# Designing Experiments to Test Planning Knowledge about Plan-step Order Constraints

**Clayton T. Morrison** and **Paul R. Cohen**
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, California 90042
{clayton,cohen}@isi.edu

### Abstract

A number of techniques have been developed to effectively extract and generalize planning knowledge based on expert demonstration. In this paper we consider a complementary approach to learning in which we may execute experiments designed to test hypothesized planning knowledge. In particular, we describe an algorithm that automatically generates experiments to test assertions about plan-step ordering, under the assumption that order constraints between two steps are *independent* of *other* step orderings. Experimenting with plan-step ordering can help identify asserted ordering constraints that are in fact not necessary, as well as uncover necessary ordering constraints not represented previously. The algorithm consists of three parts: identifying the space of step-ordering hypotheses, efficiently generating ordering tests, and planning experiments that use the tests to identify order constraints that are not currently represented. This method is implemented in the CMAX experiment design module and is part of the POIROT integrated learning system.

## Introduction

In learning from demonstration, a learning system is presented with a demonstration of how to correctly complete some task. The demonstration is typically represented as a *trace* that consists of the ordered sequence of executed actions, and usually includes the parameter values and outcomes of the action executions, with some additional information about the world state. Extracting planning knowledge from the trace requires intelligent application of background knowledge. As with all learning, the goal is correct generalization: to produce planning knowledge in the form of operators and methods that can solve problems beyond what is observed in the trace. A number of techniques are being developed that do this effectively. Here we consider a facet of the learning problem that complements these methods: What if I could take some actions and observe their outcomes, thereby getting *additional* knowledge about the domain? What aspects of my current hypotheses need testing, and what would help me better generalize what I know? Just as single-shot learning can not depend on large quantities of prior examples, neither can we ask for all the information we want. Our tests must be as informative as possible, requiring the smallest possible effort, and we should also develop methods for rating the relative importance of tests. These considerations are the subject matter of the general paradigm of *active learning* (Cohn, Ghahramani, & Jordan 1996), but rather than standard active learning techniques that use statistical properties of prior data, our focus is on analyzing existing background knowledge and using both domain-general and domain-specific heuristics to plan efficient information gathering experiments.

In this paper we report on our work in the DARPA Integrated Learning (IL) program. We are developing automated methods for identifying and efficiently acquiring (when permitted) the most useful additional information to improve generalization from a single trace. The approach we describe is implemented in CMAX, the **C**ausal **MA**p e**X**periment designer, a module in the POIROT integrated learning from demonstration system (Burstein *et al.* 2007). The POIROT system comes equipped with a domain ontology and model of the primitive operators to use in planning. This model, however, is *not complete* in the sense that it does not represent all of the preconditions and possible affects of executing the operators in different combinations and contexts. POIROT's task is to use the demonstration trace as a guide to construct *planning methods* that decompose the planning problem into a hierarchical task network (Nau *et al.* 2003) that ultimately grounds out in primitive actions, in an attempt to appropriately generalize from the trace to solve a whole class of problems. From CMAX's perspective, *other* modules in POIROT play the role of "planning knowledge extractors," and their products are *hypothesized* planning methods. It is the job of CMAX to automate testing any of the different kinds of assertions the knowledge extractors might make in their hypothesized methods. CMAX currently automates our approach to planning experiments to identify potential ordering constraints between plan steps – *ordering constraints that are otherwise implicit in the original expert trace and are not represented in the primitive operator models provided as prior knowledge*.

## Generating Step-Order Experiments

The POIROT system works in a semantic web service domain, where primitive actions are calls to web services. The

operator semantics of the primitive actions-as-web-service-calls adds to the standard STRIPS-style operator semantics an explicit representation of action input parameter values (e.g., to make a reservation, one must specify the purchaser's name, the flight number, etc.) and possible output values returned by the web service (e.g., a reservation confirmation number).[1] When planning a series of web service calls to satisfy a task, output values returned by one web service may be required as inputs to another web service. For this reason, plan representation in POIROT not only includes representation of state change dependencies but also data flow from one web service to another. (The labeled directed edges in Figure 1 are examples of such data flow.) In POIROT, planning knowledge is represented as sets of planning methods expressed in the *Learnable Task-Modeling Language* (LTML). LTML is very expressive and includes constructs for representing data flow between primitive action outputs and inputs, and control flow constructs asserting step orderings, conditional execution, and loops. In the work we present here, CMAX analyzes and tests LTML planning method data flow and plan step ordering constraints.

