Designing Experiments to Test and Improve Hypothesized Planning Knowledge Derived From Demonstration

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 experiments are designed to test hypothesized planning knowledge. In particular, we describe an algorithm that automatically generates experiments to test assertions about plan-step ordering. Experimenting with plan-step ordering can identify asserted ordering constraints that are in fact not necessary, as well as uncover necessary ordering constraints previously not represented. 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. We discuss the role of experimentation in planning knowledge refinement and some future directions for CMAX's development.

Introduction

Extracting planning knowledge from a single demonstration requires intelligent application of background knowledge. As with all learning, the goal is correct generalization: produce planning knowledge in the form of operators and methods that can solve problems beyond what is observed in the trace representing the demonstration. A number of techniques are being developed that do this effectively. Here we consider a facet of the learning problem that complements these methods: If I've extracted all I can from what I've seen and based on what I know, then what? What if I could take some actions and see what the outcomes are, thereby getting *additional* information about the domain and the problems we need to solve? What tests should I run? 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 data, 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*, but rather than standard active learning techniques that use statistical properties of prior data, instead 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. We describe CMAX, the **C**ausal **MA**p e**X**periment designer, a module in the POIROT integrated learning system. In the POIROT system, *other* modules play the role of "planning knowledge extractors," and their products are *hypothesized planning methods* – planning methods that attempt to appropriately generalize from the trace to solve a whole class of problems. Our design goal for CMAX is to automate testing any of the different kinds of assertions the knowledge extractors might make in their hypothesized methods. In POIROT, these methods are expressed in the *Learnable Task-Modeling Language* (LTML). LTML is very expressive and includes constructs for representing data flow between step outputs and inputs, and control flow constructs asserting step orderings, conditional execution, and loops. So far we have focused on analyzing and testing planning method data flow and plan step ordering constraints. 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 primitive operator models provided as prior knowledge*.

In the next section we describe the three phases involved in CMAX's approach to designing step-ordering experiments. The first phase identifies assertions about step ordering in hypothesized methods in order to characterize the space of possible reordering experiments. Phase 2 efficiently generates the set of possible step-reordering tests for the hypothesis space identified in Phase 1. The last phase plans experiments using the tests to isolate (to the extent possible) conditions under which the hypotheses about the existence of step ordering constraints are shown to be true or false. We then describe the role of CMAX in the larger POIROT integrated learning architecture, place CMAX in the context of prior and ongoing work related to planning experiments to improve domain knowledge, and conclude with a brief discussion of next steps in CMAX's development.

Generating Step-Order Experiments

In the POIROT domain, planning knowledge is represented as sets of planning methods expressed in the LTML task modeling language.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[1], 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.[2] (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 in the IL project assertions about step order constraints are common. 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. 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.

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. Throughout this discussion we will focus on how to generate the specific subsequences of steps that we wish to re-order. However, we assume that these subsequences are portions of an overall "complete" workflow whose other steps remain in the same order they would be using the original, unmodified methods to generate a workflow.

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. In order 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 $a \prec c$. If we add one more element, d, to the right-end of $a\ b\ c$, then we introduce three more order relations: $a \prec d$, $b \prec d$ and $c \prec d$. 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 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 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. The need for data flow representation arises because the POIROT system operates in a semantic web service domain, where primitive actions are calls to web services with input parameter values and possible return value outputs. LTML provides a *links* construct to specify how output values resulting from web service calls are used as input values to later service calls.

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

[2] 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.

