

Modeling the plan: states of the planning entity. With the notion of what a planning decision is, we can now describe (at least partly) what the decisions are about by describing the decision domain. *First*, planning is a synthetic (design) rather than an analytic (diagnosis) or modification (repair) task (Clancy, 1985). *Second*, planning involves decisions about the future and not the execution of these decisions. *Third*, an important feature of planning is that it is about choosing one alternative out of a huge number of alternatives that are structurally similar. Determining why a motor does not work is not planning (it is a diagnosis task), building a house is not planning (it is a synthetic task: however, it does not concern decisions making), but routing trains, making a production schedule, making a staff schedule, and determining the trajectory of an

automatic vehicle are planning tasks (these are synthetic tasks and concern choosing one out of a number of similar alternatives of future states without executing them).

With this demarcation we can further define planning, by explaining what is meant by 'structurally similar alternatives'. The vague connotation of the word 'similar' already indicates that it is not inherently clear whether a problem is a planning problem or not, but that in itself is not important. We propose to model a planning problem as follows. *A planning problem consists of groups of entities, whereby the entities from different groups must be assigned to each other. The assignments are subject to constraints, and alternatives can be compared on the level of goal realization*. For example, production scheduling is a problem where orders must be assigned to machines, in a shift schedule people are assigned to shifts, and in task planning tasks are assigned to time slots and resources. Different plan alternatives have the same structure (e.g., orders are assigned to machines), but a different content (e.g., in plan alternative A, "order 1" is assigned to "machine 1", and in plan alternative B, "order 1" is assigned to "machine 2"). This definition also precludes some areas that are commonly regarded as planning, for example strategic planning and retirement planning. Although the boundaries are debatable, such problems do not exhibit the third feature of planning, that is to say that alternatives are structurally similar.

Two types of sub-plans can be distinguished in a planning hierarchy: aggregation and decomposition. In *aggregation*, the dimensions that exist in the planning problem stay the same, but individual entities of a dimension are grouped. Aggregation can be used to establish boundaries or constraints for individual assignments of entities that fall within an aggregated group. In this way, several stages of aggregation can be sequentially followed whereby each stage creates boundaries for the next stage.

In the second type of sub-plan, *decomposition*, a subset of the entities that must be planned is considered as a separate planning problem. Decomposition can deal with all entities of a subset of the dimensions, all dimensions with a subset of the entities, or a combination of subsets of dimensions and entities.

The planning definition reconsidered. We now have described what planning is:

1. Planning means that a *planning entity* determines a future course of actions for an *executing entity*. These actions should lead to the desired future situation. This is based on the *model of the future* of the planning entity. The future course of actions or the desired future state is expressed by the *plan*.
2. Planning is a complex activity and often involves reasoning with incomplete information. Plans are usually made hierarchically.
3. A plan contains the assignments of entities of different categories.
4. The assignments are subject to constraints.
5. Alternatives can be compared on the level of goal realization.
6. During the process of plan creation, sub-plans can be created at other hierarchical levels than the final plan exposes.
7. Constraints and goals are visible at each hierarchical level.
8. Decisions either provide constraints for other decisions, or assign entities.
9. Partitioning takes place by disaggregation or by decomposition.

Figure 2 summarizes the elements of planning.

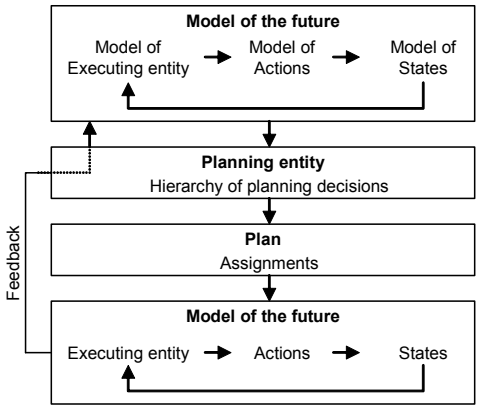


Figure 2: Elements of planning

In section 4 and section 5, we first introduce a number of generic characteristics that can be derived from this model, and then discuss some planning approaches concisely. The short introduction of the approaches serves to demonstrate our point of view that planning can be described from a generic perspective. The diversity in the planning approaches will become clear by stating questions that are based on the model in Figure 2, for example:

1. What is the planning entity? Is it a natural entity (human) or artificial entity? How does it make decisions? How is the planning decomposed; what are the partitioning criteria? In what order are the decisions made?
2. What is the executing entity? Is it an organization or an individual? Is it perhaps the planning entity itself? Do multiple executing entities have to coordinate or are they independent?
3. What kind of model of the future does the planning entity have? How flexible is the model with respect to adjustment?

The answers to such questions will bring forward not only the differences between, but also the similarities of (at first sight differing) planning approaches.

4. Generic planning characteristics

In this section, we describe a number of processing characteristics of the planning entity and of its relation to the environment. The characteristics that will be discussed are: a) closed versus open world assumption; b) the information processing mechanism of the planning entity and its architectural components, e.g., memory and

attention; c) representation; d) communication, meaning and interpretation; e) characteristics of coordination; and f) aspects of the execution of the plan.

"Closed world" versus "open world": As we already indicated the planning task itself can be called a synthetic or configuration task. In the previous section, we described a way to model plans and planning decisions. Each assignment problem consists of choosing from alternatives that are structurally the same. In classical terms this means "searching through a problem space", or, more specifically, finding a sequence of operators that modifies the current state into the goal state. From a decision perspective realizing a suitable plan or solving a planning problem requires three nearly decomposable phases. In state space descriptions the first phase is the design of a (complex) initial state, of goal state(s), and of admissible operations to change states. The second phase is, given the admissible operations, to search for an (optimal) solution. The search process may be done by exhaustive computation or by adequate evaluation functions. In many cases search does not give an optimal solution. The most one may get is a satisfying solution and even that is often not possible. Then, the third phase starts in which one goes back to the initial state and the admissible operations. Another route is chosen in the hope that a solution is found. Formulated in other words, the phases of (1) initial state, (2) search, no solution and (3a) start again with the initial state, follow the so-called "closed world" assumption. This is the necessary sequence if algorithms are applied. However, there is another way of dealing with the third phase which is more usual if humans have to make a plan. If, given the constraints and goal functions, the second phase does not give an optimal or satisfactory outcome, the planner already is so much involved in the planning process, that because he has a glimpse of the solution given the constraints, he takes his "idea" of a solution for compelling. He therefore changes the initial state(s) and the admissible operations, that is the constraints, in such a way that they fit the preconceived solution. This order of phases can be named the "open world" approach. It consists of (1) initial state, (2) search, including not finding a "real" solution and (3b) adjustment of the initial state according to the "fixed or preconceived" solution reality. In other words, the model of the future in Figure 2 is not fixed because rules are adjusted. This sequence of activities is what human planners whether in industry, in transportation planning, or in staff scheduling frequently and with great success do, but formalizing such knowledge for use in a computer program or robot is difficult. In AI the issue described above is also known as the reformulation problem.

Information processing mechanism and architectural components: During problem solving, the planning entity must process information. An information processing mechanism operationalizes the way information is selected, combined, created, and deleted. The mechanism itself needs a physical or physiological carrier. Examples are the brain as our neurological tissue, the layered connection system of a chip in a computer, a human individual in an organization, or a group of interconnected individuals in an organization. This is of course relevant when we realize that the contents of the model of the future can be restricted by the physical, physiological, or functional properties of the carrier. The most relevant distinction is the one in internal and external mechanism. With internal we mean that there is no direct access to the system from outside. Internally controlled, but not directly visible processes (not cognitively penetrable as Phylyshyn (1984) called it) take place in the system. The cognitive system and the chip are internal and consist of various architectural components, but they differ in the sense that the latter is designed which means that its operations are verifiable. External are information processing mechanisms such as groups of individuals or organizations. With respect to planning, this distinction is of course relevant if one realizes that if the plan needs to be communicated, a translation is necessary between the physical carrier and the receiver, which must be taken into account during planning. This is the case when a planning entity makes a plan that is executed by others.

(Internal) representations: Mostly, plan execution takes place in the environment of an entity. An entity that makes a plan for itself can of course misinterpret its position in the environment. Furthermore, an entity that makes a plan for others can additionally have this problem with respect to the entities that must execute the plan. In other words, the model of the future might not be accurate enough to accurately predict the result of actions on states. Representations are also immediately relevant for anticipation. A description of a future state in whatever symbol system or sign system is the core of any discussion on anticipation.

In cognitive science the conceptual framework to deal with representations can be found in the approaches of classical symbol systems, connectionism, and situated action. (Posner, 1989; Newell, 1990, Smolensky, 1988; Jorna, 1990). The basic idea in classical symbol systems theory is that humans as information processing systems have and use knowledge consisting of representations and that thinking, reasoning, and problem solving consist of manipulations of these representations at a functional level of description. Representations consist of sets of symbol structures on which operations are defined. Examples of representations are words, pictures, semantic nets, propositions or temporal strings. A representational system learns by means of chunking mechanisms and symbol transformations (Newell, 1990). A system is said to be autonomous or self-organized, if it can have a representation of its own position in the environment. This means that the system has self-representation. Connectionism and situated action are attacks on missing elements within the classical symbol system approach. Connectionism criticizes the neglect of the neurological substrate within the symbols approach and defends the relevance of sub-symbolic processing or parallel distributed processing. Situated action criticizes the neglect of the environment within the symbol approach and

emphasizes the role of actions, situatedness and "being in the world".

