Object Histories: object-oriented triumph over action-nets

Donald H. Mitchell

Proactive Solutions, Inc.
10814 S. Quebec Ave.
Tulsa, OK 74137
dmitchell@trc.amoco.com

ProAct is our planner for direct end-user use in project management and contingency planning. We're building it after extensive experience with SIPE-2 (Wilkins, 1989) and discussions with Austin Tate and Brian Drabble. We rely on untrained end users to create the plan knowledge. Due to the nature of their planning needs, we've made several significant changes to the functionality and representation over other classical planners. These changes have rewarded us with additional benefits. Among the benefits is the advantage of using a fully object-oriented representation.

We no longer use an action-net but instead use what we call "object histories" to represent the modifications to objects over the plan. Experiments and formal analyses show that object histories almost always outperform action nets for question-answering (or Modal Truth Criterion (MTC)) and detection of precondition clobberers.

In this paper, I'll explain why we've chosen a radically object-oriented approach, how we represent "actions", and how the MTC algorithm works. Due to space, I will be unable to address how task insertion works including precondition testing and detection of precondition clobberers. I will also be unable to explain how we handle order linking and other issues related to using this representation.

1.0 Why object-oriented?

As we move automated planners out of the laboratory and into practice, we're seeing two distinct application arenas: robotics (situated automaton) and end-user planning.

Robotics planning appropriately sees real-time reaction and rich formal causal representations as critical for success. To achieve this success, robotics planners are developing architectures that require extensive AI knowledge to program and build. Each application is painstakingly crafted by experts for long-term and repeated use.

End-user planning, on the other hand, requires accessible programming and debugging, rich interfaces, and a minimum of representational distinctions. In our investigations, end-user planning does not require the reactivity that robotics requires. It does, however, impose unique and challenging requirements. Most new plans introduce new initial states, goal states, and actions. That is, these planning elements cannot be defined ahead of time; thus, the planning environment must provide accessible methods for the user to define their own states and actions. As in other planning domains, change occurs. The user may change the initial or goal states at any time. These changes can include adding new objects.

In general, users are more familiar with object-oriented concepts than declarative concepts. To facilitate the user's ability to program our planner, we've opted for a fully object-oriented representation. The user has full access to all the domain knowledge and can create, edit, and delete any domain information they wish including the class hierarchy, instances, and task templates (which are objects). The instances define the "current" or "initial" state. Users can also create and edit plan objectives.

Of critical importance to our users is the ability to see what exists in the domain, to see what can be said about those objects, to extend what exists and what can be said, and to see the changes predicted by the plan. Using objects and class hierarchies enables us to meet these needs.

By defining slots on classes rather than using arbitrary predicates, we enable the user to see at a glance everything that is known and can be said about each object. The user does not need to guess what each predicate applies to. Instead, the slots of the objects play the role of the predicate. Using CLOS's MOP, the planner can show the user the list of applicable slots at each place that the user needs to define a goal, effect, or condition. In addition, using a slot facet we've defined for recording the legal value-type of each slot,[1] the planner can also show the user all the legal values. These capabilities enable us to build a powerful, structured interface for defining plan knowledge, building plans, and analyzing plans.

2.0 Plan representation

Our plans are hierarchical and non-linear. The planner does not insist on uniform expansion (i.e., that each "level" be fully expanded or that if there is an action at a given level, then all actions have either a hierarchical descendent or ancestor at that level). There is not a strict notion of level. As far as non-linearity goes, the planner allows any ordering of actions as long as the order remains consistent: that is, as long as the imposition of the order does not create a temporal circularity. Thus, there may be ordering constraints between actions at different levels of the plans or in different abstraction subtrees of the plan.

A "completed" plan does not have to be completely ordered, does not have to have specific assignments for each "variable," and does not have to have an operator for each goal. That is, any plan is complete as long as it is consistent. A plan prescribes and describes the anticipated changes in the world. The user may begin execution at any time.

2.1 Domain objects

Users create and modify the classes representing the objects in their domain. These classes are full-fledged CLOS classes. For each class, the users can create and modify the slots and their attributes. We provide four attributes or facets for each slot: the type of object that it takes as a value, whether it takes multiple values or can only have one value at a time, who can see the slot, and an optional documentation string describing the slot. Users can also create and modify instances for any class. We use this object, slot, value representation in place of traditional planning system's sentential representation.

