Domain-Independent Planner Compilation

Derek Long and Maria Fox
D.P.Long@dur.ac.uk Maria.Fox@dur.ac.uk

Department of Computer Science,
University of Durham, South Road, Durham, UK.

Abstract

The contrast between domain dependent and domain independent planning shares several characteristics with the contrast between interpreted and compiled programming languages. This observation has inspired an effort to construct a compiler for declarative planning domain descriptions which produces the code for a planner within the target domain. Thus, domain independent planning advantages are combined with the efficiency of domain-dependent code. The compilation procedure is described, together with the target abstract machine and the process by which it is implemented in C++ and a brief summary of results is provided.

Introduction

Research in automatic planning has broadly divided into two directions: domain-independent and domain-dependent. Constructing domain-dependent planners offers opportunities to tailor the mechanisms to the particular domain for far greater efficiency (typically, several orders of magnitude). For example, game-playing algorithms, such as for chess, can plan strategic and tactical sequences of moves, allowing for an opponent's responses, and do so in seconds. Some of these algorithms can also work with uncertainty (playing games in which chance plays a role). By sharp contrast, domain-independent planning is far behind in terms of efficiency and also flexibility (although Smith et al. report recent successes in applying an HTN planner to bridge-playing (Smith, Nau, & Throop 1996)). However, domain-independent planning offers far greater opportunity for reuse of planning technology, abstracting the mechanisms that give powerful and effective planning capability from the domain-specific tricks and short-cuts which allow the efficient behaviour within particular domains.

Domain independent planning separates the planning mechanism from the domain on which it operates by providing a (typically) declarative representation of the domain as input for the planner at execution time. Much of the recent work in domain-independent planning has been concerned with identifying what should be encoded in these domain descriptions to enable the planner to operate with greater efficiency and with greater flexibility. Of course, as the representations of domains are extended with more information to enable more efficient performance, so the construction of these representations becomes more complex and difficult for the non-specialist.

One of the reasons that domain-dependent planners can be so much more efficient than domain-independent planners is that the information which might be captured about a domain to enhance performance can be compiled into the domain-dependent planner, with no excess baggage to supply machinery for domain properties which simply do not apply to the particular domain being considered. The domain-dependent planner can be hard-coded to consider the particular domain operators and need not provide the machinery which would enable an arbitrary set of operators to be handled. These advantages are similar to the advantages of a compiled program over an interpreted program. Indeed, a domain-independent planner has much in common with an interpreter for a specialised language - a planning domain description language. It is this observation which has inspired the work described in this paper, the construction of a compiler for this planning domain description language, to enable relatively simple declarative expression of planning domains to be compiled into domain-dependent planners which combine the basic architecture of a domain-independent planner with the content of each particular domain description.

Some work has been carried out considering compilation in the context of domain-independent planning (McCluskey & Porteous 1994; McCluskey & Porteous 1995; McCluskey & Porteous 1996). However, this work is concerned with compiling rich domain descriptions (that is, declarative domain descriptions

which contain more information than the traditional simple operator specifications) into heuristic information to improve the performance of a domain-independent planner operating on that domain. The planner itself is still essentially an interpreter for the enriched language, with the compilation improving the access to some of the information encoded in the domain description. Work which has far more obvious common ground with that described in this paper is described in Srivastava's, Mali's and Kambhampati's paper (Srivastava, Mali, & Kambhampati 1997) on the synthesis of planners from specifications. Their work involves the use of a collection of tools within a semi-automated software synthesis system, KIDS (Smith 1992), supported through a development system, CLAY. They describe the process of construction of domain-dependent planners from domain-independent components together with domain-descriptions, and domain-dependent heuristics which are provided, at least in part, in declarative style. The construction leads to code generation in the first-order language REFINE, which can, in turn, be compiled into C or Common Lisp. REFINE is a general language, supporting first-order logic, set theory, pattern-matching and transformation rules, making it a powerful language in which to express the planning process, but at the same time imposing significant demands on the compilation process which will almost certainly include overheads which are unnecessary in the construction of planners. The process Srivastava *et al* describe is not a trivial one, clearly involving a close knowledge of the development system: an advantage of the work undertaken by the present authors is that the compilation process works directly from an unmodified domain description input, so no technical knowledge is required beyond that for the usual process of domain construction. However, the construction process in CLAY is far more flexible than that described here, allowing a range of planning strategies to be selected and combined in different ways. The present work should be seen as complementary to that of Srivastava *et al*, in that it initiates an exploration of a compilation strategy which is more focussed than the KIDS system, and offers the prospect of a planner-construction tool which demands little or no additional specialised knowledge of either planner-mechanisms or of the construction process, but works solely on the declarative domain description itself.

