A General Approach to Synthesize Problem-Specific Planners

Okhtay Ilghami and **Dana S. Nau**
Department of Computer Science, Institute for Systems Research, and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742-3255 USA
{okhtay,nau}@cs.umd.edu

Technical Report CS-TR-4597 and UMIACS-TR-2004-40
October 27, 2003

Abstract

In this paper, we describe a way to improve the performance of hand-tailorable planners by compiling each domain description into a separate domain-specific planner. We discuss why and when this approach can be useful, and we present experimental results showing that our approach produces significant increases in the speed of planning.

Introduction

In this paper, we describe a way to improve the performance of hand-tailorable planners by compiling each domain description into a separate domain-specific planner.

So far, hand-tailorable planners have mainly been considered to be *interpreters* of their domain description language. In our approach, the planner is a *compiler* of its domain description language. The input to a planner is a domain description like the one in an ordinary hand-tailorable planner, and the output is a program specifically generated for that planning domain. The program can then be run to solve problems in that domain. This can give the planner all of the advantages that a compiled program can have over an interpreted program.

This is a general technique that in principle can be applied to any hand-tailorable planner—and by compiling domain descriptions directly into low-level executable code, we can do implementation-level optimizations that are not otherwise possible and have not been explored in previous research on AI planning. These optimizations can be coupled with the other speed-up techniques that the AI-planning community has developed (domain analysis and other automated domain information synthesis techniques) in order to obtain additional speedups.

In this paper, we describe the compilation process, and provide empirical results to show the improvements made possible by our approach.

Background

Hand-Tailorable Planning

Existing planning technology can be divided into the following categories:

- Domain-specific planners, which only work in a single problem domain. In domain-specific planning, a planner is built from scratch for each new problem, using techniques specific for that problem to improve the efficiency of the planning process. Examples include ICAPS (Nau, Regli, & Gupta 1995), Remote Agent (Muscettola *et al.* 1998), Bridge Baron (Smith, Nau, & Throop 1998), and Mars Rover (Dias, Lemai, & Muscettola 2003).
- Fully automated domain-independent planners (e.g., nearly all classical planning systems). The biggest advantage of using such planners is that by abstracting the planning mechanism, they provide the user with the ability to reuse them for other domains without requiring much effort. However, as shown in (Bylander 1991; Erol, Nau, & Subrahmanian 1995), even with reasonable restrictions, general-purpose planning is at least PSPACE-hard. This makes general-purpose planning a less attractive choice in real-world domains.
- Hand-tailorable domain-independent planners. A hand-tailorable planner is usually a general-purpose planner. However, in contrast to automated planners where the domain description contains only the planning operators, hand-tailorable planners provide a richer domain description language so that the input to the planner can include advice to the planner on how to search for a plan. Hand-tailorable planners can be described therefore as domain-independent planners which give their user the option to specify domain-specific advice in domain description.
  Most hand-tailorable planners are Hierarchical Task Network (HTN) planners; some of the best-known examples include Nonlin (Tate 1977), SIPE2 (Wilkins 1990), O-Plan (Currie & Tate 1991; Tate, Drabble, & Kirby 1994), UMCP (Erol, Handler, & Nau 1994), and SHOP2 (Nau *et al.* 2003). More recently, hand-tailorable planners have been built in which the domain knowledge consists of control rules; the best-known examples are TLPlan (Bacchus & Kabanza 2000) and TALplanner (Kvarnström & Doherty 2001).

Hierarchical Task Network Planning

Although the compilation technique described in this paper can be applied to any hand-tailorable planner, we chose to apply it to SHOP2, an HTN planner. Because of this, understanding how the compilation process works will require the reader to understand what HTN planning is., so we now

