AIPS 2002

Sixth International conference
on
Artificial Intelligence Planning and Scheduling (AIPS2002)

Workshop on Knowledge Engineering Tools and Techniques for AI Planning

Toulouse, France

24 April 2002

http://scom.hud.ac.uk/planet/aips02kett/programme.html

Workshop Chair

Lee McCluskey, University of Huddersfield

Working Notes for distribution to workshop attendees only.

Event Sponsors:

CONTENTS PAGE

Knowledge Engineering: Issues for the Planning Community

T. L. McCluskey
School of Computing and Mathematics,
University of Huddersfield, UK
email: lee@zeus.hud.ac.uk

Abstract

Knowledge engineering for AI planning is the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimization of appropriate planning machinery to work on them. Evidence from the growing body of experience in applying planning technology suggests that knowledge engineering issues are crucial to an application's success. The Knowledge Engineering Technical Co-ordination Unit of PLANET[1] has been active for several years now in carrying out workshops and sponsoring cross-site visits on the subject. Here I briefly summarise some of the material in our roadmap document(McCluskey *et al.* 2000), selecting some of the important research questions from it, and introduce the papers that are to be presented in this workshop.

Introduction

Knowledge Engineering (KE) for AI Planning is the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimization of appropriate planning machinery to work on them. Hence, knowledge engineering processes support the planning process - they comprise all of the off-line, knowledge-based aspects of planning that are to do with the application being built.

KE issues have to be engaged by our community if we are to get non-AI Planning people to use our technology. These issues are recognised as a major problem in the application of planning systems. Experience with planners adapted for aerospace and military applications (Wilkins 1999; Tate, Drabble, & Dalton 1996; Muscettola *et al.* 1998) has pointed to KE aspects as being those most in need of attention.

In the field of Knowledge-based Systems, from whence the term 'Knowledge Engineering' originates, the need for modelling knowledge at the conceptual level has long been accepted in the development of KBS methodologies. Specifying components and their interfaces at the knowledge rather than implementational level leads to the kind of abstractions that facilitates interoperability and re-use. The *knowledge-level principle* of Alan Newell (Newell 1982) influenced and directed much KBS work into this direction. Hence the pursuit of KE within planning may be seen as a special case of KE within the general knowledge-based system field. It may prove useful to derive methods and adapt tools from KBS, as the work of Tate et al (Tate, Polyak, & Jarvis 1998) has attempted. There are peculiarities of planning that clearly distinguish engineering planning knowledge from general expert knowledge:

- the ultimate use of the planning domain model is to be part of a system involved in the 'synthetic' task of plan construction. This makes it very specific in the world of KBS, where many successful systems are, in contrast, aimed at solving diagnostic or classification problems.
- the knowledge elicited in planning is largely knowledge about actions and how objects are effected by actions. This knowledge has to be adequate in content (and ultimately in form) to allow efficient automated reasoning and plan construction.

In branches of requirements capture in software engineering knowledge is elicited about processes or actions in the domain of interest, similar to that in Planning. Very expressive and formal languages and development environments have been introduced for this purpose. In software engineering however, the purpose of this capture in very different - it is to help in the analysis and understanding of a system, and to be used in the creation and validation of a new system model.

Despite the peculiarities mentioned above, it seems fruitful to pursue research and developments in KE for planning in the context of related developments in KBS and software requirements engineering. In particular, it is appears inevitable that research in AI Planning must adopt multi-disciplinary approaches to knowledge-based planning.

The Growth of Tool Support

Not too many years ago tools for planning domain acquisition and validation amounted to little more than syntax checkers. 'Debugging' a planning application would naturally be linked to bug finding through dynamic testing. The two pioneering KBS planners O-

[1]The EU-funded Network of Excellence in Planning

Plan and SIPE have of course, by necessity, developed methods and tool support. The O-Plan system has for example its 'Common Process Editor' (Tate, Polyak, & Jarvis 1998) and SIPE has its Act Editor (Myers & Wilkins 1997). These visualisation environments arose because of the obvious need in knowledge intensive applications of planning to assist the engineering process. They are quite specific, however, having been designed to help overcome problems encountered with domain construction in previous applications of these planning systems.

One of the earliest types of tool support for AI Planning grew from research into Machine Learning (ML). From an ML point of view, siting a learning mechanism with a clean, classical planner led to an attractive way to evaluate the automated learning algorithms. Hence ML tools for planning have received considerable attention over the last 20 years. For example, tools have been built to induce operator descriptions from traces of actions, and to acquire or tune heuristics in the form of macro-operators, state evaluation functions and goal orders.

More recently interest has grown in the area of *domain analysers*. These tools process a domain model with the goal of making explicit useful information, and they may be embedded in an online planner or be standalone. In the latter case, they can function as part of a modelling environment, helping a user to perform *static validation* on an acquired model. For example, tools may: check that a planning operator is consistent (e.g. it never inputs a valid state and outputs an invalid state); reason with operators and output state invariants to be visually checked by a user; or output necessary goal orderings, to check for impossible goal combinations. Also, domain analysis tools can help in the acquisition of heuristics that customise a general planning engine to an application, or more importantly to identify the kind of planner appropriate for solve problems within the application domain.

A further step is to produce *tools environments* for acquiring, modelling and prototyping planning applications in such a way that the tools are integrated and the environment is *open*. An open environment means that users can attach their own tools which integrate with the other tools. Research into and the development of such an environment for classical AI planning was one of the goals of the PLANFORM project (Planform 1999). GIPO, an outcome of this project, is a GUI designed to integrate tools in support of knowledge engineering for AI planning. GIPO is purely a 'laboratory' system aimed as a testbed for knowledge acquisition techniques and planning tool integration. Users can attach their planners to GIPO via a PDDL (AIPS-98 Planning Competition Committee 1998) interface, but other more knowledge-rich interfaces are yet to be developed. Truly open environments are essential for the development of planning technology, but this requires a level of standardisation not yet present in the community.

Representation Languages and Standardisation

Both to help the Planning field mature, and to help engineers apply and integrate the technology, representation language conventions should be sought. This has been achieved in a very limited way with PDDL, a community accepted standard for communicating minimal dynamical models of a domain. PDDL has been a lowest common denominator for planning systems running under many of the classical STRIPS-assumptions, allowing sets of domain models to be distributed and certain classes of planning engines to be compared in competitions. There is a need to exchange and to some degree standardise much more than bare domain dynamics - for example, domain structures and heuristics. Specifically, it would be useful to share and exchange generic object structures that could be re-used over a range of applications. More generally, creating a planning knowledge base *without* re-use seems at best inefficient. There is some work emerging on planning ontologies (for example see (Gill & Blythe 2000)) but there is still a long way to go.

When considering standard representations one must consider the function and content of the representation itself. For example, in contrast to domain specification languages, there have been attempts at creating standard languages for *plan specifications* - notably the work surrounding the creation of SPAR (Tate 1998). Another class of representation language, which concerns knowledge engineers in particular, is one in which languages are specifically designed with pragmatic features that help the process of domain acquisition and modelling. Our roadmap (McCluskey *et al.* 2000) postulates criteria for such languages, asserting that they should be well structured, tool supported, expressive, customizable, well founded, and finally, embedded within a modelling method. Languages that have been developed from the point of view of knowledge acquisition, and fulfill some of these criteria, include DDL.1 (Cesta & Oddi 1996), Act (Myers & Wilkins 1997), TF (Tate, Polyak, & Jarvis 1998) and OCL_h (McCluskey & Kitchin 1998).

A Note about Terminology

Research papers that concern KE in planning often appears to use inconsistent, confusing or imprecise terminology. I will choose a simple but pervasive example to illustrate the point. In this volume of papers phrases such as 'domain description', 'domain specification', 'domain definition', 'domain model', 'domain theory' and simply 'domain' are used. Firstly, a distinction: let the *domain* denote the reality being modelled within the corresponding planning system. Let any form of symbols representing parts of the domain be called a *domain description* – for example a document containing natural language describing the domain. Domain description is the most vague term of the set.

The term *domain specification* is less vague than a

description. It implies something that is finished, and something that can be reasoned with. In other words, we expect a domain specification to be complete and precise, and often formal in the sense that valid inferences can be made using it, about the domain itself. Of course most specifications fail in these aspects!

The term *domain model* implies that we have a representation that can be used to perform operations in the same manner that occur in the domain; and that there is a well-known operational semantics for constructs in the model. Further, the term 'model' implies that named objects within it correspond directly to named objects in the domain, and there is an obligation on the developer to check that the model accurately predicts changes in the domain. I would argue that the traditional operator-based domain descriptions fall exactly into this last case - they are domain models. In software engineering there is a divide between implicit or property-based formal specifications on the one hand, and executable formal specifications on the other. While the former might state *properties* of the domain, it may or may not contain operational details. Hence, a domain *model* is rather like the idea of an executable specification in software engineering.

In summary, we call the outside reality the Domain; the Domain Description is any set of documents about the Domain, possibly in natural language; the Domain Specification is an abstract, formalised account of the Domain; and the Domain Model is a Domain Specification which is in an operational form, containing explicit details of domain dynamics, and suitable for processing by a planning engine.

Issues for the Planning Community

Planet's KE Roadmap (McCluskey *et al.* 2000) lists a number of concerns and research challenges for the future in the area. Here I list several but expect that many more will arise as a result of the workshop.

evaluation of knowledge engineering methods: How do we evaluate knowledge engineering methods, tools and techniques? Case studies and controlled experiments are very expensive compared to evaluation of a planner against a set of benchmarks. Is the introduction of *challenges* or *competitions* feasible or desirable to promote this area?

improved representation languages: Pragmatic aspects of programming languages (objects, types, modules) are very well developed to help one to program. On the other hand, it can be argued that PDDL is at the level of a 'machine code' for domain description. What kind of standard, pragmatic structures are needed in domain modelling languages?

further standardisation: There are many reasons why standardisation can help advance a field - one in particular is to help us develop components of a planning system flexibly. Should we be developing languages for standardising the exchange of heuristics, and other planning - related knowledge? Given the potential for applying planning technology through the internet, should we not be developing web-friendly 'semantic' mark-up languages for this purpose?

ontologies: Given the emphasis on Ontologies in Knowledge-based Systems, should we be developing Planning Ontologies, and if so, in what form? The availability of libraries of components from which to assemble planning knowledge bases and planning systems seems very desirable (Gill & Blythe 2000), but how do we go about funding and evaluating this work? Hertzberg in the last section of reference (Hertzberg 1996) declares that there is a lack of a 'vocabulary for describing the characteristics of domains, plans ...'. In the context of Knowledge Engineering, the pursuit of such a classification system and/or vocabulary is still on the to-do list, and well worthy of action.

The Workshop Papers

Aylett and Doniat tackle the very difficult area of knowledge acquisition for planning, using an approach inspired by the KBS community. Their focus is on helping a domain expert rather than a planning expert perform such a task. The aim of Simpson et al's work is also knowledge acquisition, but at a more detailed specific level where libraries of generic types could be used to aid the acquisition of new domains. Murray's paper too concerns generic types, but rather than for domain structure, he attempts to use them as abstract control rules that could form a generic control rule library.

The papers by Cresswell et al and Varrentrapp et al both concern support tools. The first deals with a much needed extension to an existing domain analyser, while the second postulates an open environment specifically for evaluating planners using dynamic testing.

The work of Fernandez et al falls into the category of using learning techniques to tune planning heuristics. Of note is their use of a Neural Network as the learning technique, resulting in interesting coding issues centering on the representation of states and goals as inputs to such a network.

Bartak's paper is ambitious in that it proposes the creation of a modelling framework which spans both planning and scheduling, and which regards resources and activities with durations as fundamental. Influenced by scheduling applications, Bartak's work provides a good counterpoint to emerging modelling platforms aimed at AI planning.

Finally, Jarvis's paper calls for a change in AI Planning's research direction away from the easily evaluated 'stand-alone' knowledge sparse planner (of the AIPS competition variety), to the more embedded, mixed-initiative expressive kind. He introduces the idea of 'computer aided planning' rather than 'computer replaced planning' and argues most convincingly that this is both a more feasible and useful direction for main-

stream planning research. Perhaps the AI Planning Community will split up along these lines, with a gap emerging between AI scientists, interested in planning capabilities per se, and AI Planning engineers, interested in exploiting the technology. These issues will no doubt be discussed at the Panel session!

References

AIPS-98 Planning Competition Committee. 1998. PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Cesta, A., and Oddi, A. 1996. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press. 341–352.

Gill, Y., and Blythe, J. 2000. PLANET: a shareable and reusable ontology for representing plans. In *Proceedings 17th International Conference on AI*.

Hertzberg, J. 1996. On Building a Planning Tool Box. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press. 3–18.

McCluskey, T. L., and Kitchin, D. E. 1998. A Tool-Supported Approach to Engineering HTN Planning Models. In *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*.

McCluskey, T. L.; Aler, R.; Borrajo, D.; Haslum, P.; Jarvis, P.; and Scholz, U. 2000. Knowledge Engineering for Planning ROADMAP. *http://scom.hud.ac.uk/planet/*.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–48.

Myers, K., and Wilkins, D. 1997. The Act-Editor User's Guide: A Manual for Version2.2. SRI International, Artificial Intelligence Center.

Newell, A. 1982. The Knowledge Level. *Artificial Intelligence* 18:87 – 127.

Planform. 1999. An open environment for building planners. http://scom.hud.ac.uk/planform.

Tate, A.; Drabble, B.; and Dalton, J. 1996. O-Plan: a Knowledged-Based Planner and its Application to Logistics. AIAI, University of Edinburgh.

Tate, A.; Polyak, S. T.; and Jarvis, P. 1998. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh.

Tate, A. 1998. Roots of spar - shared planning and activity representation. *Knowledge Engieering Review* 13:121 – 128.

Wilkins, D. 1999. Using the SIPE-2 Planning System: A Manual for SIPE-2, Version5.0. SRI International, Artificial Intelligence Center.

Supporting the Domain expert in planning domain construction

Ruth Aylett, Christophe Doniat

University of Salford
Centre for Virtual Environments
Business House
Salford M5 4WT
Fax: +(00 44) (0) 161 295 2925
r.s.aylett @salford.ac.uk; m.soleil@wanadoo.fr

Abstract

This paper discusses work aimed at allowing domain experts to generate a domain model for an AI planning system as part of a larger project to build an integrated set of tools for supporting AI planning. It outlines the overall methodology and discusses the tool in which this is embodied. A Domain model is generated in which can be represented by cluster of constraints shaping an Ontology of each studied case. Progress has been made towards automatic conversion into the modelling language OCL and integration with the OCL tool GIPO. We illustrate the methodology by applying it in two examples of planning

Introduction and motivation

The effort required to construct a domain model for an AI planning system has long been recognised as a major barrier to the take-up of this technology outside the AI planning community. The PLANFORM project, which is supported by the UK Engineering and Physical Sciences Research Council, involves researchers collaborating between the Universities of Huddersfield, Salford and Durham [Planform 99] who are tackling this problem. Its aim is to research, develop and evaluate a method and supporting high level research platform for the systematic construction of planner domain models and abstract specifications of planning algorithms, and their automated synthesis into sound, efficient programs that generate and execute plans. Figure 1 shows the high-level architecture of the PLANFORM system.

Within Planform, the domain model is represented in the language OCL [McCluskey & Porteous 97, Liu & McCluskey 99] which supports validation and checking tools as well as translation to other formalisms such as PDDL [McDermott et al 98]. The toolset GIPO [Simpson et al 01] has been produced to help in the iterative construction and validation of this model. However GIPO still currently requires too much specialist knowledge of OCCL and of AI planning in general to be a suitable interface for a domain expert – one who understands the domain in which planning is to take place but lacks any specific expertise in AI planning. The KA-Tool discussed here is aimed at such domain experts.

The problem of supporting knowledge acquisition directly from the domain expert, without the intervention of a knowledge engineer, has been discussed in the field of Knowledge-Based Systems (KBS) for many years [Musen 98,Valente 93]. A consensus has been reached that this may be feasible where a skeletal domain model can be provided to guide the knowledge acquisition process and both the skeleton model and the process itself can be defined through a methodology embodied in the knowledge acquisition tool [Musen 98]. The key components of the skeleton model are seen as domain ontologies combined with domain-independent problem-solving methods which have often been thought of as *generic tasks*. The best-known – but far from the only –

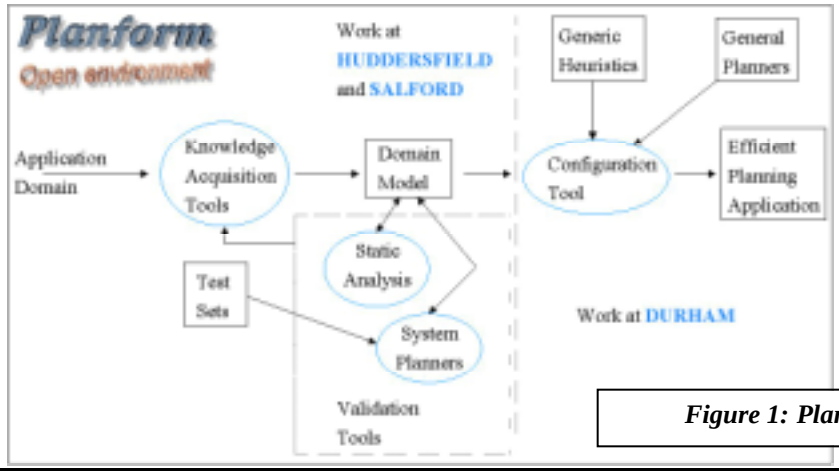


Figure 1: Planform architecture

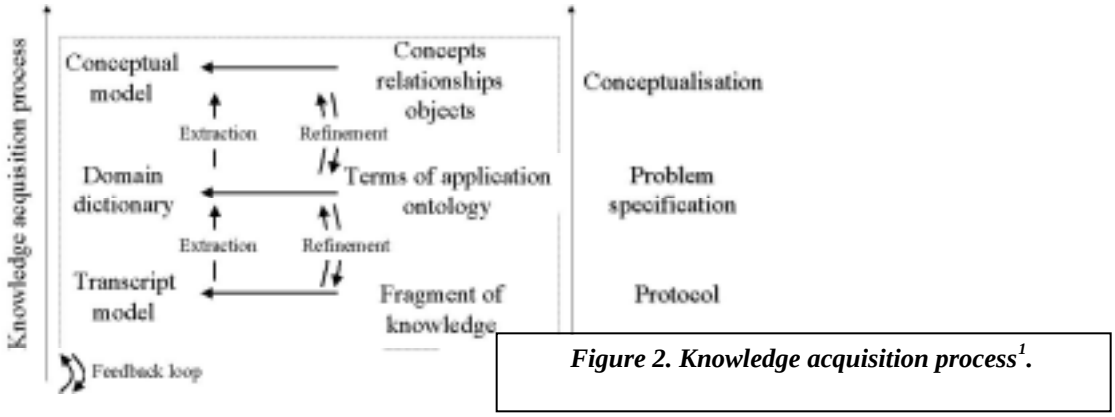


Figure 2. Knowledge acquisition process[1].

example of a methodology is Common KADS [Breuker & Wielinger 89, Shrieber et al 94], which provides libraries of configurable problem-solving components together with stereotypical configurations which can be thought of as corresponding to types of abstract problem-solving tasks such as *diagnosis by heuristic classification* or *interpretation*.

It is noticeable that AI Planning has rarely been considered as part of this research (see Valente 93 for a rare exception). While in theory planning could be considered as one or more generic tasks, in practice the Knowledge Engineering community has concentrated on other generic tasks – diagnosis in particular [Benjamins 93] – while AI Planning researchers have hardly been involved at all, tending to concentrate on the development of planning algorithms.

The approach discussed here draws on this work in the KBS community, and sees the combination of ontologies, logics and generic problem-solving methods as a way of addressing knowledge acquisition for planning [Musen 98]. It supports the capture and structuring of relevant knowledge about a domain and its intelligent behaviours [Hayes-Roth & Hayes-Roth 90] because they play an important role in the choice of an appropriate problem-solving method, possibly configured from complex components stored in a library [Valente 93].

Knowledge Acquisition Process

Since the tool being constructed automates a knowledge acquisition (KA) process, first it is necessary to model the process itself. The KA process is shown in Figure 2, embodying two different extraction/refinement processes.

The first of these (bottom - right) moves from protocol to problem specification. By protocol we mean raw domain knowledge - transcripts, documents, interviews, observations[1]. A protocol is created by a problem-solving episode, where the expert is provided with an AI Planning problem, of a kind that they normally deal with, and are asked to solve it. As they do so, they are required to describe each step, and their reasons for doing what they do. The transcript of their verbal and/or text account is, in this case, called a protocol. By problem specification we mean a definition or description of an application domain represented as a set of choices at a particular level of abstraction in an ontological hierarchy. Thus 'Entertaining a foreign visitor' and 'Drumstore', the domains used for the experiments reported later, are problem specifications.

The second extraction/refinement process moves from problem specification (middle - right) to conceptualisation (top – right). By conceptualisation, we mean a stable and restricted formal representation (concepts, relationships and objects) with defined structure and behaviour[2]. Clearly movement between these levels is iterative rather than linear.

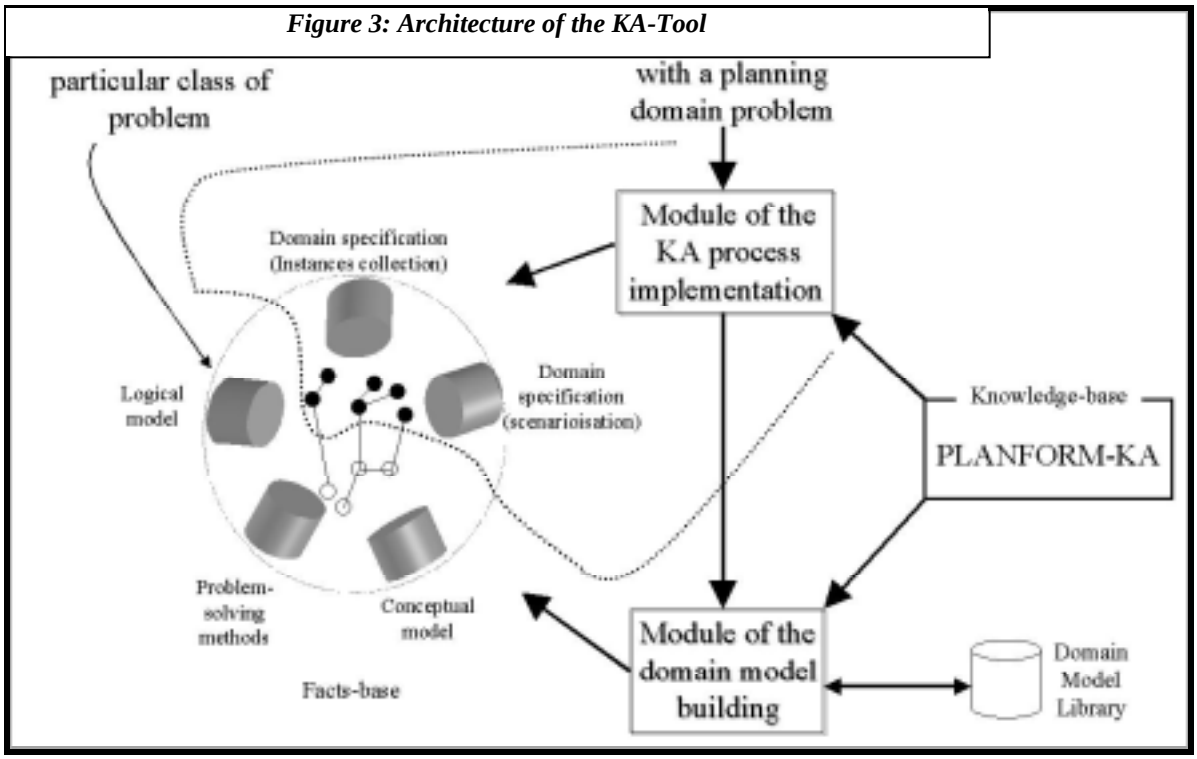
The conceptual model (top - left) is represented using a hierarchical frame system because this allows easy representation of inheritance between *sorts* (the relationship *kind-of*) and/or aggregation between *sorts* (the relationship *part-of*) for instance. Translation into a sorted first-order logic such as that used by OCL is straightforward. Frames have an advantage over a first-order logic in that both structure and behaviour can be embodied in one generic entity.

An ontology is defined [Gruber 93] as a rigorous specification of a set of specialised vocabulary terms sufficient to describe and reason about the range of situations of interest in a particular domain - a conceptual representation of the domain entities, events, and relationships. Two primary relationships of interest are abstraction (kind-of) and composition (part-of). Thus an ontology provides a grounding of the key concepts within a domain. In principle we need both an ontology of planning problem domains and of planning software to carry out knowledge acquisition since the premise is that the conceptual framework of the problem domain is not the same as that of the planning software – otherwise there would be no problem for the domain expert.

[1] We will used the term 'transcript' in the next paragraphs to mean a combination of transcripts, documents, interviews, observations as a whole.

[2] Note here that this is a basic definition of behaviour only. Complex behaviours are not covered in this present work.

Figure 3: Architecture of the KA-Tool



A Domain dictionary (middle of figure) is a partial ontology - using the experimental approach, it is hard to make an exhaustive analysis of all domain objects. Nevertheless, the problem specification can be used to define relevant objects and relationships, using macroscopic properties that support appropriate choices. Broadly, the Domain Dictionary is associated with (i) a particular domain, (ii) specification of a problem or problems that we want to solve, (iii) the reasoning that belongs to the studied domain and allows the specified problem to be solved.

Overview of PLANFORM-KA Tool architecture

Figure 3 shows the main architecture of the PLANFORM-KA tool – an intelligent system that contains the KA process. The user applies the *module of domain model building* to a particular problem specification. The building of a new conceptual model might be carried out with or without an existing problem specification from the *Domain model library*. The result is recorded in this library. On the right-hand side, the overall knowledge base consists of the conceptual model of the knowledge acquisition process itself, called PLANFORM-KA and the *KA-Expertise* belonging to the particular conceptual model being constructed.

Case studies and methodology

In this section, we present two case studies created with our methodology, using the problem specifications: (i) 'EVentus: Entertaining a foreign visitor to your lab at the weekend' and (ii) 'Drumstore: a logistics problem in a nuclear waste factory'. We conducted these experiments, respectively with ten people and six people who verbalised their knowledge about how they would solve this problem during interviews. We chose EVentus because (i) people knew about it (drew on general rather than specialised knowledge) and it was not difficult to capture it, (ii) it was an example of a planning domain. Drumstore was chosen because it had already been implemented as an AI planning domain within the group. The interviews contained the unstructured knowledge (discourse) and sometimes some notes such as graphics, plans and other material describing knowledge and activity (explicitly/implicitly) both about the case studies and the KA process itself.

It is important to understand the level of abstraction at which such a sample problem must work. The PLANFORM toolkit as a whole will be used to create a domain model within which a number of specific tasks can be planned. Thus the experiment does not start with a specific task, but with the generic problem specification. Subjects were asked to explore the generic domain model that would be needed to plan within the domain of the problem specification and to support the solving of a number of specific tasks. Note that a more abstract version of this problem would be to replace 'your lab' with 'a lab' where this might be anywhere in the world potentially. An instance of a specific task would be something like 'Professor Stein from GMD Germany is to be entertained on Saturday May 9th'.

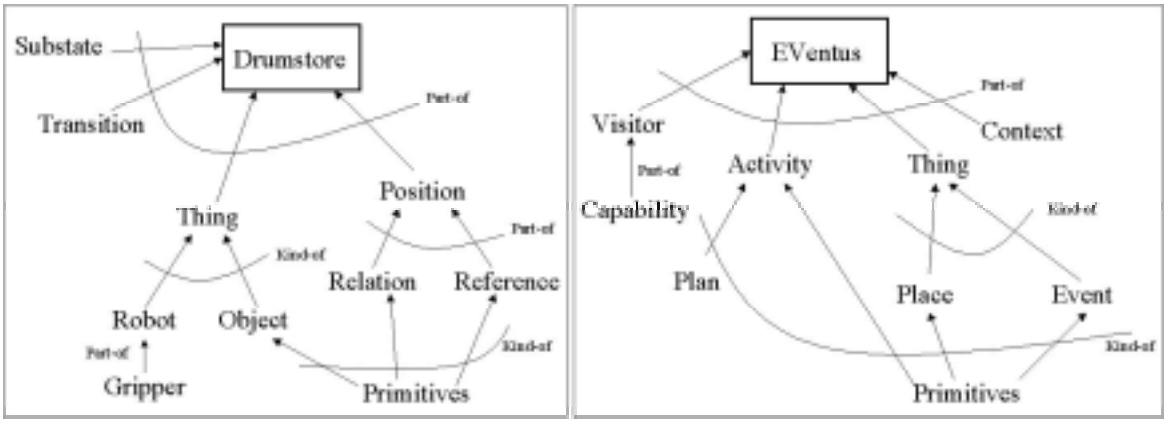


Figure 4. Frame systems of Drumstore and EVentus

Building of a domain dictionary

The first extraction phase gives us a domain dictionary (Table 1) that puts together a set of terms according to the problem specification.

Drumstore	EVentus
Robot	Thing
Thing	Activity
Gripper	Context
Object	Visitor
Relation	Capability
Reference	

Table 1. Domain dictionary

Next, we built a set of scenarios with the shared knowledge of these domain experts to find out how each expert defines reasoning strategies to solve the problem specification. We used a part of the KOD (Knowledge Oriented Design) [Vogel 88] method to obtain an accurate process for knowledge acquisition and to build the conceptual model through the set of examples and scenarios (see section 2.2). Table 2 and 3 show the number of instances of each term in each scenario. We will call these outcomes *instance coverage.*

Drumstore	Terms[1]					
	R	T	G	O	Rel	Ref
1	5	1	3	7	2	3
2	10	2	2	5	2	3
3	20	5	5	12	1	3
4	10	3	3	5	1	3
5	5	1	4	7	2	2
6	6	1	3	13	2	2
7	8	1	5	7	2	3
8	7	1	4	11	2	2

Table 2.Instance coverage of Drumstore.