Planner-compilation: The Abstract Machine

In compiling planning domain descriptions into planners, a decision about the technology on which the basic planner architecture will be built must be made very early on, unless the system is to allow choices between different technologies at compilation-time. This decision influences the entire structure and process of compilation. In the work described here it was decided to begin with linear-planning technology. One reason for this is that linear planners are relatively straightforward technology, so the compilation process is (presumably) simpler than for other planning technologies. Another reason is that linear planning has much in common with the process of theorem-proving in Prolog and there is a well-established technology for the compilation of Prolog programs based on the Warren Abstract Machine (WAM) (Warren 1983).

Thus, the starting point for this work was the examination of the architecture of the WAM as a basis for the development of a Planning Abstract Machine (PAM). This led to the construction of an abstract machine based on a collection of stacks used to track state (both for forward progress and for possible backtracking) and with an instruction set containing some 30 instructions. Space prevents a complete description of these instructions here, but the central instructions will be described (several are essentially housekeeping-task instructions, such as those used in the control of the simple-establishment loops in goal-satisfaction code).

The planning domain-description language used in this initial exploration has purposefully been kept succinct. The language is a classic STRIPS language, with operators having preconditions, add-lists and delete-lists, all represented by sets of simple atomic propositions. Operators generally use variables to represent place-holders within the actions they encode, and the language used in this work allows types to be assigned to both these variables and to language constants which is a convenience in constructing domain descriptions, allowing same-name predicates to be used to capture similar properties of objects of different types.

The Data Structures

Rather than attempting to describe the instruction set, this description will begin with an account of the main data-structures which support the PAM. One of the core data structures of the PAM is an *activation stack* on which are allocated the activation records, or frames, for operators as they are initiated. Initially, an activation record contains variables corresponding to the variables in the operator schema. These are instantiated: some as the result of the initial invocation of the operator in order to ensure the operator instance has the desired effect and others as a result of satisfying preconditions of the operator. Thus, an operator can be seen as analogous to a procedure in the compiled code, with the activation frame analogous to the stack frame invoked on procedure entry. The main difference between a stack frame and an activation frame is that stack frame variables are initialised by the calling procedure (except, possibly, for reference parameters), while activation frame variables are only initialised through unification of an add-list entry of the operator and a goal which leads to initiation of the operator call. This is exactly analogous to the process of clause invocation in the WAM, where an activation frame can also include both instantiated and

uninstantiated variables.

A second key structure is the *choice stack*, used to record information required in backtracking. Each choice point in the planning process generates a choice stack entry of an appropriate type. There is a choice stack entry for choices at simple-establishment, enabling backtracking through alternative state-entries in the simple-establishment of a goal. There is a choice stack entry for choices between alternative step-additions, enabling backtracking to attempt alternative actions to achieve a goal. Finally, there are choice stack entries for operator applications enabling the planner to unwind an operator application before attempting other alternatives in its search for a plan. The choice stack also has its analogue in the WAM, although there choice frames are homogenous, dealing purely with backtracking through alternative clauses (although optimisations can lead to some complication of this picture).

The third vital data structure is the *state stack* which records the current domain state. This stack is constructed as an interlocking collection of doubly-linked lists which each record the status of propositions linked with a single predicate. The lists are constructed on the stack so that they contain the valid propositions of the current state, but so that the previously deleted elements of the states visited earlier in the plan are still present. This allows the planner to backtrack, restoring the state to earlier stages in the planning process. There is, of course, a trade-off between the storage of records of older components of state to enable rapid restoration of the earlier states and the reconstruction of those states by careful unwinding of actions from a much smaller record. This space-time trade-off has been deliberately made in favour of time, since the storage requirements of the state records for benchmark problems considered so far have all been insignificant (under 3Kb!) and time efficiency has been put at a premium. A second consideration is that the order of the elements in the lists representing the states is significant in the current implementation, since backtracking to attempt alternative simple-establishments requires that it be simple to keep track of which alternatives have been considered and which remain to be considered. If states are retrieved by reconstruction rather than simply restoration then the ordering must somehow be reestablished too and this would demand additional records, possibly undermining any space-saving on the state records themselves. State management is not an issue in the standard WAM. The use of the Prolog meta-predicates *assert* and *retract* has some common features with the use of add and delete effects in operators but these are not supported within the basic WAM. Prologs do not, as far as is known to the authors, allow backtracking over *assert* or *retract* in a way that restores the original state, and this is a crucial feature of the search machinery of linear planners.

