Domain Compilation for Embedded Real-Time Planning

Anthony Barrett

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
anthony.barrett@jpl.nasa.gov

Abstract

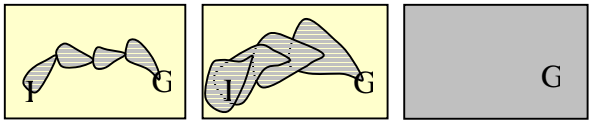
While universal plans tell a system how to reach a goal regardless of what state it is in, such plans are typically too large to represent. Hybrid systems execute plans where each action is implemented to robustly produce effects if the world does not stray outside the action's control envelope. This paper presents a middle ground between these two extremes that uses plans, but also enables much larger control envelopes using a real-time planner that finds optimal *n* step plans to achieve a set of goals if one exists.

Introduction

Ever since discovering that SHAKEY lacked the speed needed to deal with the real world (Brooks, 1999), the robotics community has endeavored to develop more responsive systems based either on universal plans or hybrid planners and executives. In the case of universal plans, researchers merge a set of behaviors into a universal plan, but the system must be restricted to relatively small problem domains to avoid having to reason about too many states and represent them in the universal plan. The hybrid approach avoids the problem by implementing activities as small sets of behaviors with limited applicability and then using a planner to string these actions together into an action sequence to traverse the system's state space from the current initial conditions to a goal state. This approach works well in static environments, but it is inherently brittle when addressing a dynamic world that can easily cause a failure by escaping an action's region of applicability.

This paper presents a middle ground between these two extremes, where a system can vary from the hybrid approach to a universal plan depending on a single integer parameter. As illustrated in Figure 1, the parameter starts at 1 to denote the hybrid approach where each linked action has a small coverage of the state space. As the parameter increases, the system's state-space coverage associated with reaching each subgoal grows from the subgoal's associated action until some domain dependent value M is reached, where the system contains a universal plan.

By using an embedded real-time planner that enables this middle ground, a hybrid system can become much more robust to a dynamic environment. Such a system operates by having a planner pass the current subgoals instead of activities to an executive. The executive then uses the real-time planner to determine when to perform which action until either satisfying the current subgoals or determining that they cannot be reached within *n* steps. While the first case signals success and the system continues by giving new subgoals to the executive, the second signals failure and the planner will have to alter its activity sequence to resolve the problem.



hybrid system $n = 1$ | middle ground $1 < n < M$ | universal plan $n = M$

Figure 1. Parameterized state-space coverage – from a hybrid system to a universal plan

This paper starts by describing a universal(*n*) plan as a parameterized generalization of a universal plan. The next section shows how to compile a domain into a universal(*n*) plan. Given such a structure, the fourth section shows how it takes O(structure size) time to determine the next actions to take given the current state and subgoals if there can exist an *n* step plan to reach the subgoals. Subsequent sections present empirical results, and conclude.

Universal(*n*) Plans

Ever since discovering the performance limitations of taking a sense-plan-act approach to controlling robots, the robotics community has endeavored to develop behavior-based approaches where a behavior implements a rapid feedback loop between state estimation and motor control. While this works well for simple tasks, like controlling robots with simple activity cycles on a factory assembly line, it gets much more complicated when controlling robots that have to flexibly react in an unstructured environment. This difficulty arises from the resulting complexity in selecting/coordinating the activation of possibly conflicting behaviors. From the action arbitration circuits in Pengi (Agre and Chapman, 1987) to the universal plan on EVAR (Schoppers, 1995) and the command arbiter of DAMN (Rosenblatt, Williams, and Durrant-Whyte, 2002), controlling and fusing behaviors becomes progressively harder as the target system faces progressively more unstructured environments.

As illustrated in Figure 2, the typical hybrid approach attempts to resolve the complications of developing a behavior selector/coordinator by replacing it with a planner. The planner determines an action schedule that maps a path from the current situation to a goal state, and this action schedule is used to incrementally determine when to activate which action behaviors in order to achieve the goal state. Since action behaviors are much more limited than a global behavior, actions can fail when environmental conditions evolve to a situation not covered by their corresponding action behaviors. In such cases a failure signal tells the planner to search for an alternative action schedule. Since replanning is computationally intensive, any low overhead way to reduce failure signals results in a more responsive system.