When presented with a planning problem (an initial world state and goal), POIROT submits a set of methods to a planner and an executable workflow is generated. A *workflow* is a totally-ordered sequence of steps[2], where each step may involve executing a primitive action, and output-to-input data flow between steps is specified by data flow links. An executive then executes the workflow and reports the results of the execution.

There are three possible outcomes of workflow execution: a workflow could (1) fail to execute because one of the steps in the workflow cannot be enacted or completed; or, if execution completes, the executive determines whether the executed workflow has (2) failed or (3) succeeded in satisfying the planning goal. If the execution fails (with either outcome 1 or 2), then we conclude that one or more planning method constructs in incorrect. If, on the other hand, a workflow execution succeeds in satisfying the planning problem goal (outcome 3), then we conclude that the planning knowledge was at least sufficient for this problem instance. We define a *test* to be the execution of a workflow – a workflow that is "complete" in the sense that it is intended to completely satisfy the planning goal.[3] (From here on, unless otherwise noted, we will use the terms "test" and "workflow" interchangeably.) An *experiment*, then, consists of the execution of one or more workflows. Our goal for CMAX's experiment design procedure is to construct workflows as experiments whose outcomes inform our beliefs about the efficacy of our planning knowledge, as expressed in our planning methods.

There are many different aspects of hypothesized planning knowledge that we might try to test. We have chosen to target assertions about step ordering first because assertions about step order constraints are common in the IL project. By *step-order constraint* we mean that a step order is necessary: if step $s_1$ is constrained to occur before step $s_2$, then having $s_2$ occur *before* $s_1$ in a workflow will result in execution failure. In general, the existence of an order constraint can depend on the context in which the ordered pair of steps occurs. For example, if $s_1$ sets a condition that $s_2$ depends on and that condition is not present initially, then $s_1$ must occur before $s_2$; if that condition is present initially and nothing else removes it, then there is no necessary ordering between $s_1$ and $s_2$. Similarly, a step $s_3$ might negate that condition, so if $s_3$ occurs before steps $s_1$ and $s_2$, then $s_1$ must occur before $s_2$, but not otherwise. In the algorithm we present below, particularly the phase that relates potential experiments according to the order constraints tested, we assume that order constraints are *independent* of the order in which *other* steps occur (and also assume the initial world state is held constant). It is important to keep this in mind as it does restrict the applicability of the method; we will return to this point below.

Misrepresenting step order constraints results in at least one of two kinds of planning knowledge generalization failures:

1. Asserting an ordering constraint that is in fact *not necessary* may make later plan generation under new circumstances appear impossible when in fact a plan could be generated if the order constraint were ignored.
2. *Not* explicitly representing a step order constraint means that plans may be generated that violate the constraint and subsequently fail to successfully execute.

Assuming that necessary step ordering constraints are independent, we can test for either of these generalization failures by producing totally-ordered workflows with different step orderings, execute them, and see whether they succeed or fail.

In the next three sections we describe the three phases involved in generating experiments to identify missing or misrepresented ordering constraints.

## Phase 1: The Step-order Hypthesis Space

The first task for CMAX is to identify the space of step order constraint hypotheses we wish to test. To describe this hypothesis space we review some (likely familiar) concepts about order relations. Consider a totally ordered sequence of three elements, $a$ $b$ $c$, and let $\prec$ represent the irreflexive, asymmetric and transitive order relation (assuming left-to-right directionality). In the sequence, $a$ comes before $b$ is represented by $a \prec b$, and $b$ comes before $c$ is represented by $b \prec c$. We also have, by transitivity, the order relation $a \prec c$. In general, for each element we add *to the right-hand end* of the sequence, we introduce a set of order relations between every prior element and the new one. The total number of order relations present in a totally ordered

[1] As with standard STRIPS operators, successful web service enactment may involve world-state preconditions that must be satisfied (e.g., to make a plane reservation there must be an open seat), as well as bring about changes in the world (a seat is filled when a reservation is made).

[2] In this phase of our work we do not consider parallel plan-step execution.

[3] We could define a test to be a shorter sequence of steps aimed at achieving local goals. This raises a number of issues for experimentation (including reasoning about subgoals and their satisfaction) that are beyond the scope of the work we report here.