[1] Each Drumstore scenario is designed through the six terms as follows: Robot (R), Thing (T), Gripper (G), Object (O), Relation (Rel) and Reference (Ref).

EVentus	Terms[2]				
	T	A	C	V	Ca
1	9	4	1	1	3
2	5	6	1	1	2
3	8	7	2	2	2
4	5	5	3	1	4
5	13	7	1	2	6

Table 3.Instance coverage of EVentus.

This shows that knowledge about this particular specification varies between domain experts giving different number of examples of each term. This coverage gives us an idea of experts' practice so as to build the interface of the future intelligent system.

Building of conceptual/epistemological model

The second extraction gives us first a conceptual model, i.e.semantic relationships, objects and actions. Then the model is completed with a epistemological model, i.e., the definition of concepts, its hierarchy and structuring relationships (behaviours). A domain model is thus defined by these representations in our methodology by using a frame system as in Figure 4.

`Drumstore` relies on the nine following generic concepts: `Thing` is a root of the domain model and describes two mobile things: `Robot` and `Object`. `Robot` depicts a real robot, which can navigate and has equipment – `Gripper` – to bring and carry some `Object` according to a `Relation/Reference` address pair (e.g. `(Object,at,beacon1)`). `Primitives` depict a set of generic concepts like `Drum` (`Object`), `At`, `Near` (`Relation`) and `Beacon` (`Reference`). `Substate` and `Transition` depict respectively the conditions in which Robot does some tasks and the state of each task when it has taken place.

. `EVentus` contains the nine following generic concepts: `Visitor` is a locus of the domain model and describes a

[2] Each EVentus scenario is designed through the four terms as follows: Thing (T), Activities (A), Context (C), Visitor (V) and Capability (Ca).

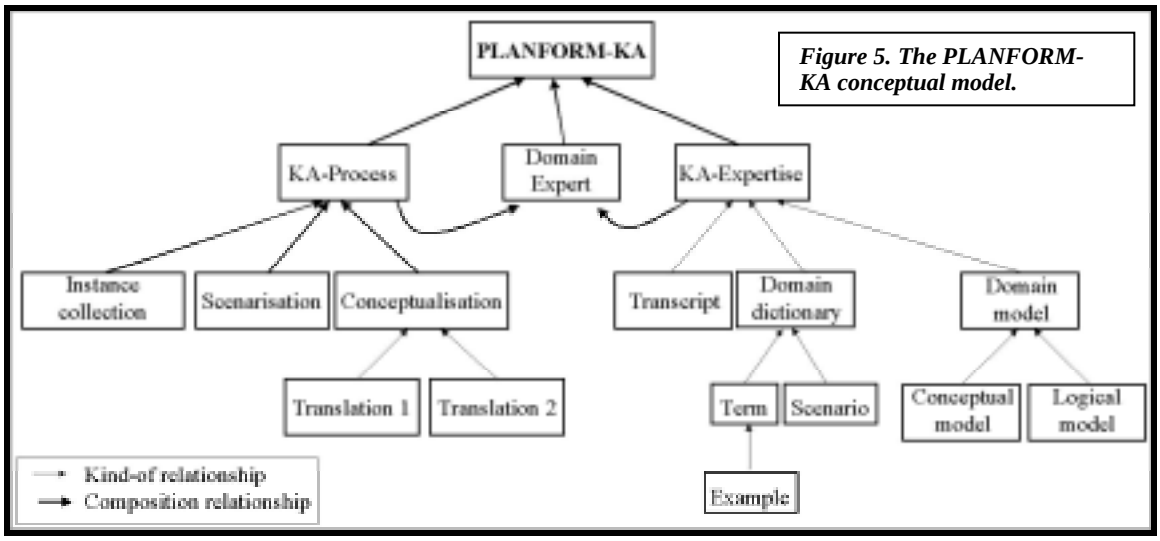


Figure 5. The PLANFORM-KA conceptual model.

real visitor according to her/his real capacities, which are depicted by `Capacity`. `Activity` and `Context` describe behaviours of a visitor, `Plan` describes a set of alternative plans used by a visitor. `Thing` describes `Places` and `Events` used during the activity. Finally, `Primitives` depicts a set of generic concepts like a restaurant, a town (place) or an exhibition (event).

Summary

A KA process has been carried out to capture knowledge and build two domain models for particular problem specifications through two case studies: Drumstore and EVentus. The generic concept `Thing` is defined in both domain models with different semantics. In `Drumstore`, this concept represents an abstraction of *mobiles* but in `EVentus`, it represents an abstraction of *locations*.

Categories	***Drumstore***	***EVentus***
Agent	*Thing*	*Visitor*
Object	*Position*	*Thing*
Task	*Substate* *Transition*	*Context* *Activity* *Plan*

Table 4. Abstraction similarity undependable the level between Drumstore and EVentus.

Table 4 shows the similarity between `Drumstore` and `EVentus` concepts using three main categories: `Agent`, `Object` and `Task` as a skeleton ontology for planning domains [9]. Note that the `Task` category is divided into two semantic sub-categories: (i) the `Drumstore` task is state-based and the `EVentus` task is action-based. This represents a first step towards an epistemological model.

An Intelligent system: PLANFORM-KA

In this section we discuss the Planform-KA tool in more detail – see Figure 5 for its conceptual model. As outlined above, the process component of the tool can be decomposed into a set of refinement processes – called phases – carried out by the domain expert according to an expertise. We envisage supporting it with a generic ontology like the Upper Cyc Ontology [Upper Cyc] (though in this work we have constructed a small ontology ourselves) to start instance collection.

This Ontology provides a sufficient common grounding for applications. Some concepts such as `Actor` or `Plan` are already supplied as generic definitions, which should help the domain expert. It also includes definitions of `Object` and `Agent` categories (as in the Summary above) and possibly a fragmentary definition of the `Task` category. That is the case for `Drumstore` for instance where there are `State` and `Transition` generic concepts as parts of OCL.

Conceptual model of PLANFORM-KA

Its conceptual model (Figure 5 above) relies on several interrelated generic concepts. The `domain expert` generic concept depicts the subject acquiring the knowledge model, the `KA-expertise` generic concept features the knowledge required to build the knowledge, the `KA-Process` generic concept describes the behaviour carried out by the domain expert. The KOD method was again used to elaborate a frame system.

Frame representations for Domain expert, KA-Expertise and KA-Process

For reasons of space we illustrate only a subset of the frame representations for these concepts.

Domain expert

We consider the domain expert (DE) as the cognitive agent carrying out the process of knowledge acquisition. DE has a mental model of the real world expressed in concepts. The `domain expert` generic concept represents the properties of this agent in relation to the carrying out of the KA-Process and is central to the overall conceptual model since there are composition relationships with concepts KA-Process and KA-expertise

KA-Expertise

The `KA-Expertise` generic concept represents the memory of our domain expert. This holds three knowledge categories: transcripts from a case study, and the related domain dictionary and domain model.

The `Transcript` generic concept represents the properties of documents such as free-text or graphics collected in a case study. The `Domain dictionary` generic concept represents the properties of a domain specification expressed as a set of choices – terms – themselves organised into a set of scenarios (Figure 6).

DOMAIN DICTIONARY Frame and its slots	**Arity**
`Kind-of value KA-EXPERTISE`	(1) (1,1)
`Name domain STRING` `If-add <TERM,createinstance(),($term)>`	(1) (1,1)
`Term domain TERM` `If-add <EXAMPLE,create-instance(),($example)>`	(1) (1,n)
`Scenario domain SCENARIO`[1]	(1) (1,n)

Figure 6. DOMAIN DICTIONARY Frame definition.

The `Domain model` generic concept depicts the properties of a conceptualisation as a set of conceptual/epistemological and logical representation levels (Figure 7).

DOMAIN MODEL Frame and its slots	**Arity**
`Kind-of value CONCEPT`	(1) (1,1)
`Name domain STRING` `If-add <CONCEPTUAL_MODEL,create-instance(),($Conceptual_level)>`	(1) (1,1)
`Conceptual_level value CONCEPTUAL_MODEL` `If-add <LOGICAL_MODEL,create-instance(),($Logical_level)>`	(1) (1,1)
`Logical_level domain LOGICAL_MODEL`	(1) (1,1)

Figure 7. DOMAIN MODEL Frame definition.

[1] Each scenario will spead through the relationship with the instances of examples.

KA-Process

The `KA-Process` generic concept represents the process which drives knowledge acquisition and refinement phases. The KA process starts with an instance collection phase, i.e. the explaining of each term by providing examples of it. For example, `Drum` and `Robot`, two terms of the terms in Drumstore, contain the following instances:

```
Drum D12 is radioactive
Drum D12 is at beacon B14
Robot R3 navigates from location S3
  towards beacon B14
Robot R3 docks at beacon B14
Robot R2 grabs from beacon B15 drum D12
```

This phase continues until the expert provides instances for each newly defined term. The process then continues with a creation of scenarios (scenariosation) phase, the description of several scenarios – particular problems to be solved – within the scope of the given global goal (for example: entertaining a foreign visitor; a logistic problem in a nuclear waste factory) using the previously defined instances.

Each scenario belongs to one expert or a group of experts. Finally, a scenario can be seen as a set of facts (predicates), which will be used to define some properties, constraints, plan and goal states samples at the conceptual level. The outcome is a terminology, i.e. a set of terms and a set of scenarios. The built-in ontology is used to prompt the expert during this phase.

This bottom-up approach has also been supplemented by a top-down approach in which the ontological categories `agent`, `object` and `action`, [Aylett & Jones 96] are used to drive a question cycle in which new terms are extracted from the expert. Questions move between the categories, so that if the expert provides an `agent` term (for example `robot`), they are then prompted for actions carried out by that agent and objects involved in the action.

At the conceptual/epistemological level, first of all, the process automatically carries out a translation phase into the frame-based representation, so that each defined term becomes a frame. Next, the domain expert defines by hand, or through the agent-object-action question cycle, the properties of each frame. For example, the term Robot becomes the `Robot` frame and belongs to the `Concept`[2] superframe..

Following the same process, we defined the `Visitor` frame – from `EVentus` – as seen below. The CAPABILITY frame depicts the properties of natural abilities and skills that make the visitor able to do some activities. A visitor could have either at least seven {Status, gender, age, budget, type, quality, nationality} or several further capabilities such as {like to try new things, accompanying other people, swim, has a budget, other}.

[2] SuperFrame CONCEPT is the generic frame, which is the root/father of all frames in the frame system.

VISITOR Frame and its slots	Relationship type	Arity
Kind-of value AGENT	Kind-of (frame-frame) Inheritance (frame–frame)	(1) (1,1)
Name domain String = {Fred,Group B,other} If-add <VISITOR,create-instance(),($group)> If-add<PLAN,create-instance(),($pref-to-do)> If-add <CAPABILITY,create-instance(),($capability)>	Has-a (frame–attribute) Is-a (frame–instance) (Behaviour) (Behaviour) (Behaviour)	(1) (1,1)
Group domain VISITOR	Part-of (frame–frame)	(1) (0,n)
Pref-to-do domain PLAN	Part-of (frame–frame)	(1) (1,n)
Capability domain CAPABILITY	Part-of (frame–frame)	(1) (7,n)

The conceptualisation finishes with a second translation phase from the frame-based representation into sorted first-order logic, in which each defined frame becomes a set of propositions. Here, we decided to use the sorted first-order logic language OCL. In OCL, substate and transition substate concepts describe respectively, the conditions before the transformation of each task and the transition when an object changes from one substate to another substate.

This translation is automatic: each frame ⇇[1] sort, each instance of frame ⇇ object, each attribute ⇇ predicate and each *part-of* relationship with its related arity ⇇ a defined predicate called 'belongs_to'. For example, table 5 shows the `Robot` frame and its translation into OCL where gripper – equipment – of the robot. The arity of this slot (column Arity, bottom) is defined by `(1)`, i.e. this slot takes one frame gripper in the relationship at the same time and `(1,1)`, i.e. this slot allows the obligatory instantiating of one gripper's instance. As a result, the relationship and its arity of this slot translates into invariant predicates (bottom) the constraint that one robot has to have one gripper only.

Evaluation and results

A first demonstrator has been implemented to validate the approach of PLANFORM-KA. Figure 8 shows the main graphical user interface during the creation of the `Robot` generic concept in the `Drumstore` domain model.

We have also generated the logical model seen in Appendix 1 with OCL semantics and syntax through a first version of a translator:

[1] ⇇ means 'is translated into the type of…'

Generalising over the different phases of the KA process, we have formulated the notion of `Constraint`. Thus the `Term` generic concept – in the instance collection phase – is a kind of constraint which allows the domain expert to make a set of choices to justify the domain specification. Next, the `Scenario` generic concept – used in the scenarioisation phase – is also a kind of constraints, allowing choices in the design of task representations. Thus the task could be state-based, action-based and so forth.

In the same way, the `Relationship` generic concept – in the conceptualisation phase – is a kind of constraint (Figure 9), which structures each concept. In addition, the `Arity` and `Daemon` generic concepts – from the epistemological phase – are also kinds of constraints (Figure 10) on the problem-solving methods (PSM) and heuristics.

Finally, the `Proposition` generic concept – from the logical phase – is also a kind of constraint (Figure 13), representing the chosen logical language. The `Constraint` is then described as something that must be `true`. Thus in the KA-process we define a cluster of constraints (Figure 11) across the several representation levels.

Capturing actions

The creation of a strong methodological framework for the Planform-KA tool was seen as a priority, and this has been accomplished. What is required now is to incorporate the planning-specific conceptual framework of agent, object and task [Aylett & Jones 96] in a more direct fashion. We have not at the time of writing attempted to generate planning operators into OCL, but the question-driven agent-action-object dialogue is seen as the basis for doing so. Given that Planform-KA sits within the overall

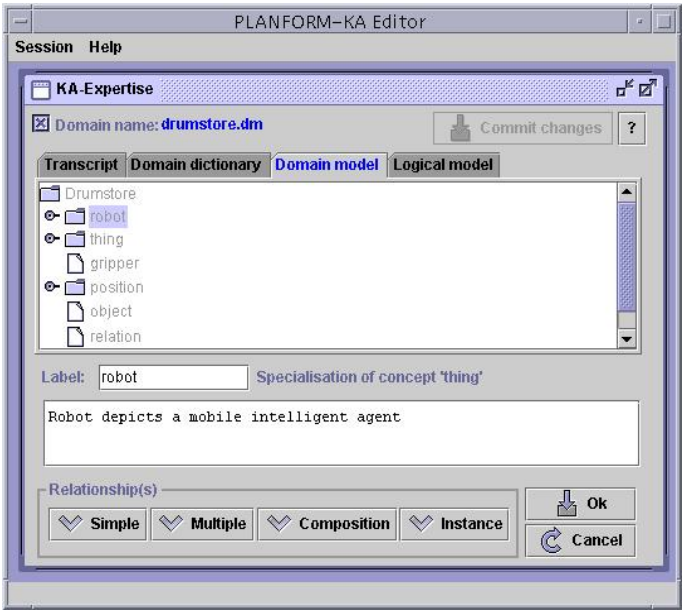


Figure 8 – Creating the term `Robot`

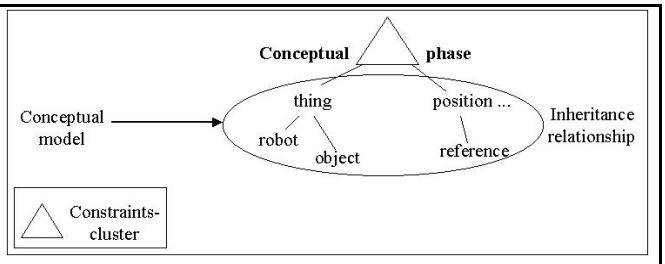


Figure 9. Constraints-cluster on conceptual model

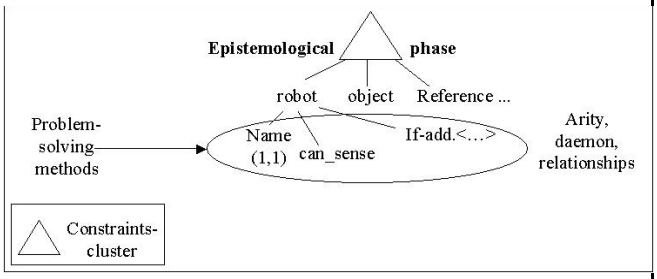


Figure 10. Constraints-cluster on epistemological model

Planform architecture, even the generation of skeletal operators would allow use of GIPO's refinement mechanisms to fill them out into a complete form. This would require an AI planning expert to supplement the role of the domain expert but would at least automate the basic knowledge acquisition process from the expert.

Related work

Many specific approaches propose a set of solutions for the acquisition, the representation and the sharing/reusing of knowledge using libraries and/or strategies, since this topic has been studied extensively in the KBS community since the 1980s. Some of them are more specialised in the first extraction of knowledge proposing a generic surrogate to capture knowledge. Protégé [Freidman-Noy et al 00] includes a suite of tools for editing ontologies, which can automatically generate customised editors that are accessible to domain experts. The Protégé library includes the problem-solving strategies (diagnosis) and also methods ontologies that describe the kinds of domain-independent knowledge used in strategies. EXPECT [Gil & Blythe 00a] used the explicit representations of problem-solving strategies (propose-and-revise strategy for the configuration design task, for example) that is used to support flexible approaches to knowledge acquisition. For instance, Protégé is an approach supported by a tool that captures new ontologies, and offers a library of problem-solving methods – For example propose-and-revise – to combine with them.

EXPECT [Gil & Blythe 00a] is a framework and knowledge based system to acquire and represent problem solving method capabilities. PLANET [Gil & Blythe 00b] is Ontology for the representation of plans in the AI Planning field and is very relevant to the more extended framework discussed here. In other approaches, the answer for a given problem is built through a combined set of different techniques (AI methodologies, for example KOD, KADS [Shrieber et al 94]) according the major aim (diagnosis for example [Mercatini et al 99, Mercatini et al 00]).

Conclusion and further work

Surprisingly, given the amount of work in the KBS community in general, knowledge acquisition has not been widely studied in AI planning. Yet applying planning systems to real-world problems requires a systematic approach to knowledge acquisition and a methodology supporting reuse rather than ad-hoc adaptations of specific planning systems by particular individuals whose expertise remains private and invisible. The work discussed here represents some steps in this direction.

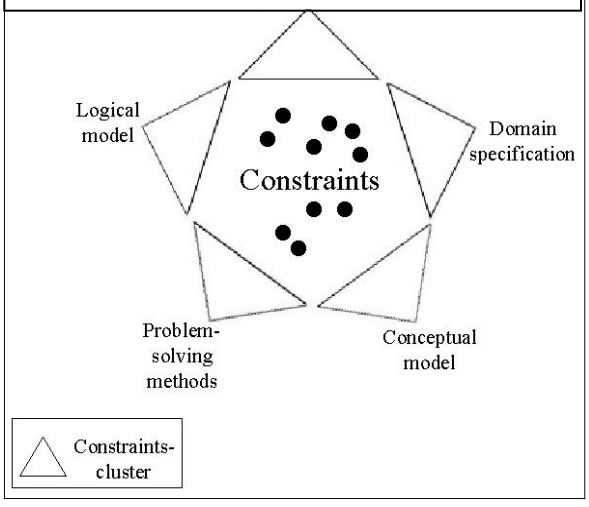
Conclusion

Our work consisted in demonstrating the value of the methodology called PLANFORM-KA in supporting a knowledge acquisition process.

First of all, we have presented the basic steps of a methodology to build a representation of AI Planning case studies according to a given problem specifications. We have described how a cluster of constraints could help domain experts during the knowledge acquisition process and how the configuration of a cluster at any representation level can formalise the knowledge of a domain expert.

Second, we have validated our KA process through the building of the case studies such as Drumstore and EVentus and shown some results as follows:

Figure 11. Clustered constraints define the KA model and process



- Instance coverage. This allows us to study the interaction with the domain expert,

Two frame system. These introduce different abstraction levels of knowledge.

- Three AI Planning categories: `Agent`, which is a mobile thing like `Robot` or `Visitor`, `Object`, for example location (`Position`, `Place`, `Event`) and `Task`, which is specialised into action-based and state-based representations.
- The Constraint generic concept. It features an abstraction of several constraints defined at different representation levels.

Finally, we are building on the question-driven interface and expect soon to generate at least outline planning operators

Further work

So far, we have built a framework for an intelligent system to solve a set of issues concerning the knowledge acquisition in AI Planning. We will make a systematic survey – at the epistemological level – of other approaches like PROTÉGÉ, EXPECT or PLANET, for instance, which focus on a similar approach with respect to reuse of ontology. A particular direction is to explore the use of generic types, [Fox & Long 00] formulated by Planform co-researchers Fox and Long, within the question-driven acquisition module. Currently, generic types are extracted from PDDL domain models, but the FSM definitions used for this might be moved towards the domain expert through incorporation into Planform-KA. Thus once an expert identifies a mobile agent for example, the system could actively prompt for the possibility of route-following. Further case-study examples will be explored in order to assess the coverage Planform-KA is able to provide for domains where a domain model has already been created by hand. Finally, supporting the expert with a much larger ontology – possibly a specialised version of the CYC Upper ontology – will also be explored. This would then enable much more widespread trials of the system

References

Aylett, R.S & S. Jones. Planner and Domain: Domain Configuration for a Task Planner. Int. Journal of Expert Systems, 9(2), 279-318, 1996.

Benjamins, V. R. (1993). Problem Solving Methods for Diagnosis. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Breuker, J. and Wielinga, B. (1989). Models of Expertise in Knowledge Acquisition. G. Guida and C. Tasso (eds). Topics in Expert Systems Design: methodologies and tools. North Holland Publishing Company, Amsterdam, The Netherlands

M. Fox, and D. Long. Automatic Synthesis and use of Generic Types in Planning. AIPS 2000 - Workshop on Analysis and Exploiting Domain Knowledge for Efficient Planning.

Fridman-Noy, N. et al. The knowledge model of Protégé-2000: combining interoperability and flexibility. 2th Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW). Juan-les-Pins (France) 2000.Y.

Gil and J. Blythe. 2000a How Can a Structured Representation of Capabilities Help in Planning? Proceedings of the AAAI – Workshop on Representational Issues for Real-world Planning Systems. 2000.

Y. Gil and J. Blythe. 2000b PLANET: A Shareable and Reusable Ontology for Representing Plan. Proceedings of the AAAI – Workshop on Representational Issues for Real-world Planning Systems. 2000.

Gruber, T.R. A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 2(5), 1993.

Hayes-Roth, B. and F. Hayes-Roth. A Cognitive Model of Planning. Representation and Reasoning. Readings in Planning. Morgan Kaufman Publishers. 1990.

D. Liu and T. L. McCluskey, The Object Centred Language Manual - OCLh - Version 1.2. Technical report, School of Computing and Mathematics, University of Huddersfield, 1999.

McDermott, D et al. PDDL --- The Planning Domain Definition Language. In Machine Intelligence 4. D. Michie, ed., Ellis Horwood, Chichester (UK). 1998.

McCluskey, T.L and Porteous J. M. Engineering and compiling planning domain models to promote validity and efficiency, Artificial Intelligence, pp.1-65. 1997.

J.M. Mercantini et al. Safety previsional analysis method of an urban industrial site. Scientific Journal of the Finnish Institute of Occupational Health, serie: People and Work, safety in modern society, pp. 105-109, 33. 2000.

N. Mercantini et al. Etude d'un systeme d'aide au diagnostic des accidents de la securite routiere. IC'99. Palaiseau (France). 1999.

M. Musen. Modern Architectures for Intelligent Systems: Reusable Ontologies and Problem-Solving Methods. In Chute (Eds), AMIA Annual Symposium, 46-52. 1998.

Planform. An Open environment for building planners. Available at http://helios.hud.ac.uk/planform. 1999.

Schreiber, G., Wielinga, B., de Hoog, R., Akkermans, H. and Van de Velde, W. (1994). CommonKADS: A Comprehensive Methodology for KBS Development. IEEE Expert, 9 (6), pp. 28-37..

Simpson, R.M; T. L. McCluskey, W. Zhao, R. S. Aylett and C. Doniat 2001 An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. Proceedings, 2001 European Conference on Planning, Toledo, Spain.

.Sowa, F. Knowledge representation: logical, philosophical and computational foundations. Brooks/Cole (eds). 2000.

Valente, A. Planning models for the CommonKADS librairy. ESPRIT Project KADS-II. 1993. Available at http://www.swi.psy.uva.nl/usr/andre/publications.html.

Vogel..C. Le genie cognitif. Masson (Eds). 1988.

The Upper Cyc Ontology, available at http://www.cyc.com/cyc-2-1/cover.html.

Appendix 1 – OCL model

```
domain_name(drumstore).

% Sorts
sorts(non_primitive_sorts,[thing,position]).
sorts(primitive_sorts,[robot,gripper,object,relat
ion,reference]).
Sorts(thing,[robot,object]).

% Objects
objects(robot,[r1,r2,r3,r4]).
objects(gripper,[g1,g2,g3,g4]).
objects(object,[d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d1
1,d12]).
objects(relation,[near,at]).
objects(reference,[s1,s2,s3,s4,b1,b2,b3,b4,b5,b6,
b7,b8,b9,b10,b11,b12,b13,b14,b15,b16]).

% Predicates
predicates([
     can_sense(robot,object,relation,reference),
     sense_on(robot),
     position(thing,relation,reference),
     full(gripper),
     empty(gripper),
     belongs_to(robot,gripper),
     in(object,gripper),
     released(object),
     in_range(reference,reference)]).

% Atomic Invariants
atomic_invariants([

position(r1,at,d12),position(d9,at,d4),position(r
2,near,s2),

belongs_to(r1,g1),belongs_to(r2,g2),belongs_to(r3
,g3),belongs_to(r4,g4),
     in_range(s1,b12),in_range(b12,s1),
     in_range(s2,b15),in_range(b15,s2),
     in_range(s3,b14),in_range(b14,s3),
     in_range(s4,b13),in_range(b13,s4),
     in_range(b13,b1),in_range(b1,b13),
     in_range(b15,b13),in_range(b13,b15),
     in_range(b12,b14),in_range(b14,b12),
     in_range(b14,b16),in_range(b16,b14)]).
```

Generic Types as Design Patterns for Planning Domain Specification

R. M. Simpson and **T. L. McCluskey**
School of Computing and Mathematics, The University of Huddersfield, Huddersfield, UK
r.m.simpson@hud.ac.uk, lee@zeus.hud.ac.uk

Derek Long and **Maria Fox**
Department of Computer Science, University of Durham, UK
d.p.long@dur.ac.uk, maria.fox@dur.ac.uk

Abstract

In this paper we investigate the use of 'Generic Types' as design patterns to assist in the specification of planning domains. Current planning technology uses induced patterns discovered in a domain specification to speed up plan creation. We argue that such generic types can also be used to help a domain author to develop a design for a domain at specification time using concepts at a much higher level of abstraction than is normally provided by domain specification languages.

Introduction

Research into domain independent AI Planning and Scheduling, has traditionally focused on the development of algorithms to efficiently find solutions to planning problems within the domain. The problems of dealing with what is perceived to be realistically large problems has been very difficult but recent advances in algorithms appear to make the problems more tractable. Perhaps because of the difficulty in developing capable solution generating algorithms knowledge engineering for applications of AI Planning technology is still very much in its infancy. Recent successful AI planning applications (Muscettola *et al.* 1998; A. Tate (editor) 1996) have nonetheless highlighted the problems facing knowledge engineering in planning. Questions raised by such work include issues of how to encode knowledge into domain models for use with planning algorithms. Subsequently concern over the development of knowledge engineering issues in AI Planning has resulted in a set of workshops and initiatives, including (Benjamins, Nunes de Barros, Shahar, Tate and Valente (eds) 1998; PLANET 1999).

In this paper we describe a domain definition strategy and tools to support the knowledge acquisition phase, to be carried out by domain experts rather than experts in AI Planning. We show that planning domains can be constructed using concepts at a much higher level of abstraction than has traditionally been the case in domain independent planning. Traditional languages for the specification of planning domains allow the authors of a new domain great freedom in their choice of representation of the domain details. This freedom is we contend for the most part unnecessary and provides an unwanted conceptual barrier to the development of effective domain definitions. As part of our ongoing project to enhance the tools available for knowledge engineering in planning we recently released a "Graphical Interface for Planning with Objects" called GIPO (Simpson *et al.* 2001). This is an experimental GUI and tools environment for building classical planning domain models, providing help for those involved in knowledge acquisition and the subsequent task of domain modelling. The current work is an enhancement to the *GIPO* tools environment which is supported by EPSRC grant GR/M67421, within the PLANFORM project *http://scom.hud.ac.uk/planform*.

Generic Types

The primary purpose of *generic types* in planning as introduced by Fox and Long (Fox & Long 1997; Long & Fox 2000; 2001) was to provide control information to planning algorithms to boost the speed of finding solutions to planning problems. The underlying conjecture in that work is that if common structural elements can be detected in a domain definition then specialised algorithms can be brought to bear on those elements of the problem to speed up the detection of solutions. With this in mind researchers have now identified a number of candidate generic types that can be found in a range of the domains publically available. The purpose of our research is to investigate the possibility of using generic types as *design patterns* to assist the domain modeller in the construction of an initial specification. Defining a design pattern in his seminal work (Alexander *et al.* 1977), Alexander states that:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice. *Christopher Alexander*

There are many potential advantages to presenting the domain modeller with a range of patterns around which a domain can be structured. Just as the concept of design patterns has promoted greater reuse at a higher level of abstraction in software engineering,

it can equally be beneficial for planning domain engineering. The domain modeller can benefit from encapsulated clean and elegant solutions to representation of common domain structures. Another advantage is, again as with software engineering practice, the designer is encouraged to conceptualise the domain at a higher level of abstraction than has typically been the case for modellers working in STRIPS derivative languages such as PDDL. A further benefit is that the use of canonical, "normalised" domain representations supports the opportunity for reuse of deeper domain knowledge associated with the patterns, such as control knowledge, specialised algorithmic problem-solvers and so on.