Communication, meaning, and interpretation: Communication means the exchange of information between different components. Depending on whether we are talking about internal or external information processing entities, communication means possibilities for and restrictions on the kinds of symbols or signs (the codes) that are used for the exchange. If we relate this to the aforementioned discussion about representations, the various kinds of signs have different consequences. Unambiguous communication requires sign notations (Goodman, 1968; Jorna, 1990), but we know that all communication between humans is not in terms of notations. If computers require sign notations and humans work with sign systems, then if the two have to communicate, the one has to adjust to the other. Until recently, most adjustments consist of humans using notations. Now, interfaces are designed that allow computers to work with (in terms of semantic requirements) less powerful, but more flexible sign systems. This means that computers can now better deal with ambiguity. For mental activities no explicitness (channels, codes etc.) is necessary; for planning as an external task it is essential.

Coordination: Coordination concerns attuning, assigning, or aligning various entities that are not self-evident unities. Information processing in a cognitive system is a kind of coordination mechanism (with no direct access). It is internal (or mental). The coordinating processor is cognition itself. No explicit code is necessary. If the code is made explicit and obeys the requirements of a notation we can design an artificial intelligent agent that in its ultimate simplicity could be a chip. In case of a set of entities that not by themselves are a coherent unity (for example individuals in an organization), various coordination mechanisms can be found, such as a hierarchy, a meta-plan, mutual adjustment, a market structure, and many others (Gazendam, 1993). The important difference with the single agent is that these coordination mechanisms are external and of course with direct access.

Planning, execution, and control: Making a plan, executing it, and monitoring the outcomes in reality are valued differently in planning your own actions and in planning actions of others (i.e., organizational processes). Planning in organizations usually is decoupled from the execution of the plan. There are two main reasons why the planner is someone else than the one who executes the plan. First, planning is a difficult job that requires expertise and experience. This is the organizational concept of task division. Second, a planner must be able to weigh the interests of many parties. Therefore, he must have knowledge about things that go beyond the limits of the individual tasks that are planned. The consequence of this decoupling is almost always inflexibility with respect to adaptation. For simple tasks as doing errands the possible division in terms of sub-tasks may be interesting, but can in reality be intertwined with flexible adaptation after unforeseen events. If the controlling entity is itself a unity, discussions about transfer, communication, sign systems to do the communication, and representations are almost trivial. This does not make the planning task itself simpler; it only prevents the occurrence of ambiguity, interpretation, and meaning variance between making the plan and executing the plan.

In the following section, we discuss a number of planning approaches. Our starting point is that planning is always in essence about the same thing: anticipating the future and determining courses of action. Still, the various planning approaches use different languages, ontologies, and descriptions of real world objects. The above discussed characteristics allow us to determine the similarities and dissimilarities of the various planning disciplines and perspectives.

5. Planning entities and planning approaches

In our definition of planning as decisions related to anticipating the future we discussed several characteristics of planning in general, and we made a distinction in decisions and states of the planning entity. Planning approaches in literature basically describe how a planning entity searches for solutions of planning problems, i.e., makes decisions about states. For example, a mathematical algorithm that calculates the optimal plan, a human that makes a shopping list, or a robot that mows my lawn. A first distinction in planning approaches is the kind of planning entity. It can either be natural or artificial. A second distinction is the executing entity. It can be the planning entity itself or it can be executed by one or more other entities. In this section we will discuss four forms: planning for yourself in a) a natural agent and b) an artificial agent, and planning for others in organizations by c) a natural agent, and d) an artificial agent (i.e., computer software).

Planning in the cognitive and behavioral sciences: Many studies in cognitive sciences deal with planning. Miller et al. (1960) define planning as a "hierarchical process in the organism that can control the order in which a sequence of operations is to be performed". Das et al. (1996) relate planning to the control of human information processing. They state that the plan is essential to connect knowledge, evaluation, and action. Newell & Simon (1972) go one step further, by describing a planning heuristic that is used by their General Problem Solver (GPS), which is used "to construct a proposed solution in general terms before working out the details. This procedure acts as an antidote to the limitation of means-ends analysis in seeing only one step ahead." (op. cit., p. 428). Basically, this heuristic uses abstract reasoning to overcome the complexity of the real world.

Early models of planning presume that planning is always a hierarchical process that proceeds according to successive refinement. Sacerdoti (1975) implemented such an approach in his computer program NOAH. Hayes-Roth & Hayes-Roth (1979) found empirical evidence that humans do not always follow such a hierarchy, but that

humans tend to plan opportunistically. Humans do not work solely linear but appear to switch in levels of abstraction and move both forward and backward in time in successive reasoning steps. Some behavior that can be explained by their model is multi-directional processing in addition to top-down processing, incremental planning, and heterarchical (i.e., network) plan structures.

Riesbeck & Schank (1989) argue that planning is based on scripts. Instead of thinking up a new plan for each problem, humans try to find a plan that is used for a previously solved comparable planning problem. Then, the basic planning activity is adaptation rather than construction. In this paradigm, planning is about memory, indexing, and learning (Hammond, 1989; Veloso, 1996). These issues are very much interrelated. Plans should be stored in such a way that it becomes easy to find an existing plan on the basis of a comparison of the new goal with already handled goals.

The discussion until now describes a number of planning issues from a cognitive perspective. Although they are sometimes approached as contradictory they are, in fact, not. More likely, the different approaches are complementary in the sense that they apply to different stages or phases of the planning process. Together, they compose a comprehensive (but not complete) model of human planning.

Analyses of human problem solving and planning have been used as input for simulations of human problem solving, and after that as a way to direct the behavior of artificial agents such as robots. The results of such simulations and applications are sometimes used in cognitive sciences to further analyze and explain behavior models. This is partly the cause that the demarcation between models of human reasoning and models of reasoning by artificial agents is not very clear.

Simulations and Robots: The planning entities that are dealt with within Artificial Intelligence (AI) are very much related to the entities that occur in the human sciences. This is not surprising, since the aim in AI is to mimic natural intelligence (Meystel, 1987). As a result, the cognitive architecture that is commonly used to describe human reasoning, is more or less simulated in AI. Artificial agents that plan their own behavior need (just as humans that plan their own task) to be able to deal with uncertainty and incomplete information. For such agents, planning is a means to reach the goal, just as it is with human problem solving. Due to the close resemblance of humans and artificial agents, planning of artificial agents is very much related to the problem solving approaches. Techniques from AI are used to let such agents function more or less independently in their environment, and react to unforeseen events (Sacerdoti, 1975; Curry & Tate, 1991; Beetz, 2000). Much of the planning research in AI stems from the wish to let autonomous actors or agents (such as robots) perform tasks without prescribing how the task should be carried out (Fikes & Nillson, 1971). Important in this respect is the work of Brooks (1999). He showed that the implicit sub-division of an intelligent system into perception, cognition, and action (motor) components does not hold. The intelligent systems he developed only had perception and action parts. "It posits both that the perception and action sub-systems do all the work and that it is only an external observer that has anything to do with cognition, by way of attributing cognitive abilities to a system that works well in the world, but has no explicit place where cognition is done." (Brooks, 1999, p. X). Recently this approach also emerged in cognitive science, especially from a physiological and neurological perspective. The 'cognition box' is opened in such a way that this box consisted of further sub-divided perception and action parts.

The planning task in organizations: In the same way as with the cognitive and behavioral sciences, the organizational sciences deal with planning at multiple time scales differently (Anthony, 1965). A common ground for planning problems in organizations is that they basically concern the coordination of supply and demand, whereby (a) the supply consists of scarce capacity and (b) the way in which this capacity is put to use can make a difference with respect to the goals in the organization (Smith, 1992; Verbraeck, 1991). Examples are producing at low costs at a production facility, having enough phone operators at a call center, or taking care that all employees work the same amount of night shifts. The way in which the coordination takes place (in other words, the planning process or the planning task) determines to a large extent the plan that eventually is executed. Not much literature or theory exists about the relations between the planning domain, the planning task, the organization of the planning, and the performance of plan execution (Jorna et al., 1996; Van Wezel, 2001). Most analyses are limited to task models, for example, McKay et al. (1995), Mietus (1994), and Sundin (1994). Lack of a theory to explain the relation between planning complexity, planning organization, task performance, and planning support makes it difficult to pinpoint the cause of the planners' dissatisfaction, to attribute the causes of poor organizational performance to the planning, or to analyze and design planning practices.

In order to make generic statements about the planning task, it is important to know what the task performance depends upon. It should be noticed that by performance we mean execution without a qualitative connotation. According to Hayes-Roth & Hayes-Roth (1979), the determinants of the planning task are problem characteristics, individual differences, and expertise. That the task performance depends upon individual differences and expertise is no surprise. This applies to all tasks. But the fact that the task performance also depends on problem characteristics leads to the statement that it is possible to describe a planning problem, at least partly, independent from the planner.