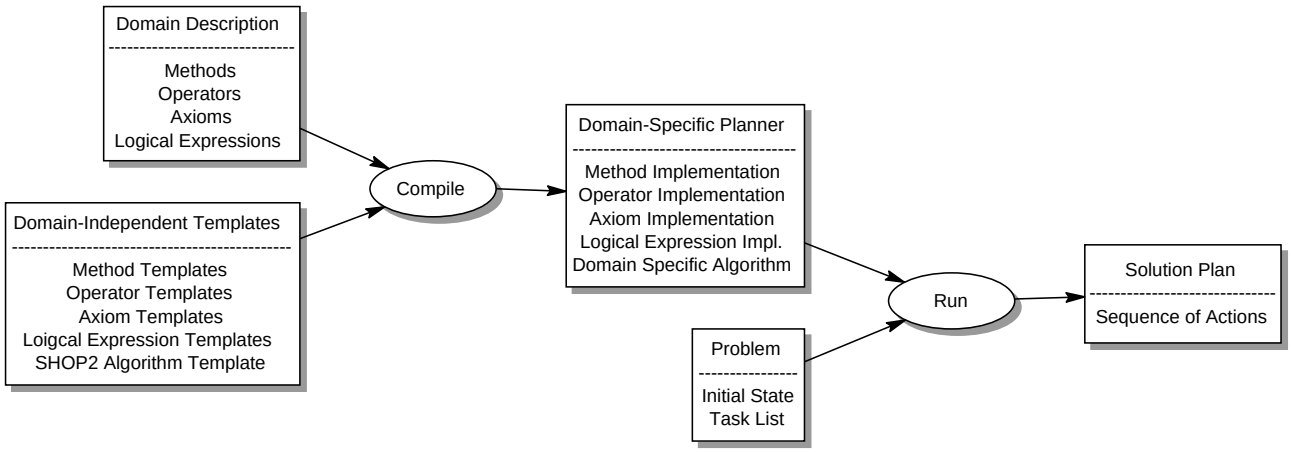

Figure 1: Compilation Process

describe it briefly.

In an HTN planning system, instead of the traditional goal formula, the objective is to accomplish a partially-ordered set of given *task*s. Each task can be decomposed into several *subtask*s using predefined *methods*. Each possible decomposition represents a new branch in the search space of the problem. At the bottom level of this hierarchy lie *primitive tasks*, whose actions can be executed using atomic operators. The plan still consists of a sequence of actions, but the correctness definition of the plan differs. In traditional planning, a plan is correct if it is executable, and produces a state that satisfies the goal formula. In HTN planning, a plan is correct if it is executable, and it accomplishes the given tasks. In other words, the main focus of an HTN planner is to perform tasks, while a traditional planner focuses on achieving a desired state.

Compilation Process

The compilation process is depicted in Figure 1. We have implemented it in Java; our system is called JSHOP2. As can be seen, JSHOP2 has two different sets of inputs:

- First, a domain description is fed to JSHOP2. JSHOP2's domain descriptions are identical to those of SHOP2, and are composed of methods, operators, and axioms. Both the methods and operators include logical expressions that describe their preconditions. The axioms are horn-clause like statements that can be used to infer preconditions that are not explicitly present in the current state of the world. JSHOP2 contains domain-independent templates for methods, operators, axioms, and logical expressions. Each time one of these elements (method, operator, axiom, or logical expression) occurs in the domain description, JSHOP instantiates the corresponding template accordingly. The result is a piece of code that implements the behavior of the corresponding element. JSHOP2 also has a template for the general SHOP2 planning algorithm, and it instantiates and tunes this template according to various characteristics of the domain. Thus, the resulting code base includes the code for each element of the domain, and a specific instance of SHOP2 algorithm for that domain.
- To solve the actual problems in a compiled domain, one has to compile the problem description. Again, there is a problem template that is instantiated according to the problem description. The result is an executable code which can be run to solve the problem. This executable code makes use of the domain code base created in the last step, and also a small library of domain-independent code that includes basic operations and data structures that are common to every SHOP2 domain.

In much of the work done on planner compilation and synthesis (some of which are discussed in the related work section), the planners are usually generated in some kind of an intermediate abstract language. This intermediate code can then be translated into a lower-level language to be executed. In contrast, we decided to produce our planner directly in a low-level executable language (namely Java). This gave us much greater power and flexibility to optimize the planner. Here are two examples of optimizations that this decision has made possible:

- *Using static data structures instead of dynamic ones:* Several of the elements in the SHOP2 planning process are maintained in dynamic data structures such as lists rather than static ones such as arrays. This is the case mainly because SHOP2 is a general purpose algorithm and does not know in advance how big these elements are going to get, so it cannot use static data structures. This is a huge waste of memory and time in many cases, because very often the size of these elements can be determined given the domain description. So by compiling those domain descriptions, we acquire the ability to fix the size of those elements at compile time and thus use more efficient arrays rather than lists. Being able to simplify the data structures in hand-tailorable planners is of great importance because usually, a good domain description in such planners is very elaborate and complex and therefore requires complex data structures.