The language OCL (Liu & McCluskey 2000) which has a design suite of tools *GIPO* (Simpson *et al.* 2001) itself tries to lift the level of generality at which the modeller can design the domain by making the concept of an object and the changes of state that they undergo central to the conceptualisation of the domain. However the research group acknowledges that the task for the domain modellers who are not themselves experts in the field of A.I. planning is still too difficult. Along with other approaches being investigated by the team, this current research is seen as having the potential to help bridge the gap between the tools and techniques usable by a domain expert, who is not necessarily steeped in the technologies of AI, and those that may only be used by experienced practitioners in the field of AI Planning.

In software engineering, design patterns (Gamma *et al.* 1995) are described using stylised natural language templates, combined with UML class diagrams, describing the relationships between the fundamental building blocks of object-oriented software. In planning domains the corresponding notion of a generic type is described using relationships between the fundamental building blocks of planning domain descriptions: sorts (or types), predicates, object states and state transitions (associated with operators). These relationships can be captured graphically, in a diagram rather like a UML class diagram (Figure 1), or more formally, using declarative specifications of the necessary relationships between the components. The formalisation of these descriptions is still the subject of current research, since the precise language should combine expressiveness with precision and tractability. The role of generic types has expanded from being patterns to be identified and exploited, which demands a description that can be matched efficiently against domain descriptions, to that of domain engineering construct, which is not concerned with pattern-matching, but with expressiveness and ease of instantiation.

Broadly, a generic type then defines a class of classes of objects all subject to common transformations during plan execution. Within OCL we refer to *sorts* which are sets of objects all subject to the same characterisation and transformations, in typed-PDDL the range of a type identifies a set of objects all subject to the same characterisation and transformations. A generic

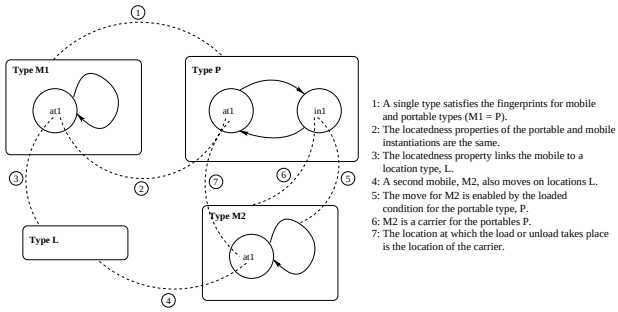


Figure 1: Pattern description for the driver generic type.

type accordingly ranges over the types or sorts of individual domains. The degree of commonality in the characterisation and transformations that these types or sorts must share have been described in the literature in terms of state machines describing the patterns of transformations that the objects undergo.

In the following section we illustrate the way in which a generic type can be used as a design pattern to support a more abstracted view of the structure within a planning domain during the engineering process.

An Extended Example

We will describe the generic type for a "mobile". A "mobile" can initially be thought of as describing the types of objects that move on a map. They can very simply be characterised by the state machine shown in figure 2

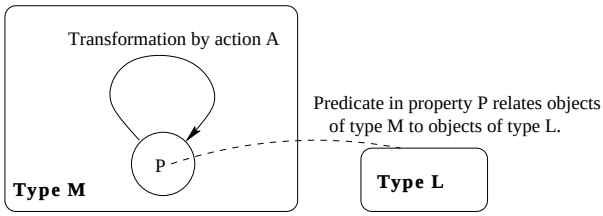


Figure 2: The Mobile Generic Type

In this one-state-machine the state is characterised by the property of the mobile object corresponding to its *locatedness* and the transition is identified by the action causing it to move. For this generic type to be applicable to a type or sort in a particular domain there must be a type such that there is an action that changes the truth value of a n placed predicate $N >= 2$, let us call it at, where one argument identifies the type in question, M, and another a value L which changes as a result of the application of the operator. That is the value of L differs in the pre- and post-conditions of the action in the reference to the at predicate. No other predicate referencing M should change truth value in the same action definition in this most basic form of the mobile prototype. The namings of predicates and arguments

may (and will) be different in different instances of the generic type. This is a very weak characterisation of a mobile, in that domains that describe actions that perform transformation on some property of the objects in question might fulfill the above requirements and hence be characterised as a mobile. From the point of view of the designer of planning algorithms this does not represent a problem as it just means that any specialised algorithms identified to speed up processing domains containing such structures will have wider application. From the point of view of someone trying to produce tools to assist domain developers it poses a problem in conveying to the user precisely what is the scope and potential uses of the pattern we have described as mobile. This problem of communication is probably exacerbated when we realise that subtle variations of the pattern need to be distinguished from one another.

Flavours of Mobiles First we characterise the transition made by such a "mobile" in terms of the formula $[at(Mobile0, LocA) \wedge LocA \neq LocB] \Rightarrow [at(Mobile0, LocB)]$ where the compound predicate within square brackets to the left of $\Rightarrow$ describes the object $Mobile0$ prior to application of the action and the formula to the right describes it after the application of the operator. The question arises as to the nature of the relationship between the values of LocA and LocB. In the formula they are required to be distinct and, we will assume, of the same type but their relationship is not otherwise constrained.

In some domains a more complex relationship might be required to hold between the two locations in order to allow movement between them. For example if "a" and "b" are locations then the transition from "a" to "b" can only be made if there is a "road" from "a" to "b". In the case of the logistics domain transition are allowed if both locations are in the same city. In yet other cases e.g. the rocket world the only restriction may be the one we have assumed anyway that the locations are all of the same type and that they are distinct from one another. Distinguishing these differences may be necessary when analysing existing domains with the intention of speeding up solution detection but it does not follow from that that we should provide the domain developer with the freedom to create each variant, with the consequent additional burden of forcing the developer to distinguish conceptually between the different patterns. We must decide whether or not the differences are essential to capturing distinct behaviours of objects in the respective domains or whether they are merely alternative ways of describing the same behaviour and represent nothing deeper than differences in encoding strategies. We believe that there are advantages to requiring the developer to come to terms with a minimal toolkit of canonical concepts to allow them to model their domains. We should only deviate from this if minimality means that the developer must conceptualise a problem at a level of abstraction that is too far removed from a natural way of thinking of the domain, in which case we may introduce features which from the minimalist point of view are redundant. Determining what represents a "natural" way of thinking about domain structures is a matter of experience and of judgement — we anticipate the need to refine the collection of generic types offered as a domain design patterns as their use develops.

Data Structures

Consideration of the very simple model of a mobile described above leads us further to distinguish different elements of a generic cluster. In particular, we need to distinguish between *data structures* that are referenced in the cluster and the dynamic generic types. Data structures are elements within the domain that are captured in predicates that do not change truth value during the application of a plan. In PDDL, data structures are given in the initial state of a problem. An example is the set of connectedness propositions that define the road structure in the truck-world which form a connected graph. Such data structures may be referenced by multiple operator definitions within the planning domain. The dynamic generic types are types or groups of types characterised by the changes in state they make during plan application.

Data structures in planning domains are not normally identified in terms of their structure but are captured implicitly in collections of predicates. In PDDL data structures are to be found in a subset of the propositions defined as *true* in the initial state of a problem definition. In *OCL*, these static propositions that do not change truth value during the planning process are collected together in the *atomic_invariants* section of the domain specification.

Examples

- *Sets* In the logistics domain the proposition *in-city(pgh-po,pgh)* is used as a way of asserting that *pgh-po* is a member of the *set* of locations defined as part of the city *pgh*. Sets may be identified more simply than this. A set may simply be represented as the values of a particular argument in a predicate. We would describe the locations that can be visited in the brief-case world as forming a set, though they are never referred to in the domain specification other than as values of the location argument in the predicate that relates an object such as the "briefcase" to a location. Given this example it might seem that the range of every *typed* variable could be regarded as forming a *set* but we distinguish between *dynamic* and *static* types and only the ranges of static types are regarded as candidates to be identified as sets. The distinction between dynamic and static objects we have discussed in (Simpson *et al.* 2000). To summarise: *dynamic* objects are those that would normally be regarded as changing their properties or relations during plan execution, where as *static* objects do not change. Again referring to the briefcase world,

in the $at(briefcase1, home)$ predicate that describes the location of the briefcase1 as being at home, it would be normal to regard briefcase1 as changing location when moving from "home" to the "office", but we would not think of the locations themselves as changing state simply as a result of the arrival or departure of a briefcase. Hence we regard the "briefcase" as dynamic but the "home" as static.

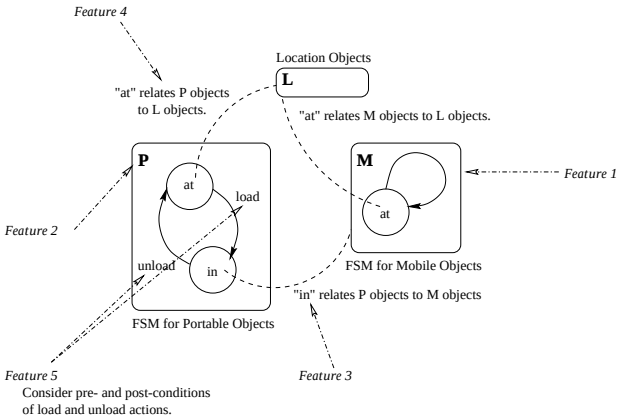
- *Maps* Examples of *maps* can be found in the "travel" world that contains collections of predicates such as $road(a, b)$ in the "init" sections of the problem specifications. The collection taken as a whole for each problem specifies a directed graph which has to be navigated by some *dynamic* object, the locations are again regarded as static.
- *Sequences* Examples of *sequences* are rarer and less obvious in the public domain planning domains. They occur in such domains as the "elevator" domain to show the relationships between floors but we would probably use a map in such an instance to model the relationship. In domains such as "truck" and "ferry" and "rocket" worlds there are single step sequences moving from "full" to "empty", which provides a very primitive way of representing resources in those domains. We generalise this and provide the notion of a sequence to enumerate the stages in the consumption or production of a resource. The use of this idea can be seen in the encoding of fuel levels and space resources in the Mystery and Mprime domains (AIPS'98 Planning Competition) and also in aspects of the encoding of the FreeCell domain (AIPS'00 Planning Competition).

Definitions

- A *map* we define as a named directed graph with nodes identified by simple labels/names and edges identified by a tuple containing the map name and node names. The tuple {x,a,b} identifies that there is an edge in map "x" from node "a" to node "b".
- A *sequence* we define as a fully ordered set with members identified by simple labels/names and a unique named $<$ relation. The tuple $\{<x$,a,b$\}$ identifies that "a" immediately precedes "b" in the "$<x$" sequence.
- A *set* we define as a *set* of items uniquely identified by labels and a predicate name identifying the set. The tuple {x,a} identifies "a" as a member of the set "x".

Defining Core Generic Types

The work done to date primarily identifies a family of types clustering around the notion of a "mobile". Figure 3 shows an initial pattern language for this collection of patterns. We distinguish two forms of mobile, those constrained to move on "maps" and those that move on "sets". The first we call "mobiles" the second we call "carriers". There is then a number of optional components that can be added to both mobiles and carriers. First both may be used to transport other objects. In which case the other objects will make a transition analogous to the "mobile" when the mobile moves but they will also make transitions when "loaded" into the mobile and "unloaded". These objects which we call "portables" can be characterised by state diagrams as shown in figure 4 The behaviour of portables are to be



The following components form the fingerprint for portability:

1. A previously identified mobile generic type, M, and its linked location generic type, L.
2. A new type, P, with a FSM containing two states linked by transitions in both directions.
3. One state of the FSM for P must include a property formed from a predicate linking the P type objects to the M type objects.
4. The *other* state of the FSM must contain a property formed from a predicate linking the P type objects to the L type objects.
5. The operators from which the two transitions in the P type FSM are derived must require an M object to be located at the same location as the P object is located at the appropriate end of the transition.

Note that the *names* of the operators and predicates are irrelevant and that the name of the predicate in feature 4 need not be the same as the locatedness predicate for type M.

Figure 4: The Portable Generic Type

determined by three actions, the action to move the mobile, an action to load the portable into the mobile and an action to unload the portable from the mobile. The state diagram for the portable does not however specify how the movement of the portable is to relate to the similarly structured movement state diagram for the mobile. Given that both describe the same "movement" action there are two plausible ways that they may relate to one another. First the transitions may both be required to take place together, in which case it will be a precondition of the "movement" action that the portable be "in" the mobile before any movement can take place. Portables of this sort we call "Drivers" and a specific transition needs to be defined for each driver participating in the action. The second

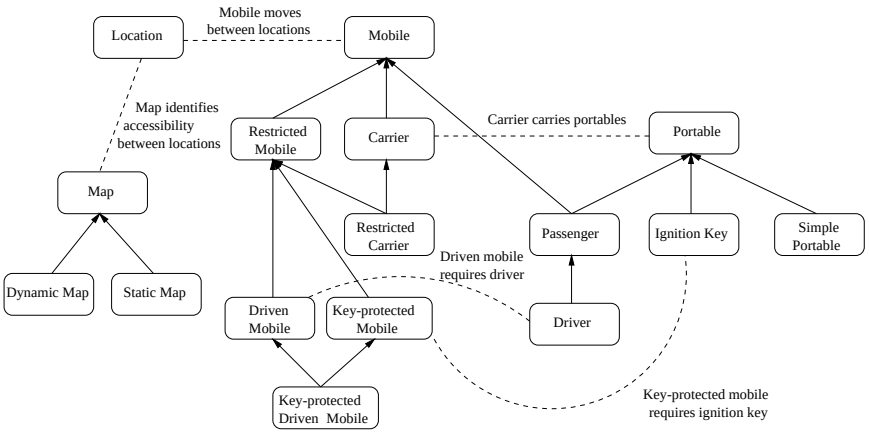


Figure 3: A Hierarchy of Mobile-related Generic Types

case arises when the transition of the portable is conditional on the portable being in the mobile but it is not required that a portable be in the mobile to allow the mobile itself to make the movement transition. In this case the conditional transition will apply to any instance of the portable "in" the mobile. We have therefore distinguished two types of mobile (a) carriers and (b) mobiles both being capable of being associated with two types of portable (i) drivers (ii) portables.

We have not yet exhausted our elaboration of the "mobile" generic type. In a number of domains, including one of the simplest, the "rocket" world, there is a notion of fuel which is a resource to be consumed as a result of the mobile moving. In the rocket world the fuel is consumed in one shot. The rockets starts with fuel full but any movement action results in the rocket being empty of fuel. We can easily see that the consumption of the resource could have been staged and to accomplish this we model the movement action as traversing a step of a sequence on each movement. Given the sequence {full, half, empty} a single movement action might take the rocket from *full* to *half* and from *half* to *empty*. With the notion of resources we can augment any transition as any transition might consume or produce some resource. The movement action may consume fuel, but equally loading or unloading some portable may consume energy.

In the discussion above we have not described features of mobiles that require "dynamic maps" nor "key" enabled actions. We have given an indication of the complexity and flexibility of the "mobile" generic type, viewed as a design pattern.

Composition of Generic Types

The problems of the composition of patterns falls broadly in two. The simple case is that already explored where a complex pattern has many optional but predictable variations. Examples are where a mobile requires a driver or consumes a resource. The more problematic case is where the domain contains two or more patterns where the same object type plays a role in more than one pattern instance. This can happen even in domains just containing mobiles. In a variation of the "hiking" domain we may have cars which can be used to transport the hikers from one centre to another but the hikers themselves may be mobiles in that they also walk from some locations to neighbouring mountain tops. In relation to the car(s) mobile pattern the hikers will play the role of either, or both, the roles of "drivers" and "portables" but in relation to their walking they play the role of mobiles perhaps even with their own portables such as the "tent" which they may carry on some walks.

The problem of composition also occurs where we have conceptually independent patterns. To illustrate, one of the patterns we are working with we call a "bistate" and it represents objects that typically exist in one of two states and there are actions to change back and forth between the states. A canonical example of a bistate would be a switch that can be "off" or "on". In the hiking domain the tent that the hikers sleep in may play a role as "portable" relative to the car and the hikers themselves, but may additionally be modelled as a bistate in that it is typically either "up" i.e. erected or "down". In this case the "erect" transition may be captured by the formula $[down(Tent0)] \Rightarrow [up(Tent0)]$
The "load" and "unload" transitions associated with the tent as portable may be captured as:
$[at(Tent0, LocA)] \Rightarrow [in(Mobile0, Tent0, LocA]$ and
$[in(Mobile0, Tent0, LocA] \Rightarrow [at(Tent0, LocA)]$.
The combination that we require is to associate the "down" state with the "at" state but we cannot simply replace the designation of the "down" state with that of the "at" state because the "at" state carries extra information about the location of the tent. We could not adequately describe the transition of the tent when we take it down as $[up(Tent0)] \Rightarrow [at(Tent0, LocA)]$ as there is no indication as to how the "LocA" variable is to be bound. Obviously the location of the tent is the same as that it had when the tent was

erected. Accordingly to preserve the location information we must merge the arguments in the "at" and "down" states and then propagate additional arguments to the other state descriptors in the merged patterns. In this case the "erect" transition now becomes $[at(Tent0, LocA)] \Rightarrow [up(Tent0, LocA)]$ and the "take down" transition is similarly enhanced.

The strategy shown here is the one that we generally follow in combining instances of patterns where a common state exists between the roles of the merged patterns.

Domain Definition using Generic Types

To enable the domain developer to use the identified generic types to structure a domain we have developed a series of dialogs which we have integrated into the *GIPO* domain development tool. The dialogues allow

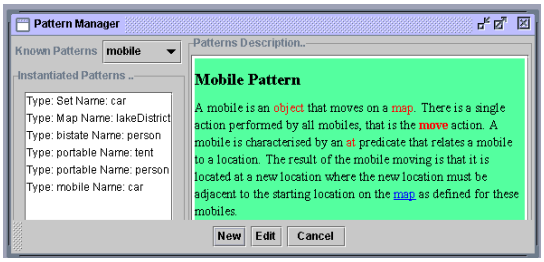


Figure 5: The Pattern Manager

the user to choose the relevant patterns and then tailor them to the problem in hand. In simple cases tailoring is simply a matter of naming the components of the pattern in an appropriate way. In more complex cases the user must add optional components to the pattern again by form filling and in the most complex cases ensure that domains using multiple patterns allow them to interact with each other in the correct way. The set of dialogues form a domain editor in such a way that the user committing her choices in the editing dialogues will result in the formal domain specification being automatically generated. We illustrate the process with snapshots taken from the "Pattern Manager" in figure 5 which is used to control the addition and editing of patterns known and instantiated within the domain. We also show the main dialog for defining the parameters of the "mobile" pattern in figure 6.

Evaluation

The implemented pattern editors that we have produced currently give good coverage of domains featuring "mobiles" of one sort or another. Our evaluation is currently limited to testing to see if we can produce using the pattern editors versions of the domains that have been made available as part of previous *AIPS* competitions. We are judging equivalence of domains not at

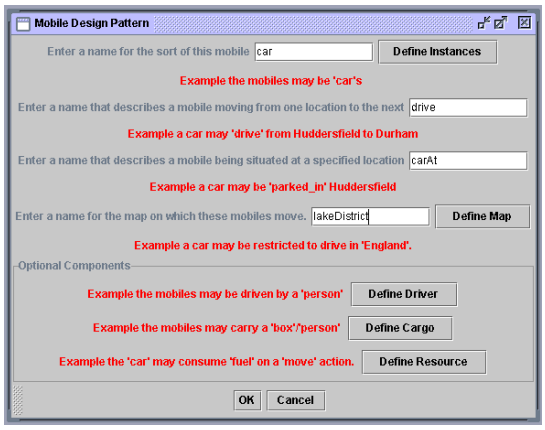


Figure 6: The Mobile Dialog

the level of encoding the operators and problems but rather at the level of generality that would allow us to say that derived solutions to equivalent problems are equivalent. We do not require for example that any planner that works with the original will work with our generated version, as this is not the case even in the rocket world as our encoding uses conditional effects whereas the commonly available originals typically do not. We would also judge domains to be equivalent even where they do not contain the same number of operators or predicates, for example a *move* operator may be either expanded into multiple *move* operators each responsible for moving objects of different sorts or conversely we may contract multiple operators into a single operator dealing with a more general *sort*.

An interesting observation is that though the pattern-directed reconstructions of classic domains are not always identical to the familiar encodings we consider it a strength of the use of patterns that a canonical encoding, with its attendant well-understood behaviours, is used to encode the domains. Nevertheless, it raises an important point about the expected behaviour that a domain is intended to capture. In reconstructing domains we expect to find that there is a correspondence between legal states of the reconstructed domain and the original encoding, with an induced correspondence between plans in the two domains (see figure 7). Confirmation that this correspondence exists forms a reasonable element of the evaluation of the use of the pattern-directed approach to domain construction.

Evaluation of the provision of support for domain construction by domain design patterns is difficult. It is intended that they make domain construction easier, but this is a matter of HCI and could only be empirically evaluated with access to a reasonable sample of potential users. Of course, the developers consider the

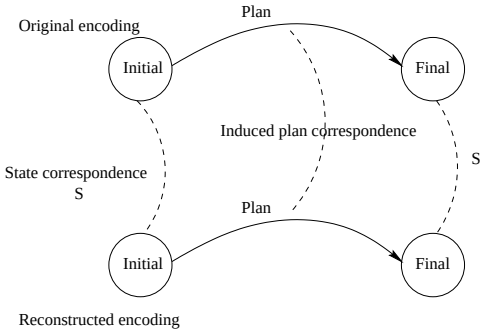


Figure 7: Equivalence between domain encodings.

approach to be an efficient and convenient way to generate domains. A separate dimension of evaluation is to consider the extent to which the patterns provide support across a wide range of domains. For example, one might consider how many benchmark domains can be reconstructed using the patterns, with the patterns providing support for a high-level view of the domain objects and their behaviours. It is of interest to note that many of the benchmark domains include a transportation element (Logistics, Gripper, Briefcase, Grid, Mystery and MPrime are all examples). The Tyreworld domain consists chiefly of interlocking bistate elements (hubs can be up or down, nuts are on or off, loose or tight, and wheels are on or off, inflated or deflated). Many of the other domains contain a construction component (Hanoi, Blocksworld, Assembly and Freecell) and we are currently exploring the implementation of a generic cluster to support construction.

Further Work

The work described above is still work in progress. We continue to develop it at a number of levels. We continue to work on incorporating known "generic types" into the *GIPO* tool and to enhance the facilities within the tool for creating, editing and combining patterns. At the level of the patterns themselves there is still more work to be done in identifying new patterns and elaborating further the existing patterns. We are also working at formulating more formally the rules for combining patterns with an ultimate goal of providing a formal description of patterns and an "algebra" for their composition. Ideally the outcome of this work would be tools to allow the domain designer to develop a wide range of domain definitions without the need to develop the domain in any way at the level of the underlying specification language such as *OCL* or *PDDL*. A further goal of the work is also to provide planning algorithms with information on the instantiated patterns to allow them to use this as control information to inform the planning process itself. We do not expect however that this will eliminate the need for further domain analysis to assist in speeding up planning solution production.

References

A. Tate (editor). 1996. *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*. IOS Press.

Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; and Angel, S. 1977. *A Pattern Language*. Oxford University Press.

Benjamins, Nunes de Barros, Shahar, Tate and Valente (eds). 1998. Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice. Proceedings of AIPS.

Fox, M., and Long, D. 1997. The Automatic Inference of State Invariants in TIM. *JAIR* 9:367–421.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of reusable software*. Addison-Wesley.

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield .

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Proc. of 5th Conference on Artificial Intelligence Planning Systems (AIPS)*, 196–205. AAAI Press.

Long, D., and Fox, M. 2001. Planning with generic types. Technical report, Invited talk at IJCAI'01 (forthcoming Morgan-Kaufmann publication).

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–48.

PLANET. 1999. *PLANET Knowledge Technical Coordination Unit*. http://scom.hud.ac.uk/planet.

Simpson, R. M.; McCluskey, T. L.; Liu, D.; and Kitchin, D. E. 2000. Knowledge Representation in Planning: A PDDL to OCL_h Translation. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*.

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*.

Integrated modelling: when time and resources play a role

Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské námestí 2/25
118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz

Abstract

A formal model of the planning or scheduling problem is the first step in the design of a solver for such a problem. In the paper we propose a basic framework for modelling planning and scheduling problems that involve reasoning about time and resources. In this framework we go beyond the traditional definitions of planning and scheduling and, from the beginning, we expect integration of both these areas.

Introduction

Traditional AI planning tackles the problem of sequencing operators to achieve some goal. In STRIPS-like planning, the operator is defined by pre-conditions and effects, i.e., the pre-conditions must be satisfied to use the operator, and the effects hold after using the operator. The task is to find a sequence of operators starting from a given set of pre-conditions and achieving a given set of effects.

There is no explicit usage of time and resources in traditional planning. In fact, there are no numeric values used so planning methods are based mostly on symbolic manipulation. That is the reason why planning is assumed to be an AI problem rather than a number crunching task. Nevertheless, we can find time and resources behind the traditional planning notions. At least relative time must be assumed if speaking about operator sequencing, i.e., the pre-conditions hold just before we execute the operator and the operator's effect will be true since we execute the operator (until another operator annihilates the effect). Still, traditional planning uses instantaneous operators, i.e., no duration of the operator is assumed. This is OK if we are just sequencing the operators, but, it may cause problems when overlaps of operators are allowed. Moreover, in reality executing the operator takes some time so the planning system should assume this time when looking for a valid sequence of operators. The above observations are reflected in so called durative actions that are included in the recent version of PDDL (Fox and Long 2001), a modelling language for planning problems, and that are studied in (Coddington, Fox, and Long 2001).

While time is hidden in semantics of operators, the resources can be encoded in formulas defining pre-conditions and effects. Even one the earliest planning problems - a block world problem - involved a resource, the robot's hand that moves the blocks over the table. Encoding resource in pre-conditions and effects is a standard way of modelling resources in traditional planning. However, this technique covers only a limited number of resources, we can call them *state resources*. Pre-conditions describe a required state of the resource to execute the operator, e.g., an empty hand, and effects describe a state of the resource after executing the operator, e.g. holding a block A.

In reality, the interaction between resources and operators and the integration of time and resources is more complex, e.g. a single resource may execute several operations in parallel. This brings planning to a new level where the quality and feasibility of the plan depends on time and resources too. Planning community is aware of such real-life demand and handling of time and resources is a hot topic in AI planning.

Time and resources play a key role in the areas of scheduling and timetabling too. The scheduling task is to allocate a known set of activities to available resources over time respecting precedence, capacity and other constraints. Timetabling can be seen as a special case of scheduling (Wren 1996) with different view of space-time (slots) and different objectives. Thus, we will not speak about timetabling separately.

The main difference of scheduling (and timetabling) from planning is that in scheduling we know the structure of activities while planning has to construct this structure. Therefore, when solving real-life problems planning and scheduling modules can be kept separated: first, we plan which activities (operators) are necessary to satisfy the demands and, second, we schedule the activities to available resources. This could be useful in some problems due to efficiency issues (Srivastava and Kambhampati 1999) but in other areas, integration of scheduling and planning seems necessary (Barták 1999) or (Smith, Frank, and Jónsson 2000). Note that this integration is not easy because of rather different techniques used to solve problems in planning and scheduling. While planning is

based mainly on symbolic manipulation, scheduling uses number crunching techniques from operations research. Recently, constraint satisfaction seems to provide a bridge between these two different technologies so discussions about integration of planning and scheduling are becoming more realistic now. Constraint programming is a widespread technology in scheduling (Wallace 1994); application of constraint satisfaction techniques to planning problems is described in (Binh Do and Kambhampati 2000), (Laborie 2001), (Nareyk 2000), or (Van Beek and Chen 1999) among others.

When speaking about integration of planning and scheduling, a formal modelling framework to describe such problems is one of the first issues. There exists a de facto standard modelling language PDDL for description of planning problems (Ghallab et al. 1998) and this language is being extended to model time (Fox and Long 2001). Other approaches in planning attempts to model resources (Brenner 2001) or (Koehler 1998). Still, all these approaches have their limitations when describing real-life resources and time.

Surprisingly, there is no system independent language for scheduling problems; at least we are not aware of any such language. There exists a well-known classification of scheduling problems using the triple (machine environment | job characteristics | optimality criterion) by Graham et al. (Brucker 2001). However, this is an academic classification, not a modelling language to describe a particular problem. Some modelling languages, like STTL (Kingston 2001), exist for timetabling problems but these languages can hardly be extended to general scheduling problems or to planning problems.

In this paper, we describe a framework for integrated description of both planning and scheduling problems. This framework is based on our previous works on modelling scheduling problems enhanced by planning capabilities (Barták 1999) and (Barták and Rudová 2001) so time and resources play an important role there. We have abstracted from a particular scheduling problem to cover a wider class of problems including pure planning and pure scheduling problems. This paper is a bit refined version of our proposal from (Barták and Rudová 2001). Here, we concentrate on a basic structure of the framework rather than on particular attributes (even if we mention some attributes to illustrate how the objects are used). This gives us a freedom of designing a generic framework that can be filled by attributes and that way adapted to a particular problem area. We also describe how such formalisation can be used to support planning/scheduling.

The paper is organised as follows. In Section 2, we highlight the main roles of formal models. In Section 3, we describe basic modelling requirements to capture real-life planning and scheduling domains and in Section 4 we specify how to model a particular problem in the given domain. Section 5 is dedicated to pre-scheduling and pre-planning techniques that prepare the formal model for solving.

A Context for Formal Models

The design of a formal model is a crucial step to understand all the details of the problem and to find a solution of the problem. However, having a formal model or more precisely having a modelling language to describe problems has other advantages. Basically, such modelling language serves as an interface (see Figure 1).