Plan generation for organization planning: It is widely accepted now that computer programs will not be able (in the foreseeable future) to replace human planners that plan organizational processes. Still, much research focuses on plan generation techniques, as it is seen as an important

part of computerized planning support. There are two mainstream approaches in plan generation techniques.

The first is about making a quantitative model that can search efficiently for good solutions. At first glance, the same kind of reasoning is used as in cognitive science: a problem space is set up and the aim is to find a state that satisfies all constraints and scores on goal functions. The states are (just like in the cognitive problem solving approaches) transformed by operators. The difference is that states and operators comprise something else than the ones in cognitive science, namely values on variables and mathematical operations (Sanderson, 1989; Baker, 1974). Most techniques are somehow limited in the kinds of characteristics they can handle. A linear programming model can not deal with nonlinear constraints, and temporal reasoning is difficult to implement in many mathematical techniques. Therefore, the domain analysis must somehow be translated into the quantitative model, and the solution must be translated back to the application domain (Prietula et al. 1994; Fox, 1983).

Second, approaches can focus on imitating the human problem solving processes in so called rule bases or expert systems, also called the transfer view (Schreiber et al., 1993), because the knowledge is extracted from a human and transferred into a computer program. For this approach, the problem solving behavior of the human scheduler must be analyzed. In terms of the human problem solver (Newell & Simon, 1972), this means that the problem space and operators must be traced and implemented. In the resulting plan generators, the available computational capacity is not used since the computer is used as a symbolic processor. It is, however, understandable for the human planner why a generated plan looks as it looks, because he would have made it more or less in the same way.

Table 1 contains the scientific planning fields that were discussed in this section and how they relate to general planning characteristics described in Section 4.

6. Conclusions

Planning approaches and planning perspectives are very different. That is an obvious conclusion if one studies planning literature. We want to challenge this conclusion by starting at the basic notion of planning. Planning is anticipating the future, implying that one has a model or representation of a future and a set of actions to get to a goal state in (the model of) the future. Within this circumscription three bifurcation points are relevant. The first relates to the distinction in modeling the plan (states of the planning entity) and making the plan (decisions of the planning entity). The second relates to planning for yourself or planning for others. The third point of bifurcation refers to the kind of entity that is making the plan: natural agent and artificial agent. This makes the confusion and the seemingly incommensurable positions more transparent.

We have three reasons for presenting the above binding perspective. In the first place there exists a growing distance between the academic world dealing with planning and real world planning issues. Misunderstanding is a bad teacher for successful mutual fertilization. In the second place the need for adequate planning analysis and planning support is growing. The simple reason is that more and more organizations are becoming closer connected to other organizations and firms require a better kind of planning (and scheduling) at all levels. In the third place we see that good ideas in one planning approach are not used in other planning approaches, simply because for example organization science (dealing with planning in organizations) has nothing to do with operations research (dealing with algorithms to solve planning problems). We do not believe in this kind of rigid separation. In this paper we wanted to present a conceptual framework in order to bridge the gaps and the seemingly incommensurabilities.

Scientific field	Cognitive & behavioural sciences	Organization planning	Robotics	Plan generation
Kind of planning entity	Natural	Natural	Artificial	Artificial
Kind of executing entity	Same as planning entity	Group of humans / organizational processes	Same as planning entity	Group of humans / organizational processes
Close vs. open world	Fixing the reality to the solution that is found; reformulate the starting-point		Searching for a solution that fits the (modelled) reality	
Information processing mechanism	Neurological: memory structures, attention processors	Translation of internal internally coded information is necessary	Information processing needs not to reckon with the outside world	Translation of internally coded information is necessary. Is designed explicitly
Architectural components	Memory, perception, motor, and central processors	Individuals and artefacts	Electronic: memory structures, attention processors	Program components: procedures, variables
Representations	Self-representation	Representation of others	Self-representation	Representation of others
Communication, meaning, and interpretation		Mostly communication with sign systems or sign sets		Communication with sign notations
Coordination	Only with respect to anticipated actions	Coordination of actions of others	Only with respect to anticipated actions	Coordination of actions of others
Planning, execution, and - control	Intertwined	Separated	Intertwined	Separated

Table 1 Characteristics of kinds of actors related to what they are planning

This is not to say that research in the planning approaches should stop, on the contrary. However, it might be the case that unknown fellow travelers exist in parallel research domains. A common conceptual framework is the basis for understanding.

References

Anthony, R.N. (1965). *Planning and Control Systems. A framework for Analysis*. Boston: Harvard.

Baker, K.R. (1974). *Introduction to sequencing and scheduling*. New York: John Wiley & Sons.

Beetz, M. (2000). Concurrent Reactive Plans. *Lecture Notes in Artificial Intelligence*. Berlin: Springer-Verlag.

Brooks, R.A. (1999). *Cambrian Intelligence: The early history of the new AI*. Cambridge: The MIT Press

Clancey, W.J. (1985). *Heuristic classification*. Stanford, CA: Stanford University.

Curry, K.W. & A. Tate (1991). O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 51, 1.

Das, J.P., Karr, B.C. & Parrila, R.K. (1996). *Cognitive Planning*. New Delhi: Sage.

Fikes, R.E., & N.J. Nillson (1971). STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence,* 2, pp 189-208.

Fox, M.S. (1983). *Constraint-Directed search: A Case Study of Job-Shop Scheduling*. Ph.D. Thesis, Carnegie Mellon University.

Gazendam, W.H.M. (1993). *Variety controls variety: on the use of organizational theories in information management*. Groningen: Wolters-Noordhoff.

Goodman, N. (1968). *Languages of Arts*. Brighton, Sussex; The Harvester Press.

Hammond, K.J. (1989). Chef. In: Riesbeck, C.K. & Schank, R.C. (1989). *Inside case-based reasoning*., Chapter 6. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Hayes-Roth, B. & Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science, 3*, pp. 275-310.

Jorna, R. J. (1990). *Knowledge representation and symbols in the mind*. Tübingen: Stauffenburg Verlag.

Jorna, R.J., Gazendam, H., Heesen H.C.& Wezel W. van (1996). *Plannen en Roosteren: Taakgericht analyseren, ontwerpen en ondersteunen*. Lansa: Leidschendam. (*Planning and Scheduling: task oriented analysis, design and support*).

McKay, K.N., F.R. Safayeni & J.A. Buzacott (1995). 'Common sense' realities of planning and scheduling in printed circuit board production. *International Journal of Production Research.* Vol. 33, Nr. 6, pp. 1587-1603.

Meystel, A.M. (1987). Theoretical foundations of planning and navigation for autonomous robots. *International Journal of Intelligent Systems*, 2, pp. 73-128

Mietus, D.M. (1994). *Understanding planning for effective decision support*. Groningen: Ph.D. thesis, University of Groningen.

Miller, G.A., Galanter, E. & Pribram, K.J. (1960). *Plans and the structure of behavior*. Holt, Rinehart & Winston.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge: Harvard University Press.

Newell, A. & Simon, H.A. (1972). *Human Problem Solving*. Englewood Cliffs, New Jersey: Prentice-Hall.

Posner, M.I. (Ed.). (1989). *Foundations of cognitive science*. Boston, Mass.: The MIT-Press.

Prietula, M.J., W. Hsu & P.S. Ow (1994). MacMerl: Mixed-Initiative Scheduling with Coincident Problem Spaces. In: Zweben, M. & Fox, M.S. *Intelligent Scheduling*. San Francisco: Morgan Kaufman.

Pylyshyn, Z.W. (1984). *Computation and Cognition*. Cambridge: The MIT Press.

Riesbeck, C.K. & Schank, R.C. (1989). *Inside case-based reasoning*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Rosen, R. (1985). *Anticipatory systems; philosophical, mathematical, and methodological foundations*. IFSR International Series on Systems Science and Engineering – Volume 1. Oxford: Pergamon Press.

Sacerdoti, E.D. (1975). The Nonlinear Nature of Plans. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 206-214.

Sanderson, P.M. (1989). The Human Planning and Scheduling Role in Advanced Manufacturing Systems: An Emerging Human Factors Domain. *Human Factors,* 31 (6), pp. 635-666.

Schreiber, G., B. Wielinga & J. Breuker (1993). *KADS. A principled approach to knowledge-based system development*. London: Academic Press.

Simon, H.A. (1985). *The sciences of the artificial*. Cambridge, Massachusetts: The MIT Press.

Smith, S.F. (1992). Knowledge-based production management: approaches, results and prospects. *Production Planning & Control*, 3, 4, pp. 350-380.

Smolensky, P. (1988). On the proper treatment of connectionism. *Behavioral and Brain Sciences, 11*, pp. 1-74.

Starr, M.K. (1979). Perspectives on disaggregation. In: L.P. Ritzman, L.J. Krajewski, W.L. Berry, S.H. Goodman, S.T. Hardy & L.D. Vitt (eds.). *Disaggregation. Problems in Manufacturing and Service Organizations*. Bosten: Martinus Nijhoff Publishing.