- *Producing customized backtracking code:* Implementing backtracking has been a difficulty for planner designers, because of the fact that whenever a planner backtracks on a decision, it needs to undo all the changes made by that decision to the state of the world. This problem is even worse for HTN planners like SHOP2 because they have to keep track of a task network also, and task networks are much more complicated data structures compared to states of the world which are merely a collection of atoms.

  There are generally two different approaches to address this issue: One is two copy the entire data structure for the state of the world (and task networks for that matter) whenever a decision is made and use that copy from there on, and simply discard the copy and use the original version in case of a backtrack. The second approach is to design the data structures in a way that changes made to them can be kept track of and be undone in case of a backtrack. The first approach uses a lot of memory, while the second approach uses a lot of time (because the data structure is much more complex and therefore it is harder to change or update it). The first approach is more useful if there is not that much backtracking while the second approach is more useful if there is lots of backtracking and therefore there is not enough memory to copy the entire data structure over and over again. Moreover, if there are decision points where planner can make sure that no backtracking is going to happen, there is no need to keep track of anything. While a general purpose planner uses the same technique for keeping track of data structures at each decision point, a problem-specific planner can adjust its technique for each specific decision point. In our system, we introduce a family of data structures each of which is suitable for decision points where backtracking happens with a certain frequency, and according to our assessment of the frequency of backtracking in each decision point, we use one of those data structures in the produced code.

Experimental Results

To test JSHOP2, we did experiments comparing it to two other planning systems, both of which are available from their authors as open-source software:

- JSHOP. This is a Java implementation of the SHOP planning algorithm (Nau *et al.* 1999). As input, it takes the same domain descriptions as SHOP. These domain descriptions are a restricted version of SHOP2's domain descriptions, the biggest restriction being that all sets of tasks must be totally ordered.
- SHOP2 (Nau *et al.* 2003). This planner is a Lisp implementation of the planning algorithm described in (Nau *et al.* 2001). It has more capabilities than the original SHOP algorithm, the main enhancement being that sets of tasks may be partially ordered so that subtasks of different tasks can be interleaved.

Below we describe the reasons for choosing these planners for our comparison.

Comparing SHOP2's and JSHOP2's running times should give a rough sense of what kind of a speed-up can be achieved using our approach, because SHOP2 and JSHOP2 are essentially two different implementations of the same planning algorithm—the difference being that one acts as an interpreter of its input and the other one acts as a compiler of its input.

If the only experimental comparison were between SHOP2 and JSHOP, it would be difficult to tell whether the difference in speed between JSHOP2 and SHOP2 was because of compilation versus interpretation, or because of speed differences between the languages in which they are implemented (JSHOP2 is implemented in Java and SHOP2 is implemented in Common Lisp). To address this issue, we also wanted to compare JSHOP2 with a Java implementation that runs interpretively the way SHOP2 does. The only available such implementation was JSHOP.

Since JSHOP is an older implementation, it does not have all the capabilities of SHOP2. Many of the domain descriptions that were written for SHOP2 cannot be run with JSHOP, nor can they be adapted to run with JSHOP, because those domain descriptions require some of the methods to have partially ordered subtasks. Thus, in order to do a full comparison, we were forced to run our experiments on two older domains: blocks world and logistics.

For our experimental comparisons, we used two different versions of each domain: (1) a simple and relatively straightforward version of the domain description, and (2) a second and more sophisticated version that contained some extra information to guide the planner in order to improve its efficiency. The reason for this was to investigate whether the optimizations made by our approach were independent of the kinds of optimizations a knowledgeable domain author might make (in which case JSHOP2 should provide the same kinds of speedups on both versions of the domain description), or were optimizations that a knowledgeable domain author would write anyway (in which case the JSHOP2 would give less of a speedup on themore sophisticated version of the domain).

The results are shown in Figures 2, 3, 4 and 5. Each data point time is an average of time spent to find a plan on ten different randomly-generated problems of the given problem size. In the blocks world, the problem size is the number of blocks in the world. For logistics, the problem size is the number of packages to be delivered. These experiments were conducted on a Sun Ultra 10 machine with a 440 MHz SUNW UltraSPARC-IIi CPU and 128 megabytes of RAM.