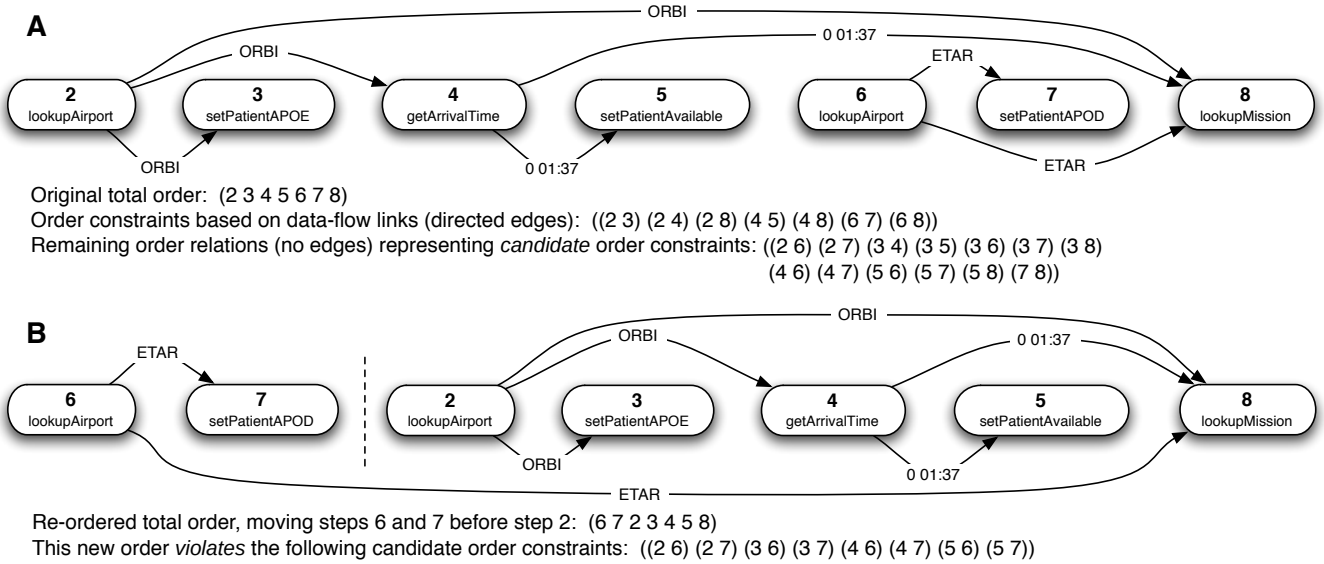


Figure 1: **A** represents the link-based order constraint relations (directed edges) identified for a *sequence* in a method fragment; the pairs of step numbers in parentheses represent individual order relations; labels on the edges represent values that would be assigned to the corresponding links were the method used to reproduce the demonstration trace as an executing workflow. **B** represents a re-ordering of the step sequence that *preserves* the link-order constraints identified in **A**, but *violates* (and therefore tests) eight candidate order constraints.

sequence of $n$ elements is $K = (n^2 - n)/2$. We will refer to this complete set of order relations as the *transitive closure* of order relations over the totally ordered sequence. Without prior knowledge about order constraints, any of the $K$ order relations in the transitive closure over a sequence of steps is a *candidate* step-order constraint.

CMAX distinguishes between two classes of order constraints expressed in LTML methods and treats them differently for the purpose of step-order experimentation. The first class consists of assertions about step ordering that range from fully specifying step sequences (in *Sequence* clauses) to leaving steps unordered (in *Activity-graph* clauses); Activity-graphs may include predicate clauses whose conditions must be maintained, thus asserting some constraints. CMAX identifies groups of steps associated with these clauses and experiments with their order. Depending on the experiment outcome, CMAX may recommend revising the current method's step order constructs.

The other class of order constraints consists of data flow assertions, as mentioned in the previous section. LTML provides a *links* construct to specify how output values resulting from web service calls are used as input values to later service calls. In the version of the step order experiment design algorithm we present here, CMAX generates orderings of steps that *preserve* these data flow links, thus treating them as order constraints that *cannot* be violated. In later versions of CMAX, ordering experiments will be generated that also test data-flow links.