The remaining stacks are used for minor details of backtracking and storage during forward progress of the planning process, including tracking the linkage between successive goals. Several of these stacks correspond to control stacks in the WAM, such as a *trail* to track the instantiation of variables through unification. The trail can be unwound to allow backtracking through alternative search paths, undoing the effects of unifications along earlier paths.

The PAM Code Structure

The code which runs on a PAM is structured in two main blocks. One block is responsible for the attempt to satisfy propositions and the second is responsible for the sequencing of preconditions of operators and their subsequent application. These blocks are subdivided so that each predicate has a separate entry point in the proposition satisfaction code and each operator its own entry point in the operator code block. Rather like the WAM, arguments to a goal are placed in PAM machine-registers at entry to the code for a given predicate. All predicate sequences begin with the code which attempts to simply-establish the current goal. There then follows a sequence of instruction groups, one for each way in which an operator in the domain description could be used to achieve the given goal. Each sequence begins by setting up an activation frame for the candidate operator, placing the arguments from the registers into these frames at the appropriate points, corresponding to the arguments instantiated by matching the goal against the correct add-list entry for the operator, and then calling the entry point for the correct operator. The call creates a choice frame which records the registers and the next instruction group entry point for return should this step-addition fail for any reason. After all step-addition instruction groups there is a final instruction, *Fail*, which causes backtracking when there is no way left to consider to satisfy the current goal. In case a predicate does not appear on the add-list of any operator the corresponding code sequence for the predicate will contain the sequence for simple-establishment followed by the instruction *Fail*.

The code for each operator consists of a sequence of code groups which each set up the registers for a goal precondition and then call the appropriate predicate code entry point, followed by a block responsible for the effects of the operator. The code for the effects of the operator treats delete effects first and then add effects, updating the state stack accordingly. There is no explicit conjunction-check as is common in linear planners. Instead, propositions which are used to satisfy goals (and preconditions) are *marked* and operators are not allowed to delete marked propositions. This prevents harmful interactions from undoing goals and means that there is no need for a conjunction-check prior to step execution. This process is equivalent to creating a causal-link between the achieving step and the goal in SNLP/ UCPOP-style (Penberthy & Weld 1992).

If an operator attempts to delete a marked proposition in the state then it must unwind its effects and a

backtrack is then executed. Once the add effects of an operator have been completed then unwinding the operator becomes slightly more involved, since then state must be recovered in two stages - removal of added effects and then restoration of deleted propositions.

The search machinery of the planner is depth-first, but it uses a staging parameter, so that the search cannot go deeper than the current depth constraint. The current implementation of the planner does not increment this parameter automatically, so the user must rerun the planner with successively larger values if a plan is not found at a given depth - but a modification to allow automatic iteration would be a very straightforward change to the existing code. Because depth-first search does not necessarily find the shortest plan first, the planner will not automatically report an optimal plan. However, successively reducing the search depth parameter will, of course, enable the user to find an optimal plan once a plan has been found. Equally, incrementing the search depth automatically from 1 would, in principle, allow optimal plans to be found. Of course, the high cost associated with such slow deepening of the search and repeated construction of upper parts of the search space make this search control strategy an unlikely candidate for continued development.

The management of initial state and of goals was considered carefully, and it was decided that both of these components should be separated from the main body of the domain description. Therefore, the compiled planners do not include specific initial states or goals and these are both entered at execution time making the compiled planners flexible. This appears to be an important decoupling if the compiler overhead is to be considered an off-line cost, since it is unlikely that many planners would be run repeatedly from the same initial state, or with the same goal sets. The initial state is entered as a file given as a command-line argument to the planner when it is invoked (together with optional arguments which set run time parameters such as search depth and stack sizes). Goals are entered interactively to a prompt once the planner is running, although this input scheme could, of course, be modified very easily.