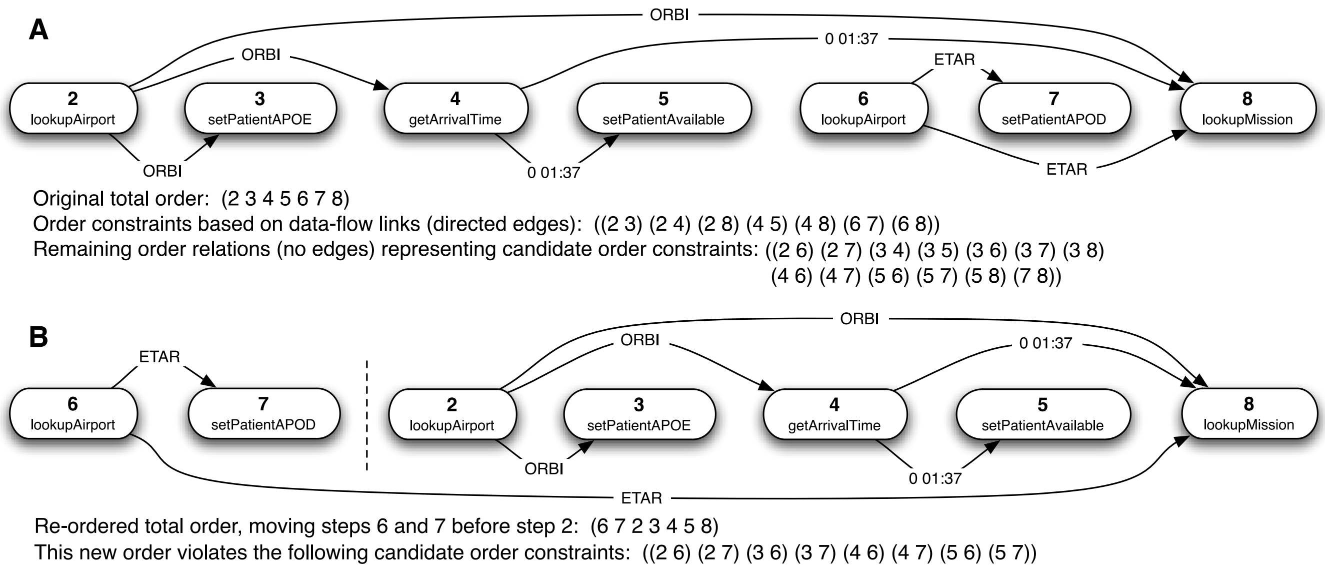

Figure 1: **A** represents the link-based order constraint relations (directed edges) identified for a *sequence* in a method fragment; the pairs of step numbers in parentheses represent individual order relations; labels on the edges represent values that would be assigned to the corresponding links 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.

In the version of the step order experiment design algorithm we present here, CMAX generates orderings of steps that *preserve* these 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.[3] For the seven steps in Fig. 1, there are actually only 144 permutations that

[3] While there is no known formula for calculating the number of permutations that preserve arbitrary ordering constraints (Brightwell & 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 $n!/c!$.

satisfy all of the identified link order constraints – an order of magnitude fewer than the full set of unconstrained permutations. The following algorithm generates all, and only, the permutations satisfying the set of (consistent) ordering constraints.

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 and b thus produces: 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.

The operation above distributes elements that have no order constraints with elements already in the sequences. There are three cases to consider when distributing an element that *does* have order constraints involving one or more elements in the sequence:

- Case 1: Suppose we add element x so a sequence containing elements of types y and z. If x only has order constraints with the y's, such that for every y: $y_i \prec x$, but no constraints with the z's, then x can be distributed over any of the positions to the right of the right-most y. Note that if there are no z's to the right of the right-most y, then distributing x results in only one new sequence: the original sequence with x appended to the right-end.
- Case 2: This is the converse of case 1; suppose instead that for every y: $x \prec y_i$. Now x can only be distributed over any of the positions to the left of the left-most y.
- Case 3: Now suppose there are three types of elements y, w and z. Again, x has no order constraints with the z's. If x is constrained with y's and w's such that for every y: $y_i \prec x$ and for every w: $x \prec w_j$, then x can only be distributed over positions between the right-most y and the left-most w.

For Cases 1 and 2 we noted that x has at least one position it can occupy – at one end of the original sequence – while not violating it's existing order constraints with the y's. This is important because while we are presenting the distributing operation as a constructive method of generating sequences that result from *adding* elements, what we are eventually producing is a set of permutations where every resulting permutation involves *all* of the individual elements. We therefore need to make sure Case 3 does not preclude x from appearing somewhere if x is a member of any of the other sequences. To show this, we make the following two observations:

1. If x is in the original set of elements and has the constraints that for all y: $y_i \prec x$ and for all w: $x \prec w_j$, then for all y and w: $y_i \prec w_j$. This follows from transitivity the of the $\prec$ relation: $y_i \prec x \prec w_j$.
2. Given observation 1, it follows that there must be at least one position for x between the right-most y_r and the left-most w_l because simply placing and element between y_r and w_l does not violate $y_r \prec w_l$, and neither does it violate $y_r \prec x$ and $x \prec w_l$.

The final extension to the distribution operation involves handling the case where we add *two* new elements, p and q, to the sequence while also maintaining $p \prec q$. In the base case, where neither p nor q have constraints with any of the elements already in the sequence, we distribute them as follows. Round 1: starting at the left-most end of the sequence, insert p, then distribute q over the positions around the elements to the right of p, including the position between p and the elements to the right. Round 2: starting with the original sequence (in the state before we inserted p and q), this time insert p to the first free position directly the right of the position we inserted it in round 1; this means the first element in the original sequence is to the left of p. Again distribute q, but this time over the remaining elements to the right of p. And we keep doing this: for the $i + 1^{th}$ step, apply the procedure, each time starting with p in the next position to the right of its starting position in the i^{th}, until p is finally placed in the right-most position of the original sequence and q is placed to the right of p. This completely distributes p and q while maintaining $p \prec q$ in each generated sequence. What about cases where either p or q have different ordering constraints with elements already in the sequence? We could enumerate each possibility in a separate case, but we don't actually need to because we only end up needing the base case of the two-element distribution operation while constructing our set of order-constraint-preserving permutations. We now present the complete permutation generation algorithm.

The first step in the algorithm is to generate all of the permutations of elements involved in just the link-order constraints. (This *can* involve all of the elements in the original sequence; we deal with any remaining elements below). We do this by incrementally adding each ordered pair of steps in the list of ordering constraints to the accruing set of permutations. Using the example in Figure 1 **A**, we start with $2 \prec 3$. Next we distribute $2 \prec 4$. Since 2 is already in the sequence, we simply distribute the single element 4 over the position(s) to the right of 2. Next we distribute 8 following the $2 \prec 8$ constraint, and so on.