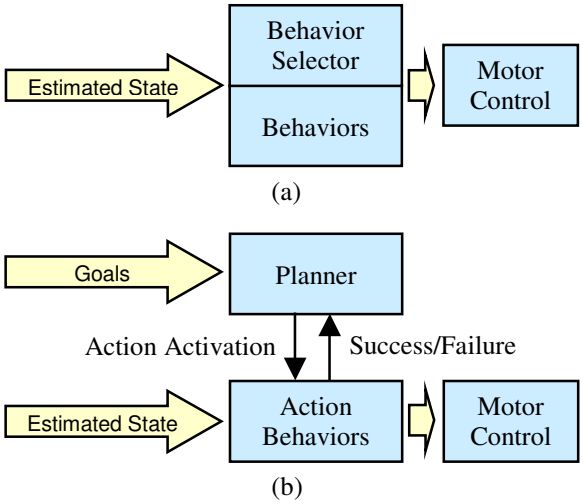


Figure 2. Comparing a universal plan (a) with a hybrid system (b)

The approach presented here reduces the number of failure signals by inserting a relatively limited real-time action selector between the planner and behaviors (see Figure 3). This approach only signals failure when the action selector fails to find the appropriate action for the current situation, and the action selector is guaranteed to resolve a failure if it can be resolved within *n* steps, where *n* is a user specified parameter. This guarantee is facilitated by evaluating a universal(*n*) plan against the current situation and subgoals.

Definition 1: A universal(*n*) plan is a structure that can be evaluated in linear time to generate an optimal *n* level plan to reach any set of goals from any current situation if such a plan exists.

Universal(*n*) plans are more general than universal plans by virtue of their not being tied down to a specific end goal. They are more restricted than universal plans by virtue of the *n* level requirement, where a level is any number of simultaneous non-interacting actions. As a user increases *n*, the universal(*n*) plan becomes less restrictive until reaching some domain dependent value M – where there is a guarantee that an goal can be reached from any situation with an M level plans. In practice *n* is kept relatively small because universal(*n*) plans tend to grow rapidly with *n*.

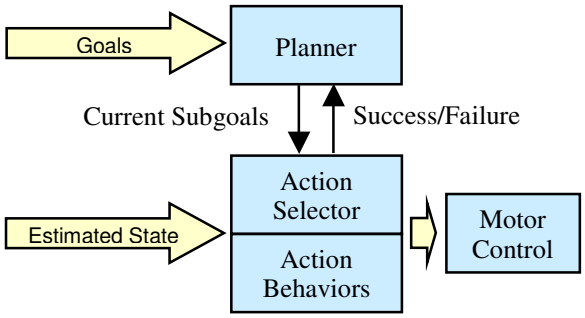


Figure 3. A universal(n) system

Compiling Domains

The approach to generating universal(*n*) plans presented here is a two step process. Step one utilizes results from developing SATPLAN (Kautz and Selman, 1992) to convert a domain into conjunctive normal form (CNF) whose satisfaction solves an *n*-level planning problem, and step two utilizes results from research on knowledge compilation (Darwiche and Marquis, 2002) to convert the CNF representation into Decomposable Negation Normal Form (DNNF). It turns out that this form of logical expression can be evaluated in linear time to compute an optimal *n* level plan.

To make this more concrete, consider the following simple two-operator domain that moves a system between states `a` and `b`. Each operator has a precondition that the variable `at` has the value `a` or `b` and an effect that assigns the value `b` or `a` respectively.

```
(make-op :name do[a->b]
         :prec (at=a) :post (at=b))

(make-op :name do[b->a]
         :prec (at=b) :post (at=a))
```

This state variable approach to representing a domain was motivated by the fact that most NASA planning domains represent spacecraft in terms of state variables, depletable resources like propellant, and nondepletable resources like electric power. It also mirrors the SAS+ domain language (Jonsson and Bäckström, 1994) with known complexity results.

Domains to CNF

In SATPLAN generating a CNF encoding of an *n*-level planning problem is fairly straightforward. There are actually several possible encodings (Ernst, Millstein, and Weld, 1997). The encoding used here has a boolean variable for each ⟨*action, level*⟩ tuple and an enumerated variable for each ⟨*state variable, level*⟩ tuple, where there are *n* action levels and *n*+1 state levels. The generation of the CNF derives from an observation that executing an action at level *i* implies that the action's preconditions hold at level *i* and its effects hold at level *i*+1. Also, not executing an action that affects a variable at level *i* implies that the variable's value persists to level *i*+1.