Sundin, U. (1994). Assignment and Scheduling. In: Breuker, J. & W. van der Velde (Eds.) *CommonKADS library for expertise modeling: reusable problem solving components*. Amsterdam: IOS Press.

Van Wezel, W. (2001). *Tasks, hierarchies, and flexibility. Planning in food processing industries*. Capelle a/d IJssel: Labyrint Publication.

Veloso, M.M. (1996). Towards Mixed-Initiative Rationale-Supported Planning. In: Tate, A. (Ed.). *Advanced Planning Technology*. Menlo Park, CA: AAAI Press.

Verbraeck, A. (1991). *Developing an Adaptive Scheduling Support Environment*. Delft: Ph.D. Thesis, University of Delft.

Planning with Complex Actions

Sheila McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
sam@ksl.stanford.edu

Ronald Fadel
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, 2A wing
Stanford University
Stanford, CA 94305, USA.
rfadel@ksl.stanford.edu

Abstract

In this paper we address the problem of planning with complex actions. We are motivated by the problem of automated Web service composition, in which planning *must* be performed using predefined complex actions or services as the building blocks of a plan. Planning with complex actions is also compelling in primitive action planning domains because it enables the exploitation of reusable subplans, potentially improving the efficiency of planning. This paper provides a formal, semantically-justified account of how to plan with complex actions using operator-based planning techniques. A key contribution of this work is the definition, characterization, and computation of preconditions and conditional effects for complex actions. While we use the situation calculus and Golog to formalize the task and our solution, the results in this paper are directly applicable to most action theories and planning systems. In particular, we have developed a PDDL-equivalent compiler that computes the preconditions and effects of complex actions, thus enabling wide-spread use of these results. Finally we provide an approach to planning that enables us to exploit deductive plan synthesis or alternatively ADL planners to plan with complex actions. Our approach to complex-action planning is sound and complete relative to the corresponding primitive action domain.

1 Introduction

Given a description of an initial state, a goal state, and a set of actions, the planning task is to generate a sequence of actions that, when performed starting in the initial state, will terminate in a goal state. Typically, actions are primitive and are described in terms of their precondition, and (conditional) effects. Our interest is in planning with complex actions as the building blocks for a plan. Complex actions are actions composed of primitive actions using typical programming language constructs. E.g.,complex actions move(obj,orig,dest) and goToAirpt(loc) are defined as:

move(obj,orig,dest) $\doteq$[1] pickup(obj,orig);putdown(obj,dest)
goToAirpt(loc) $\doteq$ **if** loc=Univ **then** shuttle(Univ,PA);
train(PA,MB);shuttle(MB,SFO) **else** taxi(loc,SFO)

Our primary motivation for investigating complex action planning is to automate *Web service composition* (e.g., [13]). Web services are self-contained Web-accessible computer programs, such as the airline ticket service at www.ual.com, or the weather service at www.weather.com. These services can be conceived as complex actions. Consider ual.com's buyAirTicket($\vec{x}$) service. This service can be described as the complex action locateFlight($\vec{x}$); **if** Available($\vec{x}$) $\wedge$ OKPrice($\vec{x}$) **then** buyAirTicket($\vec{x}$);...[2]. The task of automated Web service composition is to automatically sequence together Web services such as buyAirTicket($\vec{x}$) or getWeather($\vec{y}$) into a composition that achieves some user-defined objectives. The task of automated Web service composition is, by necessity, a problem of planning with complex actions. But how do we represent these complex actions (Web services) and how do we plan with them?

What makes planning with complex actions difficult is that the traditional characterization of actions as operators with preconditions and effects does not apply, making operator-based planning techniques such as Blackbox, FF, GraphPlan, BDDPlan, etc., inapplicable, at least at face value. In this paper we provide a formal, semantically-justified account of how to characterize, represent and precompile the preconditions and effects of complex actions, such as buyAirTicket($\vec{x}$), under a frame assumption [16]. This enables us to treat complex actions such as buyAirTicket($\vec{x}$) as planning operators and to apply standard planning tech-

[1] Denotes "defined as."
[2] Example is simplified for illustration purposes.

niques to planning with complex actions. Planning results in a plan in terms of complex actions from which a plan in terms of primitive actions is easily expanded, if desired[3].

A secondary motivation for this work is to improve the efficiency of planning by representing useful (conditional) plan segments as complex actions. As we show, our approach to planning with complex actions can dramatically improve the efficiency of plan generation by reducing the search space size and the length of a plan.

The idea of planning with some form of abstraction or aggregation is not new, and there has been a variety of work in this area including ABStrips (e.g., [17]), planning with macro-operators (e.g., [11] and [6]), and most notably HTN planning (e.g., [5]). Our work is fundamentally different from these approaches, and in particular from HTN planning, both in terms of i) the representation of complex actions (aka HTN *non-primitive tasks*), and ii) the method of planning. In this paper we precompile complex actions into planning operators described in terms of preconditions and effects that embody all possible evolutions of the complex action. In contrast, HTN planners do not use a declarative representation of the preconditions and effects of tasks. Rather, methods are associated with tasks, and tasks are pre-arranged into a network of compositions, without the full programming constructs we use to describe complex actions [18]. Further, HTN planners operate by searching for plans that accomplish task networks using task decomposition and conflict resolution. In contrast, having precompiled our complex actions, we can apply standard operator-based planning techniques to generate a plan, followed by plan expansion.

Our work is somewhat similar in methodology to [2], which proposes to encode planning constraints by compiling the constraints together with the original planning problem into a new unconstrained problem. The resultant planning problem can be solved using classical planning methods, and the resultant plan *decompiled* to provide a solution in the original problem domain. The general methodology of compilation and subsequent expansion is similar to what we propose. Nevertheless, the general problem is different. We are compiling complex actions into new plan operators. These complex actions represent Web services that we wish to reason with as black-box components. The constraints used in [2] are constraints upon the domain, and thus capture different types of planning information than our more procedural complex actions. Further the formal treatment and results are different.

We also contrast our work to the use of Golog (e.g., [12]) in planning. In this paper we use Golog as the formal language to describe complex actions, however the role these actions play in planning is very different. Golog complex actions are traditionally used to specify non-deterministic programs. In combination with deductive plan synthesis [7], a Golog program expands to a situation calculus formula which constrains the search space for a plan. This is similar to the role of domain-specific knowledge, as exemplified by systems such TALPlanner [4], BDDPlan [10] and ASP [18]. In all these systems, complex actions constrain the search space, but are not used as operators in plan construction.

The research presented in this paper is of both theoretical and practical significance. From a theoretical standpoint, we provide a semantically-justified means of characterizing the preconditions, effects and successor situations of complex actions under a frame assumption, that embodies all possible trajectories of a complex action. This enables us to not only use operator-based planning methods to plan with complex actions, but also to prove formal properties of our approach. In particular, we prove that our approach to planning is sound and complete relative to corresponding primitive action domains. From a practical perspective, analysis shows a significant increase in the efficiency of planning with complex actions, relative to primitive action planning. We illustrate potential speedup with some experiments on the briefcase domain, using the FF planner ([9]). Finally, this paper provides a principled approach to automating Web service composition, that has far-reaching application to automated component-based software composition

2 Background: Situation Calculus & Golog

We use the situation calculus and Golog to formalize the task and our solution. The expressive power and formal semantics of the situation calculus provide the theoretical foundations for our work, and for the later translation to PDDL.

Briefly, the situation calculus is a logical language for specifying and reasoning about dynamical systems [16]. In the situation calculus, the state of the world is expressed in terms of functions and relations (fluents) relativized to a particular situation s, e.g., $F(\vec{x}, s)$. A situation s is a history of the primitive actions, e.g., a, performed from an initial, distinguished situation S_0. The function $do(a, s)$ maps a situation and an action into a new situation. A situation calculus theory $\mathcal{D}$ comprises the following sets of axioms:

- domain independent foundational axioms, Σ.
- successor state axioms, $\mathcal{D}_{SS}$, one for every fluent F.
- action precondition axioms, $\mathcal{D}_{ap}$, one for every action a in the domain,which define $Poss(a, s)$.
- axioms describing the initial situation, $\mathcal{D}_{S_0}$.

[3] For many Web service applications, expansion is not relevant.

• unique names axioms for actions, $\mathcal{D}_{una}$.

Successor state axioms, originally proposed [15] to address the frame problem, are created by compiling effect axioms into axioms of this form[4]: $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$ where $\Phi_F(\vec{x}, a, s) = \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))$. (See [16, pg.28-35] for details.)