Naturally, the modelling language forms an interface between the real problem and the solver. Having such interface brings several advantages. First, the solver is independent from the problem description, i.e., it is possible to exchange the solver for a better one without changing the problem specification. For example, we can use a special solver for a particular domain without changing the user interface of the system or the problem description. Second, we can have several user interfaces for modelling different problems and all these user interfaces may share a common generic solver via the unified interface. In fact, we can use an automated modeller that converts the problem description from an ERP system or, generally, from a database describing the problem to a formal model. The solver does not need to know what is the source of the model. To summarise it, the formal modelling language provides an interface between various modules in a complete planning/scheduling system.

Universal description of planning and scheduling problems brings also the advantage of sharing problem domains and problems between researchers. Thus, it simplifies maintenance of benchmark sets. We sketch some other usages of the formal model later in the paper.

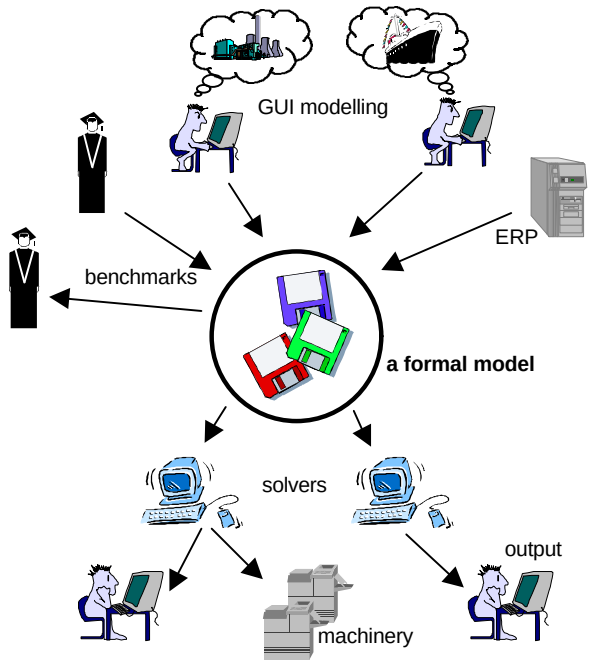


Figure 1. The role of a formal model.

Domain Modelling

When describing a problem, we can start with the description of the problem area - a domain. This makes the model more general, because it simplifies changes of the model. What is it a domain? Let us start with a real-life example of industrial scheduling. When scheduling processes in the factory, the problem description consists of the description of the factory, i.e. machines and processes, and the description of demands (orders). In this case, the domain corresponds to the description of the factory and the particular problem consists of the domain and a set of demands. We can say that the domain is a static part of the whole problem that is not changing or the changes are less frequent.

We propose the model for a domain to consist of three basic elements: activities, resources, and recipes. Activity is a basic scheduled/planned object that usually occupies some time and space. Resources define space for processing the activities and recipes describe direct relations between the activities.

Resources

Resource is an object that defines space for processing the activity. We will speak about connection between resource and activity later, so let us now concentrate on resource-only features.

Life of the resource, i.e., evolution of the resource in time can be described using a sequence of *states*. For example, the resource oven uses four states load - heat - unload - clean and these states are repeating in a cycle. Some resources, e.g. classroom in timetabling, have only one state. We expect that resource is an object (machine, room etc.) so consumable resources like fuel are modelled using a tank etc. The resource appears in a single state at a given time so the schedule for the resource consists of the sequence of non-overlapping states.

Basically, the model of resource consists of the set of states and transitions among the states (see Figure 2). The transition describes how the resource can change a state. Typically, information about timing is included so we can define minimal and maximal duration of the state, working time for the states, and transition time.

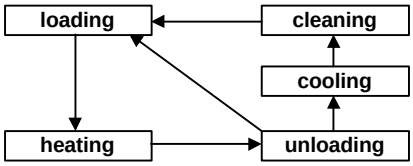


Figure 2. A state transition diagram for the resource.

Because the resource defines a space for activities, we should describe how much space is available in each state - a state capacity. The state capacity restricts the number of activities that can be processed together. We can also restrict the alignment of activities in the state. Basically, we distinguish between parallel processing, where there is no restriction about the alignment of activities, and batch processing, where the overlapping activities must start and complete at identical times (see Figure 3).

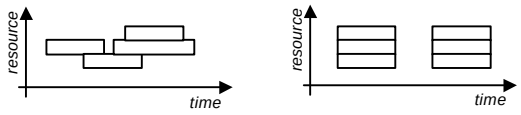


Figure 3. Parallel (left) vs. batch (right) processing.

To summarise the above discussion, the model of resource consists of the states with some attributes and the transitions between the states (see Figure 4).

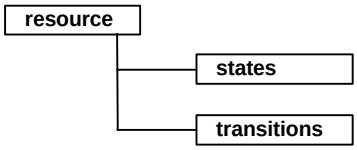


Figure 4. A basic structure of the resource model.

Activities

Activity is a basic scheduled/planned object so when modelling the problem we should specify which activities can be used in the solution. The basic attribute of the activity is its duration, i.e., time occupied by the activity. We can also use time windows to restrict when the activity can be processed.

In many cases, the activity requires some resources for processing. For example, a lecture in timetabling requires a classroom and a teacher, a heating activity in industrial scheduling requires an oven, and a moving activity in transport planning requires fuel. So for each activity we can assign a set of resource requirements. In the *resource requirement* we describe the way of using the resource. Some resources are consumed or produced, we call them consumable resources, and some resources are just used, we call them renewable resources (see Figure 5).

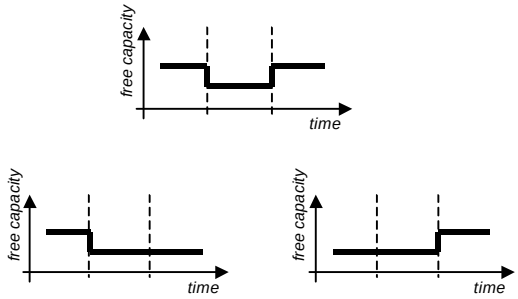


Figure 5. Renewable (top) and consumable (bottom) resources. Dashed lines indicate start and end of the activity.

Naturally, we should also describe what capacity of the resource is consumed/used/produced. We can also describe what state of the resource the activity requires. Note that the states with batch processing are meaningful for renewable usage of the resource only while parallel processing can be used both for renewable and for consumable usage of the resource.
When specifying the resource requirement, we usually have alternative resources that can satisfy the requirement. Thus we attach a list of resources to each requirement (see Figure 6).

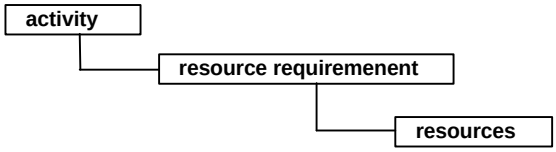


Figure 6. A basic structure of the activity model.

Recipes

The model of activities and resources can describe an indirect relation between the activities only. In particular, the only modelled relation between the activities is via a shared resource, e.g., two activities cannot run in parallel if they share a resource with capacity 1. Such modelling is usually enough for (most) timetabling problems. However, in planning and scheduling we need to model direct relations between the activities (and between the resources), for example a supplier-consumer dependency or a precedence.

Traditional planning uses STRIPS-like rules (Fikes and Nilsson 1971) to model relations between the activities: each activity has some pre-conditions and it generates some effects that may become pre-conditions of another activity. If we add some attributes to the pre-conditions and effects (typically logical terms are used to describe both pre-conditions and effects) we have a general mechanism for information passing between the activities. In HTN (Hierarchical Task Network) Planning (Erol, Hendler, and Nau 1994) the activities are connected into a task graph so more constraints can be expressed over the activities. Moreover, the tasks can be part of another task graph so planning is done via task decomposition and conflict resolution.

To simplify description of relations between the activities we introduce a notion of *event*. Each activity requires some events to precede it, we say that the activity consumes the events, and each activity generates some other events, we say that the activity produces the events. We call a triple (activity, consumed events, produced events) an *activity environment*. Note that we may have several environments for a single activity, e.g., there exists various combinations of input items consumed by the activity that produces another item. Moreover, we can put constrains between the event and the activity, for example to describe the allowed delay between the event and the activity.

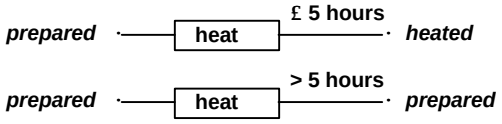


Figure 7. Two activity environments for a single activity; consumed events are on the left side and produced events are on the right side. Notice also the timing constraint between the activity and the produced event.

To provide richer modelling capabilities we propose to combine activity environments into a *recipe*. Basically, a recipe is a DAG (directed acyclic graph) where nodes are marked by activities and events. The edge goes either from an activity to an event produced by the activity or the edge goes from an event to the activity that consumes the event. In particular there are no direct edges between the activities and no direct edges between the events. The activity must be connected to all its produced and consumed events (for a given activity environment). So an activity environment forms a sub-graph in the recipe. If there are more environments for the activity then the activity may appear more times in the recipe (each appearance corresponds to one activity environment). However, there are no duplicate events in the recipe. There is one exception when the event may appear two times in the recipe. If the event is produced by one activity and consumed by another activity and connecting both activities to the same event node forms a cycle in the graph. To break the cycle (we require the recipe to be a DAG) we divide the event into two events, one is used as a consumed event only and the other one is used as a produced event only. Let us call such event a *broken event*. Such situation may appear if we want to model recycling or similar features of the real problem (see Figure 8).

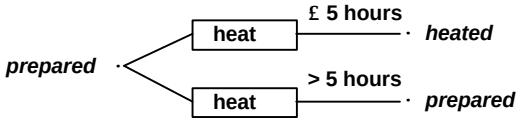


Figure 8. A primitive recipe Heating; the edges goes from left to right. There is also a broken event "prepared".

In the recipe, there exist three types of events: events that are both produced and consumed (by different activities), events that are produced only, and events that are consumed only. In case of recycling described above, the broken event is part of both consumed-only and produced-only sets of events. Together, the recipe behaves like a meta-activity and thus we can use the recipe within another recipe like an activity environment (see Figure 9).

During planning we are decomposing the required recipes to individual activities but we can also connect different recipes via common events (one recipe produces the event and another recipe consumes the event). Still there could be some events that are consumed only (there is no action that consumes such event); these events may

correspond to purchases of raw material etc. Similarly, there could be produced only events, e.g. describing appearance of the final product. We call such produced-only and consumed-only events *one-way events.*

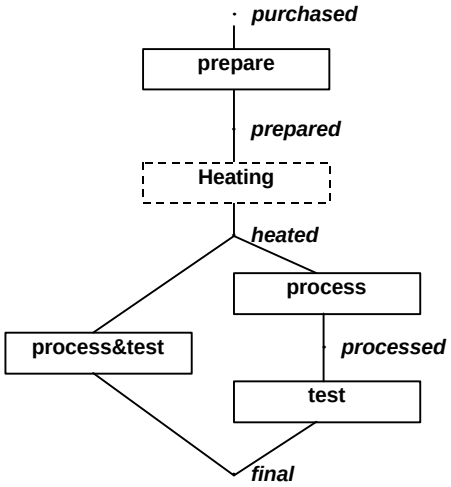


Figure 9. A recipe using another recipe (dashed).

If we expect that all the events have unique names then we can represent the recipe as a set of activity environments and recipes. In such a representation it is clear how the activities and recipes are connected via common events.

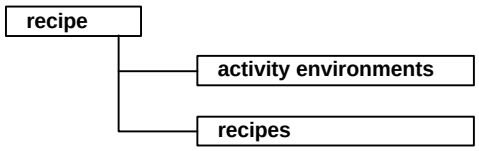


Figure 10. A basic structure of the recipe.

Problem Modelling

A domain model describes the problem area i.e. which resources are available, what activity types can be used, and what are the relations between the activities. To specify a particular problem we need to describe the actual activities. This could be done explicitly, like in traditional scheduling and timetabling, where the set of activities is given as the input and the task is to allocate the activities to resources respecting the resource and recipe (precedence) constraints. In traditional planning, the input consists of some events and the task is to generate the activities in such a way that the events are connected via activities i.e. the activities in the plan are described implicitly via the events. In our framework, we propose to combine both these ways of input specification, i.e., depending on the input we will solve either a pure scheduling (timetabling) problem or a pure planning problem or a mixture of both.

Initial data

If we are using resources in the problem, it is a good manner to describe the initial situation/state of each resource. In timetabling this is useless because there are no states. In pure scheduling this is done via specification of the activity with pre-allocation of the activity to the resource and to initial time.

In our framework we allow description of the initial state(s) of each resource as well as specification of activities that are known before we start scheduling. These activities may be pre-allocated, i.e., some of the parameters of the activity are known (like time and used resources) or the parameters are unknown and the task is to find their value (allocate the activity to resources and time). Using such initial data allows us to model pure scheduling and timetabling problems or to use the system to complete partially known schedules. In the second case, new activities are introduced during scheduling to fill gaps in recipes.

Goals

To further extend the planning features of the framework, we allow specification of known events in the description of the problem. Remind that the events make a connection node between the activities. If there appears an event in the system then this event must be produced by some activity and consumed by another activity. Only the one-way events may have either the consumer or the producer. To start planning, we can put some initial events to the system and the system will try to cover them, i.e., to find an action that produces the event and/or the action that consumes the event. Introduction of the action may cause introduction of new events and the task is to cover all the events. As we said above it means that there must be an action producing the event and an action consuming the event. A missing action (producer or consumer) in a one-way event is substituted by including the event among the initial events. Note that this process is similar to STRIPS planning where we have to find activities generating the final effects using the initial pre-conditions.

It is possible that some one-way events are introduced during the process of planning and these events are not included among the initial events. For example we can introduce an event describing a purchase of raw material. To allow such situation we can mark some one-way events as free events. Then, we can introduce a free event during planning if some activity requires it even if the event is not among the initial events.

To summarise the above paragraphs, the problem is described by specifying the domain (a problem area) and by describing some objects in the final schedule, namely some activities and initial events. The task is to fill the gaps in the schedule following the recipes and respecting the resource constraints (see Figure 11). It means that the resulting plan consists of the activities allocated to resources and connected with other activities via events.

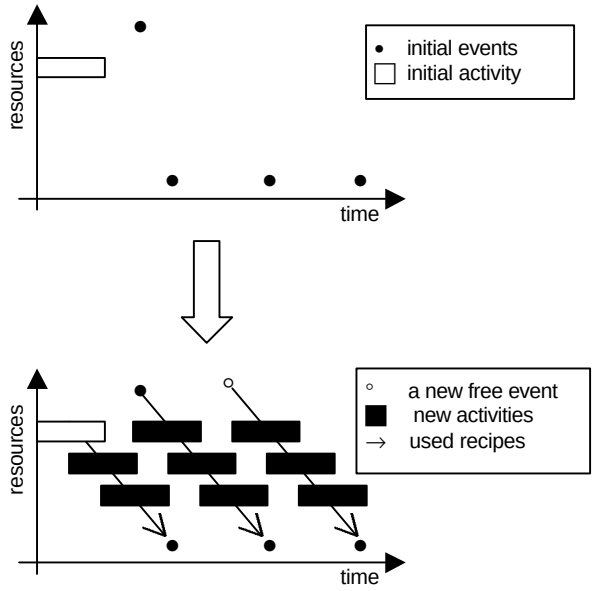


Figure 11. Gantt charts - from the problem description (top) to the solution (bottom).

Before Planning/Scheduling

When the problem is described formally, the next step is to solve the problem. However, we can look in data and insert some addition steps that may simplify the process of problem solving. Basically, we distinguish three steps that could/should be done before we start solving the problem:

- data checking that reports errors in data,
- data pre-processing that simplifies the model,
- data analysing that finds useful information for the solver.

Data Checking

The modelling framework may provide a formal language for problem description. This language is typically based on some underlying language like LISP or Prolog so we can use tools of the underlying language to ensure syntactic soundness of the model. Nevertheless, syntactic soundness does not guarantee that the model is semantically correct, i.e., that there are no bugs prohibiting finding a solution. Especially, when a less-experienced user designs the model we can expect many such bugs. Semantic bugs can be discovered during planning/scheduling but it is often a tough process leading to a very long computation (the system tries to find a solution even if "visibly" no solution exists).

Our and others [personal communication to Helmut Simonis] experience says that it is very important to check data before we start scheduling. Such data checking can be automated in some way, for example to discover (some) clashes in data. Some data-checking can be general, i.e., designed for all models. For example, we can check if resources required by activities are present in the model or if all states of the resource are accessible from the initial state. Note also that the data checker may identify parts of the model that could cause problems, i.e., it is not an error but it could be an error. For example, the graph of states for the resource consists of more components that are not connected etc.

Other data checks may be designed for particular instances of the modelling framework. Assume that time windows are defined for activities and for states of the resources. We can check if the activity can be processed by a given resource in a given state by comparing time windows of the activity and the state.

We mentioned just few data checks, many other checking techniques can be proposed for a particular class of models. Generally, data checking is not a complete technique that guarantees existence of the solution; otherwise complete data checker includes full planner/scheduler which is not the goal of data checking. The point is that as much as possible (polynomial) data checks should be done before we start (exponential) planning or scheduling.

Data Pre-processing

When the developer designs a formal model of a non-trivial problem then he or she should take in account the details of the solving algorithm. Sometimes the modelling language guides the user to design "reasonable" models but if the modelling framework is general like our framework then it is hard to integrate all good modelling skills into the modelling language itself. Moreover, the end users prefer the models that are close to what they know in reality rather than the models with "low-level" tricks that make the model easier for scheduling. Finally, the formal model of the problem may be designed automatically from an ERP system or a database describing the domain. All in all, the pure formal models may contain features that are sound but that make scheduling more complicated.

To remove "bad" features of the pure model we can smooth it out by applying some pre-processing techniques that change the model into a model easier for scheduling. The only requirement about the pre-processed model is that it must be equivalent to the original model. Such equivalence is defined in the following way: the post-processed schedule of the pre-processed model is a schedule of the original model (see Figure 12).

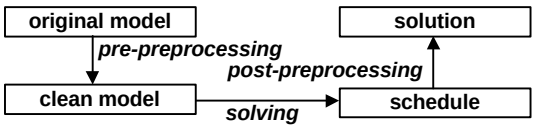


Figure 12. Using pre- and post-processor

Some pre-processing techniques change significantly the model, e.g. by using different structure of activities. For example, when the end user describes a resource using

states, he or she tends to describe all possible states, e.g. loading - processing - unloading. Or all these states are saved in a database describing the resource so it is natural that they appear in the automatically generated model as well. It implies that there must be loading, processing, and unloading activities as well. However, if such sequence is unique, then the experienced modeller abstracts from these states and uses just one abstract state/activity to describe the situation. Visibly, using just one activity is easier for scheduling then using three activities. Moreover, if we substitute such activity by a triple of activities in the final schedule then we get a schedule for the original problem so the conversion is sound.

Other pre-processing techniques are less invasive and they just remove some unfeasibility from the model. Assume that time windows are defined for activities and for resources (their states). If we know that some resource must be selected from the set of alternative resources then we can make an intersection of the time window for the activity with the union of the time windows of these resources to get a new time window for the activity. It means that some values may be removed from the time window for the activity, which decreases the search space.

Note finally that pre-processing is closely related to data checking so both techniques can be applied together.

Data Analysing

When we have a sound and complete formal description of the problem, there remains one "small" step - to solve the problem. There exist many solving algorithms for particular classes of planning, scheduling, and timetabling problems; the hard job is to choose the one that best fits the problem or to extract some information from data that the solver can use.

At top level, by analysing the data we can decide automatically whether the problem belongs to traditional planning (no resources and time), to traditional scheduling (all activities are known), or if it requires both approaches. In case of traditional problems we can further classify the problem. For example, in traditional scheduling there exists a Graham's classification of scheduling problems and a catalogue of efficient algorithms for solving these problems (Brucker 2001). Theoretically, if we classify the problem, which can be done by analysing the data, then we can find a solving algorithm automatically. Unfortunately, Graham's classification is rather academic so we can hardly expect that a real-life problem fits into a category in this classification. Still, it is possible to find sub-problems that can be solved using existing efficient algorithms and the rest of the problem is solved using some generic technique like constraint satisfaction.

Note that data analysis may be used also to find additional information for the solving algorithm. For example, we can go beyond simple activity joining described in the previous section and we can identify some required dependencies between the activities, so called landmarks (Porteous and Sebastia 2000). A planning algorithm can then use information about landmarks to improve its efficiency.

Conclusions

In the paper we describe an integrated framework for modelling planning and scheduling problems. We concentrate on an informal description of such a framework rather than on a precise specification of all the attributes and solving algorithms.

This paper extends the work from (Barták and Rudová 2001) in the way of more precise specification of objects, in particular recipes. We also separated the model of domain from the problem and we put our framework into a context of existing frameworks for planning (like HTN, STRIPS) and scheduling (resources). Finally, we showed how such a formal framework might automate some data processing before we start planning/scheduling.

Acknowledgements

The research is supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942. The author would like to thank Hana Rudová, Roman Mecl, and the team of VisOpt Ltd. for useful discussions concerning modelling real-life scheduling and timetabling problems. The author is also grateful to Ondrej Cepek for proof-reading of the paper draft.

References

Barták R. 1999. On the Boundary of Planning and Scheduling: A Study. In *Proceedings of the 18th Workshop of the UK Planning and Scheduling SIG*, 28-39, Manchester, UK.

Barták R. and Rudová H. 2001. Integrated Modelling for Planning, Scheduling, and Timetabling Problems. In *Proceedings of the 20th Workshop of the UK Planning and Scheduling SIG*, 19-31, Edinburgh, UK.

Binh Do M. and Kambhampati S. 2000. Solving planning-graph by compiling it into CSP. In *Proceedings of AIPS 2000*, 89-91.

Brenner M. 2001. A Formal Model for Planning with Time and Resources in Concurrent Domains. In *Proceedings of IJCAI-01 Workshop Planning with Resources*, Seattle.

Brucker P. 2001. *Scheduling Algorithms*. Springer Verlag.

Coddington A., Fox M., Long D. 2001. Handling Durative Actions in Classical Planning Frameworks. In *Proceedings of the 20th Workshop of the UK Planning and Scheduling SIG*, 44-58, Edinburgh, UK.

Erol K., Hendler J., and Nau D. 1994. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network

Planning. In *Proceedings of 2nd International Conference on AI Planning Systems*, 249-254.

Fikes R. and Nilsson N.J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2: 189-208.

Fox M. and Long L. 2001. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. Technical Report, Department of Computer Science, University of Durham, UK.

Ghallab M., Howe A., Knoblock C., McDermott D., Ram A., Veloso M., Weld D., Wilkins D. 1998. PDDL - The Planning Domain Definition Language, Tech Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Kingston J.H. 2001. Modelling Timetabling Problems with STTL. In *Proceedings of The Practice and Theory of Automated Timetabling*, 309-321, LNCS 2079, Springer Verlag.

Koehler J. 1998. Planning under Resource Constraints. In *Proceedings of 13th European Conference on Artificial Intelligence*, 489-493, Brighton, UK.

Laborie P. 2001. Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results. In *Proceedings of 6th European Conference on Planning*, 205-216, Toledo, Spain.

Nareyek A. 2000. AI Planning in a Constraint Programming Framework. In *Proceedings of 3rd International Workshop on Communication-Based Systems*.

Porteous J. and Sebastia L. 2000. Extracting and Ordering Landmarks for Planning. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling SIG*, 161-174, Milton Keynes, UK.

Smith D.E, Frank J., and Jónsson A.K. 2000. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*, 15(1): 61-94.

Srivastava B. and Kambhampati S. 1999. Scaling up Planning by teasing out Resource Scheduling. Technical Report ASU CSE TR 99-005, Arizona State University.

Van Beek P. and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In *Proceedings of AAAI-99*, 585-590.

Wallace, M. 1994. Applying Constraints for Scheduling, in: *Constraint Programming*, Mayoh B. and Penjaak J. (eds.), NATO ASI Series, Springer Verlag.

Wren A. 1996. Scheduling, Timetabling and Rostering - A Special Relationship. In *Proceedings of The Practice and Theory of Automated Timetabling*, 46-76, LNCS 1153, Springer Verlag.

Extending TIM domain analysis to handle ADL constructs

Stephen Cresswell, Maria Fox and **Derek Long**
Department of Computer Science, University of Durham, UK.
{s.n.cresswell,maria.fox,d.p.long}@durham.ac.uk

Abstract

Planning domain analysis provides information which is useful to the domain designer, and which can also be exploited by a planner to reduce search. The TIM domain analysis tool infers types and invariants from an input domain definition and initial state. In this paper we describe extensions to the TIM system to allow efficient processing of domains written in a more expressive language with features of ADL: types, conditional effects, universally quantified effects and negative preconditions.

Introduction

The analysis of a planning domain can reveal its implicit type structure and various kinds of state invariants. This information can be used by domain designer to check the consistency of the domain and reveal bugs in the encoding. It has also been successfully exploited in speeding up planning algorithms by allowing inconsistent states to be eliminated (Fox & Long 2000), by revealing hidden structure that can be solved by a specialised solver (Long & Fox 2000), or as a basis for ordering goals (Porteous, Sebastia, & Hoffmann 2001).

Many other researchers are also interested in domain analysis. Planners based on propositional satisfiability (Kautz & Selman 1998) and CSP (van Beek & Chen 1999) often require hand-coded domain knowledge, much of which could be derived from domain analysis. TLPlan (Bacchus & Kabanza 2000) and TALPlanner (Doherty & Kvarnstrom 1999) rely on control knowledge which might be inferrable from the underlying structures describing the behaviour of the domain. Our analysis produces not just invariants, but the underlying behavioural models from which they can be produced. These models provide a basis for further analysis, giving an advantage over other invariant algorithms, such as DISCOPLAN (Gerevini & Schubert 1998), which do not produce these structures.

The TIM system performs its analysis by constructing from the planning operators a set of finite state machines (FSMs) describing all the transitions possible for single objects in the domain. The type structure and domain invariants are derived from analysis of the FSMs.

Earlier versions of TIM have accepted planning problems expressed only in the basic language of STRIPS. A more expressive language for describing planning domains using features derived from ADL (Pednault 1989) is in widespread use. The use of these extensions makes life easier for the domain designer, but it is difficult to handle them efficiently in planning systems.

In this paper we describe work to extend the language handled by TIM to include the main features of ADL, whilst also attempting to preserve the efficiency of TIM processing. The features we describe here are types, conditional effects, universally quantified effects and negative preconditions.

For example, in the briefcase domain, consider the `move` operator, shown below. The operator includes a universally quantified conditional effect which says that when the briefcase is moved between locations, all the portable objects which are inside the briefcase also change their location. [1]

```
(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l)
               (not (is-at ?m))
               (forall (?x - portable)
                  (when (in ?x)
                     (and (at ?x ?l)
                     (not (at ?x ?m)))))))
```

A typical invariant that we would like to get from this domain is that portable objects are at exactly one location:

$$\forall x : \text{portable} \cdot \forall y \cdot \forall z \cdot at(x, y) \wedge at(x, z) \rightarrow y = z$$
$$\forall x : \text{portable} \cdot (\exists y : \text{location} \cdot at(x, y))$$

TIM

For a full account of the TIM processing, see (Fox & Long 1998). Here we briefly summarise those parts of the algorithm which are relevant to the extension to ADL.

[1] This encoding of the domain is from IPP problem set.

We are concerned with extracting state descriptions for individual objects. These are described using *properties*, which describe a predicate and an argument position in which the object occurs. For example, if we have $at(plane27, durham)$ in the initial state, then we consider the object $plane27$ to have property at_1, and the object $durham$ to have property at_2.

To explain the processing of domain operators by TIM, we must introduce two kinds of derived structure — the *property relating structure* (PRS) and the *transition rule*.

The PRS is derived from an operator, and records the properties forming preconditions, add effects and delete effects for a single parameter of an operator. For example, consider the operator `fly`:

```
(:action fly
  :parameters (?p ?from ?to)
  :precondition (and (at ?p ?from) (fuelled ?p) (loc ?to))
  :effect (and (not (at ?p ?from))
               (at ?p ?to)
               (not (fuelled ?p))
               (unfuelled ?p)))
```

The parameters of the operator give us the following PRSs:

`?p`		`?from`		`?to`	
pre:	$at_1, fuelled_1$	pre:	at_2	pre:	loc_1
add:	$at_1, unfuelled_1$	add:		add:	at_2
del:	$at_1, fuelled_1$	del:	at_2	del:	

A *transition rule* is derived from a PRS, and describes properties exchanged. Transition rules are of the form:

$$Enablers \Rightarrow Start \rightarrow Finish$$

where $Enablers$ is a bag of precondition properties which are not deleted, $Start$ is a bag of precondition properties which are deleted, and $Finish$ is a bag of properties which are added. Empty bags are shown as $null$, or may be omitted in the case of enablers. The transition rules corresponding to the above PRSs are:

$$fuelled_1 \Rightarrow at_1 \rightarrow at_1$$
$$at_1 \Rightarrow fuelled_1 \rightarrow unfuelled_1$$
$$at_2 \rightarrow null$$
$$loc_1 \Rightarrow null \rightarrow at_2$$

The part of the TIM processing that we discuss in the rest of the paper consists of the following main stages:

1. Construct PRSs from operators.
2. Construct transition rules from PRSs.
3. Unite the properties in the *start* and *finish* parts of transition rules, to group properties into property and attribute spaces. Property spaces arise where transition rules define a finite number of possible states (as the rules involving at_1, $fuelled_1$, and $unfuelled_1$) — these define an FSM. Attribute spaces arise where transition rules allow properties to be gained without cost (as for at_2) — these spaces must be treated in a separate way, as it is not possible to enumerate all of their possible states.
4. Project the propositions from the initial state of the planning problem to form bags of properties representing the initial states for each state space.
5. Extend the states in each space by application of transition rules.