For instance, suppose that n=2 when compiling the example domain. The variable logic encoding, where all literals are variable assignments, of the planning problem appears below, where variable levels are reflected in subscripts. Notice that the first disjunct refers to the fact that the variable `at` persists from level 1 to 2 if neither of the two actions are performed at level 1. The second and third disjunct reflect the implications of executing one of the two listed operators at level 1 respectively. The last three disjuncts respectively mirror the first three, but for level 0 instead of 1.

```
(and
  (or do[b->a]1=t do[a->b]1=t
      (and at1=a at2=a)
      (and at1=b at2=b))
  (or do[a->b]1=f (and at1=a at2=b))
  (or do[b->a]1=f (and at1=b at2=a))
  (or do[b->a]0=t do[a->b]0=t
      (and at0=a at1=a)
      (and at0=b at1=b))
  (or do[a->b]0=f (and at0=a at1=b))
  (or do[b->a]0=f (and at0=b at1=a)))
```

As the example implies, the variable logic encoding grows linearly with n. In this case, the number of disjuncts would be three times n. In general the number of disjuncts is $n(|O|+|V|)$, where $|O|$ and $|V|$ are the domain's number operators and state variables respectively. Also, since each level is a set of disjuncts, the CNF that derives from the variable logic encoding only grows linearly with n. This size complexity is typical for a SATPLAN encoding.

Once the initial variable logic equation is expanded into CNF, removing disjuncts that are subsumed by other disjuncts reduces the result. It turns out that a disjunct subsumes another if its variable assignments are a subset of the other disjunct's variable assignments. This subset property implies that any satisfaction of the first disjunct satisfies the second.

CNF to DNNF

Unfortunately finding a minimal satisfying assignment to a CNF equation is an NP-complete problem, and more compilation is needed to generate a universal(n) plan. Since a DNNF equation can be evaluated in linear time, the second step converts the CNF to DNNF to represent a universal(n) plan. DNNF has been defined previously in terms of a boolean expression where only literals are negated and the literals appearing in sub-expressions of an conjunct are disjoint. The following definition slightly extends Boolean DNNF to variable logic equations, where the negation of a variable assignment has been replaced by a disjunct of all other possible assignments to that same variable.

Definition 2: A variable logic equation is in Decomposable Negation Normal Form if (1) it contains no negations and (2) the subexpressions under each conjunct refer to disjoint sets of variables.

Just as in the boolean case, there are multiple possible variable logic DNNF expressions equivalent to the CNF and the objective is to find one that is as small as possible. Since Disjunctive Normal Form is also DNNF, the largest DNNF equivalent is exponentially larger than the CNF. Fortunately much smaller DNNF equivalents can often be found. The approach here mirrors the Boolean approach to finding a d-DNNF by first recursively partitioning the CNF disjuncts and then traversing the partition tree to generate the DNNF.

The whole purpose for partitioning the disjuncts is to group those that refer to the same variables together and those that refer to different variables in different partitions. Since each disjuncts refers multiple variables, it is often the case that the disjuncts in two sibling partitions will refer to the same variable, but minimizing the cross partition variables dramatically reduces the size of the DNNF equation. This partitioning essentially converts a flat conjunct of disjuncts into an equation tree with internal AND nodes and disjuncts of literals at the leaves, where the number of propositions appearing in multiple branches below an AND node is minimized.

Mirroring the boolean compiler, partitioning is done with by mapping the CNF equation to a hyper-graph, where nodes and hyper-arcs respectively correspond to disjuncts and variables. The nodes that each hyper-arc connects are determined by the disjuncts where the hyper-arc's corresponding variable appears. Given this hyper-graph, a recursive partitioning using a probabilistic min-cut algorithm (Wagner and Klimmek, 1996) computes a relatively good partition tree for the disjuncts. See Figure 4 for an extremely simple example with two disjuncts and three variables. From the equation tree perspective, there is an AND node on top above disjuncts at the leaves. The branches of the AND node share a variable `b`, which is recorded in the top node's *Sep* set.

Once the equation tree is computed, computing the DNNF involves extracting each AND node's associated shared variables using the equality

$$eqn = \bigvee_{c \in domain(v)} \left(v = c \wedge eqn \setminus \{v = c\} \right),$$

where $eqn\backslash\{v=c\}$ is the equation generated by replacing disjuncts containing `v=c` with *True* and removing assignments to `v` from other disjuncts. If a disjunct ever ends up with no assignments, it becomes *False*.