Example: In the interest of simplicity, we illustrate concepts in this paper in terms of an action theory with three actions $pickup(x), putdown(x)$ & $drop(x)$, and three fluents $holding(x), broken(x)$ & $hot(x)$. (1)-(3) comprise $\mathcal{D}_{ap}$, and (4)-(6) comprise $\mathcal{D}_{SS}$[5].

$$Poss(pickup(x), s) \equiv \neg holding(x, s) \quad (1)$$
$$Poss(drop(x), s) \equiv holding(x, s) \quad (2)$$
$$Poss(putdown(x), s) \equiv holding(x, s) \quad (3)$$
$$holding(x, do(a, s)) \equiv a = pickup(x) \vee holding(x, s) \wedge a \neq putdown(x) \wedge a \neq drop(x) \quad (4)$$
$$broken(x, do(a, s)) \equiv a = drop(x) \vee broken(x, s) \quad (5)$$
$$hot(x, do(a, s)) \equiv hot(x, s) \quad (6)$$

Golog (e.g., [12, 16, 3]) is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing extralogical constructs for assembling primitive situation calculus actions, into complex actions δ. [3] shows how these complex actions can be considered to be first-class objects in the language. $Do(\delta, s, s')$ is an abbreviation that macro-expands into a situation calculus formula, as defined inductively below. The formula says that it is possible to reach s' from s by executing a sequence of actions specified by δ [16].

Prim. action: $Do(a, s, s') \doteq Poss(a, s) \wedge s' = do(a[s], s)$
Test: $Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$
Seq.: $Do(\delta_1; \delta_2, s, s') \doteq \exists s''. Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$
Nondet. act.: $Do(\delta_1 \mid \delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$
Nondet. arg.: $Do((\pi x)\delta(x), s, s') \doteq \exists x. Do(\delta(x), s, s')$

The construct, **if** ϕ **then** δ_1 **else** δ_2 **endIf** is defined as $[\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$. The Golog language also includes nondeterministic iteration, δ^*, which executes δ zero or more times. The while-loop construct,**while** ϕ **do** δ **endWhile** is defined in terms of nondeterministic iteration as $[\phi? : \delta]^*; \neg\phi?$. For now, we exclude nondeterministic iteration, and while-loops, whose macro-expansions are second order, and which may be non-terminating. Instead, we define a bounded notion of while, **while** $_k(\phi)$ δ, which is guaranteed to terminate, and is commonly used in Web services. **while** $_k(\phi)$ δ executes like the original while-loop except that it loops at most k times, even if ϕ still holds after the k^{th} iteration. Formally, **while** $_k(\phi)$ δ corresponds to k conditional branchings as follow:

$$\textbf{while}\ _1(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ \delta\ \textbf{endIf}\ [6] \quad (7)$$
$$\textbf{while}\ _k(\phi)\ \delta \doteq \textbf{if}\ \phi\ \textbf{then}\ [\delta; \textbf{while}\ _{k-1}(\phi)\delta]\ \textbf{endIf} \quad (8)$$

[4]For space, we will only consider relational fluents here.

[5]Notation: formulae are universally quantified with maximum scope unless noted. Action arguments suppressed.

A deterministic version of the choice construct (π') is defined in a longer paper. These constructs are used to specify complex actions such as buyAirTicket($\vec{x}$) or goToAirpt(loc). Traditional usage of Golog is to apply deductive plan synthesis to find a sequence of actions $\vec{a} = [a_1, \ldots, a_n]$ that realizes a Golog program, δ relative to domain theory, $\mathcal{D}$. I.e., $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\vec{a}, S_0)$, where $do(\vec{a}, S_0)$ is an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

3 Problem: Planning with Complex Actions

Given a set of primitive actions, $\mathcal{A}$ together with an associated set of complex actions, $\Delta_{\mathcal{A}}$, our objective is to use an operator-based planner to compose complex and primitive actions to achieve some goal. To do this, we must characterize the preconditions, effects, and the situation resulting from performing a complex action.

3.1 Preconditions, Effects, Resulting Situations

For analysis, our actions $\mathcal{A}$ are axiomatized in a situation calculus action theory $\mathcal{D}$, and our complex actions $\Delta_{\mathcal{A}}$ are described in Golog. For now, we restrict our focus to terminating complex actions described in Section 2.

Resulting Situation: We wish to characterize the situation resulting from performing the complex action δ. Observe that many complex actions are nondeterministic. They may have several different executions, each terminating in a different situation. As such, we can't define a function analogous to $do(a, s)$. Instead, we introduce the abbreviation $do_{ca}(\delta, s)$ to denote a situation resulting from performing complex action δ in s. $do_{ca}(\delta, s)$ ranges over the set of *executable* situations and corresponds to a so-called *ghost situation* [16, pg.52-53], when δ is not physically realizable. The interpretation of $do_{ca}(\delta, s)$ is constrained by the following axiom, which is added to $\mathcal{D}$ producing theory $\mathcal{D}_{ca}$.

For all complex actions δ and situations s:

$$Do(\delta, s, do_{ca}(\delta, s)) \vee (\neg\exists s''. Do(\delta, s, s'') \wedge \neg executable(do_{ca}(\delta, s))) \quad (9)$$

where $executable(s)$ denotes a situation, all of whose actions in the situation action history are $Poss$-ible [16]. I.e., $executable(s) \doteq (\forall a, s^*). do(a, s^*) \sqsubseteq [7] s \rightarrow Poss(a, s^*)$ It follows that:

$$\mathcal{D}_{ca} \models \forall s. executable(s) \wedge Do(\delta, s, do_{ca}(\delta, s)) \rightarrow executable(do_{ca}(\delta, s)) \quad (10)$$

[6]**if -then -endIf** is the obvious variant of **if -then -else -endIf**.

[7]The order relation on situations in the situation tree [16].

Preconditions: $Poss_{ca}(\delta, s)$ denotes the preconditions of complex action δ. Intuitively, the preconditions of a complex action are the preconditions of all the actions that make up the execution of δ. E.g., for $a_1; a_2$,

$$Poss_{ca}(a_1; a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, do(a_1, s)).$$

This is captured tidily in the inductive definition of Do. We define the precondition of complex action δ , $Poss_{ca}(\delta, s)$ as:

$$Poss_{ca}(\delta, s) \equiv \Pi^*_\delta(s) \tag{11}$$

where $\Pi^*_\delta(s) \equiv \exists s'.Do(\delta, s, s')$. These are *intermediate* action precondition axioms.

Proposition 1 (Properties of $Poss_{ca}(\delta, s)$)
These axioms follow from $\mathcal{D}_{ca} \cup (11)$.

$$\begin{aligned} Poss_{ca}(\delta, s) &\equiv Do(\delta, s, do_{ca}(\delta, s)) \\ executable(s) \wedge Poss_{ca}(\delta, s) &\equiv executable(do_{ca}(\delta, s)) \end{aligned}$$

Effects: Intuitively the effects of a complex action are the effects of each action in the execution of δ, modulo the effects of subsequent actions. We assume that fluents whose truth value is not changed by an action, persist. $F(\vec{x}, do_{ca}(\delta, s))$ denotes that fluent F is true in the situation resulting from performing complex action δ in s. We capture the effects of complex actions as successor state axioms. Since all but trivial complex actions involve multiple intermediate situations, strictly speaking, we cannot define successor state axioms for complex actions. Rather, we define the notion of a pseudo-successor state axiom. Here we define *intermediate* pseudo-successor state axioms, making them "Markovian" in the section to follow via regression.

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi^*_F(\vec{x}, \delta, s)], \text{ where,}$$
$$\Phi^*_F(\vec{x}, \delta, s) \equiv \exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge s' = do_{ca}(\delta, s). \tag{12}$$

We need the $s' = do_{ca}(\delta, s)$ since some complex actions are nondeterministic. This enables us to identify the particular sequence of actions in the instantiation of the complex action that leads to the truth/falsity of the fluent F.

3.2 Pseudo-Markovian Complex Actions

In order to plan with complex actions as operators, we must make our characterization *pseudo-markovian*. That is, we wish to characterize the preconditions strictly in terms of the situation in which the complex action execution is initiated, and the effects, strictly in terms of the initiating and terminating situations of the complex action. To do so we appeal to regression rewriting [19], regressing over the successor state axioms for the *primitive actions* in our domain theory $\mathcal{D}$. Unfortunately, the formulae over which we need to regress are not, by definition, regressable using $\mathcal{R}$ [16, pg.62], since we are not regressing to S_0, and since the macro-expansion of $Do(\delta, s, s')$ does not yield a nested representation of situations. Since regression is a syntactic rewriting, this is problematic. We define a suitable (small) variant of Reiter's regression operator, $\mathcal{R}^s$, that first rewrites the macro-expansion of Do so that situations are expressed as nested do's, and that enables regression to an arbitrary situation s, rather than to S_0. We define the preconditions and effects of δ in terms of a set of action precondition axioms, $\mathcal{D}_{caap}$, of the form of (13) and a set of pseudo-successor state axioms, $\mathcal{D}_{caSS}$, of the form of (15).

Preconditions:
Action Precondition Axioms, $\mathcal{D}_{caap}$, one for every $\delta \in \Delta$:

$$Poss_{ca}(\delta, s) \equiv \Pi_\delta(s) \tag{13}$$

where $\Pi_\delta(s) \equiv \mathcal{R}^s[\Pi^*_\delta(s)]$ from (11), i.e. $\mathcal{R}^s[\exists s'.Do(\delta, s, s')]$.