This process derives information that can be used to extract types and invariants. Each unique pattern of membership of property and attribute spaces for the domain objects defines a type. From the generated FSMs, domain invariants can be extracted.

ADL extensions to TIM

The original version of TIM (Fox & Long 1998) deals only with untyped STRIPS domains. We are interested in extension to a subset of ADL, and our extensions fall into four areas:

- Types
- Universally quantified effects
- Conditional effects
- Negative preconditions

A generic means of transforming ADL domain descriptions into simple STRIPS representations has been given by Gazen and Knoblock (Gazen & Knoblock 1997). Although it would be possible to apply this directly as a preprocessing stage to TIM, an undesirable feature of this approach is that several stages of the processing may lead to an exponential blow-up in the size of domain descriptions or initial state descriptions.

In particular, universally quantified effects lead to an expansion in the size of the operator proportional to the number of domain objects that can match the quantified variable. Conditional effects lead to generation of multiple operators, with one operator for each possible combination of secondary conditions.

In the following work, we seek to avoid the combinatorial blow-up with a combination of simplifying assumptions, and by avoiding operator expansions which are irrelevant to the TIM processing.

TIM analysis only needs to work with structures describing what transitions are available to individual objects, and what states they can reach. The transitions generated must capture all possible transitions, and generate all possible states of individual objects. The transitions are only partial descriptions of operators, and the object states are only partial descriptions of a planning state. This representation does not preserve or make use of all the conditions present in the original operator, and could not be correctly used in a planner. Hence, some of the information which is painstakingly preserved in the Gazen-Knoblock expansion (e.g. instantiation of `forall` effects) makes no difference to the resulting TIM analysis, and can be avoided.

Now we consider the processing of each language feature in more detail. In most cases, we describe the conversion by showing a transformation on the operator and domain description before the standard TIM analysis is performed on the modified descriptions.

In general, the transformations carried out on operator descriptions will not yield operators that are correct for use in a planner. They are correct for the TIM analysis, because they yield all possible legal transitions for objects in the domain.

Types

Part of the processing performed by TIM is to derive a system of types for the domain. Clearly, if types have already been specified in the domain, these should be taken into account. Type restrictions on the parameters of an operator are transformed into additional static preconditions on the operators.

Types declared for the objects in the domain also give rise to additional propositions in the initial state. Note that since PDDL allows for a hierarchy of types, each object must have a proposition for every type of which it is a member.

The predicates which are automatically introduced to discriminate will be recognised as static conditions by TIM, and will then be taken into account when determining the derived types inferred by TIM. Hence, we can be sure that TIM types will always be at at least as discriminating as the declared types.

Conditional Effects

Treatment of conditional effects is rather more challenging, but offers far more scope for interesting results from the analysis. Consider the following example operator:

```
(:action A
 :parameters (?x)
 :precondition (p ?x)
 :effect (and (not (p ?x)) (q ?x)
              (when (r ?x) (and (not (r ?x)) (s ?x)))))
```

From this operator we would like to infer (assuming no other operators affect the situation) that parameter `?x` can make transitions $p_1 \rightarrow q_1$ and $p_1 \Rightarrow r_1 \rightarrow s_1$. This would enable us to identify two potential state spaces and consequent invariants: p and q are mutex properties and s and r are mutex properties.

In considering treatment of conditional effects, several proposals were examined and these are discussed in the following sub-sections.

Separated dependent effects One technique by which we hoped to harness the power of existing TIM analysis was to construct a collection of standard transition rules that capture the behaviour encoded in conditional effects. The previous example shows that this is possible in certain cases. The first proposal for achieving this was based assuming all `(when ... ...)` clauses in an operator to be mutually exclusive, in which case there is no problem with exponential blow-up. Under this assumption, we can generate a separate pseudo-operator for each conditional effect. The pseudo-operator has the primary preconditions and effects from the original operator, and the `when` clause is absorbed by merging its preconditions into the primary preconditions, and its effects into the primary effects. We also create an operator with only the primary effects. These operators are pseudo-operators because they do not always encode the same behaviour as the original operators and could not be used for *planning*. However, this does not prevent them from being used to generate valid transition structures that reflect the transitions described by the original conditional effects operators.

Unfortunately, conditional effects do not always satisfy the assumption that at most one conditional effect is triggered during a single application of an action. In fact, the assumption required for correct behaviour can be weakened: it is only necessary that at most one conditional effect is triggered *to affect the state of each parameter of the operator*. Even this condition is stronger than is appropriate for some domains.

An example where the assumption is violated is the following:

```
(:action op_with_non_exclusive_conditions
 :parameters (?ob)
 :precondition (a ?ob)
 :effect (and (not (a ?ob)) (x ?ob)
              (when (b ?ob)  (and (not (b ?ob)) (y ?ob)))
              (when (c ?ob)  (and (not (c ?ob)) (z ?ob)))))
```

For an object with initial properties $\{a_1, b_1, c_1\}$, this operator should allow the state $\{x_1, y_1, z_1\}$ to be reachable.

Instead, we actually generate the rules $(a_1 \rightarrow x_1)$, $(a_1, b_1 \rightarrow x_1, y_1)$ and $a_1, c_1 \rightarrow x_1, z_1$, since the operator leads to the creation of three pseudo-operators, one with the pure simple effects of deleting a and asserting x and the others each taking a precondition from their respective conditional effect and the appropriate additional effect. With these rules it is not possible to reach the state $\{x_1, y_1, z_1\}$.

Failure to correctly generate all reachable states leads to the generation of unsound invariants, so this approach cannot be used unless it is possible to guarantee the necessary conditions for valid application.

Separating conditional effects The strong assumption, that at most one of the conditional effects of an operator will apply, can be replaced with a much weaker assumption: that any number of the conditional effects of an operator could be applicable. This assumption can be characterised using the original TIM machinery by creating pseudo-operators, one with the complete collection of primary preconditions and effects from the original operator and one for each conditional effect, adding the condition for the effect to the preconditions of the original and *replacing* the effect of the original with the conditional effect. In the example above, this would lead to the following transition rules: $a_1 \rightarrow x_1$, $b_1 \rightarrow y_1$ and $c_1 \rightarrow z_1$.

With these rules it is possible for extension (the process by which TIM generates complete state spaces from the initial state properties of objects) to generate all the states we want, and more besides. For instance, we could generate a state $\{a_1, y_1, c_1\}$, by applying one of

the conditional effects without the corresponding primary effect.

The weakened assumption leads to correspondingly weaker invariants, since the opportunity to apply rules that do not correspond to actual operator applications allows apparent access to states that are not, in fact, reachable. More seriously, however, we have separated the primary and secondary effects of the operator into distinct transitions which can only be applied sequentially. This does not fit with the intended meaning of the operator, which is that all the preconditions (primary and secondary) are tested in the state before the effects take place. In the previous example this does not make any difference to the behaviour of the rules.

The circumstances under which it makes a difference are:

- When any (primary or secondary) effect deletes a secondary precondition (for a different conditional effect). This is because sequentialising the rules will cause the deleted effect to be unavailable for the application of the rule with the secondary precondition if the deleting rule is applied first. However, rules can be applied in all possible orders so this leads to a problem only when the second rule deletes a precondition of the first rule, so that applying the rules in either order prevents them from both being applied.
- When any (primary or secondary) effect deletes a (primary or secondary) effect (where not both are primary components or are in the same conditional effect). In this case, as an operator, the classical semantics (Lifschitz 1986) causes the add effects to occur after the delete effects and the apparently paradoxical effects are resolved.

A simple example of the first case is an effect of the form:

```
(when (and (q ?y) (p ?x))
      (and (q ?x) (not (p ?x))))
(when (and (p ?y) (p ?x))
      (and (r ?x) (not (p ?x))))
```

In both secondary effects, `(p ?x)` is deleted. If both `when` conditions apply, the condition is deleted once. The proposed compilation of the operators into transitions will not deal with this correctly. The rules created from these effects (supposing no primary pre- or post-conditions affect them) will be of the form: $p_1 \rightarrow q_1$ and $p_1 \rightarrow r_1$. It looks as though a p_1 property must given up to gain either q_1 or r_1, but in fact both can be purchased by giving up a single p_1.

To handle this problem we can identify the common deleted literal and collapse the rules to arrive at a third rule: $p_1 \rightarrow q_1, r_1$. Notice that we cannot replace the other rules with this one, since the conditions cannot be assumed to always apply together. More generally, we cannot assume that two rules generated from the same operator that have a common element on their left-hand-sides will always be exchanging different instances of the property (even if derived from different variables – the variables could refer to the same object). Therefore, such rules have to be combined to collapse the exchanged property into a single instance. (It is interesting to observe that the pathological case, in which the rules derive from primary effects affecting different variables, can lead to a similar problem in the original TIM analysis). There is a minor problem in that the process of collapse could, in principle, be exponentially expensive in the size of the operator, since every combination of collections of rules sharing a left-hand-side element must be used to generate a collapsed rule (in which the shared collection is collapsed into a single instance of each property). In practice the size of these sets of rules is very small and the growth is not a problem. A more significant problem is that the combinations of these rules can lead to further weakening of the possible invariants through over-generation of states.

An example of the second case is:

```
(:action A
  :parameters (?x)
  :precondition (p ?x)
  :effect (and (not (p ?x)) (q ?x)
               (when (q ?x) (and (not (q ?x)) (r ?x)))))
```

Notice that in order to delete an effect that is added by the primary effect of an operator, or another conditional effect, the deleted condition must be a precondition of the effect. The overall behaviour of `(A a)` applied to a state in which `(p a)` and `(q a)` hold is to generate a state with `(q a)` and `(r a)`. This is because the delete effect is enacted before the add effect, so that the net effect on `(q a)` is for it to be left unchanged. Application of the operator to a state in which only `(p a)` holds will yield the state in which only `(q a)` holds.

The rules generated from this operator using the proposal of this section are $p_1 \rightarrow q_1$ and $p_1 \Rightarrow q_1 \rightarrow r_1$. Testing the enablers in application of these rules allows a precise generation of states in both cases. However, it is generally not possible to consider enabling conditions without compromising correct behaviour. Suppose that the additional primary precondition `(s ?x)` and primary effect `(not (s ?x))` are added to the previous operator. Then the rules will become: $p_1, s_1 \rightarrow q_1$ and $p_1, s_1 \Rightarrow q_1 \rightarrow q_1$. It is now impossible to apply the rules sequentially to the property space state $\{p_1, s_1\}$ if we take enablers into account, because the first rule will consume the s_1 property and prevent the second rule from being applied. Consequently, this approach cannot restrict rule application using enabling conditions and we will therefore be forced to generate the unwanted states, weakening the invariants.

Conditional transitions The most radical treatment of conditional effects involves a significant extension of the TIM machinery in order to extend the expressive power of the rules in parallel with the extended expressive power of the operators. This is a less attractive option, since it requires new algorithmic treatments, but this price must be offset against the improved analysis and the more powerful invariants that

can be inferred from domain encodings.

The proposal is to extend the expressiveness of the transition rules to include *conditional transitions*. The conditional component of the transition rule is an additional transition denoted by the keyword `if`. Satisfaction of the condition depends on the presence in a state of both enablers and the start conditions.

$$\begin{array}{lll} a_1 \rightarrow & x_1 & \\ & \text{if} & b_1 \rightarrow y_1 \\ & \text{if} & c_1 \rightarrow z_1 \end{array}$$

Generating conditional transitions Firstly, we must generalise the notion of a PRS into a nested structure to represent operators including `when` conditions. We include an extra field `cnd` to record an embedded PRS for the conditional part of the operator. `op_with_non_exclusive_conditions` then yields the following PRS.

$$\left[\begin{array}{lll} pre: & a_1 & \\ add: & x_1 & \\ del: & a_1 & \\ cnd: & \left[\begin{array}{ll} pre: & b_1 \\ add: & y_1 \\ del: & b_1 \end{array}\right] & \\ & \left[\begin{array}{ll} pre: & c_1 \\ add: & z_1 \\ del: & c_1 \end{array}\right] & \end{array}\right]$$

Now we use the generalised PRSs to generate conditional transition rules. PRS analysis for secondary conditions is essentially the same as the analysis for primary conditions, except that only the adds and deletes of the secondary rule are considered, but all the preconditions of the containing structures must be included.

In general this construction is straightforward, but there is an important case that presents a minor complication. If a conditional effect deletes a primary precondition then the primary precondition will be seen as enabling the outer rule, but it will not appear as a precondition for the conditional effect. One solution is to simply handle the proposition as if it were a precondition of the conditional effect, so that the property appears as a start condition for the conditional rule. However, this leaves the enabling condition outside, apparently required as an additional property for the application of the conditional transition rule. This presents the problem that if enablers are used in determining applicability of rules it will not be clear whether the rule demands one or two copies of the property to be applied. One way to solve this problem is to *promote* the precondition, so that deleted conditions in conditional effects are always explicit preconditions of the conditional effects. To achieve this, a new conditional effect must be added to the original operator, with the deleted literals as its preconditions and all of the original primary effects as its effects. The deleted literals are now removed from the primary preconditions and added explicitly as preconditions to all the other conditional effects. This transformation yields an operator which is equivalent in its effects on a state to the original, although it can, in principle, be applied to a larger collection of states (all states in which the deleted effects are not true – the effect of application is null). Analysis of this new operator yields a conditional rule with the correct structure, distinguishing the case where a property is genuinely an enabling condition from the case where it is actually a copy of the deleted condition in a secondary effect.

Uniting transition rules In TIM the process of uniting is that of combining the collections of properties into a partition such that each transition rule refers, in its start and end components, only to properties in a single set within the partition. This ensures that the construction of extensions of states in property spaces works within a closed subset of the properties used in the domain and that the initial state properties are properly divided between the property spaces to seed the extension process.

In the case of conditional rules, the properties which change between the start and end of each conditional component of each rule must be united into the same subset of the partition, and properties within different conditional components of the same rule are also combined.

$$\begin{array}{l} uniters((Ens \Rightarrow Start \rightarrow End) + Subrules) = \\ \quad (Start - End) \cup (End - Start) \cup \\ \quad \bigcup_{Sr \in Subrules} uniters(Sr) \end{array}$$

This form of uniting ensures that the property spaces remain as small as possible, which improves the quality of the invariants that can be generated and also the efficiency of the analysis. It does, however, impose additional difficulties in the use of rules, since the same rule can now affect the behaviour of objects in multiple spaces (conditional elements might refer to properties in entirely different spaces to the primary effects of the rule). Each rule must be added to every space that it applies to and considered during the extension of each of those spaces separately.

A further change from the process of setting the initial collection of property spaces in STRIPS TIM is that conditional rules can appear to contain attribute rules when, in fact, they are half of a transition rule that is completed by a second "attribute" rule in a conditional effect. For example:

```
(:action B
  :parameters (?x)
  :precondition (p ?x)
  :effect (and (not (p ?x))
               (when (s ?x) (q ?x))
               (when (not (s ?x)) (r ?x))))
```

This operator leads to the rule:

$$\begin{array}{lll} p_1 \rightarrow & \text{null} & \\ & \text{if} & s_1 \Rightarrow \text{null} \rightarrow q_1 \\ & \text{if} & \neg s_1 \Rightarrow \text{null} \rightarrow r_1 \end{array}$$

which might suggest that p_1, q_1 and r_1 should all be considered to be attributes and, consequently, have no useful invariant behaviours. However, it would be better to observe that the behaviour of these properties is actually equivalent to a pair of transitions: $s_1 \Rightarrow p_1 \rightarrow q_1$ and $\neg s_1 \Rightarrow p_1 \rightarrow r_1$ (exploiting negative preconditions, discussed below). Although it might be possible to convert the rules automatically, it becomes much harder to do this in the context of multiple rules referring to other parameters. It is actually easier to manage the rules during extension. The important thing, during initial property space construction, is to avoid labelling properties as attributes on the basis of the structure of *conditional* rules. A decision about which properties to label as attributes must be postponed to the extension phase.

The fact that the properties in a property space can be distributed between primary and conditional rules creates a complication for uniting: it is not enough to put together properties in the same rule. Properties must be combined when they appear in "attribute" rules such as the previous example. It will be noted, however, that the example relies on the form of the conditions of the conditional effects and this feature is further discussed later in the paper.

Extending the state spaces with the conditional transitions Extension is the stage most impacted by the introduction of conditional rules. Conditional rules must be applied to each state in the property space containing them in order to generate a set of reachable states. Thus, the key to exploiting these rules is to understand how to apply the conditional rules to a state.

When a standard transition rule is applied the start conditions are removed from the state and the end conditions added to the result to yield the (single) new state. Conditional rules are applied by removing the primary start properties and then, for each conditional rule, continuing expansion under the assumption that the condition applies and under the assumption that it does not apply. Therefore, there will be 2^n new states generated for n conditional effects (subject to repetition of previously visited states). Although this is potentially exponentially expensive, n is typically a very small value, so that the cost is not a problem. Nevertheless, conditional effects represent a potential source of considerable cost in the TIM analysis (just as they can in planning itself). The hope is that we will have saved significant cost by deferring this combinatorial aspect to the latest possible time, and some redundant processing has been avoided.

There are, in fact, several special cases that can be used to reduce the number of combinations of conditional rule elements that must be considered. Conditional rules can only be applied if the start properties are present in the state to which they are applied. Conditions that are restricted to propositions concerning only the variable that is affected by the transition can usually be restricted by the enabling or start conditions of the rule. Further, the observations of the next section provide for an important collection of situations.

Having allowed possible attribute conditional rules to be entered into property spaces it is extremely important that the extension process monitors the possible existence of increasing attributes in a property space. This possibility also exists in the original TIM analysis and is handled by checking to see whether newly generated states are proper super-states of previously generated states. In such cases, if there is path of transition rules leading from the sub-state to the super-state, the difference between the states represents a collection of attribute properties and they must be stripped from the property space and its rules before extension can be continued. This process might iterate several times before the space settles on a fixed collection of properties. If this collection is actually empty then the properties will, in fact, all be attributes due to the effects of the conditional rules in the space.

Hidden exclusivity In some operators, conditional effects are mutually exclusive because it is not possible for their preconditions to hold simultaneously. In such cases the extension process described above will generate permissive property spaces and weaker invariants. In order to improve the generation process it is necessary to avoid allowing rules to be applied simultaneously when their conditions will prevent it. Further improvement can be made by observing that in many cases the conditional effects are created to ensure that precisely one effect of a collection will be triggered. For example:

```
(:action op1
    :parameters (?y)
    :precondition (a ?y)
    :effect (and
              (not (a ?y))
              (b ?y)))

(:action op2
    :parameters (?y)
    :precondition (b ?y)
    :effect (and
              (not (b ?y))
              (a ?y)))

(:action op3
    :parameters (?x ?y)
    :precondition (p ?x ?y)
    :effect (and
              (when (a ?y) (q ?x))
              (when (b ?y) (r ?x))
              (not (p ?x ?y)) ))
```

In this example, properties a_1 and b_1 may only be exchanged for each other, as illustrated in Figure 1. If, in the initial state, objects only have at most one of the two properties, then the two `when` conditions in `op3` are mutually exclusive.

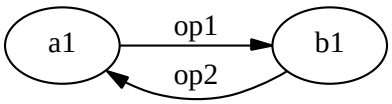


Figure 1: Property space and transitions for properties a_1 and b_1.

The generated transitions would be: $a_1 \rightarrow b_1$, $b_1 \rightarrow a_1$,

$$\begin{array}{ll} p_1 \rightarrow & \text{null} \\ & \text{if null} \rightarrow q_1 \\ & \text{if null} \rightarrow r_1 \end{array}$$

and $p_2 \rightarrow$ null. Failure to observe that the conditional effects are governed by preconditions of which precisely one must be true will mean that no invariants can be generated using the properties p_1, q_1 and r_1. In order to discover these we need two pieces of information to be available during extension: the fact that each conditional effect is governed by a property that, in this case, apply to another parameter and form part of another property space and the fact that the properties governing these effects are the two alternative states in that space. We therefore mark conditional rules with all of the properties that govern their application. Enabling conditions that are properties of other parameters are called *aliens*. We enclose them in square brackets to distinguish them from enablers applying to the parameter governed by the transition rule.

$$\begin{array}{ll} p_1 \rightarrow & \text{null} \\ & \text{if} \quad [a_1] \Rightarrow \text{null} \rightarrow q_1 \\ & \text{if} \quad [b_1] \Rightarrow \text{null} \rightarrow r_1 \end{array}$$

This provides the first piece of information. The second piece is derived from an analysis of the property space containing properties a_1 and b_1. It is important that the analysis makes the latter space available before the use of the rule that depends upon it. Therefore, we perform a dependency analysis to order the property spaces for appropriate expansion order. Where one state space depends on another, an order can be imposed between them that would allow information from the first to be used in the second. Space z depends on space y if transitions belonging to space z have enabling conditions which belong to space y.

Circular dependency amongst state spaces must be handled carefully. One way is to break the cycle arbitrarily and then follow the dependencies that remain. The first space in the chain will then be expanded with the more conservative assumption that no invariants affect the conditions governing the application of conditional effects, possible leading to weakened invariants. Because these weakened invariants can propagate their impact up the chain, it would obviously be best to break the chain in such a way as to minimize the impact that these weakened invariants might have. An alternative solution to the problem of cyclic dependency is to carry out extension on these property spaces in an interleaved computation, leading to a fixed-point. The approach is to apply conditional rules relative to the dependencies on property spaces using the states that have been generated *so far*. The extension process must then iterate around the cycle of interdependent spaces, adding new states as new enabling states are added to the spaces on which later spaces depend. This iterative computation will have to be restarted if any of the properties in a space is identified as an attribute, as described previously.

Managing combinations of subrules In this section we give more detail about how to use information from spaces for which the extension process is already completed, together with annotation of the subrules with alien enablers, to restrict the combinations in which subrules of a conditional rule can fire.

For a subrule to fire, we require that for each variable-space combination occurring as an alien enabler in the subrule, there is at least one state that satisfies the enabling property.

The coupling between subrules is captured because valid combinations of subrule firings are those in which the enabling conditions can be simultaneously satisfied. Enabling conditions are satisfied, if, for each variable, and each space in which it occurs, there is a non-empty set of satisfying states.

The approach adopted depends on amending the TIM algorithm with the following steps

1. During rule construction, each subrule is annotated with alien enablers, each consisting of a triple $aliens(subrule)$ is a set of $\langle property, space, var \rangle$, where var is the variable from which the enabler was generated in the planning operator.
2. A graph of dependencies between spaces is constructed from the alien enablers.
3. Spaces are constructed in order, according to topological sort of the dependency graph. Cycles must be broken as discussed above.

Now we define the valid combinations of rule firings by describing a set of variables, and constraints which must hold between them.

For each variable-space combination in alien enablers, we have a variable $state(variable, space)$ whose domain ranges over possible states of $variable$ in $space$.

$$state(var, sp) \in states(sp)$$

For each property-variable-space combination in alien enablers, we have a boolean constraint variable, $sat(property, variable, space)$, whose value indicates whether the property is satisfied.

$$sat(prop, var, sp) \in \{0, 1\}$$

For each subrule, we have a variable $f(subrule)$, whose domain is {0,1}, which records the whether or

not the subrule can fire. Note that since the possible values remaining in the domain are attached to the variable, we can describe never {0}, sometimes {0,1}, and always {1}.

Now we attach constraints between these variables as follows:

Between the $sat(property, variable, space)$ variables, and the $state(variable, space)$, we require that the property is satisfied iff var is in a compatible state in $space$.

$\forall subrule \in subrules$
$\quad \forall \langle prop, var, sp \rangle \in aliens(subrule).$
$\quad\quad sat(prop, var, sp) \Leftrightarrow has\_prop(state(var, sp), prop)$

Between the $f(subrule)$ variables and the $sat(property, variable, space)$ variables of its alien enablers, we require that $subrule$ fires iff the conjunction of its alien enablers is satisfied.

$$f(subrule) \Leftrightarrow \bigwedge_{\langle prop,var,sp\rangle \in aliens(subrule)} sat(prop, var, sp)$$

These constraints are used to determine which combinations of subrules may fire together.

Universally Quantified Effects

Since we are concerned with the transitions made by individual objects, it is not necessary to expand the operator in advance with every instantiation of the quantified variable, as is done by (Gazen & Knoblock 1997).

```
(:action one_forall
   :parameters (?a - t1)
   :precondition (p ?a)
   :effect (and (not (p ?a))
                (q ?a)
                (forall (?b - t2)
                   (when (r ?a ?b)
                         (and (not (r ?a ?b)) (s ?a ?b) )))))
```

The effect inside the quantifier may occur as many times as there are instantiations for the quantified variable `?b`.

For `?b` itself, we can generate a transition rule exactly as if it was an ordinary parameter of the operator. The number of objects making the transition is not relevant, as any object belonging to this state space experiences only a single transition. In the above example, analysis for the variable `?b` gives the transition $r_2 \rightarrow s_2$.

Now consider the transitions for the variable `?a`. Outside the quantifier, `?a` undergoes a single transition $p_1 \rightarrow q_1$. Inside the quantifier, `?a` undergoes a transition $r_1 \rightarrow s_1$, but the number of times the transition may occur depends on the number of instantiations the quantified variable can take. This reasoning applies to any variable occurring inside the scope of the quantifier, which occurs in an effect together with the quantified variable.

We must again resort to a new notation to describe the resulting transition rules. We use * to indicate that the transition inside the quantifier may be performed an unknown number of times.

For the variable `?a`, we have:

$$\begin{aligned} p_1 \rightarrow \quad & q_1 \\ & (r_1 \rightarrow s_1)^* \end{aligned}$$

In the extension process, the interpretation of the starred rule is that the inner transition may occur any number of times.

Observe that in this example, the transition inside the quantifier is an exchange of properties. It may also occur that the transition inside the quantifier involves properties simply being gained or lost. Such properties are considered to be attributes in the TIM system, and are processed in a separate way. The presence of such rules otherwise leads to non-termination of the extension process, as attributes may be added without limit. Attribute rules are less useful for discovering invariants, and it is unfortunate if properties are considered as attributes unnecessarily. This is harmful to the TIM analysis because any property which becomes an attribute also makes anything for which it can be exchanged into an attribute.

It is our contention that properties will not be made into attributes unnecessarily as a result of analysing the quantified effect. Where properties are exchanged, the whole exchange will normally take place inside the quantifier, as in the example above.

An example where a quantified effect correctly gives rise to attribute rules can be seen in the briefcase `move` operator given above. There the at_2 properties are attributes — the portables at a location may be gained and lost without exchange.

An awkward example can be found in the Schedule domain, in which predicates of the form `(painted ?object ?paint)` are typical. All operators in the domain which mention `painted` include a deletion which is universally quantified over possible paint colours, as in the example operator shown below. Some of the operators also add a single `painted` effect. Hence all operators which touch an object's $painted_1$ property result in a state with either 0 or 1 instances of the property for that object. If this holds also in the initial state, it is an invariant.

```
(:action do-immersion-paint
    :parameters (?x ?newpaint)
    :precondition (and
                    (part ?x)
                    (has-paint immersion-painter ?newpaint)
                    (not (busy immersion-painter))
                    (not (scheduled ?x)))
    :effect (and
                (busy immersion-painter)
                (scheduled ?x)
                (painted ?x ?newpaint)
                (when (not (objscheduled))
                      (objscheduled))
                (forall (?oldpaint)
                    (when (painted ?x ?oldpaint)
                          (not (painted ?x ?oldpaint))))))
```

It is problematic that the quantified effect makes it appear that $painted_1$ is a decreasing attribute. In earlier versions of TIM, the appearance of decreasing attribute transitions always led to the creation of a separate attribute space. However, if the attribute may only decrease, it can lead to only a finite number of states, so it is safe to have rules of this kind in a property space. It is important to take this approach, as invariants will otherwise be lost.

Negative preconditions

Our treatment of negative preconditions is currently the most restrictive described here. We use a similar technique to (Gazen & Knoblock 1997): for each predicate that may appear as a negative precondition, we create a new predicate to represent the negative version of that precondition. This predicate must exactly complement the positive use of the predicate and it replaces, in preconditions, the use of the negative literal.

For those predicates which occur anywhere as a negative precondition, we must do the following to the effects of every operator:

- Wherever the positive version of the predicate appears as a delete effect, the negative version must appear as an add-effect.
- Wherever the positive version of the predicate appears as an add effect, the negative version must appear as a deleted effect.

We complete the initial state of the problem description as follows: For each predicate and each object that may instantiate the predicate, if there is no positively-occurring fact, we add the negative version of the fact.

In the case of negatively-occurring predicates with multiple arguments, the completion of initial state is very expensive, and would, in any case, lead to a much weaker domain analysis. For our processing we impose the restriction that only predicates of a single argument are handled. This also allows the transformation to be performed, not on the operator itself, but to be processed at the level of PRSs.

We believe that predicates with more than a single argument could be handled efficiently using a representation in which properties which properties are counted, but this remains an area of further investigation.

Results

A prototype system has been implemented which successfully produces the expected invariants for all the cases considered in this paper, except those relying on a full treatment of universal quantifiers. Work on the handling of universal quantifiers is currently in progress. Additional computational overheads for handling the conditional effects are negligible.

In the example of hidden exclusivity between conditional transitions, the system successfully detects that exactly one of the subrules may fire, and thus extension does not over-generate states or unnecessarily weaken the analysis by creating an attribute space. The resulting space for the properties p_1, q_1 and r_1 is shown in Fig. 2.

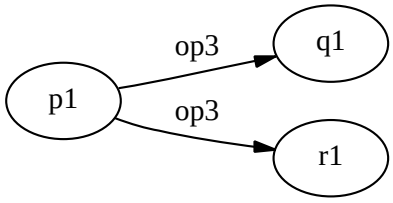


Figure 2: Property space and transitions for properties p_1, q_1 and r_1.