The planner includes a full modal representation for "variables." For each variable used in the plan that does not have a unique value due to unification, we create a new object. Following Wilkins, we name these objects "indefinite objects." We use a defeasible truth maintenance system ala Doyle (1979) to track the necessary and possible matches (we call them "codesignations" after Chapman (1987)). Bill Davis is publishing a paper in this workshop on this codesignation system (Davis, 1993). We require the user to specify a class for each variable. Each class in the user's class hierarchy then keeps track of its indefinites as well as its instances.

2.2 Actions

Users create and modify operator instances (task templates) which are the schemata for events or actions in the domain. They manipulate the operators through an object-oriented interface using the presentation capabilities of CLIM. They move objects or classes into the panes representing **use-only-when**, **steps**, **main-effects**, **side-effects**, resource characteristics, and prerequisites of an operator. They then specify which slots of these objects this operator cares about and what, if any, value constraints the operator imposes on those slots. The user specifies value constraints by moving in objects or classes from the type hierarchy.

The planner then instantiates these task templates into actions in the plan. Actions keep track of the operator that expanded them (if any), the mapping between the variables in the template and the indefinite or definite objects in the plan, how the planner satisfied each **use-only-when** and **value-binding** condition, the purpose of the action (a goal), what action the action refines, what actions refine this action, and what changes the action produces.

In addition, there are two hash tables for each action that record that action's predecessors and successors. The tables do not merely encode an action net but the whole precedence relation. That is, in the plan $a \rightarrow b \rightarrow c$, a's successors table would include both b and c. When the planner imposes a new precedence relation between the two nodes e and f, it copies the contents of f's "successors" hashtable into e's and into all the predecessors of e (needing to copy into all the predecessors' tables is why there

1. CLOS allows programmers to specify types for slots; however, in our planner, all slots can hold not only elements of the slot's type but also plan variables. Thus, we defined our own slot facet to hold the slot's "type."

must be two tables). Similarly, it copies *e*'s predecessor table into and beyond *f*. It stops copying the hashtables wherever it finds that there are no new elements to add. Thus, in a simple plan, looking up the precedence relation of any pair of nodes is a constant time operation while imposing a new precedence relation can take up to $O(n^2)$. There is a complication, however.

When the planner adds an expansion (or "refinement") to a node, it does not copy the precedence tables down to the new action nodes. Similarly, as the planner adds new elements to a node's tables, it does not copy those elements up or down the abstraction hierarchy. Thus, to find out whether any action comes before, after, or is unordered with respect to another action, the planner must check the action's hashtables. If it does not find the answer there, it must successively check the abstraction ancestors until it finds a record. If it does not find a record, then the nodes are unordered where "unordered" includes nodes on parallel branches of a traditional action net and nodes in ancestral relationship to one-another. This precedence test takes $O(\log^2 n)$ where n is the number of actions in the graph and thus $\log n$ is the expected height. The storage space for all the hashtables is $O(n^2)$.

In a standard action net, precedence tests require finding paths and take at least $O(n)$. Precedence tests are very frequent operations; thus, the constant time to $O(\log^2 n)$ complexity is very desirable.

The planner includes two other functions for making distinctions among the "unordered" relations: a function to test whether one action is an abstraction ancestor of another $O(\log n)$ and a function to determine whether one cannot come after another. An action cannot come after another if it must come before, it is an ancestor, one of its descendents needs to come before the other action, or one of the other action's descendents needs to come after it. These last two relations are unique to our planner and make the non-uniform expansion and ability to impose arbitrary precedence relations very powerful. The cost of this test can be as high as $O(n^2)$ — the square of the number of descendants from the test action.