Example (continued): Consider the complex action $pickup(x)$; **if** $hot(x)$ **then** $drop(x)$ **else** $putdown(x)$ **endIf**, which we denote as δ_1 for parsimony. Its action precondition axiom is defined as follows.

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s''))] \end{aligned} \tag{14}$$

Following our regression, $Poss_{ca}(\delta_1, s) \equiv \neg holding(x, s)$.

Successor State Axioms: Observe that while a situation calculus axiomatization has one successor state axiom for every fluent, we currently define one pseudo-successor state axiom for every fluent-complex action pair.

Pseudo-Successor State Axioms, $\mathcal{D}_{caSS}$, one for every fluent-complex action pair:

$$Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \Phi_F(\vec{x}, \delta, s)] \tag{15}$$

where $\Phi_F(\vec{x}, \delta, s) \equiv \mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)]$, $\mathcal{R}^s[\Phi^*_F(\vec{x}, \delta, s)] \equiv \mathcal{R}^s[\exists s'.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge do_{ca}(\delta, s) = s']$

Example (continued): The pseudo-successor state axiom for fluent $broken(x, do_{ca}(\delta_1, s))$ is:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow [broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \mathcal{R}^s[\exists s', s''.Poss(pickup(x), s) \wedge s'' = do(pickup(x), s) \wedge \\ &\quad ((hot(x, s'') \wedge Poss(drop(x), s'') \wedge s' = do(drop(x), s'')) \\ &\quad \vee (\neg hot(x, s'') \wedge Poss(putdown(x), s) \wedge \\ &\quad s' = do(putdown(x), s'))) \wedge \\ &\quad broken(x, s') \wedge do_{ca}(\delta_1, s) = s'] \end{aligned} \tag{16}$$

Applying our $\mathcal{R}^s$ regression operator, (16) becomes:

$$\begin{aligned} &Poss_{ca}(\delta_1, s) \rightarrow (\ broken(x, do_{ca}(\delta_1, s)) \equiv \\ &\quad \neg holding(x, s) \wedge [hot(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(drop(x), do(pickup(x), s)) \\ &\quad \vee\ \neg hot(x, s) \wedge broken(x, s) \wedge \\ &\quad do_{ca}(\delta_1, s) = do(putdown(x), do(pickup(x), s))] \ ) \end{aligned}$$

Though the computation looks complex, regression rewriting is a powerful tool and the final pseudo-successor state axiom is simple. Observe that a pseudo-successor state axiom not only defines the conditions under which fluent F is true after performing complex action δ, but it also defines the action trajectory upon which the truth of F is predicated. This is most valuable with nondeterministic actions.

Note that when the definition of $Poss_{ca}(\delta, s)$ and the intermediate pseudo-successor state axiom, ((11) and (12), respectively) are conjoined to $\mathcal{D}_{ca}$, they entail the complex action precondition axioms and the complex action pseudo-successor state axioms.

Proposition 2 $\mathcal{D}_{ca} \cup (11) \cup (12) \models \mathcal{D}_{caap} \cup \mathcal{D}_{caSS}$

Effect axioms: While we have encoded the effects of our complex actions, together with a solution to the frame problem in terms of pseudo-successor state axioms, many planners use effect axioms, rather than successor state axioms, solving the frame problem in the procedural code of their planner, rather than representationally. Hence, for completion we define effect axioms for complex actions, $\mathcal{D}_{caef}$.

Effect Axioms $\mathcal{D}_{caef}$, up to one positive effect axiom and one negative effect axiom for every fluent - complex action pair, where the execution of δ can potentially change the truth value of fluent $F \in \mathcal{F}$:

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^+(\vec{x}, s) \rightarrow F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (17)$$

$$Poss_{ca}(\delta, s) \wedge \epsilon_F^-(\vec{x}, s) \rightarrow \neg F(\vec{x}, do_{ca}(\vec{x}, \delta, s)) \quad (18)$$

Proposition 3 (Effect Axioms Entailment)

$$\mathcal{D} \cup \mathcal{D}_{caap} \cup \mathcal{D}_{caSS} \models \mathcal{D}_{caef}$$

I.e., the positive and negative complex action effect axioms are entailed by the pseudo-successor state axioms. Hence, we can easily extract effect axioms for complex actions from our pseudo-successor state axioms.

In this section we have provided a representation of the preconditions, successor state axioms and effects of complex actions under a frame assumption. They are characterized in terms of $\mathcal{D}_{caap}$, and $\mathcal{D}_{caSS}$, and follow from the semantically-justified account of actions in the situation calculus. In the section to follow, we show how these representations of complex actions lead to a simple approach to planning with complex actions.

4 Complex Actions Planning

Given our operator-based characterization of complex actions in terms of their preconditions and effects, we turn to the problem of operator-based planning with these complex actions. For now, we restrict our consideration to the subset of complex actions that are deterministic, I.e., primitive actions a, sequences $\delta_1; \delta_2$, conditional **if** ϕ **then** δ_1 **else** δ_2 **endIf**, and **while** $_k(\phi)$ δ, plus others described in a longer paper.

Following the problem statement in Section 3, our approach is to take as input [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] – an action theory $\mathcal{T}_\mathcal{A}$ and a set of complex actions $\Delta_\mathcal{A}$, both defined in terms of actions in $\mathcal{A}$. Following the results in the previous section, we **COMPILE** [$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] into a new theory $\mathcal{T}_{\mathcal{A}'}$, in terms of actions $\mathcal{A}'$ (generally $\mathcal{A} \subseteq \mathcal{A}'$), where each complex action in $\Delta_\mathcal{A}$ corresponds to a new primitive action in $\mathcal{A}'$. Next, **PLAN**ning is performed in $\mathcal{T}_{\mathcal{A}'}$ to produce a plan in terms of $\mathcal{A}'$. To extract a plan in terms of the primitive actions, we **REWRITE** the theory, replacing primitive actions from $\mathcal{A}'$ by their corresponding complex actions, $\Delta_\mathcal{A}$. Finally, using $\mathcal{T}_\mathcal{A}$, the resulting sequence of primitive actions is **EXPAND**ed from the plan in $\mathcal{A}'$ into a plan in terms of $\mathcal{A}$.

Next, we show how this approach is realized, first using the situation calculus and deductive plan synthesis, and then using an arbitrary operator-based planning system that allows conditional effects of actions in PDDL.

4.1 Deductive Plan Synthesis and Expansion

The following is the theory with primitive actions $\mathcal{T}_\mathcal{A}$.

$\mathcal{T}_\mathcal{A}$= $\Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{SS} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$.

(1) COMPILE[$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$] $\rightarrow$ $\mathcal{T}_{\mathcal{A}'}$:
- Define $\mathcal{D}_{caap}$ and $\mathcal{D}_{caSS}$ as described in Section 3.2.
- $\mathcal{D}_{ap}^{'} \leftarrow \mathcal{D}_{caap} \cup \mathcal{D}_{ap}$. $\mathcal{D}_{SS}^{'} \leftarrow \mathcal{D}_{caSS}$. $\mathcal{A}' \leftarrow \mathcal{A}$.
- $\forall \delta_i \in \Delta_A$: Create a primitive action $a_i^{'}$. Substitute "$a_i^{'}$" for "δ_i" in $\mathcal{D}_{ap}^{'}$& $\mathcal{D}_{SS}^{'}$. $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{a_i^{'}\}$.
- $\mathcal{D}_{SS}^{'} \leftarrow$ MERGE($\mathcal{D}_{SS}^{'}, \mathcal{D}_{SS}$). Update $\mathcal{D}_{una}$ to $\mathcal{D}_{una}^{'}$.

COMPILE produces a situation calculus theory in actions $\mathcal{A}'$, comprising all the original primitive actions $\mathcal{A}$ plus new primitive actions corresponding to each complex action in $\Delta_\mathcal{A}$. $\mathcal{T}_{\mathcal{A}'} = \Sigma \cup \mathcal{D}_{ap}^{'} \cup \mathcal{D}_{SS}^{'} \cup \mathcal{D}_{una}^{'} \cup \mathcal{D}_{S_0}$

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal] $\rightarrow$ plan[$\mathcal{A}'$]: Given a goal formula, $Goal(s)$ in the language of $\mathcal{T}_\mathcal{A}$, planning can be achieved via deductive plan synthesis in $\mathcal{T}_{\mathcal{A}'}$. Following [7, 16], $\mathcal{T}_{\mathcal{A}'} \vdash \exists s.Goal(s)$. From the binding of s, we can read off a plan $[a_1^{'} \ldots, a_n^{'}], a_i^{'} \in \mathcal{A}'$, a plan in $\mathcal{A}'$. [16] describes a variety of situation calculus planners implemented in Prolog.

(3) REWRITE[plan[$\mathcal{A}'$]] $\rightarrow$ plan[$\mathcal{A}$,$\Delta_\mathcal{A}$]: Rewrite the plan $[a_1^{'} \ldots, a_n^{'}], a_i^{'} \in \mathcal{A}'$ as a plan $[\alpha_1, \ldots, \alpha_n]$ in ($\mathcal{A}$,$\Delta_\mathcal{A}$), where $\alpha_i = a_i$, for all $a_i^{'} \in \mathcal{A}$, otherwise α_i equals the corresponding δ_i from the compilation in Step (1).