We run into trouble, however, when we add $6 \prec 7$. Since neither step is constrained by steps already in the sequences, we use the two-element base-case distribution operation. While there's nothing wrong with the two-element distribution method itself, there is a problem with having distributed $6 \prec 7$ this early. By doing so, one of the sequences generated is (2 3 4 5 8 6 7). In the next step, we then try to add $6 \prec 8$, which we now see constrains 6 to only occur *before* 8. To avoid this kind of problem, we ensure the following does not happen: when adding $x \prec y$ and neither x nor y appears already in the sequence, test whether there is another, yet-to-be-added order constraint such that one of its elements is x or y *and* the other element already appears in the sequence. If so, do not add $x \prec y$ yet; push it to the end of the list of to-be-added constraints and select another order constraint, again testing for this condition. What this ensures is that we don't completely distribute two new steps thereby violating a constraint that will be added later.

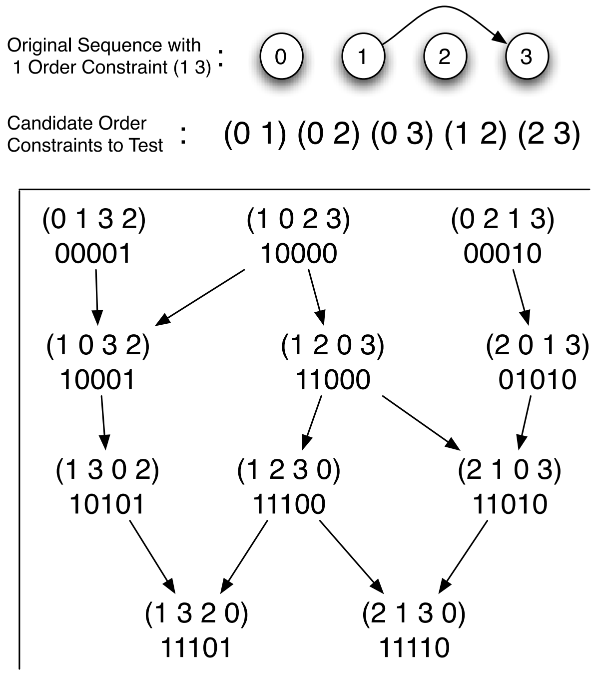

Figure 2: Test Dependency Graph

After generating all of the permutations involving the ordering constraints, any remaining steps, by definition, have no constraints with any of the steps in the sequences. We distribute them, one at a time, over the set of 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 $O(n!)$ (both in time and for the space required to store the generated sequences for the next phase). As long as there are a significant number of order constraints, time and space is much better.

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 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 orderings of these steps that can be used to test whether the ordering constructs are valid. Each experiment generated in this phase 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).

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.[4] Any

[4]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).

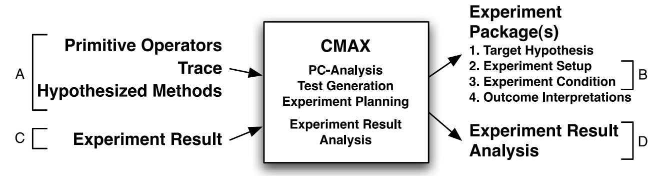

Figure 3: Protocol for Interacting with CMAX

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 $O(\log n)$ 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 executed and 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.

CMAX as a Module in a Learning System

Figure 3 provides a schematic diagram of the protocol for interacting with CMAX as an experiment generating and testing module. Because experiment generation is an iterative process that depends on the outcome of executed tests, there are several phases to calling CMAX as an experiment generating service. The arrows in the figure represent data flow. When CMAX is initially called (Step A), it is given as input a set of primitive operators, the demonstration trace (this is used in other processing not covered here), and a set of hypothesized methods. CMAX then goes through the phases of method analysis, test generation and experiment construction, resulting in the output of an *experiment package* (Step B). The experiment package includes a set of workflows representing the tests to be run. CMAX may identify several different method order constructs to test and each of these generates a separate experiment, output in a separate experiment package. In the POIROT system, a control facility selects which experiment packages to run. When selected, the workflows are sent to an executive and executed. Workflow execution success or failure then sent back to CMAX for the next phase of experiment design (Step C). In most cases, this involves selecting the next set of tests according to the experiment generation policy, and sending a new experiment package out. When an experiment has exhausted the relations to be tested, CMAX issues a final report describing any changes that may need to be made to the order construct of the method (Step D).

Conclusion

We have presented an approach to generating experiments that test planning knowledge about step ordering constraints. There are several directions to go in future work with CMAX. We have just begun to characterize the test dependency graph and there may be additional structure that we can put to work. Also, there are many other kinds of planning knowledge hypotheses that we would like to test. While much of the step-order generating machinery can be applied to testing data-flow link constraints, we also need to consider how we can generate viable alternative links without rendering a method unusable. Other constructs include branching conditions and looping constructs. In general, these extensions will require moving beyond the purely syntactic combinatorics of generate and test.

Acknowledgments

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

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