Code Generation

The target language selected for the compiler is C++. This has the advantage of being extremely portable. Planners compiled using the planner-compiler have been run on a PC under Borland C++ and on Sun workstations using UNIX with Gnu C++. The compiler currently produces a single file output with the entire self-contained C++ source for the planner in it. Typical test-bed domains produce executables of some 40Kb.

The planning domain descriptions are compiled into PAM-code which is then placed in C++ procedures enabling entry points at the necessary points within the PAM-code. An inner-interpreter loop is used to control the PAM-code, which controls jumps to the PAM-code entry points. This uses a clever trick to avoid growth of the C stack which was used in the Glasgow Haskell to C compiler (Peyton Jones 1993) and attributed by its author to (Steele 1978).

The compiler itself is written in Haskell. It constructs PAM-code as an intermediate stage and then outputs this as C++. Much of the translation of the PAM-code to C++ is actually trivial, since the C++ routines themselves are almost entirely written as PAM-code. The C++ code contains definitions of the PAM operation codes which expand on C++ compilation. Under Gnu C++ the compilation of the planner code is completed using the *caller-saves* option which generates marginally more efficient code for the inner interpreter loop.

Performance

With the optimisations described below, planners have been constructed and compiled for a range of domains, including familiar test-bed domains, such as Towers of Hanoi, Blocks World, the Ferry Domain and The Molgen Domain. Performance figures are promising, but it should be emphasised that the figures provided here are not intended to demonstrate that the PAM should be treated as a competitor, in its current form, to UCPOP or other public-domain planning-systems. Rather, it is intended to give a clear guide to performance and to indicate the initial success of the project and to support the intention to continue to explore this avenue of research. Interestingly, a search of web-sites has not revealed the existence of any live-links to simple linear planners, in order to guage more directly the advantages of compilation. Therefore, the intuitions regarding the benefits of compilation remain to be fully tested, but the analogy between the PAM and the WAM is strong enough that all of the improvements of compiled Prolog over interpreted Prolog can reasonably be expected to apply.

All of the compiled plannners are produced from simple declarative domain descriptions with no domain-specific heuristic additions to the code (although with heuristic optimisations compiled in as discussed below). Specific results in comparison with UCPOP and Graphplan (Blum & Furst 1997) can be found in the following table (figure 1). This comparison is made because these planners represent key leading performance indicators, but it should be emphasised that the comparison should be seen in the light of the fundamental limitations of linear-planning as a planning strategy. The comparison clearly indicates the potential for development of the compilation approach, without implying that this approach has somehow overcome the usual constraints on linear-planners. The basic mechanisms of the PAM can be exploited across other planning algorithms and, indeed, progress has already been made on simple modifications to the machinery to

Problem	UCPOP	Graphplan	Compiled Linear Planner
Ferry	0.49	0.03	0.03
Molgen	0.83	0.27	0.04
Roads	0.02	0.01	0.01
Hanoi-3	80.13	0.13	0.02
Hanoi-4	*	1.54	4.86‡
Monkey-1	0.14	0.07	0.02
Plumbing	0.02	0.05†	0.01
Sussman	0.04	0.04	0.06
Invert-3	0.06	0.05	0.07
Invert-4	0.43	0.18	0.51
Robot-1	0.02	0.04†	0.04
Robot-2	9.76	0.07†	0.04
Fridge-1	0.42	0.54	0.02
Fridge-2	*	1.45†	0.02
Tyre-1	0.01	0.04†	0.02
Tyre-2	0.02	0.02†	0.02
Tyre-3	0.66	0.07†	0.03
Tyre-4	0.03	0.02†	0.05

Figure 1: Performance

Comparison of results: Figures for UCPOP and Graphplan are taken from (Gazen & Knoblock 1997), and are reported for a execution on a SUN ULTRA/1. The compiled linear planner figures are measured user-time for execution on a SUN SPARC/10 (the ULTRA is about 1.5 times faster than a SPARC/10).

* : These problems were not solved within the resource constraints set for UCPOP.

