TL-SHOP: Temporal Logic Control Rules in a Hierarchal Task Network Planner

Term Project Report for CMSC 722, Spring 2007

Author:
Nathaniel Ayewah
ayewah@cs.umd.edu

Author:
Derek Monner
dmonner@cs.umd.edu

April 30, 2007

1 Introduction

Hierarchal Task Network (HTN) planning requires describing a goal state as a set of high-level tasks which will bring about that state, and involves achieving the required tasks by recursively breaking them into smaller and smaller subtasks [4]. Put simply, HTN planning, describes in detail a subset of solutions the planner is allowed to try. On the other hand, the control rule planning paradigm is a standard forward search procedure enhanced by control rules written in a modal logic such as Linear Temporal Logic (LTL) that are used to prune the search space by describing which states are undesirable or will never lead to a solution [4]. In short, control rule planning tells the planner which branches not to follow.

The product of this work, a planner called TL-SHOP, combines these two paradigms in an effort to increase the flexibility afforded to writers of planning domains and problems.

2 Related Work

TLPlan[1] is a well-known implementation of the control rule planning paradigm. Our approach to processing Linear Temporal Logic is based on a simplification of the strategy used in TLPlan[4].

JSHOP2[2] is an existing state-of-the-art domain-configurable HTN planner written in Java. JSHOP2 is unique in that it compiles a planning domain and problem into a domain-specific planner in order to increase efficiency [3]. TL-SHOP builds upon JSHOP2 by adding temporal constraints in Linear Temporal Logic at several levels, as discussed in the following section.

3 Implementation

We augmented JSHOP2 with control rules in three simple ways. First, TL-SHOP allows the addition of global control rules to the specification of the planning domain. As time progresses, if any of the control rules become verifiably false, the planner explores no further down that path. This behavior is analogous to the way control rules are expressed in TLPlan, and allows us to prune partial plans whenever a new operator is inserted (as this changes the current state on which the planner is focusing).

TL-SHOP allows a second type of control rule which is a simple extension of the first: problem-specific control rules. These rules are more or less treated the same as global domain control rules, except that they can be changed from problem to problem without the user needing to change the domain specification. They are meant to allow users to impose additional, non-domain-specific constraints. For example, in the well-known blocks world domain, a user could specify that a certain block must never be placed on the table. This control rule is not appropriate for the blocks world domain itself, since it may remove some or all valid plans, but in effect it creates a customized subdomain for this particular problem instance.

Finally, TL-SHOP allows the specification of method and operator postconditions as LTL formulas. When an operator or method with a postcondition is applied, its postcondition is added to the set of applicable control rules for every path in the search space below the applied method or operator. Returning to the blocks world domain for another example, the $stack(a, b)$ operator might have the postcondition $\Box on(a, b)$ ("always $on(a, b)$"), which is a simple but effective way to satisfy the constraint that if one stacks a block, it should be in its goal location.

The following sections will cover the main pieces of the implementation in detail, as well as some limitations of the software.

3.1 Extended Domain and Problem Parsers

The domain and problem input is provided using a lisp-like syntax which is parsed by an ANTLR parser. TL-SHOP builds on the syntax in JSHOP2 and follows many of the same conventions. Each constraint is comprised of a `constraint` label followed by a logical expression optionally including temporal operators. The temporal logic operators are symbolized by `next`, `always`, `until`, and `eventually` labels. These five new labels are prefixed by a single colon to distinguish them as keywords. In the example above, the postcondition $\Box on(a, b)$ would be encoded as:

```
(:constraint (:always (on ?a ?b)))
```

When this statement is used as a postcondition to a method or operator, the variables are bound to the parameters of the method or operator. If the statement is part of the domain or problem specification, only ground atoms or variables enclosed in a `forall` or `exists` expressions are allowed.

3.2 Linear Temporal Logic Inference Engine

Our LTL inference engine is based on the *progress* function from [4]. Applying an operator during the HTN planning process changes the state of the world, which necessitates a call to the *progress* function. The *progress* function simultaneously evaluates the current set of control rules in the new state, and derives the set of control rules that will be applicable to the next state encountered. It relies on the same atom entailment and variable binding procedures used during the HTN planning process.

In addition to the *progress* function, we were required to implement a function to be called whenever a solution plan is found, that verifies that the final world state is a valid one. The semantics of the $\Diamond p$ ("eventually p") expression require that p become true at some point, at which point $\Diamond p$ disappears from the control rule. As such, an $\Diamond p$ expression present in the control rules progressed through the final state can potentially invalidate the solution plan.

3.3 Limitations

The LTL constraint computations in TL-SHOP are not compiled in the same way that the HTN problems were in JSHOP2. In fact, we feel that the progressions of LTL formulas computed by the LTL inference engine do not lend themselves to compilation in any obvious way. As such, the LTL inference engine in TL-SHOP takes a considerable performance hit because it needs to allocate and deallocate memory during the planning process, which JSHOP2 managed to avoid. This fact renders TL-SHOP uncompetitve with JSHOP2 on large problems.

The LTL operators $\forall$ ("forall") and $\exists$ ("exists") take as their first argument not an unbound variable, but instead a logical atom that contains a single unbound variable. Ideally, this atom has a small number of ground instances, and so the combinatorial explosion generally produced by these statements is controlled.

TL-SHOP does not support function symbols in LTL expressions, as we lacked enough time to properly implement such functionality.

4 Examples

The source and examples for TL-SHOP are at `http://code.google.com/p/tl-shop/`. Included with TL-SHOP are two new examples, `basicltl` and `dishoneststudent`. They are meant to demonstrate how constraints can be used in TL-SHOP to prune plans. See the domain specification file of each for more details.

5 Conclusion

TL-SHOP does what we set out to do—combine Hierarchal Task Network planning with the control rule planning paradigm in such a way as to increase domain and problem specification flexibility. It currently lags far behind JSHOP2 in performance, but this could be remedied by future work which discovers a way to extend JSHOP2's domain and problem compilation process to include progressions of LTL formulas.

References

[1] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. *New Directions in AI Planning*, pages 141–156, 1996.

[2] O. Ilghami. Documentation for JSHOP2. Technical report, Technical Report CS-TR-4694, Department of Computer Science, University of Maryland, 2005.

[3] O. Ilghami and D.S. Nau. A general approach to synthesize problem-specific planners. 2003.

[4] D.S. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann; Elsevier Science, 2004.