Figure 1 **A** represents data flow links as directed edges between steps. In this example, the space of candidate order constraints that CMAX will test is the set of order relations *other than* the set of link order constraints. Put another way, the target set of relations to test consists of the relations that remain after *removing* the link order constraints from the transitive closure of order relations over the total ordered sequence of steps. This target set of relations is represented beneath Figure 1 **A** as the list of thirteen ordered pairs of step numbers. Figure 1 **B** shows an example of a reordering that preserves the original link orders while violating eight of the target candidate order constraints.

## Phase 2: Efficiently Generating Tests

If we could test each candidate order constraint individually, then in the example we would only need thirteen tests: for each order relation, simply swap the two elements in the relation, thereby *violating* any underlying order constraint, and see if the workflow successfully executes. However, this is only possible for order relations between adjacent steps. For example, for the $a\ b\ c$ sequence we have three options for testing the necessity of the $a \prec c$ order: (1) $c\ a\ b$, which violates $a \prec c$ and $b \prec c$; (2) $b\ c\ a$, which violates $a \prec c$ and $a \prec b$; and (3) $c\ b\ a$, which violates all three original order relations. In no case can we violate *only* $a \prec c$. As we will see in the next section, we may combine tests so as to (potentially) isolate order relations between non-adjacent steps. For example, if we first run an experiment with the step order $b\ a\ c$ and it succeeds, then we can eliminate $a \prec b$ as a candidate order constraint. Now that we know that it does not matter in what order $a$ and $b$ occur, we can use ordering (2) $b\ c\ a$ and be certain that if it fails, it is due to violating $a \prec c$. If, however, the $b\ a\ c$ workflow fails, then the $a \prec b$ order is necessary and $b\ c\ a$ is guaranteed to fail irrespective of the relationship between $a$ and $c$. In this case, we must try a different sequence of tests to isolate the $a \prec c$

relation. We won't know ahead of time which sequences of tests we will need so we must be prepared to consider any of them. We need a general method for generating orderings that violate different combinations of relations representing candidate order constraints while preserving data flow link constraints.

The naive approach to generating the desired set of sequences is to generate the complete set of permutations (with no constraints) and then select only those that satisfy the link order constraints. Unfortunately, for $n$ elements there are $n!$ unconstrained sequences. For the seven steps in Figure 1 this involves generating 7! = 5040 sequences. We can greatly reduce our work by generating *only* those permutations that satisfy the link order constraints.[4] For the seven steps in Fig. 1, there are actually only 144 permutations that satisfy all of the identified link order constraints – an order of magnitude fewer than the full set of unconstrained permutations.

A better way to proceed is to start with the constraints we wish to preserve. First, consider a single order constraint $a \prec b$. All this relation asserts is that $a$ has to come before $b$. If element $c$ has no order constraints involving $a$ and $b$, then we are free to place $c$ in any region (indicated by '_') around $a$ and $b$: _ $a$ _ $b$ _. Doing so does does nothing to the original $a \prec b$ order relation. We say that we *distribute* $c$ over the sequence $a\ b$ if we generate a new sequence for each placement of $c$. Distributing $c$ over $a\ b$ thus produces the sequences: $c\ a\ b$, $a\ c\ b$, and $a\ b\ c$. We can repeat this operation, distributing a new unconstrained element $d$ over the four positions around each of the three new sequences that include $a$, $b$ and $c$. We can also distribute an order constraint pair $x \prec y$ over the sequences, producing new sequences; if $x$ already exists in the sequence, then $y$ is distributed to all position *after* $x$, and likewise if $y$ already exists, $x$ is distributed to all positions *before* $y$. And if neither $x$ nor $y$ is constrained, than we first distribute $x$, then distribute $y$ over all positions *after* $x$.

Using this simple operation of *distributing* elements (or pairs of order-constrained elements) over sequences, the following algorithm generates all, and only, the permutations satisfying the set of (consistent) ordering constraints. We present the basic algorithm here; see (Morrison & Cohen 2007) for more developed justification of each step.