† : These figures include a small amount (typically 0.01s) involved in a preprocessing phase to allow the expressive power of Graphplan to be extended.

‡ : All the figures presented for the compiled planner are taken with plan depth search set to a reasonable margin (beyond the actual plan length), and without using the facility to control resources used in separate goals. By sharing resources in the Tower of Hanoi problem, Hanoi-4 can be solved in 1.01 seconds.

offer a compilation strategy for the graph construction process used in Graphplan.

Since the compiler produces C++ output, it is possible to take the compiled code and extend or modify it to include domain-specific heuristic mechanisms. This is not a trivial task since the code is obviously automatically generated PAM-code, and therefore integrating new code into it or modifying it in any way requires a thorough understanding of the PAM and its operations. Nevertheless, it is feasible to add modifications such as code to make execution of operators also manage physical hardware effectors or simple establishment code include readings of physical state sensors. Code could be integrated to attempt satisfaction of goals by calculation using C++ procedure calls rather than through the standard unification with state propositions. Equally, the search space could, in principle, be manipulated (although it is represented implicitly through the sequence of choice frames on the choice stack) adding domain heuristics to modify or prune the search space.

An Example

To illustrate the compilation and execution processes in a little more detail, a simple example is presented.

Consider the following domain description. The syntax used is exactly that read by the compiler.

```
x::type.(op1(x)
pre:    p(x)
add:    q(x)
del:    p(x)
)
```

```
p/1-0: SE prelude
       ...
p/1-1: Fail

q/1-0: SE prelude
       ...
q/1-1: Countop()
       Alloc(1,1)
       Getvar(0,0)
       Callfor(q/1-2,1,op0)
q/1-2: Realloc(2,1)
       Getvar(0,0)
       Callfor(q/1-3,1,op1)
q/1-3: Dealloc(1)
       Fail

r/2-0: SE prelude
       ...
r/2-1: Fail

op0:   Setvar(0,0)
       Call(op0-1,p/1-0)
op0-1: Effects(1,theop0)
       Setvar(0,0)
       Del(pred_p,1)
       Add(pred_q,1)
       Putarg(0)
       Proceed()

op1:   Setvar(0,0)
       Setvar(1,1)
       Call(op1-1,r/2-0)
op1-1: Setvar(0,1)
       Call(op1-2,p/1-0)
op1-2: Effects(2,theop1)
       Setvar(0,1)
       Del(pred_p,1)
       Add(pred_q,1)
       Putarg(0)
       Proceed()
```

Figure 2: Code produced by the compiler

x::type.y::type.(op2(x,y)
pre: *r(x,y)*
p(y)
add: *q(x)*
del: *p(y)*
)

Note that the values used in the domain are *typed* (in this case with the simple type ''type''). The compiler uses types in disambiguation of instances of operators and predicates, so that two predicates of the same name and with the same argument count will be treated as distinct if they take arguments with different types. The distinction is compiled into the code for the planner and then entry points are chosen appropriately at run time according to the types of arguments in goals.

The code produced by the compiler for this example has the structure shown in figure 2.

This example illustrates the two-block-structure of the code for the PAM, with the proposition satisfaction code first and then the operator code. The code sequences for both the predicates *p* and *r* have no achieving steps, so consist of only simple establishment code and then the instruction *Fail*. The two operators each consist of sequences to set up the calls for the satisfaction of their preconditions and then the code creating their effects. The instructions *Alloc* and *Realloc* are the activation frame allocators. *Dealloc* reclaims the last frame. The *Getvar* instruction is used to set up the appropriate frame entries before an operator is called by the *Callfor* instruction. Note that this instruction includes a continuation amongst its arguments - this is in order to lace together the C++ procedures. The other arguments are the add-list entry being used and the operator entry point. The add-list entry is required to allow protection of the correct effect if the operator is subsequently applied. The operator code uses *Setvar* instructions to set up the goal parameters before *Call*ing the appropriate goal satisfaction code sequence.

Turning attention to execution, consider the following initial state:

p(b::type)
r(a::type,b::type)

and the goal *q(a::type)*.