Several observations can be made:

- JSHOP is considerably slower than both SHOP2 and JSHOP2, and it fails to find plans for several of the bigger planning problems. This is partly because JSHOP is an older implementation and thus it does not include many of the improvements included in SHOP2. The other reason is that because of the nature of SHOP planning algorithm, Lisp is probably a better language in which to implement it. Therefore, to say that JSHOP2's speed is just a result of migrating from Lisp to Java is not a valid argument.
- JSHOP2 is as fast as SHOP2 in smaller problems and

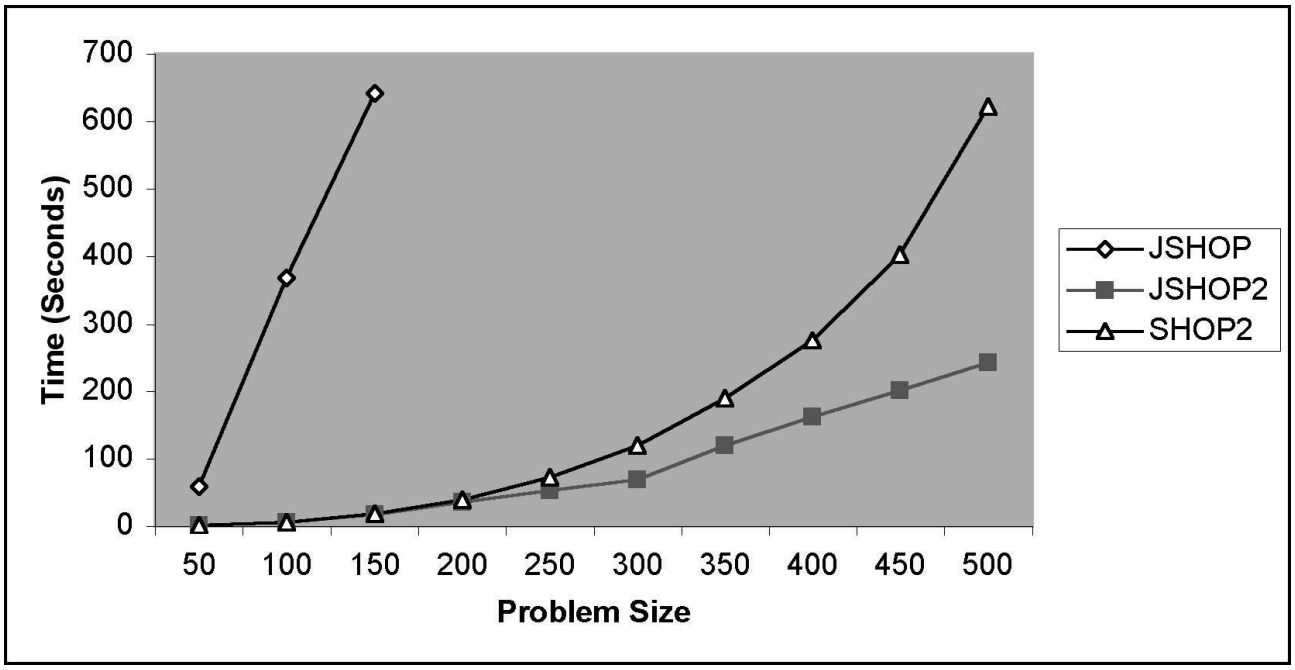

Figure 2: Blocks World, Simple Implementation

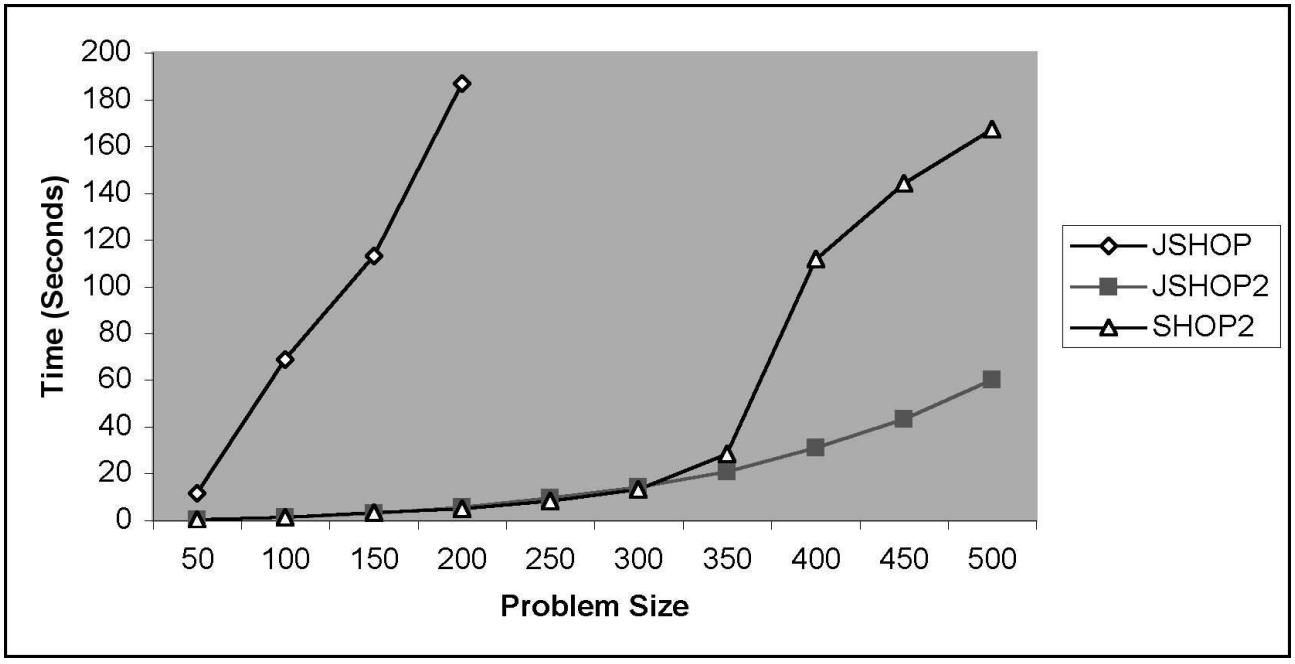

Figure 3: Blocks World, Optimized Implementation

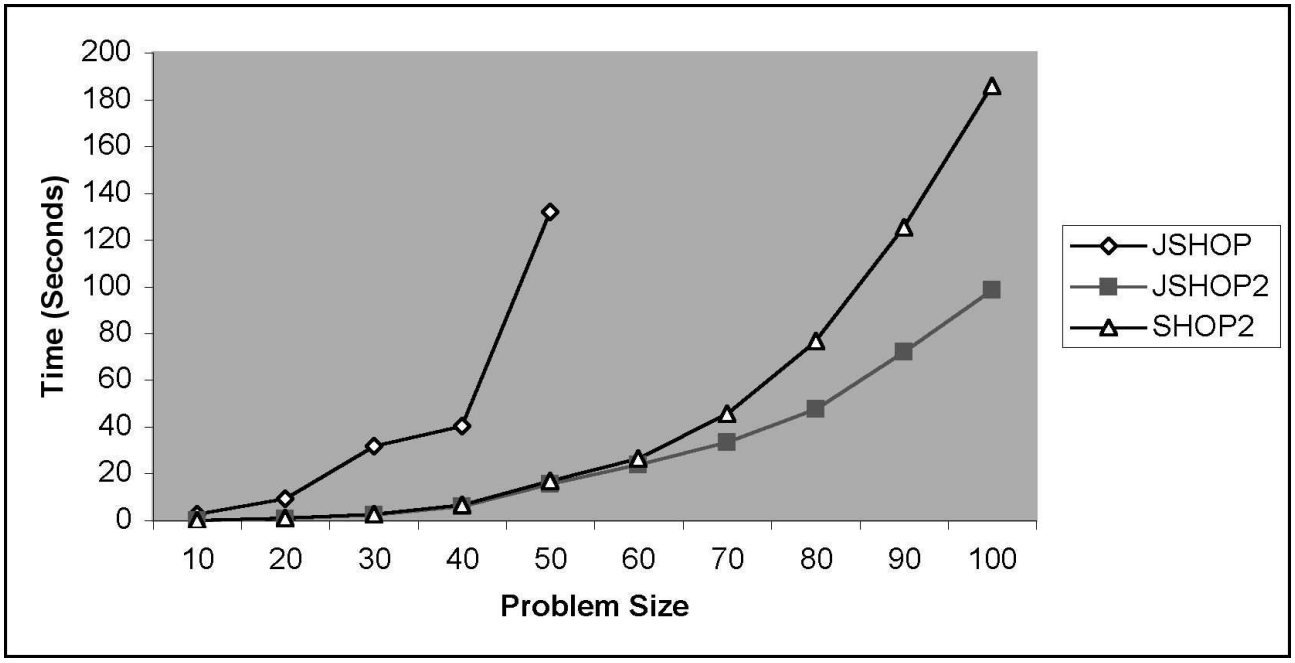