2.3 Object histories

Given the object-oriented representation, we began to think of how to represent changes to objects over the plan. We[1] developed a lightweight object type that contains only the slots whose values change and additional information needed to record how the change fits into the plan. We call these objects, **change-objects**. Each change object has the following "behaviors":

- **original-object**: returns the object that this change object represents a change to.
- **previous-change-objects**: returns list of immediately preceding changes for the same object.
- **next-change-objects**: list of immediately following changes to this object.
- **higher-level-change**: the hierarchical parent to this change.
- **refinements**: the hierarchical children of this change
- **causing-action**: the action that generated this change.
- **enabled-actions**: all actions that in some way count upon a change in this change object along with information about which change (slot-name and value). This slot represents what Wilkins (1990) calls protect-until and what Currie and Tate (1991) call the GOST.
- **slot-change-p**: given a slot name, this function indicates whether that slot changed at this change object.
- **slot-multiple-valued-p**: indicates whether a slot can take multiple values. We use this information to interpret the semantics of change in the MTC algorithm.
- **slot-values-added, slot-value-asserted**: the multiple valued and single valued form for retrieving the new values.
- **slot-values-removed**: for multiple valued slots, what values were negated at this change.

All domain objects also support the **next-change-objects** behavior. Thus, each object points to a graph containing all its changes. This graph represents the history of that object or an object-centric action net. If a change occurs to an indefinite object, then that change is inserted into the history for just that indefinite object. That is, indefinite objects also support the **next-change-objects** behavior. The object history of any given object only contains the changes that necessarily occur to that object.

Change objects that have refinements do not maintain their **next-change-objects** or **previous-change-objects** links. The planner maintains a graph for only the most detailed level of change. With only one level, the planner can

1. Jay W. Tompkins came up with the original idea and design.

maintain an accurate graph even given our flexible precedence imposition rules. With more than one level, it is very difficult to maintain a graph in a traversable form.

3.0 Deriving PERT charts

The user often wants to see the plan merely from the perspective of some object. In this case, our representation provides an immediate correspondence. More often, however, the user wants to see the "whole" plan. As long as we restrict the user to only seeing one plane through the plan, it is easy to generate this view from our representation. By one plane, I do not mean that it must be a specific level but that the user cannot view an action and its abstraction in the same graph. Generating these graphs takes $O(n^2)$ where n is the number of nodes in the graph.[1] Modifying a previously generated graph to replace a node with its refinement or abstraction requires $O(n)$. Modifying it to reflect subsequently imposed precedence relations can also be done quickly.

1. We have an $O(\log n)$ algorithm based on quicksort that we haven't implemented.

Figure 1 shows a simple single-level plan from both the traditional action-net representation and the object history representation.

4.0 The Modal Truth Criterion Algorithm

The Modal Truth Criterion (MTC) or Question Answering algorithm uses these object histories and the precedence function and does not use the actions.

The MTC algorithm starts with at least an object, attribute, action, and either the symbol **before** or the **symbol** after to indicate at which edge of the action to test the truth. If the caller supplies no value, then the MTC determines and returns all possible values distinguishing necessary from possible values and indicating the required binding and linking constraints. If the caller supplies a value, the MTC tests whether that value possibly or necessarily holds and returns any constraints required to make it necessarily hold.

The MTC algorithm first uses the codesignation lookup function to determine which base and indefinite objects possibly and necessarily unify with the object of the query. The algorithm then traverses the object histories for these