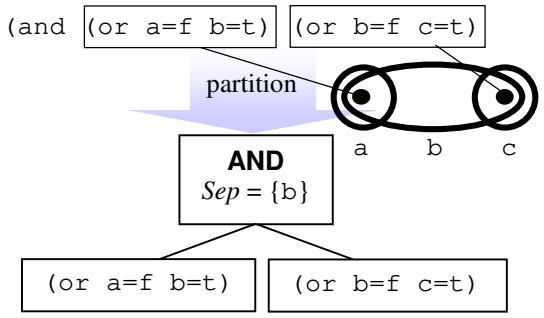


Figure 4. Example of partitioning a CNF equation

More formally, the DNNF equation is recursively defined using the following two equations, where the first and second equations apply to internal and leaf nodes respectively. In the first equation instances(*N.Sep*,α) refers to the set of variable assignments to variables in *N.Sep* that are consistent with α. For instance, running these equations over Figure 4's partition starts by calling *dnnf*(root,*True*), and the instances are `b=t` and `b=f` since the root only has `b` in *Sep*, and both assignments agree with *True*. In general the number of consistent instances grows exponentially with *N.Sep*, leading to the use of min-cut to reduce the size of *N.Sep* for each partition.

$$dnnf(N,\alpha) \equiv \bigvee_{\beta \in \text{instance}(N.Sep,\alpha)} \left(\beta \wedge \bigwedge_{c \in N.kids} dnnf(c, \alpha \wedge \beta)\right)$$

$$dnnf(disj,\alpha) \equiv \begin{cases} True & \text{if } \alpha \Rightarrow disj \\ \bigvee_{\beta \in disj \,\&\, \alpha \not\Rightarrow \neg\beta} \beta & \text{if } \exists \beta \supset \alpha \not\Rightarrow \neg\beta \\ False & \text{Otherwise} \end{cases}$$

While walking the partition does provide a DNNF equation that can be evaluated in linear time, two very important optimizations involve merging common sub-expressions to decrease the size of the computed structure and caching computations made when visiting a node for improving compiler performance (Darwiche, 2002). With respect to Figure 4, there were no common sub-expressions to merge, and the resulting DNNF expression appears below.

```
(or (and b=t c=t) (and b=f a=f))
```

Evaluating DNNF

To illustrate a less trivial DNNF expression, consider the universal(2) plan for the example two operator domain (see Figure 5). This expression's the top rightmost AND node has two children, and each child refers to a unique set of variables. From top to bottom these disjoint sets respectively are

{$\texttt{at}_2$, $\texttt{do[b->a]}_1$} and {$\texttt{at}_0$, $\texttt{do[a->b]}_0$}.

Given that AND nodes have a disjoint branches property, finding optimal satisfying variable assignments becomes a simple three-step process:

1. associate costs with variable assignments in leaves;
2. propagate node costs up through the tree by either assigning the min or sum of the descendents' costs to an OR or AND node respectively; and
3. if the root's cost is 0, infinity, or some other value then respectively return default assignments, failure, or descend from the root to determine and return the variable assignments that contribute to its cost.

When evaluating a universal(*n*) plan, a cost is assigned to each variable using a number of planning dependent preferences. First, not performing an action has zero cost. This results in associating zero with all leaves that set an action variable false. Second, performing required actions earlier is preferred. This results in associating lower costs with leaves that set lower level action variable true. In Figure 5 the values five and ten were associated with these leaves. In general the values can be both level and operator dependent for optimal planning. Third, the current state determines the costs associated with level 0 state variable assignments. While leaves with assignments corresponding to the current state have zero cost, leaves with assignments contradicting the current state have infinite costs. Finally, associated costs of top-level variables similarly depend on the current subgoals. Assignments consistent with the subgoals have zero costs, and assignments contradicting subgoals have infinite costs.

Figure 5 gives a detailed trace of evaluating a universal(2) plan for the simple move domain. The leaf costs are first computed using the planning preferences, and the results appear in the left hand column. Cost propagation then progresses through the tree, and the propagated cost associated with a node appears above it. Finally the descent from the root is a simple matter of comparing its cost with the cost of its immediate children. In this case the current state and goal were `at=a` and `at=b` respectively. Upon evaluating the universal(2) plan, `do[a->b]` is identified for execution.