The goal is set up by placing the constant *a* in the first argument register and then the code entry point for predicate *q* is called (*q/1-0*). Simple establishment fails on this goal, so the code sequence starting at *q/1-1* is executed. This begins by incrementing the operator count (*Countop()*). This instruction causes backtracking if the current search depth limit is exceeded. The *Alloc* instruction that follows it creates a new activation frame, its arguments indicating how many entries to place in the frame and how many arguments there are to the current predicate. This is used to allow the arguments to be stored in case they must later be retrieved for backtracking.

The *Getvar* instruction loads the argument from the register into the activation frame for the operator and then *Callfor* enters the operator code (in this case *op0*). On entry to the operator code (*op0*) the entry in the activation frame is used to set up the register for the call to achieve the first (and only) precondition - *p(a)*. Calling the entry point for *p* (*p/1-0*) executes an attempt to simply-establish the goal, which fails, and then the instruction *Fail* is executed. This causes a backtrack to

```
whitby:dcs0dpl[7]: compile
Compile which operator set? testops
Output file? t.cc
Done
whitby:dcs0dpl[8]: g++ -O3 -fcaller-saves t.cc -o
testplan
whitby:dcs0dpl[9]: time testplan test.ini
Reading initial state from test.ini
      p(b)
      r(a,b)
Enter goals, one per line (Cntl-D or '-' terminates):
q(a::type)
-

Final plan:
      op2(a,b)

0.00u 0.06s 0:08.83 0.6%
whitby:dcs0dpl[10]:
```

Figure 3: Example of use

the last choice point, which was the point at which *op0* was called in attempting to satisfy *q(a)*. The next continuation is *q/1-2*, which reallocates an activation frame - this time containing two entries (the values used in *op1*) and then sets up the first of these to be the register value (constant *a*). The entry point for *op1* is then called.

In similar fashion to the code for *op0*, this sets up the first precondition for *op1*, *r(a,?)*, and calls the code for satisfaction of it (*r/2-0*). The simple-establishment successfully achieves this, binding the second register (and therefore the second entry in the activation record) to be the constant *b*. The simple-establishment prelude then enters the continuation for the operator, *op1-1*, which sets up the precondition *p(b)* (the second entry in the activation frame having now been set to *b*). The attempt to simply-establish this is also successful, and execution continues from *op1-2*. This entry is the effect code for *op1*. It begins by recording the fact that operator op1 is being executed (*Effects(2,theop1)*), so that the finished plan can later be reported. The first argument to *Effects* is the number of preconditions of the current operator. This information is required so that preconditions can be unmarked prior to deletion. This is important not for the deletion itself, but so that the marks can be restored on backtracking in order to ensure that preconditions are protected during the satisfaction of later preconditions of the same operator. Following this, the registers are set for the delete effects. Each deletion is achieved by a similar mechanism to the simple-establishment code - the current state is searched for a matching proposition and then the mark is checked to determine whether deletion is safe. Provided it is possible to delete it, the proposition is then unlinked from its appropriate doubly-linked list. If a match is not found in the state the deletion is ignored. This means that it is possible (intentionally) to include on delete lists effects which are not necessarily present within the state at the point of execution. Add effects are achieved by creating a new entry in the state stack, linked to the correct predicate list (*Add(pred_q,1)*). The second argument for *Add* is used to identify the distinct add effects of an operator. This value is compared with the second argument of the *Callfor* instruction in the proposition achievement code which initiated the operator code call. When a match is made the add list entry is marked as it is created to ensure that the effect is protected against subsequent deletion. The arguments of the add list effect are placed onto the state stack by simply pushing the correct entries from the activation record onto the state stack.

In this example, this operator concludes the achievement of the final goal and therefore ends the planning process altogether. The final step is to report the plan.

The practical process of compilation and execution of this example is illustrated in the session illustrated in figure 3.

A fragment of the C++ produced for this example is as follows:

```
long i15()
{
        Countop();
        Alloc(1,1);
        Getvar(0,0);
        Callfor(i16,1,op0);
};
long i16()
{
        Realloc(2,1);
        Getvar(0,0);
        Callfor(i17,1,op1);
};
long i17()
{
        Dealloc(1);
        Fail();
};
```

This corresponds to the sequence *q/1-1*, *q/1-2* and *q/1-3*. The procedures return long values which are actually pointers to continuation code. The inner interpreter loop receives these continuations and invokes them.