Figure 4: Logistics, Simple Implementation

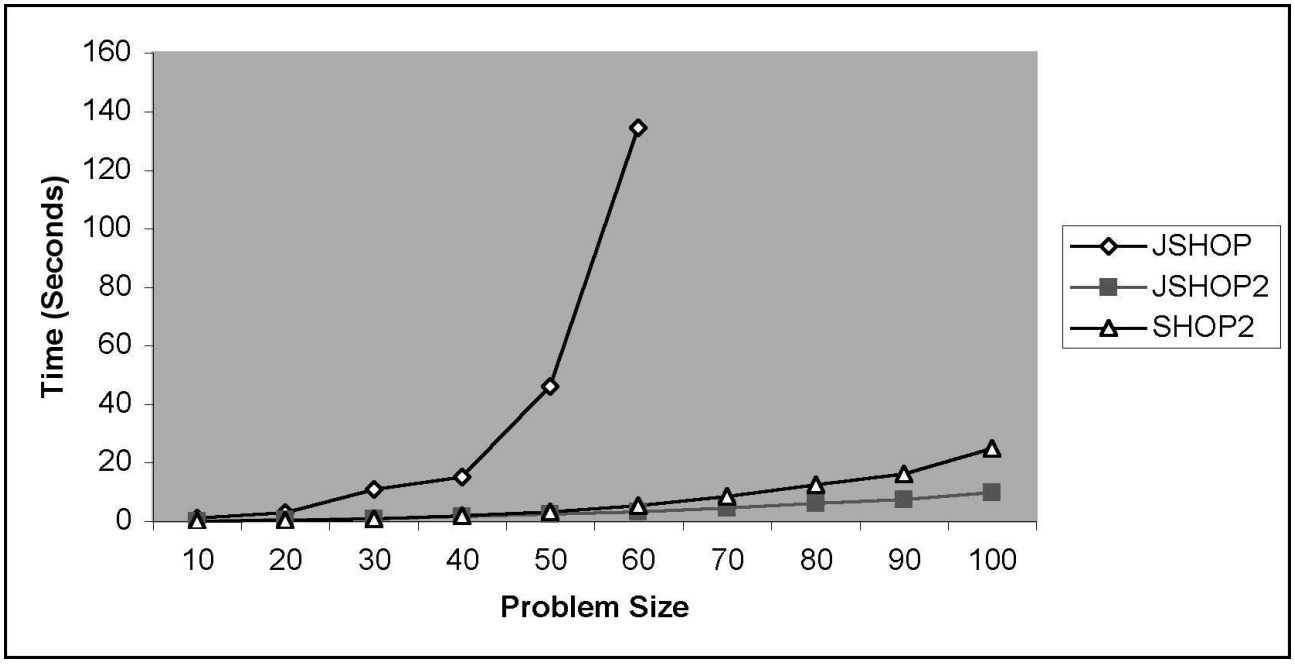

Figure 5: Logistics, Optimized Implementation

faster than SHOP2 in bigger problems in both domains, and in both versions of each domain. This suggests that the improvements that can be achieved by having a better domain description are independent of the improvements that can be achieved using our approach.