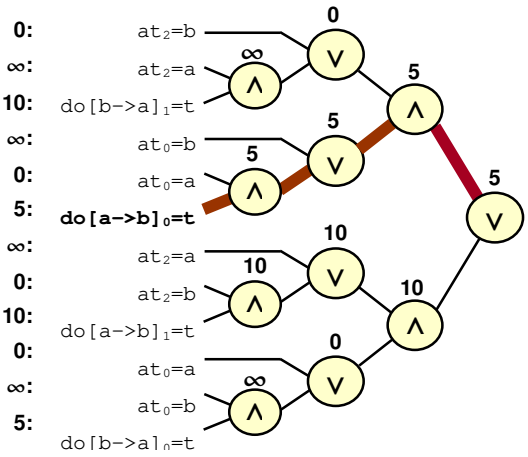


Figure 5. Evaluating a universal(2) plan when the current state is `at=a` *and the current subgoal is* `at=b`

While Figure 5 does display a universal(2) plan for the example domain, it is not precisely the DNNF equivalent to the previously listed variable logic encoding. It does not have any leaves corresponding to assignments to state variables at level 1 or false assignments to action variables. It turns out that these leaves can be safely omitted. An earlier incarnation of the compiler kept them, but the evaluation finds a plan when associating zero costs with leaves corresponding to false action variable assignments or intermediate state variable assignments. Given this, the leaves were omitted to reduce structure size by 25% on average.

Empirical Results

Both the compiler and evaluator are prototyped in fewer than 400 and 100 lines of allegro common lisp

respectively – shorter than this paper. For testing purposes a number of simple domains were encoded and compiled for varying numbers of levels. To give a first glimpse of how large universal(*n*) plans can get, consider Table 1. There are seven different domains with varying numbers of operators and state variables, and each domain was used to generate universal(*n*) plans where *n* varied from one to four. As the experiments imply, the number of nodes rapidly grows with n, but not nearly as rapidly as the number of possible *n*-level plans to consider. In the case of universal(4) for a 3-block blocksworld problem, there were 4155 nodes, but there were also 18^4 or 104976 different 4 action sequences if only one action is allowed per level. The number is even larger for arbitrary numbers of actions per level. Thus the DNNF representation of universal(*n*) plans is much more compact than listing all possible *n* or fewer level plans.

Table 1. The number of nodes in universal(n) plans for varying domains

Domain	vars	ops	*n*=1	*n*=2	*n*=3	*n*=4
Robot	3	4	40	59	89	115
Refuel	4	7	58	82	163	282
Coffee	4	8	73	208	568	1102
Wkshop	5	9	94	239	428	718
3blocks*	6	12	226	1156	1960	2799
3blocks	6	18	491	1898	3006	4155

Even with this compactness, more research is needed to address the scaling issue. While universal(*n*) plan size grows exponentially with variables, operators, and *n* in each example, all of the examples had highly interacting state variables and operators. There was less interaction in the Wkshop domain and that apparently lead to a slower universal(*n*) plan growth. For fully independent systems the universal(*n*) plan grows linearly with the number of systems.

Future Work

While the system succeeds in computing universal(*n*) plans and evaluating such structures in linear time to provide optimal *n* level plans, there are many avenues for future research. First, universal(*n*) plans tend to grow rapidly with *n*, but there are techniques for reducing structure size. While identifying and extracting zeroed leaves reduces the size by 25% on average, extra optimizations are possible. For instance, M=1 for the example domain, but Figure 5 had a universal(2) plan. Close inspection of this structure results in a discovery that the action nodes with cost 10 (i.e. the higher level action variables) are never selected, and can thus be removed from the structure. How to algorithmically perform this optimization is still undetermined. Also, the DNNF returned by the compiler has a varying size due to using a probabilistic min-cut algorithm. This implies that there is no guarantee that the compiler is actually generating the optimal DNNF equation.

Another avenue of research involves generating universal plans instead of universal(*n*) plans. If the action level width M of the state space is known, the first intermediate logic equation generated from the domain and M can be altered to insert the universal plan goal in terms of a conjunct of assignments to top-level state variables. This conjunct will actually simplify the generated DNNF equation and evaluating this equation generates the optimal *n* level plan in linear time.

A third avenue of research derives from an observation that there are DNNF based knowledge compilation results for the model-based diagnosis problem (Darwiche, 1998). Combining these results with work in this paper results in a generalization of the Livingstone executive (Williams and Nayak, 1996) that was flown on Deep Space 1. In this case the resulting evaluation engine is only 100 lines long and the diagnosis and recovery mechanisms are compiled into the DNNF structure. Both of these components are much easier to validate than the code used to implement Livingstone and reason about its models.