Directly from the state space, we get that states in this space allow exactly one of the properties $\{p_1, q_1, r_1\}$, and from this the following invariants are generated:

```
FORALL x:T1. (Exists y1:T0. p(x,y1) OR r(x) OR q(x))
FORALL x:T1. NOT (Exists y1:T0. p(x,y1) AND r(x))
FORALL x:T1. NOT (Exists y1:T0. p(x,y1) AND q(x))
FORALL x:T1. NOT (r(x) AND q(x))
```

Related work

Both the system of Rintanen (Rintanen 2000) and Gerevini and Schubert's DISCOPLAN (Gerevini & Schubert 1998) rely on generating an initial set of candidate domain invariants. These are then tested against the operators to see if they hold after the application of the operator. Candidate invariants which are not preserved are either discarded or strengthened by adding extra conditions and tested further. In both systems, discovered invariants can be fed back, to help discover further conditions.

The current version of DISCOPLAN (Gerevini & Schubert 2000) has been extended to handle conditional effects, types and negative preconditions, but does not handle universal quantification.

An interesting question is whether DISCOPLAN can detect and exploit the occurrence of mutually exclusive secondary conditions. Our tests with the current version of DISCOPLAN indicate that it cannot. The account in (Gerevini & Schubert 2000) discusses the feeding back of discovered invariants into the process, by using this information to expand operator definitions. It appears that currently, only the implicative constraints are fed back in this way. To correctly resolve this problem would require XOR constraints to be fed back into the processing of secondary conditions.

Conclusion

In this paper we have explored the extension of TIM to handle a subset of ADL features. The features we have not considered are those that allow fuller expressiveness in the expression of preconditions: quantified variables and arbitrary logical connectives. There are significant difficulties in attempting to handle these features, since

the opportunity to identify the specific properties associated with particular objects is obscured by the possibility for disjunctive preconditions combining properties of different objects, universally quantified variables that allow reference to arbitrary objects, and nested expressions that can involve exponential expansion if conversion to disjunctive normal form is used. We have also not considered a full treatment of equality and inequality propositions, nor of arbitrary negated literals.

The features that have been considered, and for which there are extensions that allow TIM to extract invariants, include conditional effects, quantified effects, types and a restricted form of negative preconditions. These features form the core of those used in existing ADL domain encodings, with the exception of the ADL Miconics-10 domain used in the 2nd International Planning Competition in 2000. The most difficult feature to handle is the use of conditional effects and we have shown that there are a variety of possible approaches, each with advantages and disadvantages. The most powerful approach is the extension of the underlying TIM machinery, rather than an attempt to preprocess conditional effects out of the operators in order to reuse the STRIPS TIM machinery. This extended machinery complicates the entire sequence of analysis phases conducted by TIM and we have described the effects that are implied for each stage in turn.

The next stages of this work include completion of a full implementation of all of the features described in this paper, a further exploration of the treatment of negative preconditions and experimentation with the system on existing ADL domains.

Acknowledgements

The work was funded by EPSRC grant no. R090459.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116.

Doherty, P., and Kvarnstrom, J. 1999. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *Proceedings of the 6th Int'l Workshop on the Temporal Representation and Reasoning, Orlando, Fl. (TIME'99).*

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 2000. Utilizing automatically inferred invariants in graph construction and search. In *International AI Planning Systems conference, AIPS 2000, Breckenridge, Colorado, USA.*

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *ECP*, 221–233.

Gerevini, A., and Schubert, L. K. 1998. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, 905–912.

Gerevini, A., and Schubert, L. 2000. Extending the types of state constraints discovered by DISCOPLAN. In *Proceedings of the Workshop at AIPS on Analyzing and Exploiting Domain Knowledge for Efficient Planning, 2000.*

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning-as-satisfiability framework. In *Proceedings of the 4th International Conference on AI Planning Systems.*

Lifschitz, E. 1986. On the semantics of STRIPS. In *Proceedings of 1986 Workshop: Reasoning about Actions and Plans.*

Long, D., and Fox, M. 2000. Recognizing and exploiting generic types in planning domains. In *International AI Planning Systems conference, AIPS 2000, Breckenridge, Colorado, USA.*

Pednault, E. 1989. ADL: Exploring the middle ground between strips and the situation calculus.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of European Conference on Planning.*

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, 806–811.

van Beek, P., and Chen, X. 1999. Cplan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence, Orlando, Florida.*

Reuse of Control Knowledge in Planning Domains

Luke Murray
Department of Computer Science
University of Durham, Durham, UK
l.c.j.murray@dur.ac.uk

Abstract

In recent years, the power of domain−independent planners which use hand coded domain−specific control knowledge has been demonstrated [AIP00]. This approach, though fruitful in terms of planner performance, has several issues asociated with it. Firstly, control rules need to be hand coded for each domain. This affords no scope for reuse, and the control rules formulated rely on the control rule writer's ability to recognise and exploit structure in the domain. Secondly, in comparisons between systems, it is often unclear to what extent off−line control rule construction governs the planners overall performance. This paper presents a new approach in automatically instantiating domain specific control rules from templates, with the use of existing domain analysis techniques. Using Generic Type information concerning the domain, template rules are instantiated from a library of *Generic Control Rules*. It is hoped that these instantiated rules may offer some of the benefits of hand−coded domain specific knowledge, but without many of the drawbacks.

1 Domain Analysis

Domain analysis is used in planning ultimately as a means of reducing the search that is essentially the burden of the planner. The concept behind domain analysis is that by analysing the domain in some way, it is often possible to recognise unfruitful paths in the search space (with respect to a problem instance) without having to actually traverse it. Typically, domain analysis is in the form of pre−processing before the planner is invoked, and the information unearthed by the domain analysis tool is passed to the planner along with the domain and problem specifications.

The field of domain analysis has developed to a point at which it is able to provide planners with important and valuable information relating to the structure of certain domains. Existing domain analysis techniques are capable of discovering a variety of different forms of information, e.g. State Invariants (conditions which are necessarily true of every state reachable from the initial state), Types (classification of domain objects according to the states they can be in and the operators that they can be changed by), etc.; State Invariants are used to prune any branch of the search space that violates them (as these are not reachable states) while Type analysis can can be used to disregard branches that violate type restrictions (e.g. those states reached by applying an operator to badly typed arguments).

Automatic domain analysis has been seen to recognise structure within a domain that humans have overlooked. A notable example of this is the Paintwall domain [FL00], in which the walls can be seen as traversing a map of locations represented by different coloured paints. In the 1998 AI Planning Systems competition, the Mystery domain (an encoding of the Logistics domain, with alternative object and predicate

names [MD00]) was 'found out' by a domain analysis tool, which allowed its planning system to invoke appropriate transportation heuristics.

2 Generic Types

Notably, the work of Fox and Long (which covers Type Analysis, Symmetry Analysis and Generic Type Analysis) in the field of domain analysis has provided the international planning community with the notion of Generic Types. These are higher−order types, populated by types identified in a given domain, and are identified by an extension of the type analysis machinery [FL98], [FL00]. Type classification is carried out by looking at the planning domain as a set of finite state machines (FSMs) whose states are representative of the individual argument positions in each predicate, and whose transition rules are governed by the operators described for the domain. Those domain objects that can traverse the same FSMs are grouped together into a type. Generic Type analysis takes this further and recognises templates or topologies of FSMs associated with a particular type and provide us with information relating to the shared behaviours and properties of types in totally distinct domains.

Each particular Generic Type has features which play specific roles in its behaviour. These features can be described at either the Generic Type level or at the level of the instance of the Generic Type. For example, one established Generic Type is the Safe Portable Object Type (SPOT). The distinguishing feature of a type identified as a SPOT is that its members can be transported between locations (according to some map) but never have any other role in a plan (commonly they have a specified goal location). Safe Portable Objects (SPOs) are distinguished from other Portable Objects precisely because it is safe to transport them, without affecting other processes in the domain. A SPO (like Portable Objects in general) changes location by being transported by a Carrier, which can pickup and deposit the object at any of the locations on its map.

It is possible to talk about the *locatedness* predicate of a SPO, meaning the predicate (relationship) which relates the SPO to the location at which it is situated. For instances of a SPO, such as packages in the Logistics domain, we can say that the locatedness predicate (or at−relation) is the *at* predicate. In the case of the Gripper domain, where the balls are identified as SPOs, *in−room* is the appropriate predicate. The other features that all members of a SPOT posess are a *contained_in* predicate (to show they are being carried by a Carrier) and the ability to have *load* and *unload* operations performed on them (to be loaded onto or deposited from a Carrier). Through these relations, is possible to talk about either the location or the carrier object to which the SPO is related in any given state. However, it is important to remember that a group of objects is only identified a SPOT if it meets the requirements; that those objects form a Type and that members of that type have no other role in the plan than to be transported between locations.

3 The Language of Control Rules

Control rules can be supplied with the domain description and problem instance and generally give planners heuristics for manipulating objects in the domain more efficiently. They can be explicitly goal−directed (of the form "if *P* is in the goal then do *Q")*, but need not be. They can simply offer efficient ways to achieve some desired state from some known state.

Historically, planners have generally had their own languages for the purposes of inputting domain specific knowledge. This is evident in the recent landmarks of TLPlan [BK00] and TALPlanner [DK99], both of which had knowledge expressed in temporal logics. The following is an example of a control rule for the Logistics domain, and captures the fact that any of the packages in the Logistics domain, once at their goal location, should remain at that location:

(1)

□ ∀ X:{package1,package2}. ∀ Y: {bos−po, pgh−po, bos−airport, pgh−airport} . at(X,Y) /\ Goal(at(X,Y)) => O(at(X,Y))

A proposal has been made to provide a standard environment for the exchange of domain knowledge in the form of DKEL [SH00]. However, these languages do not readily provide support for the proposed Generic Control Rules (i.e. control rules expressed in terms of Generic Types). A temporal logic will be proposed for expressing these control rules.

The logic proposed will incorporate the modal temporal operators O (Next), in order to be able to refer to progressions of states, □ (Always) to refer to all states and Goal to refer to the goal state. The language will be used at three distinct levels: at the highest level to express Generic Control Rules as in the library, at an intermediate level to enable the output of domain specific logical formulae (resembling existing control rule logics) and at a low level to provide object−specific queries for use with planners not necessarily capable of using general control knowledge.

4 Generic Types and Control Rules

Where several different types have the same structure of behaviour (i.e. equivalent states with equivalent state changing operators), as in Generic Types, it allows us to abstract any heuristics relevant to any particular instance to the abstracted level. This also means that heuristics can be formulated in terms of the abstraction, and then interpreted to apply to the all of the instantiations of those abstractions. It follows then that performance−enhancing heuristics can be expressed in terms of the Generic Type, as opposed to in terms of instances of that Generic Type. This abstraction allows the information to be applied in any instance of the Generic Type that is identified.

This aim has, to some extent, been achieved and was originally done in an integrated way, in the domain−analysis/planner partnership of TIM/STAN [FL98], in which hard−coded heuristics were triggered by Generic Type identification. There was no temporal aspect in the hard−coded control information, though, and no formalism was presented for the heuristics expressed. Also, the integrated approach does not allow the user easy access to the heuristics themselves, which can be embedded in the implementation. This makes it awkward to add to the heuristics, or indeed to add to the Generic Types that once identified, trigger those heuristics.

The integrated approach also means that the information discovered by the domain analysis tool tends to be in a format tailored for its partner system (the planner). The introduction of an API to TIM has begun providing access to parts of the domain anaylsis, of which Generic Type analysis is only a part. The work described in this paper aims to instantiate domain specific control knowledge (from reusable hand coded templates) using Generic Type information generated by existing domain analysis techniques.

5 Generic Control Rules

As outlined earlier, it is proposed that control rules be written in terms of Generic Types. The following is an example of a Generic Control Rule:

(2)

□ ∀T: SPOT. ∀X: T. (location_of$_T$ X) == (Goal (location_of$_T$ X)) => (O (location_of$_T$ X)) == (location_of$_T$ X)

This control rule expresses the heuristic that all members of a Safe Portable Object Type should remain at their goal location upon getting there. The antecedent ensures that the location of the object *X* in the current state is the same as it is in the goal state (the state qualification, to check that the rule is applicable). The consequent

expresses that in the next state, the location of *X* is the same as in the current state (i.e the direction for *X* to stay where it is). The instantiation of the Generic Control Rule into a domain specific control rule would involve the specialisation of (2).

The *location_of* function is specific to the type *T*. This is important as every domain type *T* may have its own at−relation, argument positions within that relation and types associated with those argument positions. *(location_of$_T$ X)* should be able to return the argument that *X* is located at, expressed as *X* and its location in the at−relation specific to *T*. So the first step in instantiating (2) is to identify the appropriate at−relation (specific to *T*), with the types and positions of its arguments (this process becomes more complicated for relations with arities greater than two). This enables us to create a proposition from *(location_of$_T$ X)*, and in the Logistics domain would look like:

(3)

$at(x{:}T_1, y{:}T_2)$

where T_1 is the domain type identified as a SPOT, T_2 is the domain type identified as the SPOT's locations and *at* is the locatedness predicate. We can then re−write the rule in (2) using the substitution of (3) in place of *(location_of$_T$ X)*, as:

(4)

$\Box\ \forall X{:}\ T_1.\ \forall Y{:}\ T_2.\ (at(X,Y) \wedge Goal(\ at(X,Y)) \Rightarrow (O\ at(X,Y)) \wedge at(X,Y))$

Notice that when the equality is evaluated, it is done with respect to a state (expressions without a temporal operator have an implicit 'Now' state argument,
indicating the current state). Now that we are dealing with propositions (see (3)) not values, we want to express that those propositions are true (in whatever their specified state), as opposed to equal (which wouldn't make sense). As a result, the equalities expressed in (2) become conjunctions as in (4), where the values that were bound by the equality are expressed as instances of the same variable. The introduction of the second argument in the at−relation (*Y*) requires the second quantification, but we have available to us the type information to do this (we want quantification over a type). Finally, the lemma

(5)

$A \wedge B \Rightarrow A \wedge C \equiv A \wedge B \Rightarrow C$

can be used to reduce (4) further to:

(6)

$\Box\ \forall X{:}\ T_1.\ \forall Y{:}\ T_2.\ (at(X,Y) \wedge Goal(\ at(X,Y)) \Rightarrow (O\ at(X,Y)))$

Individual object queries could be posted at this lower level, for planners not capable of using temporal control knowledge. This structure could be queried with domain objects *A: T_1* and *B: T_2*. If the antecedent was satisfied then the planner would know that the consequent should hold (in the plan, if the heuristic was adhered to).

Of course not all the domain specific control rules can be expressed in this manner. Rules which don't exploit types recognised as Generic Types cannot be expressed, possibly because there is no Generic Type currently identified for that particular structure or possibly because that particular rule exploits something other than Generic behaviour. However, as time progresses, it is expected that more and more Generic Types will be identified, encompassing increasing numbers of behaviours. This will allow increasing numbers of control rules to be identified for domains exhibiting the appropriate structure. Caution must be excercised, though, as the overhead cost will increase with the amount of work done by the domains analysis tool. This will have to be weighed against the benefits that the tool affords, but it is is feasible that the cost of rule instantiation for large numbers of rules would outweigh the time benefits in plan construction. Once there are large numbers of control rules being instantiated, there may also be issues regarding the precedence or priority of control rules. These points will need to be looked into as the work progresses.

6 Generic Control Rule Logics

The logic with which Generic Control Rules are expressed largely inherits its syntax and semantics from standard modal logics. However, there remain some points to note. First let there be a distinction made between the logic used to express the rules in their generic form, which we shall refer to as GCRLogic, and the logic used to express the rules in their instantiated form, which we shall refer to as DCRLogic.

DCRLogic is very much a standard modal logic, whose terms are propositions and whose modal operators are Next, Always and Goal. Universal quantification is allowed over domain types (sets of domain objects). The truth value of a proposition is determined by whether that proposition holds in the current (or otherwise specified) state.

GCRLogic, on the other hand, has a few more subtleties associated with it. Firstly, the 'of type' operator (':') is overloaded. It can be used on two levels; to denote an object variable's membership in a type and also to denote a type variable's membership in a type of types. Universal quantification over these 'meta−types' allows us to generalise over types that share behaviour, i.e. generalise over types that can be identified as being of a common Generic Type. In this sense, a Generic Type can be thought of as a higher−order type, populated by all domain types that can be shown to exhibit the appropriate behaviour.

In GCRLogic, the terms are the truth values of equalities between objects. The objects can be referred to through variables (introduced through quantification) or by function application to a variable. The application of a function to an object variable is not evaluated at the level of the GCRLogic. Instead, it is used to refer to an object that is related, through the relation expressed by the function, to the variable to which the function is applied. For example, in (2), the term

(7)

$(\text{location\_of}_T\ X)$

refers to the object related to X through X's location relation. We also want to be able to refer to objects that have a particular relationship with the variable in special states, i.e. in the goal state, the next state or in every state (remembering that a term unadorned with a modal operator has an implicit 'Now' state argument). However, as in standard modal logics, in GCRLogic modal operators can only be applied to sentences, not objects. So in order to refer to, for instance, X's location (as in (7)) in the Goal state, formally we must state

(8)

$\text{Goal}\ ((\text{location\_of}_T\ X) == N)$

However, it must be noted that a short hand representation is in use for ease of reading GCRLogic statements. We allow the use of

(9)

$\text{Goal}\ (\text{location\_of}_T\ X) == O\ (\text{location\_of}_T\ X)$

in order to represent the formally correct

(10)

$(\text{Goal}\ (\text{location\_of}_T\ X) == N) \wedge (O\ (\text{location\_of}_T\ (X) == N))$

NB this short form is applicable for all modal operators.

7 Control Rule Library

Generic Control Rules will be collected into a library, which the domain analysis tool will have access to. As a result, when domain analysis discovers Generic Types in a given domain, it can retrieve the relevant rules from the library and instantiate them with the specifics of the domain (see Figure 1 for proposed architecture). It is envisaged that this library will grow over time (as new Generic Types are described or additional rules are added for existing Generic Types), allowing the domain analysis tool to instantiate more control rules and ultimately improve the performance of the planner making use of the domain analysis. It is conceivable that

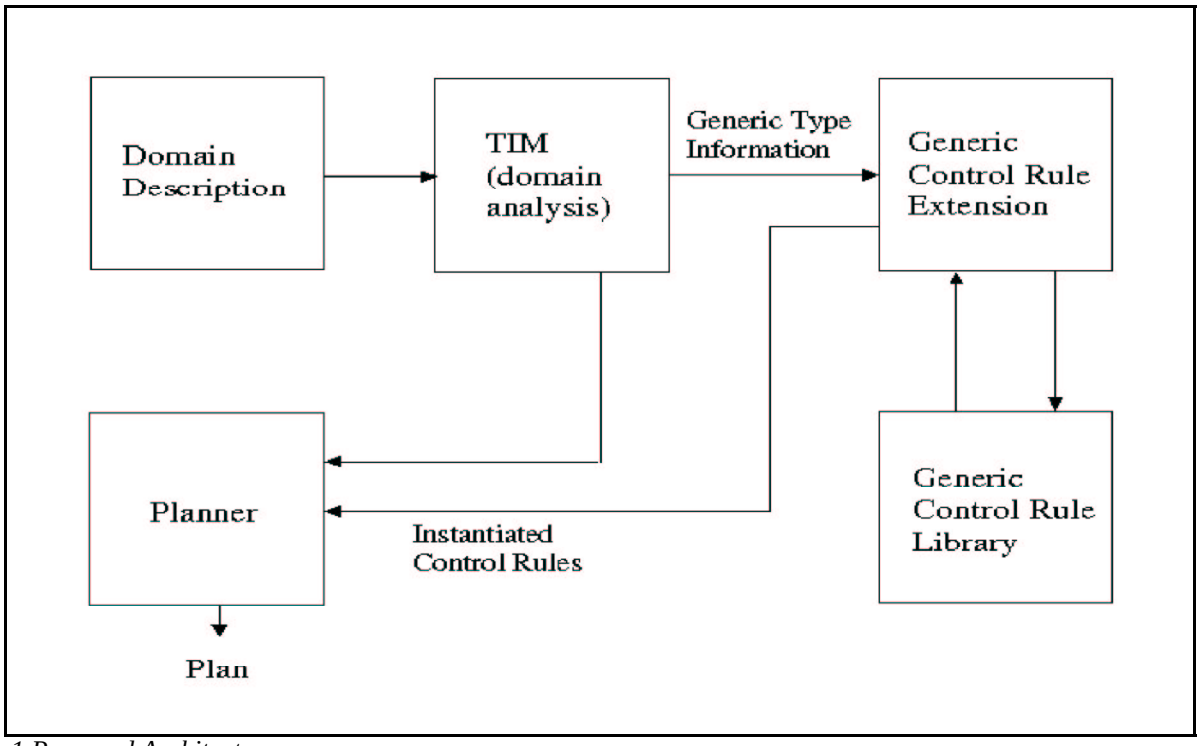


Figure 1 Proposed Architecture

future work could involve the automatic generation of Generic Control Rules. However, static domain analysis techniques are not powerful enough to achieve this; this may perhaps be attainable through an adaptive tool (one able to learn through many examples), but it is not proposed here.

The library of control rules will not remove the need for control rules to be written, rather it will allow control rule writers to write reusable control rules. However, a point worth noting is that this does involve some standardisation. Writing the control rules in terms of Generic Types forces the individual Generic Types to be standardised. The parts of the Generic Types must have standard names in order for the Generic Control Rules to be meaningful. For example, the Generic Type *mobile* has a *locatedness* predicate, representative of the fact that mobiles are always located somewhere on a map of locations. This predicate, and its contents (i.e. the name of the location involved) must be accessable in a standard way. A possible solution is that each Generic Type is supplied with a prototype, giving the names of all its fields with some description for the benefit of those using the Generic Type. The logic for expressing the control rules must also adhere to a common standard, in order for the library to be portable.

8 Conclusion

Planners which take domain specific control knowledge in addition to the domain and problem descriptions have been shown to perform well over recent years. Unfortunately though, the control rules have only yet been expressed as domain specific, with no possibilty of reuse. This means that every domain (and to some extent, every different encoding of a domain) needs a new set of rules to be formulated, and then entered, by a human control rule writer. The control rules written rely on the writer's ability to recognise and exploit structure in the domain and this is not always an easy task as the names of objects and relationships in the domain can be misleading (as in the Mystery domain [MD00]).

Work has been started on imlementing the

A First Approach to Tackling Planning Problems with Neural Networks

S. Fernández, I.M. Galván, R. Aler
Universidad Carlos III de Madrid
Avenida de la Universidad, 30
28911 Leganés (Madrid), España
sfarregu@inf.uc3m.es, igalvan@inf.uc3m.es, aler@inf.uc3m.es

Abstract

Many different machine learning techniques have been used to learn control knowledge for planning. However, subsymbolic techniques have not been widely used, in particular artificial neural networks. In this paper, a feedforward neural network (FNN) will be used to learn a heuristic function for improving planning efficiency. The main aim of this paper is to present a preliminary study about the issues and the ability to use FNN in this context.

Introduction

It is well known that domain independent planning, which is most usually based on search, is not efficient. In order to make planning efficient, knowledge about the domain must be injected into the system. A popular approach in the machine learning field is to supplement domain independent planners with control knowledge learned automatically. A large amount of work has been done in this area. Please, see (Zimmerman & Kambhampati 2001) for a very good summary of most of the relevant work. However, not all machine learning techniques are equally well represented.

The main aim of this paper is to explore new learning mechanisms and control knowledge representations which have not been widely used. In particular, we study the performance of FNN (Rumelhart, Hinton, & Williams) to solve this kind of problems.

In this paper we train a FNN to be used as a heuristic function to improve forward search performance for planning. The network will learn what operator to use next, from examples that represent planning problems (i.e. initial and final planning states).

However, control knowledge must be applied to problems of varying sizes (for instance, in the blocks world, different problems can have an incrasing number of blocks). When symbolic representations (like control rules (Minton *et al.* 1989, Aler, Borrajo, & Isasi 2001) or PRS's (Khardon 1999b, Khardon 1999a)) are used, this is not a problem. But it presents a challenge if FNN are to be used, because their inputs are of fixed size.

In this paper, we have faced this problem by decomposing a planning problem in different instantiations. Every instantiation has constant size and can be fed into the neural network. Then, the different outputs of the network must be combined somehow to produce the final answer. This is similar, in symbolic terms, to allowing PRS rules to have only a limited number of variables (Khardon 1999a).

Actually, this problem can be seen in a more general way as follows: machine learning techniques which use the propositional ("attribute and value") representation assume problems of fixed size. In the case of planning (in the blocks world, for instance), this means that only problems with up to a fixed number of blocks can be solved. If it is desired to solve problems with any number of blocks but learning with only simple problems, it is neccesary to use relational representations, of the ILP kind. In this paper, we are trying to apply a propositional representation to solve a relational problem.

So far, we have done only some preliminary testing of our approach in the blocks world. Here, we are interested in how the performance of the FNN changes as problem complexity increases.

The structure of this paper is as follows. In Section we describe our approach. Section presents somre preliminary experiments.

Description of the Approach

Representing planning problems for neural networks

In this paper we want to train a FNN to be used as a heuristic function to improve search performance for planning. In order to do so, it is necessary to define the inputs and the outputs of a FNN and how the training patterns will be codified. In our case, the inputs to the FNN will represent the current state in the search process and the goal state to be reached, and the output will be the instantiated planning operator that should be applied to the current state, to get to another state which is, hopefully, closer to the goal state. This process will be iterated until the goal state is reached.

In the STRIPS formalism, states are represented us-

ing predicates and objects. For instance, in a version of the blocks world, states are represented using five predicates (`arm-empty`, `holding(X)`, `clear(X)`, `on-table(X)`, and `on(X,Y)`) and names for the blocks (`A`, `B`, ...). States in the blocks world can be represented by instantiating the predicates with all the blocks. However, the size of a state increases with the number of blocks. But our goal is that the FNN can be applied to any world containing any number of blocks. Since FNN can accept only a fixed number of inputs, we have introduced the concept of 'variable': at any time the FNN will be only aware of a fixed number of variables which later will be instantiated in all possible ways with the blocks of the state. For instance, in order to represent a state containing three blocks (A, B, C) with only two variables (X, Y), we would have to consider 6 instantiations: (X=A,Y=B), (X=A,Y=C), (X=B,Y=A), (X=B,Y=C), (X=C,Y=A), and (X=C,Y=B).

Now, all the possible combinations of the predicates and the variables are computed. With 2 variables and 5 predicates, the combinations are: {`on(X,Y)`, `on(Y,X)`, `on-table(X)`, `on-table(Y)`, `clear(X)`, `clear(Y)`, `arm-empty`, `holding(X)`,. `holding(Y)` }. Each one of these predicates is represented with 1 if they are true in a particular instantiation and as 0 if they are false. Hence, one instantiation requires 9 bits to be represented. Therefore, in this case, the input to the FNN is made of 18 bits, 9 for the initial state and 9 for the goal state. Given an initial state and a final state, there is a binary 18-bit input pattern for every instantiation of the variables.

For instance, let us consider two variables X and Y, and three blocks. If the initial state is {on(A,B), clear(A), on-table(B), on-table(C), clear(C), arm-empty} and the final state is {on(A,B), on(C,A), clear(C), on-table(B), arm-empty}, the instantiation (X=A,Y=B) would be represented as:

100110100,100000100

In order to represent an initial and final state pair, there will be as many 18-bit binary patterns as possible instantiations depending on the number of variables. In this case there will be 6 input patterns.

As mentioned previously, the output of the FNN represents the operator that has to be applied in the current state to reach another state wich is closer to the final state. The output is made of as many bits as operators there are and another extra `no-op` bit. In the blocks world, there are four operators ({`stack(X,Y)`, `unstack(X,Y)`, `pick-up(X)`, `put-down(X)` }), so the output consists of 5 bits. The variables of the operators correspond to the variables of the instantiations. For every instantiation (X=x,Y=y), if the operator to apply to the current state is `op(X=x,Y=y)` then the bit corresponding to `op` is set to 1, and the rest of the bits are set to 0. Otherwise, the `no-op` bit is set to 1 and the rest to 0. Table 1 displays the five bits for the output of the FNN for each one of the possible instantiations, in case the operator to apply is `stack(A,B)`. In all but the first of the instantiations, the `no-op` bit is set to 1 because the variables of the instantiation are different to the variables of the operator.

Table 1: Outputs of the network for every instantiation of stack(X=A,Y=B), the 5 bit approach. S=stack, U=Unstack, PU=Pick-up, PD=Put-down.

`stack(X=A,Y=B)`

X,Y	No-op (1)	S. (2)	U. (3)	PU. (4)	PD. (5)
A,B	0	1	0	0	0
A,C	1	0	0	0	0
B,A	1	0	0	0	0
B,C	1	0	0	0	0
C,A	1	0	0	0	0
C,B	1	0	0	0	0

Table 2: Outputs of the network for every instantiation of stack(X=A,Y=B), the 8 bit approach. S=stack, U=Unstack, PU=Pick-up, PD=Put-down.

`stack(X=A,Y=B)`

	Operator			
X,Y	S. (1)	U. (2)	PU. (3)	PD. (4)
A,B	1	0	0	0
A,C	0	0	0	0
B,A	0	0	0	0
B,C	0	0	0	0
C,A	0	0	0	0
C,B	0	0	0	0
	No-operator			
X,Y	S. (5)	U. (6)	PU. (7)	PD. (8)
A,B	0	0	0	0
A,C	1	0	0	0
B,A	1	0	0	0
B,C	1	0	0	0
C,A	1	0	0	0
C,B	1	0	0	0

Another way we have used to represent the output is by using 8 bits. The meaning of the first four bits is the same than in the later paragraph. The other four bits expand the `no-op` bit. They represent the operator that should be used, in case the variables of the instantiation were the right ones as shown in Table 2.