- In several of the experiments, the gap between SHOP2's and JSHOP2's running time increases by more than just a constant factor as the problem size increases. This reinforces the conclusion that JSHOP2's speed-ups are caused by fundamental differences between two approaches, rather than just by using Java instead of Lisp.
- JSHOP2 only exhibited a polynomial-time improvement over SHOP2—but such a speedup is quite significant since both planners are already running in polynomial time on these problems. In the last planning competition (see http://www.dur.ac.uk/d.p.long/competition.html), SHOP2 and the other two hand-tailorable planners often ran in polynomial time. The amount of improvement is enough that if JSHOP2 had been used in the last planning competition rather than SHOP2, it would have made a huge difference in the outcome of the competition.

Related Work

Several researchers have worked on automatically synthesizing domain-specific information to improve the performance of domain-independent planners and schedulers:

Srivastava, Kambhampati, and Mali in (Srivastava, Kambhampati, & Mali. 1997) describe CLAY. CLAY is a planner synthesis tool based on a semi-automated software synthesis system, namely KIDS (Kestrel Interactive Development System) (Smith 1992). In this work, a *refinement planning theory* is combined with domain-specific knowledge to derive a domain-specific planner in REFINE, which is a first-order logic language. The generated code can then be translated into Common Lisp or C. KTS (Kestrel Transportation Scheduler) (Smith & Parra 1993) is another system based on KIDS which is aimed at deriving domain-specific schedulers from domain descriptions.

Planware II (Becker, Gilham, & Smith 2003) and its predecessor Planware (Blaine *et al.* 1998) are integrated development environments that are used to model complex planning and scheduling problems and deriving domain-specific code based on those models to solve those problems.

In (Long & Fox 1998), Long and Fox introduce Planning Abstract Machine (PAM). PAM is essentially a compiler for declarative planning domain description. It outputs a domain-specific planner code in an intermediate abstract language which can in turn be translated into C++ code.

In (Winner & Veloso 2002), plan examples are used to learn *plan templates*. These plan templates are pieces of code that compactly represent the domain-specific information learned during the planning process. These plan templates are then merged and generalized to synthesize domain-specific planners that can then be used to solve new problems in that domain.

One of the biggest challenges of using hand-tailorable planners is that providing them with domain-specific knowledge is a tedious and time-consuming task. To address this issue, several researchers have tried to automatize the process of acquiring such knowledge:

Domain analysis (Fox & Long 2002) is introduced by Long and Fox as a technique to automatically obtain domain-specific knowledge. This technique is based on trying to identify generic patterns of behavior in different planning domains, and then use domain-specific instances of those generic behaviors to guide the planner in its search process. This work has resulted in a system called `TIM` (Fox & Long 1998) (Type Inference Machinery). This system is used in a planner named `STAN` (STatic ANalysis planner) (Fox & Long 2001).

In (McCluskey & J. M 1997), the authors suggest an object-centered (as opposed to literal-centered) specification language to describe planning domains. They also introduce tools to operationalize those domain models. It is worth to remind the reader that McCluskey and Porteous have used the term compilation in their paper to refer to a totally different concept: What they mean by compilation is *domain compilation*, which refers to the process of analyzing a domain description in advance to acquire the information that can be used to guide the planner during its search to find a plan. However, that acquired information should be fed to a general-purpose planner which acts as the interpreter of that information. What we mean by compilation in this paper is *planner compilation*, which refers to synthesis of a planner for a given domain description which can then be run independently to solve problems in that domain. In this paper, the authors also suggest methods to automatically derive *macro-operators* from the domain description. Another system that makes use of such techniques is STATIC (Etzioni 1990) which is part of the PRODIGY architecture (Minton *et al.* 1989).

Conclusion and Future Work

In this paper, we presented the idea of compiling a domain description for a hand-tailorable planner to a domain-specific program. Moreover, we chose to compile domain descriptions directly to executable programs, rather than an intermediate abstract language. We also presented examples of how doing this gives the planner designer much more control over how the planner works. This process is transparent from the point of view of the end-user of the planning system: although the output of this process is a domain-specific planner, the overall process (compiling a domain description and running it) is domain-independent.

Our experimental comparison of JSHOP2 to JSHOP and SHOP2 shows that compilation makes the planning algorithm significantly faster. In fact, the amount of improvement is enough that if JSHOP2 had been used in the last planning competition rather than SHOP2, it would have made a huge difference in the outcome of the competition. Our experiments also show that even when the planner is provided with a good domain description that has been hand-tuned by an experienced human user, our approach still provides a similar degree of speed-up. This important observation suggests two things:

- It suggests that our approach can still be beneficial when

coupled with various other techniques that the planning community has developed for improving the search process by automatically analyzing domain description to gather information that can later be used to guide the planner.