1. Remove transitive orderings from the order constraints we wish to preserve: if $a \prec b$ is an order constraint we wish to preserve, but two or more other to-be-preserved order constraints exist that together link $a$ to $b$, then remove $a \prec b$ from the to-be-preserved list.
2. An order constraint $a \prec b$ *overlaps* another order constraint if $a$ or $b$ appears in the other order constraint. This step sorts the to-be-preserved list of order constraints remaining after step 1 so that any two related order constraints A and B, where A appears earlier in the list than B, will not be separated by one (or more) order constraints that *do not overlap* A while they *do* overlap B. The following simple example shows why we do this: if we went through the following list in order, $a \prec b$, $c \prec d$, $a \prec d$, iteratively distributing the next constraint over the growing set of sequences, then at some point we would distribute $c$ and $d$ around $a$ and $b$, generating, e.g., $c\ d\ a\ b$, but then come to $a \prec d$, asserting that $d$ cannot occur before $a$. This violates the goal of producing all and only valid orderings. By moving $c \prec d$ to come *after* $a \prec d$, we avoid this problem.
3. After step 2, we simply step through the list of to-be-preserved order constraints and incrementally distribute them: take the first order constraint and distribute the second over it; then distribute the third order constraint over the sequences just generated, and so on.
4. Finally, any other elements not represented in preserved order constraints are individually distributed one at a time over the sequences.

This algorithm generates all of the permutations that satisfy the link-order constraints, and no more. In the worst case, when there are no order-constraints, the algorithm runs in $\mathcal{O}(n!)$ (both in time and for the space required to store the generated sequences for the next phase). As long as there is a significant number of order constraints, time and space complexities are much better.

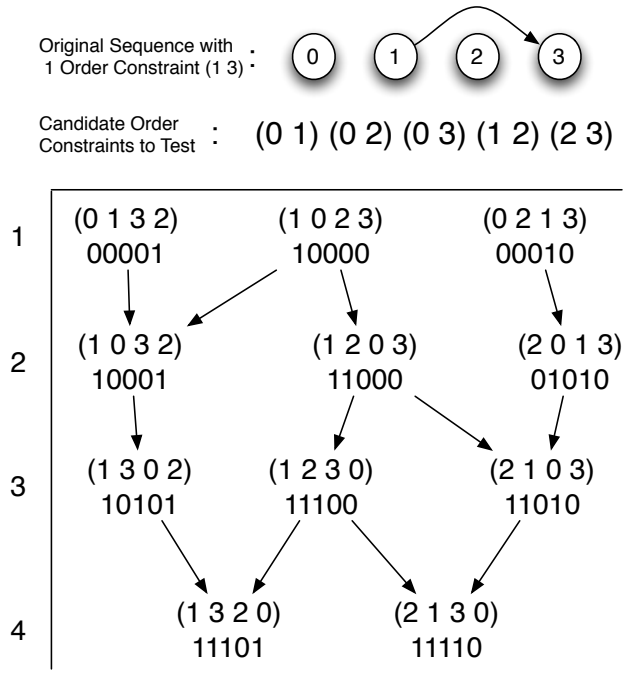


Figure 2: Test Dependency Graph

## Phase 3: Planning Effective Experiments

We now refine our definition of experiments and describe how they are constructed. Experiments are sets of tests chosen to determine whether ordering assertions made in LTML

[4] While there is no known formula for calculating the number of permutations that preserve arbitrary ordering constraints (Birghtwell & Winkler 1991), we do know that if the longest chain of ordering constraints includes $c$ elements (e.g., steps 2, 4 and 5 in Figure 1 **A** form a chain of length 3), then the number of permutations is upper bounded by $n!/c!$.

methods are valid. In Phase 1, CMAX identifies sets of steps (and their link constraints) involved in LTML method ordering constructs. In Phase 2, CMAX generates re-orderings of these steps that can be used to test whether the orderings asserted in the method are valid. Each experiment generated in Phase 3 will test only *one* ordering construct at a time. Recall that while our focus has been on permuting a sequence of steps, a *full* test in an experiment consists of these permutations in the context of other steps that together form a "complete" workflow. That is, our experiments alter, through reordering, a component of a workflow that is otherwise expected to solve the planning problem instance. If the workflow fails to execute, the cause is somewhere in the alteration.