Optimisations

The planner-compiler has been optimised in three ways. The first is that the staged depth-first search mechanism has been refined so that it is possible to associate search depths with the initial goals separately, where a conjunction of goals is set. This is illustrated in the following example.

Operators:

a::block.b::block.c::block.(Move(a,c)

pre: *on(a,b)*
clear(a)
clear(c)

add: *on(a,c)*
clear(b)
clear(a)
clear(table::block)

del: *clear(a)*
clear(c)
on(a,b)
clear(table::block)

)

Initial state:

on(a::block,table::block)
on(c::block,a::block)
on(b::block,table::block)
clear(b::block)
clear(c::block)
clear(table::block)

Goal:

on(a::block,b::block):3
on(b::block,c::block):0

The goals have been annotated with a search depth indicating that the first goal must be satisfied with 3 steps and the second goal within 0 further steps after the first goal has been satisfied. This ensures that the planner will not attempt to uselessly satisfy the second goal by changing the state following satisfaction of the first goal. In other words, it is a signal to the planner that the second goal must actually be satisfied during satisfaction of the first goal. Of course, the planner can be executed without giving this information, causing it to find the plan only after a longer search. Similarly, giving a larger search bound will, in this example, possibly yield a sub-optimal plan.

Although this information is often not available as accurately as in this trivial example, the ability to divide resources between goals, in order to restrict effort invested in the satisfaction of whhat is considered a trivial goal, is a powerful possibility.

It is worth observing, lest the reader's suspicions be aroused by the example, which bears a striking resemblance to Sussman's anomaly (notoriously unsolveable by linear-planners), the operator is defined to ensure that the *table* is always clear and it also, slightly counter-intuitively, deletes and then reasserts that the block being moved is clear. This device ensures that the given goal can be satisfied optimally by allowing movement of *b* onto *c* to be used as a way to satisfy the sub-goal *clear(b)* in preparation for the movement of *a* onto *b*.

The second optimisation is the inclusion of *hints* into the planning domain specification. Careful analysis of the planner's behaviour on tower of hanoi problems showed that the planner attempts many redundant operators which simply move a disc twice in a row. This dramatically increases the size of the search space. If the planner is told that there is never any point in moving the same disc twice then this improves its efficiency by several orders of magnitude. More generally, many domains contain pairs of operators which should not appear adjacent to one another, instantiated in particular patterns, in efficient plans, so these hints have the power to improve the planning process quite significantly. It is interesting to observe that this heuristic is one of those employed as

domain-dependent heuristics (LIMIT-USELESS-MOVES) to control the planners synthesised in the CLAY system, described in (Srivastava, Mali, & Kambhampati 1997).

The way in which this restriction is achieved includes three parts: the declarative component which affects the writing of domain descriptions, the compilation part which affects what code is generated for these hints and finally the execution process for the PAM.

The hints are provided very simply in the declarative form. An example for the blocks world specification provided above is to amend the domain specification with the hint:

Never Move(a,b) then Move(a,c)

Generally, a hint takes the form:

Never $op0(x_1,\ldots,x_n)$ then $op1(y_1,\ldots,y_n)$

where each y_i may be an instance of some x_j.

Hints are currently restricted to only allow overlap of variables between the first and second operator references, so, for example, the hint:

Never Move(a,a) then Move(b,c)

would not be allowed.

The compiler compiles code so that when an operator is about to be executed a check is made that it is not the second half of one of these hint-pairs. If it is, then a further check is made to ensure that any common variables in the two operator references in the hint are distinct values in the two operator application. If this check fails then the planner backtracks without applying the operator. In practice, this check is compiled into the satisfaction of the final precondition (if there is one) and a small further optimisation is used to prevent the satisfaction of this precondition by simple-establishment if these checks fail. This short-circuits a certain amount of work in satisfying the final precondition. It has proved frustrating that the check establishing that a hint is being violated cannot apparently be usefully carried out any earlier than the moment almost immediately before execution of the operator. This means that significant work might be done to satisfy the preconditions of an operator which will simply not be allowed because of the hints.

A final optimisation in the current version of the planner is to note which preconditions are deleted by an operator so that simple-establishment of these preconditions will not use state propositions which are marked as protected. Correspondingly, these effects can be safely deleted by the operator without checking wether they are marked, since the check is done at the time when the precondition is first satisfied. This optimisation is achieved by the compiler, which uses a different deletion operation code and a different simple-establishment prelude entry point for propositions that are affected.