Obtaining the training patterns

Given an initial situation, a final situation, and the first instantiated operator that must be applied to reach that final situation (let us call them a planning triplet), several training patterns will be obtained. The number of training patterns obtained depends on the number of possible instantiations which in turn depends on the number of variables. For instance, let us suppose that we consider two variables and three blocks. Also, we have the following planning triplet:

- Initial state: all the three blocks on the table
- Final state: a tower made of the three blocks (A on

Table 3: Obtaining training patterns.

	Input		Output
Instantiation	Initial state	Final state	
(X=A,Y=B)	001111100	100010100	10000
(X=A,Y=C)	001111100	000110100	10000
(X=B,Y=A)	001111100	010001100	00010
(X=B,Y=C)	001111100	100100100	00010
(X=C,Y=A)	001111100	001001100	10000
(X=C,Y=B)	001111100	011000100	10000

B, and B on C)

- The first operator to apply is `pick-up`(B)

From this planning triplet, six training patterns can be obtained (with a 5 bit output), as it is shown in Table 3.

There will be many cases where several training patterns with the same input have a different output associated. That is, they are contradictory patterns. To deal with this problem we have tried two approaches:

1. **OR approach:** a single training pattern is obtained from all the training contradictory patterns by using the OR function on the output. In that case, a pattern can have several bits set to 1 at the output.
2. **AVERAGE approach:** a single training pattern is obtained from all the training contradictory patterns by computing the probability of every bit in the output being 1. In this case, the bits at the output are real numbers between 0 and 1.

Using the network to solve planning problems

First of all, given an initial and final planning situation, the set of instantiated operators that could be applied in the initial situation is obtained. Now, the problem is to choose one of them. The steps to determine it are the following:

1. All possible instantiations of the initial and final situation are obtained. Therefore, a set of input patterns to the FNN are obtained.
2. For each one of the input patterns, the output of the FNN is calculated. Therefore, now there is a set of outputs that need to be combined.
3. Also, for each instantiation and for each operator that could be applied in the initial situation, the set of outputs that the network should give are determined. Then, they are compared with the actual outputs of the FNN. This is actually done by substracting the actual output of the FNN and the desired output. The instantiated operator that gets the minimum value (that is, which is closest to the desired output) is applied and an new initial situation is obtained.
4. These steps are applied until the final situation is reached.

Table 4: How the FNN is used to select the next operator to apply.

	Actual output	`PU.`(B) output	`PU.`(A) output	`PU.`(C) output
X,Y				
A,B	0.99 0.00 0.01 0.00 0.00	10000	00010	10000
A,C	0.98 0.10 0.02 0.01 0.10	10000	00010	10000
B,A	0.01 0.12 0.01 0.87 0.00	00010	10000	10000
B,C	0.01 0.12 0.01 0.99 0.00	00010	10000	10000
C,A	0.99 0.00 0.02 0.01 0.00	10000	10000	00010
C,B	0.90 0.00 0.09 0.10 0.00	10000	10000	00010

For example, let us assume that the planning problem to solve is:

- Initial state: all the three blocks on the table
- Final state: a tower made of the three blocks (A on B, and B on C)

In this case, the solution is to apply `pick-up`(B). The set of operators that could be applied in the initial situation is: {`pick-up`(B), `pick-up`(A), `pick-up`(C)}. The way to decide the operator that should be applied is illustrated in Table 4.

From this planning triplet, the following six training patterns can be obtained (when the output is made of 5 bits):

As it can be seen in Table 4 the instatiated operator which is closest to the actual output is `pick-up`(B).

Experimental Results

The experimentation was divided in two parts. First, we compared different experimental configurations of the system. And second, we selected the best configuration and then it was tested with some simplified problems.

Comparing different configurations

We have tested the following configurations (so far, we have only used the 5 bit approach):

- To compact equal patterns using the OR function
- To compact equal patterns using the AVERAGE fuction

We generated all the possible planning problems in the blocks world with 3 blocks and with 4 blocks and we have trained the FNN according to the following configurations:

- 3 blocks world using the OR function to compact
- 3 blocks world using the AVERAGE function to compact
- 4 blocks world using the OR function to compact
- 4 blocks world using the AVERAGE function to compact

Table 5: Empirical results for the different configurations.

Blocks	Var.	Outputs	Compact	Training error	Testing error
3	2	5	OR	0.009	0.052
4	2	5	OR	0.004	0.005
4	2	5	AVE.	0.001	0.001
4/3	2	5	OR	0.005	0.043
4/3	2	5	AVE.	0.004	0.033
4	3	5	OR	0.012	0.105
4	3	5	AVE.	0.033	0.112

- 3 and 4 blocks world using the OR function to compact
- 3 and 4 blocks world using the AVERAGE function to compact

Configuration were tested using 10-fold cross-validation. The results are shown in Table .

According with the results it seems slightly better to compact equal patterns using the AVERAGE approach insteed of the OR approach.

Testing the FNN

In this subsection the FNN trained with examples from the blocks world with 3 and 4 blocks and the AVERAGE approach has been chosen for testing.

We have considered two kinds of problems. The goal of the first ones is to stack a block C1 on C2. The initial situation contains several blocks on both C1 and C2, that must be unstacked before being able to stack C1 on C2. For instance, in the following there are 2 blocks on C1 and 0 blocks on C2:

- Final state: ((ON C1 C2) (ON-TABLE C2) (ON-TABLE C3) (ON-TABLE C4) (CLEAR C1) (CLEAR C3) (CLEAR C4) (ARM-EMPTY))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON C3 C1) (ON C4 C3) (CLEAR C4) (ON-TABLE C2)(CLEAR C2))

The second kind of problems consist in bulding towers from an initial situation in which all the blocks are on the table. For instance, the goal of the following problem is to build a tower of two blocks:

- Final state: ((ARM-EMPTY) (ON-TABLE C2) (ON C1 C2) (CLEAR C1))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON-TABLE C2) (CLEAR C1) (CLEAR C2))

We have decided to have two different kinds of problems, which can be easily scalated as a methodological help for testing the system. This way, it will be possible to know in which cases the FNN has learned something useful, and how well it scalates for more difficult problems. This is not possible if the system is tested only in randomly generated problems.

So far, we have tested the FNN with the two kind of problems, in the blocks world with 3, 4, 5, and 6 blocks. It has been observed that the FNN found the optimum solution for the first kind of problems. However, it is also observed that for the second kind of problems, the system is not very efficient.

Conclusions

In this paper we have used a FNN to be used as a heuristic function to improve forward search performance for planning. The network will learn what operator to use next from examples that represent planning problems (i.e. initial and final planning states).

The preliminary results show that the FNN could be an appropriate way to approach this problem. However, the experiments show that more work is required. We believe that to improve results it is necessary to find better ways to combine the different outputs of the network to obtain which operator should be applied. Also, it could be interesting to try different representations for the network.

Finally, in order to evaluate the system more accurately, it would be necessary to measure how much time is saved by using the FNN in comparison with a random search and also other domain independent planners.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2001. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation* 9(4):387–420.

Khardon, R. 1999a. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Khardon, R. 1999b. Learning to take actions. *Machine Learning* 35(1):57–90.

Minton, S.; Carbonell, J.; Knoblock, C.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40.

Rumelhart, D.; Hinton, G.; and Williams, R. *Learning Internal Representations by Error Propagation in Parallel Distributed Processing.*

Zimmerman, T., and Kambhampati, S. 2001. What next for learning in ai planning. In *Proceedings IC-AI.*

Table 5: Empirical results for the different configurations.

Blocks	Var.	Outputs	Compact	Training error	Testing error
3	2	5	OR	0.009	0.052
4	2	5	OR	0.004	0.005
4	2	5	AVE.	0.001	0.001
4/3	2	5	OR	0.005	0.043
4/3	2	5	AVE.	0.004	0.033
4	3	5	OR	0.012	0.105
4	3	5	AVE.	0.033	0.112

- 3 and 4 blocks world using the OR function to compact
- 3 and 4 blocks world using the AVERAGE function to compact

Configuration were tested using 10-fold cross-validation. The results are shown in Table .

According with the results it seems slightly better to compact equal patterns using the AVERAGE approach insteed of the OR approach.

Testing the FNN

In this subsection the FNN trained with examples from the blocks world with 3 and 4 blocks and the AVERAGE approach has been chosen for testing.

We have considered two kinds of problems. The goal of the first ones is to stack a block C1 on C2. The initial situation contains several blocks on both C1 and C2, that must be unstacked before being able to stack C1 on C2. For instance, in the following there are 2 blocks on C1 and 0 blocks on C2:

- Final state: ((ON C1 C2) (ON-TABLE C2) (ON-TABLE C3) (ON-TABLE C4) (CLEAR C1) (CLEAR C3) (CLEAR C4) (ARM-EMPTY))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON C3 C1) (ON C4 C3) (CLEAR C4) (ON-TABLE C2)(CLEAR C2))

The second kind of problems consist in bulding towers from an initial situation in which all the blocks are on the table. For instance, the goal of the following problem is to build a tower of two blocks:

- Final state: ((ARM-EMPTY) (ON-TABLE C2) (ON C1 C2) (CLEAR C1))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON-TABLE C2) (CLEAR C1) (CLEAR C2))

We have decided to have two different kinds of problems, which can be easily scalated as a methodological help for testing the system. This way, it will be possible to know in which cases the FNN has learned something useful, and how well it scalates for more difficult problems. This is not possible if the system is tested only in randomly generated problems.

So far, we have tested the FNN with the two kind of problems, in the blocks world with 3, 4, 5, and 6 blocks. It has been observed that the FNN found the optimum solution for the first kind of problems. However, it is also observed that for the second kind of problems, the system is not very efficient.

Conclusions

In this paper we have used a FNN to be used as a heuristic function to improve forward search performance for planning. The network will learn what operator to use next from examples that represent planning problems (i.e. initial and final planning states).

The preliminary results show that the FNN could be an appropriate way to approach this problem. However, the experiments show that more work is required. We believe that to improve results it is necessary to find better ways to combine the different outputs of the network to obtain which operator should be applied. Also, it could be interesting to try different representations for the network.

Finally, in order to evaluate the system more accurately, it would be necessary to measure how much time is saved by using the FNN in comparison with a random search and also other domain independent planners.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2001. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation* 9(4):387–420.

Khardon, R. 1999a. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Khardon, R. 1999b. Learning to take actions. *Machine Learning* 35(1):57–90.

Minton, S.; Carbonell, J.; Knoblock, C.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40.

Rumelhart, D.; Hinton, G.; and Williams, R. *Learning Internal Representations by Error Propagation in Parallel Distributed Processing.*

Zimmerman, T., and Kambhampati, S. 2001. What next for learning in ai planning. In *Proceedings IC-AI.*

Design of a Testbed for Planning Systems

Klaus Varrentrapp and **Ulrich Scholz** and **Patrick Duchstein**
Intellectics Group
Darmstadt University of Technology
Alexanderstraße 10
64283 Darmstadt, Germany
klaus@varrentrapp.de
scholz@informatik.tu-darmstadt.de
patrick@duchstein.com

Abstract

Conducting computational experiments and analyzing their results in a sound manner can be tedious. Experiments have to be organized, i. e. algorithms in various configurations, with several inputs and repetitions have to be run and results have to be analyzed from different perspectives, including statistical evaluation. In this paper we discuss properties of an ideal testbed for experiments and the statistical evaluation thereof, which automates recurring tasks and supports scientific soundness. We present a prototypical testbed, which will realize some of these ideas and which is short of being completed.

Introduction

Currently, we find many proposals for techniques addressing different aspects and phases of the planning process. The right combination of these techniques is necessary to fully exploit their mutual strengths and remedy their mutual weaknesses. However, such combinations mostly have been realized in monolithic blocks resulting in yet another planning algorithm. Conversely, the different phases of planning could be implemented as independent modules with a common interface. With the ability to assemble these building blocks freely, complete planning systems could be designed more easily, yielding greater flexibility and possibly better performance. With a modular structure of planning algorithms, researchers can compare and combine their work.

Experiments mainly comprise configuration and tuning of algorithms and comparison of different algorithms and modules. The process of carrying out computational experiments (experiments in short) of any kind with combinations of algorithms and modules is challenging. Organization and realization of experiments require a lot of practical details with respect of managing the different computation steps, such as providing input, operating the algorithms, storing intermediate data, and analyzing the the results. Besides, appropriate settings of experiments in order to obtain reliable results requires some insight in the problems associated with an empirical analysis of algorithms (McGeoch 1996; 2001; Hooker 1994; 1996; Gent & Walsh 1994; Moret 2002; Rardin & Uzsoy 2001). Hence, it seems desirable to have an easy to use environment for conducting experiments that automatically takes care of most of the recurring tasks.

In order to make scientifically sound decisions, one eventually has to rely on statistical tests. Consequently, an environment for experimentation should include a component for conducting statistical tests, too. Several researchers, mainly from the field of optimization, have addressed the issue of statistical testing and identified it as crucial for future empirical research on algorithms[1] (Cohen 1995; Xu, Chiu, & Glover 1998; Birattari *et al.* 2002; Coy *et al.* 2000). The need for a testbed unifying experiments for different data sets and schemes has been identified in the machine learning community before (Garner 1995; Witten & Frank 2000).

In this paper, we elaborate on the previously mentioned issues of designing a testbed for experimentation with algorithms – with emphasis on planning algorithms. We propose our vision of a general modular architecture that will meet the requirements and address the issue of statistical evaluation. We describe a prototypical implementation of such a testbed for planning algorithms, which is currently under development.

First, we define some basic notions. Experiments with algorithms can consist of a series of runs of one or more algorithms with various parameter settings in an environment that is under control of the experimenter. Combinations of various independent algorithms, say several preprocessing procedures with a planning procedures, can be seen as a (hierarchical) algorithm, too. The purpose of such experiments is, e, g. to find the best algorithm or the optimal configuration of several algorithms, i. e. the best setting of parameters.

From now on, we will denote with *algorithm* any single or composite procedure that is being investigated. With *configuration* we refer to any distinct setting of components and their parameters that yield a running entity. The notion of *experiment* then denotes a coherent conglomeration of running an algorithm in different configurations or comparing a number of algorithms in various configuration in order to gain insight in the behavior and the relationship between algorithms.

[1] www-users.cs.york.ac.uk/~tw/empirical.html

Requirements

In this section we identify several recurring processes and problems that occur when conducting experiments to study algorithms. Based on these basic processes, we propose some requirements for a flexible testbed that automatizes and generalizes the process of experimentation. Subsequently, we briefly mention some of the expected advantages of such a tool.

When studying algorithms empirically, certain recurring general processes appear:

- Provision of problem instances. All algorithms must operate on some problem instances. These must be supplied by either enabling access to benchmark repositories or by including perhaps randomized problem instance generators.
- Operation and synchronization of the algorithms used within the scope of an experiment.
- Ensuring proper flows and potentially storage of all data involved, such as problem instances, algorithm output, data describing the setting of the experiment, the generated test instances, and so on.
- Identification of all data relevant to and produced by an experiment. Delimitation of data from different experiments.
- Processing of raw algorithm output for further analysis. This involves tasks such as plotting graphs and computing statistics like mean, variance, and so on.
- Conducting statistical tests to scientifically verify any statements extracted from the experiment results.
- Supervision, status checking and perhaps modification of the experiment during execution.
- Recovering from partial crashes. The longer the runtime of an experiment, the higher the likeliness of malfunctions. Such disruptions should not corrupt the results. Instead, improper parts of the experiment should be repeatable on demand or even repeated automatically.

Following these general requirements we derive our vision of more specific demands for an ideal testbed:

- Provision of a user friendly interface. This user interface enables the user to specify experiments, supervise them, explore immediate results, trigger statistical evaluation, and visualize any required information in this respect. Ideally, all operations of the experiments can be controlled via the user interface. The interface also enables the user to retrieve any stored data from past experiments, thus allowing easy comparison of earlier results.
- Cooperation with any algorithms and modules and existing analysis tools. The testbed allows any aspect of an algorithm to be subject of investigation. Consequently, a general interface for accessing, controlling and running algorithms with standardized input and output streams will be included. This way, arbitrary algorithms and modules that comply with the interface requirements and stream data formats can be combined. A role model for this approach can be seen in scripts and pipes of Unix. Additionally, provisons for enabling subsequent analysis of data by statistical packages will be integrated.
- Extensibility and adaptability. Users are given the freedom to extent the testbed with individual modules, especially with their own tools, such as scripts for extracting data from output-files for plots. This extention will be at best be feasible without major changes to the testbed. In the ideal case, only interfaces need to be altered.
- Decoupling of the various processes of experimentation. Following this requirement, it seems necessary to give the testbed a fully modular structure as is common in modern software engineering.
- Coherent storage of all experiment relevant data. Special attention has to be given to the issue of storing and labeling of all relevant data describing all aspects of an experiment. In order to reproduce and compare data, it is vital to store data describing the setting, such as the version of operating system and algorithms. Furthermore, it is desirable to facilitate future retrieval of all experiment relevant data, such as raw output data, processed data, and a description of evaluations and tests performed. Altogether, the complete data from any experiment should be accessible cohesively and comfortably.
- Flexible specification of complete experiment settings. All aspects of conducting experiments can be subject to change by the researcher without any confinement. The testbed supports this via a general experiment specification language, which can be seen as a programming language for conducting experiments. It is independent of the algorithms under investigation. A proper design of such a language will be crucial to this venture.

Immediate advantages when using a testbed that meets the previous demands are increased efficiency and the possibility to reproduce and compare experiments. A modular object-oriented design with independent building blocks will enable a heterogeneous and iterative development of the modules. Necessary refinements, generalizations and abstractions of the everyday problems of empirical experimentation can only appear when using it. Hence, being able to independently improve the various processes is of greatest practical use. That way, such an environment can grow with the experience gained with it.

Having an experiment specification language, general templates and procedures for conduction experiments could be extracted and proliferated, putting empirical evaluation on an even more scientifically basis, since non-experts in statistics, now, can use these templates to perform a sound statistical testing. Experimenters can concentrate on the algorithms instead of having to devote energy to the rather complicated and difficult problem of setting up experiments properly.

For example, consider a planning algorithm consisting of several individually parameterized components. The aim is to fine-tune the parameter setting. Dependent on the results of earlier runs with some initial parameter values as validated by statistical tests, later runs can explore more promising areas of parameter settings. This interaction of runs and

analysis of results could be expressed within a specification language by some loops and conditional expressions and thus this process could be automated as done in (Birattari *et al.* 2002; Xu, Chiu, & Glover 1998; Coy *et al.* 2000). Each such imperative description of experiment realization can be viewed as a general template,script or program, applicable to other algorithms, too. Generally, any experimental design such as the Taguchi Design (Roy 1990; Tsui 1992) can be regarded as a kind of template or script.

Proposed Architecture

In this subsection we further elaborate on our preceeding reflections and propose a general architecture for a testbed for experimentation with algorithms, depicted in figure 1.

The envisioned testbed consists of several components or modules for the specific tasks of experimentation. These modules are independent to such a degree that they can be exchanged with newer versions without having to change the other parts of the testbed. Special care has to be addressed to the interfaces of theses modules. By interface we primarily mean part of a module that handles communication with other modules. Only secondly we denote by interface the fixed communication protocol used between a module and one of its interface parts. Interfaces in the primary meaning are needed to flexibly separate modules. The independent modules are all controlled by a central control unit (CCU) with standardized interfaces and with no or little interaction between the modules. When changing a module, only the module and perhaps its interface to the CCU has to be changed.

In case any component fails, the CCU can detect such a crash and recover it, e. g. by repeating it or by notifying the user. Modules need not be one single entity but can be a collection of loosely related small tools that are transparent to the CCU by means of the interface that controls them directly. In the simplest case, an interface only relays commands and data. Additionally, we propose to direct all flow of data via databases. By this, all data including intermediate data can be stored efficiently and safely. One database, which will store the experiment specification, can also store pointers to all experiment relevant data, so this data can be collected from the various databases and thus be retrieved afterwards.

The CCU manages all operations that have to be done in order to conduct the experiment as a whole. Its various interfaces comprise:

- Interface to the user interface (IUI).
- Interface to the unit that runs and controls the algorithms (IRU).
- Interface that reads a new specification of an experiment (IES).
- Interface for accessing a database where experiment specifications and pointers/keys to all relevant experiment data is stored (DBI).
- Interface to the module that performs statistical processing such as data analysis and statistical testing (ISU).
- Interface to the module computing all displays and graphics presented to the user (IDU).

The modules that belong to the previously listed interfaces are described next:

Run Control Unit (RCU) This module operates the algorithm in terms of the combination of algorithm components subject to the experiment. It invokes the components with correct parameter settings, ensuring proper flow of data between the components, provision of problem instances as input, and storage of any output data. As determined by the experiment specification, the CCU invokes the components in proper sequence with actual parameter values on the specified instances. This module again has certain interfaces to ensure flexibility with respect to different problem inputs and algorithm components. These are:

- Interface to the problem generation unit (IGU). The problem generation unit (PGU) typically is either a potentially randomized problem instance generator or retrieves its problem instances from a benchmark repository. Generally, the PGU is the module that provides the experiment input.
- Interface to algorithm components (IAC). Heterogeneous algorithm components might have very individual modes of control and in/output data formats. In order to give enough flexibility, this interface can be exchanged and tailored to the concrete algorithm components at hand, e, g. by means of wrappers for single components.
- Database Interface (DBI): This interface stores the output data to the appropriate database.

Display Unit (DU) The DU is responsible for computing presentations of any data regarding the experimentation results. These comprise the data from statistics such as means, variances, confidence intervals, results from statistical tests such as tables from analysis of variance procedures, and any plots and graphs illustrating aspects of the experiment such as runtime vs. solution-quality trade-off curves. Typically, this unit will employ plotting programs such as Gnu-plot.

Statistical Unit (SU) This module computes any statistics needed by the user. Additionally, it takes care of conducting any statistical testing. The data it needs to accomplish this is retrieved from the database where the ARU stores the raw output data from the applied algorithms. Typically, this module comprises a statistical package and some added tools.

User Interface (UI) The UI provides easy management and overview of the experiment to the user. It transmits orders from the user to the CCU and displays feedback from the CCU. The UI can include provisions for easy editing and processing of experiment specifications. Additionally, the presentation of results of any kind will be relayed through the UI from the DU and SU, respectively. Implementing the UI as a web interface permits a central experimentation facility equipped with appropriate hardware that is shared via the Internet.

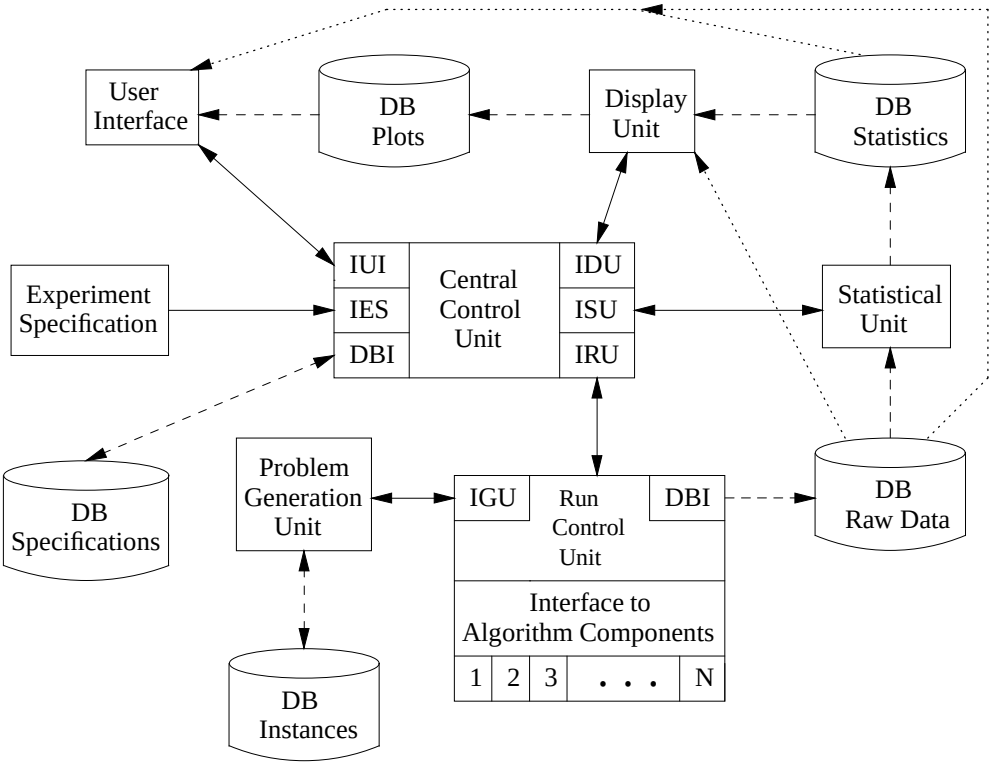


Figure 1: **Architecture of a testbed for conducting empirical experiments.** (Full lines represent control information, dashed lines indicate flow of data, and dotted lines indicate optional flow of data.) The central control unit (CCU) controls all processes of the testbed. It reads the experiment specification through its experiment specification interface (IES) and subsequently instructs the run control unit (RCU) running the algorithms, the statistical unit (SU) processing the results, and the display unit (DU) computing display such as plots and tables presented to the user via the user interface (UI). These units are accessed by the run control unit interface (IRU), the statistical unit interface (ISU), the display unit interface (IDU), and the interface to the user interface (IUI), respectively. The RCU retrieves problem instances through its problem generation unit interface (IGU) from the problem generator unit (PGU) which either accesses a database of instances or employs a problem instance generator. All intermediate data is stored persistently in various databases (DB).

This architecture is by no means fixed. It has to be tested and adapted in practical use of the testbed. Particularly, a proper design of the interfaces can only be achieved by experience. So far, the architecture mainly reflects the insight in the necessity of separating replaceable parts of the testbed.

Statistical Testing

Tuning and comparing algorithms are important aspects when designing algorithms. Usually, these tasks are performed by running algorithms with different parameter settings on a collection of problem instances. These instances either come from a benchmark repository or they are generated on demand, often randomly. On the basis of such experimentation results, postulations about the behavior and performance of the tested algorithms are made.

However, empirical results can be quite misleading. When using benchmarks as instances, for example, exact postulations can only be formulated with respect to the set of benchmarks used or with respect to the set of instances that can be generated by the problem instance generator used. Generalizations to a bigger set of problems (instances) are subject to immanent uncertainty. There is always the danger of tuning for the benchmarks or the problem generator (Hooker 1996). In fact, the excerpt of instances used for any experimentation need not be representative of all possible or of all relevant instances. When using randomly generated problems, it could happen, that, by chance, the instances generated are quite easy, suggesting the algorithm tested is very good. However, in this case, the results certainly should not be generalized, since the tested instances are too easy, which, unfortunately, is unknown to the experimenter.

There is always some kind of uncertainty about the results of experiments due to the randomness involved. Generally, there are two sources of randomness in many computational experiments. First, any choice of the test set of problem instances effectively is drawn from the underlying set of all valid or possible problem instances and as such a random experiment. Second, algorithms can themselves be randomized and hence each run is a random experiment, too. It should be clear that generalizations from the results of random experiments have to be handled with care.

A random experiment draws a number of individuals, called the sample, from a set of possible outcomes, called population. The aim of statistical inference is to control the danger of making wrong generalizations beyond the sample, since generalization from the sample is only valid when the sample is representative of the population.

The law of probabilities which governs the distribution of the observed values from a random experiment is called probability distribution. The random experiment of repeatedly throwing a coin observing heads vs. tails, for example, is distributed according to the binomial distribution with parameter p indicating the probability of observing heads. The set of all binomial distributions different only in the value of p is called the family of binomial distributions. Generally, families of distribution functions are called parameterized distributions.

Statistical inference exploits fundamental relations between the size of a sample, the variance of the observed values, and the confidence of the conclusions, following the intuition that the smaller the variance and the bigger the size of the sample, the more confidence we can have in our generalizations. Three main questions emerge when analyzing random experiments. All questions seek confident information about the true value of a parameter such as p previously or about the true value of a function of some parameters such as p^2 assuming some fixed family of probability distribution:

1. **Parameter estimation:** What is the value of a parameter or function?
2. **Confidence intervals:** What are the bounds of the value of a parameter or function given a certain confidence is expected?
3. **Hypothesis testing:** Does a parameter or function value fall into a certain interval? How high is the probability it does?

Consider running two planning algorithms pairwise on the same problem instances measuring the difference of the length of the computed plans. Assuming this difference is distributed according to a normal distribution, examples of the previous problems are:

1. What exactly is the mean difference for all problem instances? Particularly, is the mean difference positive or negative (indicating superiority of an algorithm)?
2. What are the bounds of the mean difference with a probability of 95%?
3. Is the mean difference below 0? How high is the probability it is?

Statistical inference further can be employed to answer questions about the magnitude of deviations of observed results from the expected mean. Further on, statistical inference can be employed to test whether a set of algorithms performs equally well or to regress the behavior of an algorithm depending on the problem instance size. Sometimes it is interesting to hypothesize on the underlying family of probability distribution. Finally, the influence of certain factors such as different parameter settings can be tested. One can test for mutual independence vs. correlation of a set of factors and whether a certain factor has an influence at all (Larson 1982; Lehmann 1997). When no assumption about the underlying probability distribution can be made, these questions can still be answered with so called non-parametric tests (Siegel 1956). However, they are less powerful. Existing standard statics package such as R (Ihaka & Gentleman 1996; Cribari-Neto & Zarkos 1999; Venables & Ripley 1999) supply these tests and can be used when employing statistical testing for a experimentation.

When deciding to employ statistical testing to validate conclusions, a proper design of the experiment in advance is crucial. Experimental design is concerned with proper planning of experiments involving statistical evaluation. The goal is to draw valid and objective conclusions with minimal effort. As such, the main issues of experimental design are planning of experiments to collect appropriate data and

to properly analyze these data by statistical methods (Cohen 1995; Mason, Gunst, & Hess 1989; Montgomery 1991; Dean & Voss 1999). There are several methods that describe how to properly design efficient experiments such as the Taguchi method (Roy 1990; Tsui 1992). These methods can be viewed as kind of template for conducting experiments as referred to previously. A language for specifying experiments could provide a standardized means to reusably describe such methods.