In the previous section we noted that some sequences will test more than one order constraint at the same time, and some order relations cannot be altered without altering others. We also pointed out that we can combine tests to isolate individual order constraints. In Figure 2 we show how test sequences are related. In this example, a simple 4-step sequence has one link constraint (pictured at the top of the figure). Below the sequence is the list of order relations representing the candidate order constraints we wish to test. Each node in the graph at the center of the figure represents a permuted sequence. Below each sequence is a binary vector representing which candidate order constraints are violated by the sequence – a 1 indicates a violation and the positions correspond to the positions of the candidate order constraints in the list above the graph. For example, sequence (0 1 3 2) violates the fifth candidate order constraint: $2 \prec 3$. The nodes are arranged into rows such that all nodes in a row violate the same number of order relations; the numbers in the left margin indicate how many. Finally, the directed arrows represent that the sequence the arrow is pointing toward violates the order relations of the sequence at the tail of the arrow as well as others – that is, the parent violations are a subset of the child violations. (These subset relationships are transitive, but for clarity, transitive edges have been removed.) For example, (1 0 3 2) and (0 1 3 2) both violate $2 \prec 3$, but (1 0 3 2) also violates $0 \prec 1$. The subset relationships, as represented by the arrows, are not the only kind of relation between the sequences. Sets of violated order relations for two nodes may also partially overlap. Again, for clarity these edges are not pictured. We refer to this graph as a *test dependency graph* (TDG). In the current version of CMAX, this graph is constructed explicitly, using the set of permutations generated in the previous phase.

The TDG is a useful representation for selecting tests for an experiment. After a sequence is executed, we update the graph based on the execution outcome. For example, if we execute sequence (0 1 3 2) and it succeeds, then we now know that relation $2 \prec 3$ is not necessary. This also means that any other tests violating this relation will not fail because of this violation – if another test fails it is due to some other constraint. In this example, we update the TDG by removing all 1's in index 5 of the binary vector (representing $2 \prec 3$). After the update, sequence (1 0 3 2) has only one remaining candidate order constraint, $0 \prec 1$. Running the test (1 0 3 2) will now determine with certainty whether $0 \prec 1$ is necessary. Test success may also lead to the removal of other tests: any ancestor (according to the subset relation depicted by the arrows) may be removed from the graph because all violated order relations of the ancestors have been shown to be not necessary, so running their test won't tell us anything new.

Updating due to test failure works in a complementary fashion. If a test fails, then we know that any descendant according to the subset relation will also fail, since descendants always violate *more* order relations. Again, we remove these because they can no longer tell us anything new.[5] Any other tests that overlap, however, remain tests of interest in our search to identify which violated order relations are actual order constraints.

The overall goal in experiment test selection is to run tests that determine which candidate order constraints are necessary. At the same time, we want to minimize the total number of tests we execute. There are several test selection policies to choose from, and which in the long run is cheaper depends on how many actual order constraints there are. To see this, consider binary versus linear search, two ends of a spectrum of policies. If there are likely only a few order constraints, binary search can be significantly cheaper than linearly testing each individual order. In binary search, we start by selecting a test that violates as many order relations as possible. If it fails, choose two new tests that (ideally) violate half as many relations and don't overlap in the relations they violate. As long as tests fail, keep splitting and testing. If a test succeeds, stop further testing of that set of relations. If *no* order constraints exist, then we can stop after a single test that violates all order relations succeeds. If there is just one order constraint, then we can uniquely identify it while confirming all the rest are not necessary in order $\mathcal{O}(\log n)$ experiments (for $n$ order relations). But as the number of order constraints increases, binary search quickly looses its superiority. When there are many actual ordering constraints, it is best to confirm them by testing for them one at a time. With many ordering constraints, binary search could result in twice as many tests. If we have background information about the number of likely tests, then we can use that to inform which policy to use.

We can approximate a binary search policy by starting with sequences at the bottom of the TGD, where the tests violate many order relations. In subsequent iterations, we work our way up the graph and identify pairs of tests that together cover all of the relations being tested but split the set of relations roughly in half. Linear search starts at the top of the graph and works its way down.

CMAX's default experiment construction strategy is to conduct a linear search. CMAX first selects tests from the top of the TGD that violate single order relations. These are

[5]Note that when we remove tests from the graph some order relations that were not directly tested may be no longer represented in the graph. This is due to transitivity. In the example, suppose we find that $0 \prec 1$ is necessary. Since $1 \prec 3$ is already a link constraint, $0 \prec 3$ must also be necessary, by transitivity. We see this in the TDG update: when (1 0 2 3) fails ($0 \prec 1$ is necessary), we remove all descendants, including everything with a 1 at position 3 in the binary vector: (1 2 3 0), (1 3 2 0) and (2 1 3 0).

executed and the graph is updated. Now updated tests may have better resolving power; these are selected in the next round and the cycle is repeated until all of the order relations have been confirmed to be actual ordering constraints or not necessary.