(4) EXPAND[plan[$\mathcal{A}$,$\Delta_\mathcal{A}$],$\mathcal{T}_\mathcal{A}$] $\rightarrow$ plan[$\mathcal{A}$]: Use our same deductive machinery to extract a final plan in $\mathcal{A}$ from our plan in ($\mathcal{A}$,$\Delta_\mathcal{A}$), by expanding the complex actions in $[\alpha_1, \ldots, \alpha_n]$. We do so by trivially rewriting our plan as a

sequence of complex actions in Golog $\delta_G = \alpha_1; \alpha_2; \ldots; \alpha_n$. A Golog interpreter, written in Prolog will return a binding for situation s' where $\mathcal{T}_\mathcal{A} \vdash (\exists s').Do(\delta_G, s, s') \wedge Goal(s')$. From the situation s' we can read off a plan $[a_1 \ldots, a_m], a_j \in \mathcal{A}$.

Note that every plan our approach finds is also a plan in the original primitive action theory, and vice-versa.

Theorem 1 *$\mathcal{T}_{\mathcal{A}'}$ and $\mathcal{T}_\mathcal{A}$ are defined as in Section 4.1. Let $Goal(s)$ be a formula uniform in s such that $Goal(s) \in \mathcal{L}(\mathcal{T}_\mathcal{A}) \cap \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, the intersection of the languages of $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_{\mathcal{A}'}$ respectively. For all ground situations σ' of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{T}_{\mathcal{A}'} \models executable(\sigma') \wedge Goal(\sigma')$ iff there exists a ground situation σ of $\mathcal{T}_\mathcal{A}$ such that $\mathcal{T}_\mathcal{A} \models executable(\sigma) \wedge Goal(\sigma)$ and EXPAND[REWRITE[seq(σ')],$\Delta_\mathcal{A}$] = seq(σ), where $seq(do(\vec{a}, s)) = \vec{a}$.*

Proof Sketch: First, by construction of $\mathcal{T}_{\mathcal{A}'}$, $\mathcal{L}(\mathcal{T}_\mathcal{A}) \subset \mathcal{L}(\mathcal{T}_{\mathcal{A}'})$, and, for any action a in $\mathcal{T}_\mathcal{A}$, $\mathcal{T}_{\mathcal{A}'}$ contains the successor state and action precondition axioms of a in $\mathcal{T}_\mathcal{A}$. It follows that, for any term s which denotes a situation in the language of $\mathcal{T}_\mathcal{A}$, $\mathcal{T}_\mathcal{A} \models Goal(s) \wedge executable(s)$ iff $\mathcal{T}_{\mathcal{A}'} \models Goal(s) \wedge executable(s)$. Second, since in any situation s, the expansion of an executable complex action is also executable and has the same effects, for any executable complex plan $[a'_1, a'_2, ..., a'_n]$ in $\mathcal{T}_{\mathcal{A}'}$, EXPAND[REWRITE$[a'_1, a'_2, ..., a'_n]$,$\Delta_\mathcal{A}$]$= [a_1, a_2, ..., a_m]$ is an executable plan in $\mathcal{T}_{\mathcal{A}'}$, and $do([a'_1, a'_2, ..., a'_n], S_0)$ and $do([a_1, a_2, ..., a_m], S_0)$ are the same *states* in $\mathcal{T}_{\mathcal{A}'}$(i.e. fluents has the same truth value in both situations). Finally, by definition of the REWRITE and EXPAND steps, $a_1, a_2, ..., a_m$ are actions in $\mathcal{T}_\mathcal{A}$. It follows that $do([a_1, a_2, ..., a_m], S_0)$ is a term in the language of $\mathcal{T}_\mathcal{A}$ which denotes a situation, and thus $\mathcal{T}_\mathcal{A} \models Goal(do([a_1 ... a_m], S_0)) \wedge executable(do([a_1 ... a_m], S_0))$ if and only if $\mathcal{T}_{\mathcal{A}'} \models Goal(do([a'_1 ... a'_n], S_0)) \wedge executable(do([a'_1 ... a'_n], S_0))$.

Planning in $\mathcal{T}_{\mathcal{A}'}$ is sound and complete with respect to planning in $\mathcal{T}_\mathcal{A}$. Thus our approach to complex action planning via transformation of the theory is well-founded.

4.2 Exploiting Existing Operator-Based Planners

Our approach is not limited to planners realized in the situation calculus. Most popular planners don't use a successor state axioms representation of the effects of actions. E.g., all of the planners that participate in the AIPS Planning competition use PDDL as an initial specification of the action theory. In this section we show how to exploit an arbitrary operator-based planner that accepts PDDL planning domains with conditional effects [14], in order to plan with complex actions.

(1) COMPILE[$\mathcal{T}_\mathcal{A}$,$\Delta_\mathcal{A}$]: Rather than employing successor state axioms, PDDL describes the effects of actions in terms of (conditional) effects without a solution to the frame problem. Section 3.2 provides a semantic justification for an intuitive algorithm that compiles a PDDL representation of the preconditions and effects of actions in $\mathcal{T}_\mathcal{A}$, together with complex actions $\Delta_\mathcal{A}$ into a new PDDL representation of preconditions and effects in $\mathcal{T}_{\mathcal{A}'}$, without going through the intermediate stage of creating successor state axioms. (We have such an algorithm, but space precluded its inclusion in this paper.) Intuitively, the effects of a complex action are the effects of each action in the execution of δ, modulo subsequent effects.

(2) PLAN[$\mathcal{T}_{\mathcal{A}'}$,goal]: Given a compiled PDDL representation $\mathcal{T}_{\mathcal{A}'}$, we can generate a plan with any planner that accepts PDDL with conditional effects. (We used FF [9].)

(3) REWRITE & (4) EXPAND: We can use STEP (3)-(4) from Section 4.1. Alternatively, we can write a (fairly straightforward) algorithm to expand the final plan in $\mathcal{A}'$. For maximal efficiency, we would cache the conditions that uniquely determine the expansion of each complex action in a situation.

5 Elaborations on Complex Action Planning

In this section we examine elaborations on complex action planning. In particular, we examine the conditions underwhich adding complex actions to a theory causes other actions to be redundant and thus removable. Removing redundant actions is desirable because it reduces the plan search space. In an extended version of this paper, we discuss concurrency in complex action planning.

5.1 Removing Weaker Actions

When a complex action δ_1 is compiled into a primitive action theory as a new primitive action a_1, another primitive action, a_2 may become redundant in the sense that in any situation s, if a_2 is possible, a_1 is also possible and has exactly the same effects as a_2. More generally, we define the notion that primitive action a_1 is *stronger* than primitive action a_2, $a_1 \succeq a_2$ (and conversely that a_2 is weaker than a_1) as follows:

$$a_1 \succeq a_2 \leftrightarrow [\, Poss(a_2, s) \rightarrow Poss(a_1, s) \wedge SS(do(a_1, s), do(a_2, s))\,] \quad (19)$$

where $SS(s, s')$ is an abbreviation for the first-order formulae that is true iff situations s and s' have the same state. The relation $\succeq$ is a preorder (it is reflexive and transitive). It follows that for any situation calculus theory T and goal formula $G(s)$, $T \models \exists s.G(s)$ iff $T' \models \exists s.G(s)$, where T' is T with all weaker actions removed.

Note that removal of weaker primitive actions may result in removal of the optimal plan. In particular, if $a_1 \succeq a_2$ and a_1

is a compiled complex action that can expand into multiple primitive actions, then by removing a_2, we may lose the optimal plan with respect to the number of primitive actions in our initial domain. Also note that the notion of stronger actions does not capture all the conditions underwhich an action is redundant. In particular, a_2 may be conditionally redundant, or it might be redundant relative to a_1 in some situation, and redundant relative to a_3 in others.

Example: Let a_1 and a_2 be primitive actions in $\mathcal{T}_\mathcal{A}$, let $a_2$achieve the preconditions for a_1, and let $Poss(a_1)$ be the situation suppressed expression [16, pg.112] for $Poss(a_1, s)$. Define complex action δ_3 as **if** $\neg Poss(a_1)$ **then** a_2 **endIf** ; a_1. If we compile a_1, a_2 and δ_3 in $\mathcal{T}_\mathcal{A}$ into primitive actions a_1', a_2' and a_3' in $\mathcal{T}_{\mathcal{A}'}$, following Section 4.1, then it follows that $a_3' \succeq a_1'$.