Planning Problems and Planning Systems

By now, we presented general properties of testbeds that are common for a variety of application domains. To demonstrate these ideas, we are currently working on a testbed for a specific problem domain: planning. At the time of writing, the work is about to be completed.

In planning the task is to execute actions in a world to bring it in a desired state. A world, also called domain, is defined by the set of its properties and the set of executable actions. A planning problem is a domain together with an initial world state and a set of goal states. Its solution is a sequence of actions that change the initial state into one of the goal states.

Problem descriptions commonly specify only the bare minimum of knowledge: The variable features of the domain and the available operators. In the problem domains most often addressed, however, there tends to be a rich structure "hidden" in the domain description. A planning system, now, consists of several algorithms that address different aspects of this structure. The output of one such algorithm is the input of the next and the flow of information from the initial problem to the solution can be quite complex.

To allow the modularization of planning systems, we use the Domain Knowledge Exchange Language DKEL (Scholz & Haslum 2000). It is designed to be part of domain and problem definitions in PDDL (Ghallab *et al.* 1998; Bacchus 2000), a widespread language to state planning problems. DKEL allows to decouple the extraction of domain knowledge from its use by augmenting the original domain or problem description with domain knowledge, rather than altering or reducing it right away. This way, the effect of a single module can be subject of investigation.

DKEL consists of five "general" knowledge forms which are modest generalizations of the forms of knowledge produced and used by domain analyzers and planners in existence today. The following is an example of a knowledge item for the three-operator blocksworld:

```
(:replaceable
  :optimal
    (:parallel-length :nb-operators)
  :vars (?x ?y ?z)
  :replaced ((move-from-table ?x ?y)
             (move ?x ?y ?z))
  :replacing
    (:empty (move-from-table ?x ?z)))
```

It states that it is always possible to replace the sequence of moving a block from the table onto a block and immediately onto another block by moving it directly onto the second location, at the time step of the second move. This replacement does not increase the number of operators nor does it increase plan length for parallel plans.

Prototype of a Testbed for Planning

The prototype testbed is concerned with planning algorithms composed from a number of planning modules in specific order. The testbed consists of a central control unit (CCU), a user interface (UI), one relational database for storing all relevant data in several tables, and a statistical unit. The CCU is directly responsible for handling any modules by starting the modules in proper sequence and configuration, collecting the results from the runs, and database storage of the results. Currently, all storage in the database is persistently. The raw algorithm output is stored in a table of the database, too, and accessed directly by the CCU to feed the statistical unit, in our case the R package[2] (Ihaka & Gentleman 1996; Cribari-Neto & Zarkos 1999; Venables & Ripley 1999). The results are displayed by a web user interface. As problem generators we use the generators of the FF domain collection.[3] All generated instances are stored in a table.

Any information concerning parameters of modules such as parameter flags and range of parameter values of a planning module is stored in a table. This table acts as a description of the standardized command interface as required by the testbed. New modules can easily be incorporated by adding a database entry for the new algorithm or module and placing the executable into the file system – the only prerequisite is that an algorithm provides a standardized command interface in the style of Unix. If necessary a wrapper, which can be regarded as kind of interface, has to be written for a module to adapt to the previous requirements. The testbed executes the modules via a system call. It calls the modules in proper sequence and directs the in- and outputs of the various modules via files in pipelining fashion, starting with the input from the table containing the instances, ending with the table containing the results of a run.The user interface currently accesses the database directly to read in the available modules and their parameters. So far, the prototype does not recover from partial crashes of any module.

Performing experiments is divided into three independent parts:

Instance generation The user can choose a domain and generate new problem instances by specifying several parameters defining the instance such as the size. Any instance generated that way is stored in a table for future use.

Specification and Execution The user can specify an algorithm and a configuration to run on a set of problem instances. The results and specifications of each distinct run are stored in a table and are identified by the name for the algorithm in its actual configuration, the problem instance identification and a timestamp. Algorithms can be specified by selecting a number and sequence of modules and configuring parameter values for each modules. Each

[2] www.r-project.org

[3] www.informatik.uni-freiburg.de/~hoffmann/ff-domains.html

algorithm and configuration specification created this way is named and stored in a table. Instead of specifying a new algorithm and configuration, the user can choose among the stored algorithm-configuration pairs. Problem instances can be selected from all stored instances generated in the past.

Statistical evaluation Applying statistical tests is performed by first choosing two or more named algorithm-configuration pairs. Next, a set of run results on a number of possibly common instances are picked for each pair. Then, a statistical test has to be choosen which will compare the selected algorithm-configuration pairs on the selected results. Depending on whether the user requests a test with repeated measures, i. e., tests on the same set of instances for all tested algorithm-configuration pairs, or without repeated measures, the tests are performed on the intersection of the sets of runs with respect to the same instances, or not, respectively. If the user selected more than two algorithm-configuration pairs for testing with a t-test, all possible pairwise combinations are tested. The results of the tests are displayed by the UI.

The planning modules under construction can be divided in three classes: Preprocessing techniques, which exhibit knowledge about the planning problem prior to the search for a plan, planners, and a module to connect and manipulate the input and output of the other modules in various ways. The considered planners are a reimplementation of GRT (Refanidis & Vlahavas 2001) and the planner FF (Hoffmann & Nebel 2001), adapted for use with the testbed.

We are implementing three preprocessing modules for our testbed: RIFO (Nebel, Dimopoulos, & Koehler 1997), TIM (Fox & Long 1998), and a technique to order goals (Koehler & Hoffmann 2000). RIFO eliminates irrelevant operators and TIM infers types and invariants. These techniques enrich the planning problem with knowledge which can be used by subsequent stages of the planning problem. The technique to order the goals of a planning problem is somehow different. It gives information how to divide the planning problem in to smaller ones and how to combine the partial solutions to a solution of the initial problem. This information is independent of the planning algorithm used. To account for this independence, we provide a wrapper for planners that uses the goal-ordering knowledge in a way transparent for the planner.

The use of a common language for input and output does not suffice in combining planning modules to a planning system. Each module has specific requirements that have to be met by its input, and changing these requirements can amount to a redesign of its algorithm. To account for these requirements, we provide a module, called DeDKEL, to manipulate planning problems in various ways: It allows to eliminate DKEL statements by either encoding it in the problem description or by just removing them. Furthermore, it can reduce the complexity of a problem description by eliminating specific features of PDDL, like negation or quantification.

DeDKEL has additional features that are useful for testing planing systems: concealment and randomization. In order to conceal the structure of a planning problem, e. g. for competitions, it can be helpful to replace identifiers by meaningless strings. This has been done for the mystery domain, a concealed logistics domain, which was used for the first AIPS planning competition. Randomization can be useful to test whether the performance of a planning system depends on the order of interchangeable features of the problem description.

Conclusion

In this paper, we discussed properties of an ideal environment to conduct and evaluate computational experiments. Some of these ideas are about to be implemented in a testbed for planning systems with a stress on reusability, extensibility and applicability to a wide range of problem domains.

This is not the first attempt to build such an environment: Anyone who has conducted a number of experiments on a collection of test sets will have felt the urge to automate this task. Likewise, the need for reusability of planning and preprocessing techniques has long been noticed. For example the TIM API[4] allows to easily include TIM into a planning system.

For the future, we plan to iteratively extent the testbed to meet further needs. The general direction was already outlined during this paper. In this process, the experience gained with the prototype will yield further insight in which features are necessary, which degrees of abstraction and generalization of experimentation processes are useful, and how much is necessary. This, hopefully, will also furnish us with more insight about needed features for an experimentation specification language.

This paper is inspired by our work in the meta-heuristics network,[5] which investigates heuristic search algorithms that solve combinatorial problems. This network has gained expertise in experimentation and statistical evaluation and feels the need for a tool that resembles the properties amplified within this paper. We hope that our prototype in the context of planning helps us on the way to a general purpose testbed for computational experiments.

References

Bacchus, F. 2000. Subset of PDDL for the AIPS 2000 planning competition. http://www.cs.toronto.edu/~aips2000/pddl-subset.ps.

Birattari, M.; Stützle, T.; Paquete, L.; and Varrentrapp, K. 2002. A Racing Algorithm for Configuring Metaheuristics. Technical Report AIDA-02-01, Darmstadt University of Technology.

Cohen, P. R. 1995. *Empirical Methods for Artificial Intelligence*. Cambridge, Massachusetts: The MIT Press.

Coy, S. P.; Golden, B. L.; Runger, G.; and Wasil, E. A. 2000. Using Experimental Design to Find Effective Paramterer Settings for Heuristics. *Journal of Heuristics* 7:77–97.

[4] www.dur.ac.uk/~dcs0www/research/stanstuff/TIMAPI/int.html

[5] www.metaheuristics.org

Cribari-Neto, F., and Zarkos, S. G. 1999. R: Yet another Econometric Programming Environment. *Journal of Applied Econometrics* 14:319–329.

Dean, A., and Voss, D. 1999. *Design and Analysis of Experiments*. New York, NY: Springer Verlag.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Garner, S. 1995. Weka: The Waikato Environment for Knowledge Analysis. In *Proceedings of the New Zealand Computer Science Research Students Conference*, 57–64.

Gent, I. P., and Walsh, T. 1994. How Not To Do It. Technical Report 714, University of Leeds.

Ghallab, M.; Howe, A.; Knoblock, C. A.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wikins, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hooker, J. 1994. Needed: An Empirical Science of Algorithms. *Operations Research* 42(2):201–212.

Hooker, J. 1996. Testing Heuristics: We have it all wrong. *Journal of Heuristics* 1:33–42.

Ihaka, R., and Gentleman, R. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5(3):299–314.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

Larson, H. 1982. *Introduction to Probability Theory and Statistical Inference*. New York, NY: John Wiley & Sons, Inc.

Lehmann, E. 1997. *Testing Statistical Hypothesis*. Springer Verlag.

Mason, R.; Gunst, R.; and Hess, J. 1989. *Statistical Design and Analysis of Experiments*. New York, NY: John Wiley & Sons, Inc.

McGeoch, C. C. 1996. Towards an Experimental Method for Algorithm Simulation. *INFORMS Journal of Computing* 8:1–15.

McGeoch, C. C. 2001. Experimental Analysis of Algorithms. In Pardalos, P., and Romeijn, E., eds., *Handbook of Global Optimization, Volume 2: Heurstic Approaches*. Kluwer Academic.

Montgomery, D. 1991. *Design and Analysis of Experiments*. New York, NY: John Wiley & Sons, Inc., 3rd edition.

Moret, B. 2002. Towards a Discipline of Experimental Algorithmics. In *DIMACS Monograph in Discrete Mathematics and Theoretical Computer Science*. Forthcoming. AMS Press.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the European Conference on Planning*, 338–350.

Rardin, R., and Uzsoy, R. 2001. Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial. *Journal of Heuristics* 7:262–304.

Refanidis, I., and Vlahavas, I. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.

Roy, R. 1990. *A Primer on the Taguchi Method*. Van Nostrand Reinhold.

Scholz, U., and Haslum, P. 2000. Enable your planner! Decoupling domain analysis and planning. Technical Report AIDA-00-05, Darmstadt University of Technology.

Siegel, S. 1956. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill.

Tsui, K. L. 1992. An Overview of Taguchi Method and Newly Developed Statistical Methods for Robust Design. *IIE Transactions* 24(5):44 – 57.

Venables, W. N., and Ripley, B. D. 1999. *Modern Applied Statistics with S-Plus*. Springer, 3rd edition. ISBN 0-387-98825-4.

Witten, I., and Frank, E. 2000. *Data mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann.

Xu, J.; Chiu, S.; and Glover, F. 1998. Fine-Tuning a Tabu Search Algorithm with Statistical Tests. *International Transactions in Operational Research* 5(3):233 – 244.

Profitable Directions for AI-Planning Research: A Personal View

Peter Jarvis

Artificial Intelligence Center
SRI International
333 Ravenswood Ave, Menlo Park, California 94025, USA.
Jarvis@ai.sri.com

Abstract

I reflect on the progress that the AI Planning field has made over the past 30 years and define the directions where I believe that we should focus our future efforts if we are to continue as a successful and vibrant scientific field. More concretely, I argue that Fikes and Nilsson's original framing of the planning problem is too far removed from the requirements of real-world applications. We must accept that it will not be feasible to obtain complete and consistent domain theories in the near future and that users will want to influence the plans they receive in dimensions other than goal state and shortest path. In light of this, I argue that we must focus on computer-aided planning instead of computer-replaced planning. I outline the technical challenges this course offers and the exciting work already emerging.

Keywords: Computer Aided Planning, and STRIPS Assumption.

Introduction

> "There are three principal means of acquiring knowledge available to us: observation of nature, reflection, and experimentation."
> **Denis Diderot** (1713–84)

All fields must periodically reflect upon their achievements, reexamine their goals, and direct future efforts accordingly. Here, I outline my personal reflection on the progress made in the AI planning field over the past 30 years and identify where I believe we should focus our future effort. I highlight some promising directions that have emerged before setting out some fundamental changes in research program structure and performance evaluation that we must bring about if these new directions are to mature and enable the field to move to a new level of accomplishment.

My Background

It is important for me to set out my experience in the field before proceeding. This is not an attempt to proclaim that I am a great oracle with far-reaching insights; rather, it is important that you understand the path I have taken so that you may judge appropriately my experience and bias.

During my Ph.D. program I focused on applying planning techniques to civil engineering projects (Jarvis & Winstanley 1996; Bloomfield et al. 1999) before joining the O-Plan team in Edinburgh. At Edinburgh, I worked on a range of applications, including process management for the chemical industry (Jarvis et al. 2000), army small unit tactical planning (Tate et al. 2000), and strategic military planning (Dyke et al. 2000). Since moving to SRI International, I have worked on air campaign (Myers et al. 2001) and Special Forces problems (Myers, Jarvis & Lee 2001; 2002). I have also worked on incorporating concepts from fuzzy logic into graph and SAT-based planners (Jarvis, Miguel & Shen 2000; Miguel, Jarvis & Shen 2000). To summarize, over the last 8 years I have experienced four large DARPA programs, several commercial consultancy projects, and a handful of Ph.D. programs. My goal here is to add what I have learned from this path to the debate on the directions that we should take in the future.

What We Have Achieved

AI researchers have worked primarily with Fikes and Nilsson's (1971) original framing of the planning problem for just over 30 years now. Quoting directly from their paper:

> *...the problem space for STRIPS* [or your planner] *is defined by three entities:*
>
> *(1) An Initial world mode, which is a set of wffs describing the present state of the world*
>
> *(2) A set of operators, including a description of their effects and their precondition wff schemata*
>
> *(3) A goal condition stated as a wff*
>
> *The problem is solved when STRIPS* [or your planner] *produces a world model that satisfies the goal wff.*

The publication mass of the planning field is dominated by two thrusts:

A Minimize the time a planner takes to find the shortest operator sequence necessary to produce a world model that satisfies the goal wff.

B Without damage to progress on thrust A, remove the (explicit) simplifications made by Fikes and Nilsson: atomic time, deterministic action effects, omniscience, and the planning agent being the sole cause of change.

We have made extraordinary progress over the past 30 years. Today's planners can solve problems orders of magnitude more complex than those of a few years ago (read Weld 1994, then Weld 1999 for an excellent perspective of how the field changed in those 5 years). We also have the first fully fielded and well-documented applications (Muscettola et al. 1998; [a subset of those in] Knoblock ed. 1996).

While we have a right to be proud of our accomplishments, we must continue to move forward with new and exciting innovations if the field is to remain alive and funded. I now consider the question of what is missing from our portfolio.

What We Have Not Achieved

A scientific field must produce concepts that can be taken on by engineers to produce artifacts of value to society. While it should not be a field's only focus, this link to application is important, as it provides many fascinating intellectual challenges that help keep scientists grounded and funding agencies interested.

As I noted above, our field has recently produced well-documented applications. This landmark accomplishment is rightly being celebrated in the literature and at conferences. My concern, however, is that people planning military operations still use paper or Microsoft PowerPoint™ while civil engineers use Artimis™ type tools. These tools offer benefits in facilitating the documentation and communication of plans, but they provide no assistance in the decision-making, specifically deciding what actions must be included in the plan or detecting and resolving complex action interactions.

The question I ask is, will the current focus of our field produce applications that scale to support the type of large-scale planning tasks I mention above? It is clear that we have solutions offering much impact at the physical device level. Taking a military example, we might soon be able to comfortably control a tank in achieving goals such as "stay alive" and "engage the enemy. " In the next section, I argue that the field is not tackling with sufficient emphasis the requirements of large-scale planning problems. Returning to the military example, I do not see the field's current direction leading to a technology capable of supporting a general's staff officers in designing the high-level strategy of a battle.

Where AI Planning Must Focus Its Efforts

I group my opinions on this topic into two sections. In the first, I consider promising work that seeks to address the implicit STRIPS assumptions and encourage more work in these areas. I then turn my attention to the more pragmatic issues of changes in research program structure and research evaluation necessary to encourage a broader range of work under the planning banner.

Relaxing the Implicit STRIPS Assumptions

While Fikes and Nilsson's explicit simplifying assumptions (often referred to as the STRIPS assumptions) have been the focus of much effort, little research has been devoted to the implicit assumptions in their framing of the planning problem. My thesis is that the Fikes and Nilsson's casting is too restrictive to be of value outside of device-level application domains. Here I work through the implicit STRIPS assumptions and promising work in each area. Only by bringing more effort to bear in these areas will the AI planning community reach the broad range of applications that could benefit from tool support.

Complete Operator Sets

Fikes and Nilsson's assumption that we can build an operator set that completely covers a domain is proving most difficult to realize in practice. In applications with many degrees of freedom, it is impractical to expect full coverage.

Consider the military problem of evacuating U.S. citizens from a hostile country. The number of factors that must be taken into account is simply enormous. People charged with planning operations of this type need to bring around 20 years of domain experience to bear on the problem. The AI community has long been trying to encode knowledge in these quantities, but with limited payoff (Leant & Guha 1990). Why does the AI planning community believe it can do better than its colleagues?

Consider planning's sister activity, design. Much design is now computer aided, as researchers have sought to complement rather than replace human designers. Encoding the knowledge needed to design products is several orders of magnitude more complex than encoding that required in assisting a human. Why does the AI Planning community insist on focusing solely on automated planning when computer-aided planning could provide tools of social value in a much closer time scale? Should we balance our portfolio to provide near as well as long-term payback?

Exciting work is emerging in the computer-aided planning direction. Dyer's SOFTools (GDATS 2002) provides one extreme on the continuum of computer-aided planning tools that is already in operational use with U.S. Special Forces units around the world. SOFTools provides a simple temporal planning interface with domain-specific icons. This speeds planning from the users' perspective, as it is more focused upon the types of diagrams with which

they represent plans while also providing a more structured representation for researchers to exploit than alternative documentation aids such as Microsoft PowerPoint (their previous tool of choice).

SRI's CODA system (Myers, Jarvis & Lee 2001; 2002) integrates with SOFTools to provide a higher level of computer support. With CODA, users can describe aspects of a plan where changes are likely to affect them adversely. CODA automatically generates alerts if another user changes such an area. This helps human planners coordinate when distributed in both time and space.

The mixed initiative paradigm is well suited to computer aided planning. Ferguson, Allen, and Miller's TRIPS system (1996) hooks a person and a planner together to solve travel problems, where the computer's role is to maintain constraints and inform the users when they are likely to be violated.

This direction offers significant challenges. For example, it necessitates that a user be allowed to add structures to a plan. With such editing allowed, it is no longer possible to check automatically that a plan is correct (all required causal links in place and unthreatened, for example) as the user may have neglected to input some important precondition or effect. What level of consistency checking is appropriate and possible in a plan authoring context is an interesting research question.

Goal Specification

Applications demand the specification of more than just the goals a plan is to achieve. Users may want to specify the strategies to use in solving the problem (avoid using F-14s for combat air patrols or stay in first class accommodation on the business legs of a trip) or even some of the actions that must be included in the solution (fly United between SFO and LAX). Myers has explored both types of user guidance in the form of Advisable Planners (1996) and Plan Sketch Completion (1997).

While Myers' work is an important step toward providing comprehensive mechanisms for specifying user objectives, the work assumes a complete operator base that as we argue above, is not likely to be available in practice. Some significant and interesting research challenges are still to be addressed in this area.

Plan Evaluation

We have assumed that the user of a planning system is looking for a single plan and that this should be the shortest plan. In practice, people plan for many reasons. In military planning (when, as the old adage suggests, no plan survives first contact with the enemy) planning is often used to ensure that the course of action committed to is readily adaptable to a changing situation. Planning in this context is an exploratory task where multiple plans are produced under different assumptions or advice directives (use F14s for Combat Air Patrols (CAPS), don't use F14s for CAPS) and compared.

Myers and Lee (1999) have considered this problem of generating multiple plans automatically. Again, this work assumes that complete operator sets are available. Swartout and Gil (1996) in their INSPECT system provide support for evaluating a course of action against user-defined criteria.

To move forward we must consider in more detail the need to explore multiple plans, perhaps even combining parts to form a new option. Open questions remain at many levels. How can we efficiently reason with large-scale plans that contain contingency branches? What are the salient features that users use to choose between courses of action? How can we present those features to users so that they can rapidly compare plans?

Flexible Preconditions

In mainstream planning, all an operator's preconditions must be satisfied for it to be applicable. This framework does not support the flexibility in constraint satisfaction necessary in many applications.

Consider the military problem of infiltrating a small team by swimming from a submarine to a beach. The standard operating procedure for this task contains many constraints. The submarine must remain concealed under the ocean surface (minimum operating depth and maximum illumination from the moon), the swimmer must avoid hypothermia (function of the distance to swim and sea temperature), and the infiltration must be completed within the time frame demanded by the overall mission of which it is a component.

It is rare that one can find the right combination of ocean temperatures, tide, and lunar illumination within the timescale of an operation of this type. Typically, something has to be compromised. For example, asking the swimmers to cover a greater distance keeps the submarine concealed while reducing the effectiveness of the divers when they reach their target because of the additional fatigue.

While there has been much work on looking at the probability of mission success (send two teams of divers rather than one) there has been little considering the effect of constraint violation on plan quality. Miguel, Jarvis, and Shen (2001) consider this problem. The approach is preliminary and asks more questions that it answers. In particular, we take a fuzzy-logic based approach to reasoning about the damage a violated constraint inflicts on a solution. Is this the right way to represent the importance of constraints?

Necessary Environmental Changes

Here, I examine the broader environmental issues that must be considered if we, as a field, are to produce more readily applicable technology.

Multidisciplinary Research

The problem with AI Planning research is that AI Planning researchers have undertaken it! We are focused on algorithms, as that is what interests us most. What we are

not generally interested in is modeling how people go about solving planning problems and identifying the niches for tool support. We have rather assumed that in solving Fikes and Nilsson's categorization of the problem we will produce the tool support that people require.

There are two barriers to carrying out the multidisciplinary research that I am suggesting. First, we need to build relationships with people with expertise in human factors, systems analysis, and cognitive psychology as these are the types of people who can help us understand the "as is" situation with human-level planning. We can then work on generating the "to be" process and finally the tool support needed for it. Second, we have not sought such multidisciplinary funding. This might be a function of the compartmentalization of funding agencies or just the comfort of working with colleagues with similar expertise.

Planner Evaluation and Planning Competitions

One of the attractions of Fikes and Nilsson's characterization is the ease with which progress can be measured. If Planner A solves problem 1 faster than Planner B, then we can conclude that Planner A's performance is superior. This ease of comparison has driven the community towards its current focus on planning competition where the group walking away with the most prestige from a conference is likely to be the one that provides the fastest system in the competition track.

Computer aided planning systems are going to be more difficult to evaluate. However, we must find appropriate metrics or will be difficult to determine if progress has been made. Again, we reach a difficult research question. How do we evaluate applied research? Who will pay for the evaluation effort given that it might need access to many users in controlled conditions?

Good Application Papers

A justifiable criticism raised at much of the previous applied research is that it is not well documented. Authors have not always clearly laid out the computational procedures and domain models that they have used. This has made it difficult to determine exactly what an "application" is doing and the compromises that have been made in its design.

While omitting this detail is understandable given funding constraints, it is necessary if the applied community is to maintain the respect of the more formal community. We should be careful to take the time to make our applications assessable by our more formally focused colleagues.

End the Divide on Search Control Knowledge

The planning field has been divided into the mutually exclusive "formal" and "applied" camps for too long. The applied camp has centered on Hierarchical Task Network (HTN) (Sacerdoti 1974; Tate 1977) techniques that encode knowledge about the actions available in a domain together with knowledge about how to go about solving problems in that domain. The "formal" camp has resisted the encoding of this search control knowledge as its members correctly argue that it leads to less flexible solutions.

There are two points here. First, applied work has been discounted because it has almost always has made use of search control knowledge. However, the problems posed by real applications exist independently of this design decision. Indeed, I have not had to mention this design decision until now. Second, HTN approaches couple search control knowledge tightly with operators. Huang, Selman and Kautz (1999) show that search control knowledge can be loosely coupled with the operator base, allowing it to be swapped in and out more easily.

We should proceed with the understanding that search control knowledge should be avoided. However, when we have to use it we must ensure that it is declarative so that it may be replaced as search speeds increase or the control knowledge becomes outdated.

Conclusion

As a field, we must move beyond Fikes and Nilsson's characterization of the planning problem and center our efforts upon a computer-aided rather than computer-replaced planning process.

There is no shortage of challenging research questions to answer on this path. My closing question is, how do we motivate a field to move in a new direction given that it will cause significant discomfort in the short term?

References

Bloomfield, D., Faraj, I., Jarvis, P., and Anumba, C., 1999, Managing and Exploiting Knowledge Assets in the Construction Industry, *In Proceedings of the 8th International Conference on Durability of Building Materials and Components, Vancouver, Canada.*

Dyke, D., Salt, M., Jarvis, P., and Desimone, R., 2000, Experimental Results from Integrating Planning Systems and Simulation Models. *In Proceedings of the 2000 Command and Control Research Technology Symposium, Vienna, VA.*

Ferguson, G., Allen, J., and Miller, B., TRAINS-95: Towards a Mixed-Initiative Planning Assistant. in *Proceedings of the Third Conference on Artificial Intelligence Planning Systems (AIPS-96)*, *Edinburgh, UK, 29-31 May 1996, pp. 70-77*.

Fikes, R., and Nilsson, N., 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2). North Holland Publishing Company.

GDATS, 2002. SOFTools V2.0 User Guide. General Dynamics Corporation, http://www.gdats.com.

Huang, Y., Selman, B., and Kautz, H., 1999. Control Knowledge in Planning: Benefits and Tradeoffs. *Proc. AAAI-99*, Orlando, FL.

Jarvis, P., Miguel, I, and Shen, Q., 2000, Flexible Blackbox *In Proceedings of the Workshop on Representational Issues for Real-World Planning Systems held within AAAI-00, Austin, TX.*

Jarvis, P., Moore, J., Stader, J., Macintosh, A., and Chung, P., 2000, Harnessing AI Technologies to Meet the Requirements of Adaptive Workflow Systems, In J. Filipe (ed) *Enterprise Information Systems,* Kluwer Academic Publishers, pp173-180.

Jarvis, P., and Winstanley, G., 1996, Dynamically Assessed and Reasoned Task (DART) Networks. *In Proceedings of Expert Systems 1996, the 16th Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems*, Cambridge, UK, December 1996, pp92-105. ISBN- 1-899621-15-6.

Knoblock, C., (editor), 1996, AI planning systems in the real world. *IEEE Expert,* December, p 4 – 12.

Leant, D., and Guha, R., 1990, *Building Large Knowledge Based Systems*. Addison Wesley

Miguel, I., Jarvis, P., and Shen, Q., 2001, Efficient Flexible Planning via Dynamic Flexible Constraint Satisfaction. *Engineering Applications of Artificial Intelligence*, 14, pp301-327.

Muscettola, N., Nayak, P., P, Pell, P, and William, B, 1998., Remote Agent: to boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2) 5-48

Myers, K., 1997., Abductive Completion of Plan Sketches, In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97).

Myers, K., 1996., Strategic Advice for Hierarchical Planners., In Principles of Knowledge Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96), Morgan Kaufmann Publishers, San Francisco, CA.

Myers, K., Jarvis, P., and Lee, T., 2002, CODA: Coordination of Distributed Human Planners. *In Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling Systems,* France.

Myers, K., Jarvis, P., and Lee, T., 2001, Active Coordination of Distributed Human Planners. *In Proceedings of the 6th European Conference on Planning (ECP-01)*, Toledo, Spain.

Myers, K., Smith, S., Hildum, D., Jarvis, P., and de Lacaze, R., 2001, Integrating Planning and Scheduling through Adaptation of Resource Intensity Estimates. *In Proceedings of the 6th European Conference on Planning (ECP-01)*, Toledo, Spain.

Myers, K., and Lee, T., 1999, Generating Qualitatively Different Plans through Metatheoretic Biases. in *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, AAAI Press, Menlo Park, CA, 1999.

Tate, A., Levine, J., Jarvis, P., and Dalton, J., 2000, Using AI Planning Technology for Army Small Unit Operations. *In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems, Colorado, USA, April 2000.*

Tate, A., 1977, Generating Project Networks, IJCAI, pp 888-893.

Sacerdoti., E., 1974, Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence,* 5 pp115-135.

Swartout, W., and Gil, Y., 1996, EXPECT: A User-Centered Environment for the Development and Adaptation of Knowledge-Based Planning Aids. In *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*,. Menlo Park, Calif.: AAAI Press, 1996.

Weld, D., 1999, Recent Advances in AI Planning. AI Magazine. 20(2), pages 93-123.

Weld, D., 1994 An Introduction to Least Commitment Planning. AI Magazine, 15(4), pages 27-61.