- It also suggests that because of the limitations of interpreters, there are certain improvements that cannot be made by providing a hand-tailorable planner with better and better domain descriptions. In other words, we think that *shifting from domain-description interpretation to domain-description compilation* is necessary for the potential of hand-tailorable planners to be fully realized.

What we presented in this paper is just a first step toward making use of the advantages that compilation has over interpretation. In our future work, we intend to look for common characteristics of all hand-tailorable planners that can be used to provide additional speed-ups. We also intend to extend our compilation process so that it can perform optimizations based on run-time information (such as the statistics about how frequently backtracking happens in each decision point) rather than generating code based on only static analysis of the domain description.

Acknowledgments

This work was supported by the following grants, contracts, and awards: Air Force Research Laboratory F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, Naval Research Laboratory N00173021G005, and the University of Maryland General Research Board. The opinions expressed in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

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

Becker, M.; Gilham, L.; and Smith, D. R. 2003. Planware II: Synthesis of schedulers for complex resource systems. Technical Report KES.U.03.04, Kestrel Institute.

Blaine, L.; Gilham, L.-M.; Liu, J.; Smith, D. R.; and Westfold, S. 1998. Planware: Domain-specific synthesis of high-performance schedulers. In *Proceedings of the 13th Automated Software Engineering Conference*, 270–280. Los Alamitos, CA: IEEE Computer Society Press.

Bylander, T. 1991. Complexity results for planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Currie, K., and Tate, A. 1991. O-Plan: The open planning architecture. *Artificial Intelligence* 52(1):49–86.

Dias, M.; Lemai, S.; and Muscettola, N. 2003. A real-time rover executive based on model-based reactive planning. In *The 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space*.

Erol, K.; Handler, J.; and Nau, D. S. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of The International Conference on AI Planning Systems (AIPS)*, 249–254.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1-2):75–88.

Etzioni, O. 1990. Why PRODIGY/EBL works. In *Proceedings of AAAI*.

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. 2001. Hybrid STAN: Identifying and managing combinatorial sub-problems in planning. In *Proceedings of 17th International Joint Conference on Artificial Intelligence*, 445–452. Morgan Kaufmann.

Fox, M., and Long, D. 2002. Generic types in planning. In Nebel, B., and Lakemeyer, G., eds., *Exploring AI in the New Millenium*. Morgan Kaufmann Publishers.

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

Long, D., and Fox, M. 1998. Domain-independent planner compilation. Technical Report WS-98-03, ISBN 1-57735-056-1, AAAI. AIPS98 Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice.

McCluskey, T. L., and J. M, P. 1997. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence* 95:1–65.

Minton, S. N.; Knoblock, C. A.; Kuokka, D. R.; Gil, Y.; Joseph, R. L.; and Carbonell, J. G. 1989. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

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):5–47.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 968–973. Morgan Kaufmann Publishers.

Nau, D. S.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-order planning with partially ordered subtasks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20.

Nau, D. S.; Regli, W. C.; and Gupta, S. K. 1995. AI planning versus manufacturing-operation planning: A case study. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Smith, D. R., and Parra, E. 1993. Transformational approach to transportation scheduling. In *Proceedings of 8th Knowledge-Based Software Engineering Conference*, 14–17.

Smith, S. J. J.; Nau, D. S.; and Throop, T. 1998. Computer bridge: A big win for AI planning. *AI Magazine* 19(2):93–105.

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

Srivastava, B.; Kambhampati, S.; and Mali., A. D. 1997. A structured approach for synthesizing planners from specifications. In *Proceedings of 12th IEEE International Conference on Automated Software Engineering*.

Tate, A.; Drabble, B.; and Kirby, R. 1994. *O-Plan2: An Architecture for Command, Planning and Control*. Morgan Kaufmann Publishers.

Tate, A. 1977. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 888–893.

Wilkins, D. 1990. Can AI planners solve practical problems? *Computational Intelligence* 6(4):232–246.

Winner, E., and Veloso, M. 2002. Automatically acquiring planning templates from example plans. In *Proceedings of AIPS02 Workshop on Exploring Real-World Planning*.