## Related Work

We have room here to describe only a few projects related to our work. A very similar approach to learning planning methods is Veloso and Winner's SPRAWL algorithm (Winner & Veloso 2002), which analyzes existing domain knowledge to identify step dependencies. The main difference with our work is that we focus on experimentation and can therefore refine planning knowledge in the face of incomplete domain models; we view CMAX as complementary to SPRAWL. Carbonell and Gil (1990) developed a framework of heuristics for identifying problems in planning and execution that arise due to elements missing from planning operator definitions This work was further developed in Gil's EXPO module, part of the PRODIGY planning system (Gil 1996). Other systems, such as OBSERVER (Wang 1996) and WISER (Tae, Cook, & Holder 1999) are natural extensions to EXPO, adding to operator refinement techniques that rely on less prior domain knowledge. OBSERVER shares with CMAX a focus on learning from expert trace observations. Two important differences between CMAX and these approaches are that CMAX does not require that the world state is fully observable and CMAX also explicitly considers the relationships between experiments, where multiple orderings may be tested at once (again, assuming independence). Finally, we have already indicated the connection between CMAX and active learning; similarly, the field of active sensing, particularly in robotics, involves explicit use of actions to gather information to form and refine domain models (Mihaylova *et al.* 2002).

## Conclusion

We have presented an approach to generating experiments that test planning knowledge about step ordering constraints. We plan to develop the approach described here in several ways. First, we have just begun to characterize the test dependency graph and there may be additional structure that we can put to work. At the same time, the power of the TDG rests on the assumption that individual order constraints are independent of one another. If that assumption is violated, we can no longer rely on the partial-lattice structure of TDG to represent how tests are related. Further work is needed to generalize the TDG representation to handle cases where the existence of necessary order relationships depends on context. We are also interested in generating experiments to test other method constructs, including branching conditions and looping. In general, these extensions will require moving beyond the purely syntactic combinatorics of step re-ordering and taking more of the semantics of the domain into account. Finally, we are interested in providing CMAX a probabilistic representation of order constraints. This will allow representation of prior knowledge about possible ordering constraints in terms of degrees of belief, and also moves CMAX's reasoning into the well-studied domain of statistical active learning. Along these lines, we also plan to incorporate a model of cost of actions and world state utilitiess, turning the experiment formulation task into a cost-based decision-theoretic problem.

## Acknowledgments

We thank two anonymous reviewers for their helpful comments. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) Integrated Learning program, through grant FA8650-06-C-7606, subcontract to BBN Technologies.

## References

Birghtwell, G., and Winkler, P. 1991. Counting linear extensions is #P-complete. In *Proceedings of the 23rd ACM Symposium on Theory of Computation*, 175–181.

Burstein, M.; Brinn, M.; Cox, M.; Hussain, T.; Laddaga, R.; McDermott, D.; McDonald, D.; and Tomlinson, R. 2007. An architecture and language for the integrated learning of demonstrations. In *AAAI 2007 Workshop on Acquiring Planning Knowledge via Demonstration*.

Carbonell, J. G., and Gil, Y. 1990. Learning by experimentation: The operator refinement method. In *Machine Learning, An artifical Intelligence Approach*, volume 3. Morgan Kaufmann.

Cohn, D. A.; Ghahramani, Z.; and Jordan, M. I. 1996. Active learning with statistical models. *Journal of Artificial Intelligence Research* 4:129–145.

Gil, Y. 1996. Planning experiments: Resolving interactions between two planning spaces. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*.

Mihaylova, L.; Lefebvre, T.; Bruyninchx, H.; Gadeyne, K.; and Schutter, J. D. 2002. Active sensing for robots – a survey. In *Proceedings of the 5th International Conference on Numerical Methods and Applications*.

Morrison, C. T., and Cohen, P. R. 2007. Designing experiments to test and improve hypothesized planning knowledge derived from demonstration. In *AAAI 2007 Workshop on Acquiring Planning Knowledge via Demonstration*.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Tae, K. S.; Cook, D. J.; and Holder, L. B. 1999. Experimentation-driven knowledge acquisition for planning. *Computational Intelligence* 15(3).

Wang, X. 1996. Planning while learning operators. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*.

Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS 2002)*.