5.2 Irrelevant Actions with Respect to a Goal

Let $G(s)$ be a goal predicate that is true iff s satisifies the goal formula. If the direct effect of an action a can never make $G(s)$ true, and if a cannot directly achieve the preconditions of any of the actions, then a is irrelevant with respect to goal predicate $G(s)$ and can be removed. Formally, given a primitive action a_1 and goal predicate G, we consider a_1 as G-irrelevant in $\mathcal{T}$ if and only if, for a_2 ranging over all actions in $\mathcal{T}$ except a_1, it follows from T that:

$$executable(do(a_1, s)) \leftrightarrow [\,(G(do(a_1, s)) \rightarrow G(s)) \wedge (Poss(a_2, do(a_1, s)) \rightarrow Poss(a_2, s))\,] \quad (20)$$

If a_1 is G-irrelevant, then a_1 will not be in any optimal successful plan to achieve G, and can be removed from the set of actions when planning to achieve G.

Example (continued): In the previous example, we showed that a_1' could be removed from $\mathcal{T}_{\mathcal{A}'}$. It then follows that, a_2' will never be needed to make a_1' $Poss$-ible. If a_2' can never directly achieve the preconditions for any other actions in the theory, then for all goal predicate $G(s)$ which are not among the effects of a_2', a_2' is G-irrelevant.

6 Web Service Composition

The primary motivation for our work was to be able to compose Web services using operator-based planning techniques. With the results of Sections 3 and 4, we have addressed a fundamental barrier to automated Web service composition. Service providers such as Amazon or United Airlines will describe their Web services (Web-accessible programs) as processes. In our vision of the Semantic Web, this will be done using the DAML+OIL Web service ontology, DAML-S [1], whose process description constructs are similar to Golog. (The relationship between DAML-S and the situation calculus is well-defined and has been used to define the semantics of DAML-S.) To produce black-box or compiled representations of Web services for automated composition, we can exploit the compilation techniques described in this paper. Using them, we compile process-oriented program descriptions of services into black-box component descriptions. Once Web services process descriptions have been compiled, we can use standard operator-based planning techniques to automatically compose Web services.

7 Efficiency of Complex Action Planning

A secondary motivation for our work was to potentially improve the efficiency of planning (e.g., [11, 8]) through our operator-based approach to complex action planning. We restrict our attention to complex action planning with the deterministic actions listed in Sect. 4. Compiling a complex action δ is polynomial in the number of primitive action occurrences in its definition. Note that this step can be performed offline, and is amortized over multiple planning runs. The expansion step is itself linear in the length of the plan, and in the number of branchings in the complex action definition. Of no surprise, plan generation dominates the computational cost. In particular: i) complex action operators tend to have more complex preconditions and effects than primitive actions, and ii) the size of the search space will be changed. However, i) causes only a linear slowdown and thus, the crucial point is ii).

Although the following analysis can be adapted to almost any classical planner, for simplicity of the argument, let's consider a breadth-first search forward planner. Given n ground actions, if the shortest successful plan is of length l, the size of the primitive action domain search space is $O(n^l)$. Adding d ground complex actions yields $n' = n+d$ ground actions in the compiled domain. [8] claims that adding actions that correspond to compositions of other actions will yield a larger search space. We identify conditions under which this is false.

Suppose the use of complex actions results in successful plans of length l/k, $k \geq 1$. One way to ensure this is by requiring complex actions to correspond to non-overlapping subplans in the shortest plan. In this case, the number of states visited to find a plan of length l/k will be $O(n'^{l/k})$ and the difference between the search spaces will be $O(n^l - n'^{l/k})$. If $(n^k - n')$ has a strictly positive lower bound for any problem, the new search space will be exponentially smaller than the old, as problem complexity increases. Informally, complex action planning reduces the planning search space when the complex actions significantly shorten the smallest successful plan relative to the increase they cause in the breadth of the search space.

Finally, in addition to this potential search space reduction,

some complex actions remove conflicts between the goals. This results in less backtrackings and enables the use of very efficient hill-climbing techniques (e.g., [9]).

8 Experimental Results

The techniques of Section 4.2 were implemented using the operator-based breadth-first search forward planner, FF [9]. FF supports conditional effects and uses its "enforced hill-climbing" whenever possible. We tested our approach on the ADL BRIEFCASE domain (BCD) [8]. This domain moves objects between locations using a briefcase. Three experiments were run on multiple instances of the problem, varying numbers of locations (#l) and portables (#p)[9].

The first experiment was simply BCD alone. Note that FF struggles as we increase (#p) and (#l). The next experiments involved the addition of the complex action Move-object. Move-object MO(locInit, locObj, Obj, locFinal) takes as input the location of the briefcase locInit, an object Obj, its location locObj, and a destination locFinal. It moves the briefcase to locObj, puts the object in the briefcase, moves the briefcase to locFinal, and removes Obj. MO is not a subplan of the shortest plan, so we would not necessarily expect it to do well. Further, it does *not* reduce the search space as it does not shorten the successful plan enough to compensate for the number of ground complex actions ($(n^k - n')$ is not positive). Nevertheless, adding the complex action move-object (BCD+MO) turns on FF's hill-climbing techniques, which reduce the number of nodes considered.

Finally, we designed a complex action that does correspond to subplans of the shortest successful plan and thus reduces the search space. The complex action LOC(loc-bc, loc), takes as input the location of the briefcase loc-bc, moves the briefcase to location loc, removes all the objects in the briefcase that should be at loc, and puts all other objects at loc in the briefcase. The goal defines where an object should be. To encode this complex action in PDDL, the action must know the goal at the time it executes. Hence, we added a persisting predicate $ShouldBeAt(Obj, Loc)$ to the domain. This predicate always has the same values as the $At(Obj, Loc)$ predicate in the goal statement. This complex action reduces the search space ($k \simeq \#p/\#l, d \simeq \#l$) and allows the use of hill-climbing techniques. Of no surprise, (BCD+LOC) presented the best results of all three experiment runs.

	#l:5, #p:20	#l:6, #p:30	#l:7, #p:42
BCD	5549 (1.39)	201006 (2261)	? ($>$ 40h)
BCD+MO	859 (11.83)	2345 (201.47)	5195 (2211)
BCD+LOC	75 (.08)	139 (.27)	260 (.85)

Number of nodes (and time of run in seconds).

[8] http://rakaposhi.eas.asu.edu/domain-syntax.html

[9] Experiments run on Sun Sparc v9, 2×750GHz, 4GB of mem.

9 Discussion and Summary

The work in this paper was motivated by the problem of automating Web service composition. In particular, we posed the problem of composing Web services such as UAL's buyAirTicket($\vec{x}$) or CNN's getWeather($\vec{y}$) in order to achieve a user-defined goal. These Web services are describable as simple programs, using typical programming language constructs. We conceived this task as the problem of planning with complex actions, with the restriction that the complex actions *had* to be the primitive building blocks of a plan. Consequently, we posed the problem of how to represent and plan with complex actions, using operator-based planning techniques. To this end, we embarked upon a theoretical analysis of the problem of how to represent complex actions as operators. The situation calculus provided the formal foundation for our work, enabling us to provide a formal definition of the preconditions, successor state axioms, and effects of complex actions under a frame assumption.

With this representational problem addressed we turned to the practical matter of how to plan. We proposed a method of planning that produced sound and complete plans relative to a corresponding primitive action domain. We showed how to use our results to plan via deductive plan synthesis as well as using an arbitrary operator-based planning system that accepts ADL as input.

We are currently incorporating these representation and compilation results into DAML-S [1], an AI-inspired markup language ontology for Web services. We're also incorporating the results into ongoing Web service composition work [13].

Finally, the second motivation for this work was to potentially improve either the efficiency of planning or the quality of the plans generated, by exploiting complex actions that capture some preferred subplans. We have shown how, in some domains, using relevant complex actions will result in a dramatic speedup of the planning process. We discussed the impact of our approach on the planning search space and illustrated predicted speedup with experiments.

Acknowledgements

We would like to thank Srini Narayanan for conversations related to this work. We gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DARPA Agent Markup Language (DAML) Program #F30602-00-2-0579-P00001.

References

[1] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Semantic markup for web services. In *Proc. International Semantic Web Working Symposium (SWWS)*, 2001.

[2] M. Baioletti, S. Marcugini, and A. Milani. Encoding planning constraints into partial order planning domains. In *Proc. 6th Conference on Knowledge Representation and Reasoning*, pages 608 – 616, 1998.

[3] G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[4] P. Doherty and J. Kvarnstrom. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[5] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, 1994.

[6] R.E. Fikes, P.E Hart, and N.J Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[7] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.

[8] Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In *Artificial Intelligence Planning Systems*, pages 150–158, 2000.

[9] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[10] S. Hölldobler and H.-P. Störr. BDD-based reasoning in the fluent calculus – first results. In *8th. Intl. Workshop on Non-Monotonic Reasoning (NMR'2000)*, 2000.

[11] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[12] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

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

[14] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR'89*, pages 324–332, 1989.

[15] R. Reiter. The frame problem in the situation calculus: A soundness and completeness result, with an application to database updates. In *Proceedings First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[16] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[17] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[18] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge – an answer set programming approach. In *6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Programming Approach Proceedings, pages 226–239, 2001.

[19] R. J. Waldinger. Achieving several goals simultaneously. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8: Machine Representations of Knowledge*, pages 94–136. Ellis Horwood, Chichester, 1977.