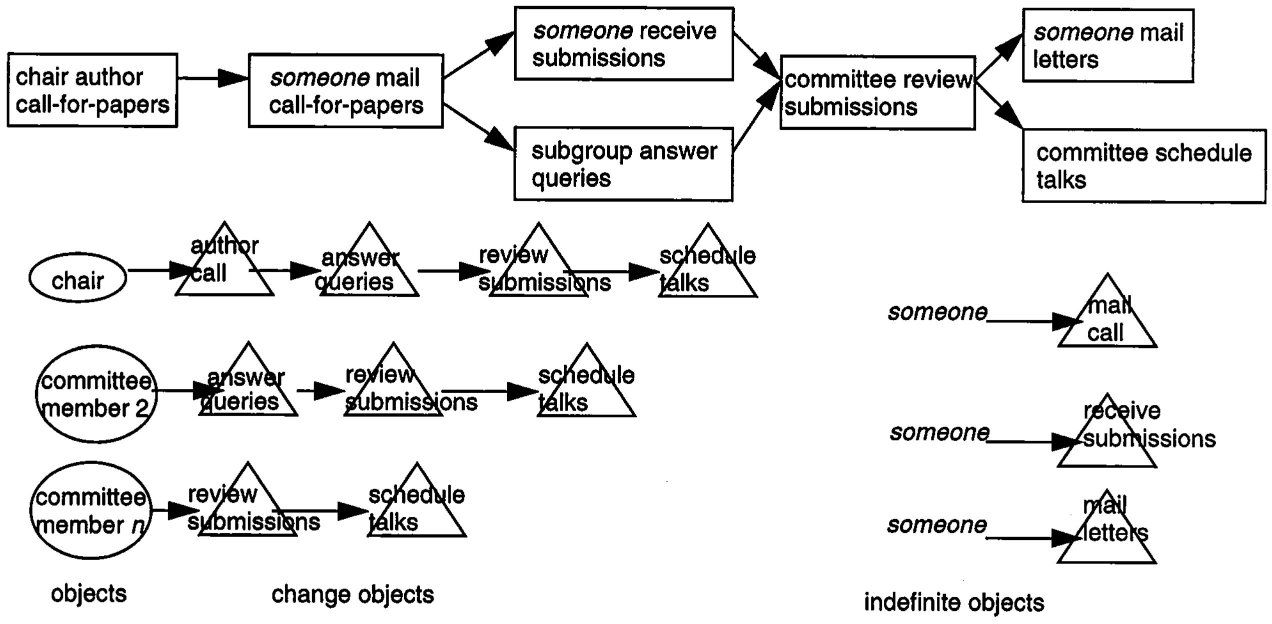

FIGURE 1. Example traditional action-net and the new object histories. Example highly simplified by showing only one level without any action-net confusing cross-links.

objects. Because these histories only have as many vertices as there are changes to the specific object in the plan, these histories will never be larger that the analogous action net would be. In most cases, the histories will be considerably smaller than the action net because it is unlikely that every action will change the same objects.

To use the example in Figure 1, a query about committee member *n* if only the last *"someone"* possibly codesignates with that committee member would look through just those two histories. The algorithm would use the nearly constant time precedence lookup function to know when to quit looking down the histories and to adjudicate among its findings between histories.

Of course, if the query were about the chair and the chair could be any of the *someones*, then the algorithm would need to look through the four object histories.

By determining the possibly unifying objects at the start, the algorithm does not have to perform unification at each change against the query object. The only unification it performs is against the value if one was supplied as part of the query.

As you can see, object histories allow us to restrict the MTC search space and reduce the unification checks. In plans that contain changes to the query object or one of its possible unifications at every action, our algorithm will do the same amount of graph traversal as a typical MTC algorithm; however, our experience shows that these plans are unusual unless they are very simple (a short set of cumulative changes to the same object). Even in this unusual case, our algorithm saves the effort of looking through all the add and delete-list items at each action and immediately restricts is attention to just the changes that possibly unify with the query object. The only added expense of our algorithm is computing all the possibly unifying objects at the beginning.

5.0 Summary

We've succeeded at finding a new and effective representation for automated planning by setting aside sentential logical formulae and looking at the implications of a fully object-oriented system. Because of space, we've only had time to describe the beneficial implications of this change for the MTC algorithm. It also benefits precondition clobbering tests (traversing all the possibly unifying histories looking for **enabled-actions** using the changed slot-name and spanning the time of the inserted effect).

A primary motivator in this switch is that we want users to be able to develop and extend their own knowledge bases. Our experience with users shows that sentential representations are confusing to them. Users want to know what they have to work with and what they can say about those things. The object-oriented representation with class browsers and explicit inclusion of slots makes these transparent.

6.0 References

Chapman, David. Planning for Conjunctive Goals. *AI, 32*(3), 333-377, 1987.

Currie, K. and A. Tate. O-Plan: the open planning architecture. *AI, 52*(1), 49-86, 1991.

Davis, William S. Defeasible Codesignation Constraints for Planning Variables. In the 1993 Spring Symposium on Foundations of Automatic Planning: the Classical Approach and Beyond, AAAI, 1993.

Doyle, John. A Truth Maintenance System. *AI, 12*(3), 231-272, 1979.

Wilkins, David E. Can AI Planners Solve Practical Problems. *Computational Intelligence, 6*, 232-246, 1990.