These optimisations have proved successful in improving performance of the compiled planners considerably, so that even in a very hard domain, the towers of hanoi, the 31 step solution to the 5 disc problem can be solved in less than 30 seconds. This is a considerable achievement for pure linear planning technology.

Conclusions

The chief contribution of the work described here is to demonstrate that compilation of planning algorithms is both possible and, as might be anticipated, offers significant efficiency gains. In addition to demonstrating the feasibility of planner compilation, the work opens some exciting new avenues for further research. Initially, there are possibilities for further optimisation of the linear planner compiler output and execution of the PAM code. There are also possibilities for extension of the expressiveness of the domain description language. In particular, it would appear relatively straightforward to include free variables in delete effects (to achieve the same effect as $ variables in delete lists of the original STRIPS planner (Fikes & Nilsson 1971)). Existentially quantified variables in top-level goals should also be possible. Other extensions along similar lines to those in the UCPOP planning language might well be worth exploring (Penberthy & Weld 1992).

A second, and probably more rewarding, avenue is to explore compilation into different planning technologies. Non-linear planning using causal links has already been considered and initial design work for an appropriate abstract machine architecture has been conducted. An interesting possibility which is also being pursued is to reuse the compilation techniques described here in the compilation of a Graphplan-style planner, compiling the graph-construction phase. These technologies present major challenges to the process of compilation, of course, but the several orders of magnitude performance boost achieved by compilation makes the possibility an exciting one which could open up whole new application areas for domain-independent planning.

The compiler has already proved a valuable tool in simply exploring the structure of the search space associated with various domains - it being a straightforward task to introduce callibration mechanisms into the planner code which allow carefully controlled viewing of the search process and the areas in which the greatest search effort is devoted (indeed, the profiling of the generated C++ is already a significant source of information). The speed of the compiled code and its flexibility have made this exploration a much more practical possiblity than would be the case with the code for a single large domain-independent planner. This exploration can, in turn, feed back into the construction of improvements in the efficiency of the planning machinery.

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281-300.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem-proving to problem-solving. *Artificial Intelligence* 2(3):189-208.

Gazen, B.C., and Knoblock, C.A. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proceedings of European Conference on Planning*, Toulouse.

McCluskey, T.L. and Porteous, J.M. 1994. Learning Heuristics for Ordering Plan Goals through Static Operator Analysis Methodologies for Intelligent Systems. *Lecture notes in Artificial Intelligence* 869, Springer-Verlag.

McCluskey, T.L., and Porteous, J.M. 1995. The Use of Sort Abstraction In Planning Domain Theories. In *Planning and Learning: On to Real Applications. Papers from the 1994 AAAI Fall Symposium.* American Association for Artificial Intelligence (AAAI Press), ISBN 0-929280-75-X , FS-94-01.

McCluskey, T.L., and Porteous, J.M. 1996. Planning Speed-up via Domain Model Compilation. In *New Directions in AI Planning, Frontiers in AI and Applications Series* 31, IOS Press, Amsterdam, Holland, ISBN 90-5199-237-8.

Penberthy, J.S., and Weld, D.S. 1992. UCPOP: A sound and complete partial order planner for ADL. In *Proceedings of KR'92 - Third International Conf. on Principles of Knowledge Representation and Reasoning.*

Peyton Jones, S.L. 1993. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, version 2.5. *Release documentation for Haskell, University of Glasgow*.

Smith, D.R. 1992. Structure and design of global search algorithms. *Kestrel Tech. Report,* KES.U.87.11.

Smith, J.J.; Nau, D.S., and Throop, T. 1996. A planning approach to declarer play in Contract Bridge. *Computational Intelligence* 12(1): 106-130.

Srivastava, B.; Mali, A.D., and Kambhampati, S. 1997. A structured approach for synthesising planners from specifications. In *Proceedings of 12th IEEE Intl. Conf. on Automated Software Engineering, Lake Tahoe.*

Steele, G.L., 1978. Rabbit: a compiler for Scheme. AI-TR-474, MIT Lab for Computer Science.

Warren, D.H.D. 1983. An abstract Prolog instruction set. Report 309, AI Centre, SRI International, Menlo Park, CA.