A fourth avenue of research revolves around distributed execution. It turns out that using DNNF structures facilitates distributing diagnosis reasoning across multiple agents (Chung and Barrett, 2003). The same should hold for distributing an action selector. The main source of complexity is the need alter the compiler to find acceptably small DNNF structures that group state reasoning and action selection components by agent in order to reduce cross agent communication. Also, the evaluation routine needs to be distributed in such a way that momentary communication losses do not fatally impact execution. The current evaluation approach of propagating costs to the root and then drilling down to find the sources of the minimum cost would be fatally impacted by intermittent communication loss.

Finally, more open-ended research directions include expanding the system to handle metric resources, time, exogenous events, uncertainty, and other representational enhancements. Conditional effects of actions can be encoded as

(or $action_i$=f (not $condition_i$) $effect_{i+1}$),

where *condition* and *effect* are from the conditional effect of *action*. Metric resources can also be included by noting how they restrict the sets of actions that can be performed simultaneously at a level of the universal(*n*) plan.

Conclusions

This paper presented a knowledge-compilation based approach to implementing an offline domain compiler that enables an embedded real-time planner for a more robust plan execution system. Past approaches either focused on generating universal plans or developing tools for implementing more robust action behaviors (Williams and Gupta, 1999) (Simmons and Apfelbaum, 1998). While the offline compiler can turn the embedded real-time planner into a universal plan, it can avoid the space requirements of a universal plan by generating a smaller universal(*n*) plan. With such a

structure, the embedded real-time planner can determine *n*-level plans to achieve any set of goals from any current state if such a plan exists. This enhances work on making action behaviors more robust by enabling rapid changes in action selection as the environment evolves.

Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author would also like to thank Seung Chung, Adnan Darwiche, Daniel Dvorak, and Mitch Ingham for discussions contributing to this effort

References

Agre, P. and Chapman, D. 1987. Pengi: An Implementation of a Theory of Activity. In Proceedings of the Sixth National Conference on Artificial Intelligence. 268-272. Seattle, WA: American Association for Artificial Intelligence.

Brooks, R. 1999. *Cambrian Intelligence*. Cambridge, MA:MIT Press.

Chung, S. and Barrett, A. 2003. Distributed Real-time Model-based Diagnosis. In Proceedings of the 2003 IEEE Aerospace Conference, Big Sky, MT: IEEE Aerospace Conferences, Inc.

Darwiche, A. 1998. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165-222.

Darwiche, A. 2002. A Compiler for Deterministic Decomposable Negation Normal Form. In Proceedings of the Eighteenth National Conference on Artificial Intelligence. 627-634. Edmonton, Alberta, Canada: AAAI Press.

Darwiche, A. and Marquis, P. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research.* 17:229-264.

Ernst, M. and Millstein, T. and Weld, D. 1997. Automatic SAT-Compilation of Planning Problems. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence. 1169-1177. NAGOYA, Aichi, Japan: Morgan Kaufmann.

Jonsson, P. and Bäckström, C. 1994. Tractable Planning with State Variables by Exploiting Structural Restrictions. In Proceedings of the Twelfth National Conference on Artificial Intelligence. 998-1003. Seattle, WA: AAAI Press.

Kautz, H. and Selman, B. 1992. Planning as Satisfiability. In Proceedings of the Tenth European Conference on Artificial Intelligence. 359-363. Vienna, Austria: John Wiley and Sons.

Rosenblatt, J. and Williams, S. and Durrant-Whyte, H. 2002. Behavior-Based Control for Autonomous Under-water Exploration. *International Journal of Information Sciences*, 145(1-2):69-87.

Schoppers, M. 1995. The use of dynamics in an intelligent controller for a space faring rescue robot. *Artificial Intelligence* 73:175-230.

Simmons, R. and Apfelbaum, D. 1998. A Task Description Language for Robot Control. In Proceedings of the Conference on Intelligent Robotics and Systems. Victoria, BC: IEEE Press.

Wagner, F. and Klimmek, R. 1996. A Simple Hypergraph Min Cut Algorithm. Technical Report, b 96-02, Freie Universität Berlin.

Williams, B. and Nayak, P. 1996. A Model-based Approach to Reactive Self-Configuring Systems. In Proceedings of the Thirteenth National Conference on Artificial Intelligence, Portland, OR: AAAI Press.

Williams, B. and Gupta, V. 1999. Unifying Model-based and Reactive Programming in a Model-based Executive. In Proceedings of the 10th International Workshop on Principles of Diagnosis, Loch Awe Hotel, Scotland.