Tech Report AI96-242, Dept CS, Univ Texas at Austin, 1996.
http://www.cs.utexas.edu/users/mfkb/papers/uttr96b.ps

A Compositional Approach to Representing Planning Operators[1]

Peter Clark, Bruce Porter and Don Batory
Dept. Computer Science
Univ. Texas at Austin
Austin, TX 78712, USA
{pclark,porter,dsb}@cs.utexas.edu

Abstract

AI has frequently been criticized for being 'stuck in the microworld' because of the common inability of AI systems to cope with the complexity of real domains. Often, adding details removes regularity, transforming a representation from a few simple structures to a large, unwieldy collection of specialized ones.

This paper addresses this problem in the context of representing planning operators (domain-specific knowledge about the effects of actions in a domain) for use by AI planning systems. We present a novel approach in which domain-specific operators are represented as a composition of general components, and show that the problem of manually building a detailed set of operators can be avoided by constructing them from a small number of such components instead. Each component encapsulates information about a domain feature that might be modeled, and each may contribute to several operators. Moreover, we describe how the choice of what to model and what to ignore in a domain can then be easily varied, simply by controlling which components are used. Finally, we show how operator sets built in this way can be used by planning algorithms.

1 Introduction

AI has frequently been criticized for being 'stuck in the microworld' (eg. [4]), referring to the common inability for methods to scale up from toy domains to more complex, real-world domains. Microworlds offer simple regularity, whereas the diversities and idiosyncracies of the real world are often complex to represent and reason about. One cause of this scaling difficulty is that adding detail removes regularity, and hence can potentially transform a representation from a simple set of general statements to a large, unwieldy collection of specialized ones.

We address this problem in the context of representing planning operators – domain-specific knowledge about the effects of actions in a domain – for use by AI planning systems. We present a novel approach in which an operator set is built from components, rather than manually enumerated. Each component encapsulates information about a feature of the domain that may contribute to many plan operators. We show how the compositional approach addresses the problem of building complex, detailed representations by factoring them, and thus can help ease the representational task in a domain. In addition, we describe how the choice of what to model and what to ignore in a domain can be easily varied with this approach, based on the particular requirements of each planning problem, simply by controlling which components are composed into the operator set. Finally, we show how planning algorithms can be easily modified to use operators specified in this manner.

Our approach draws heavily on a recent, novel approach to software reuse, called the GenVoca method, developed in software engineering [1, 2, 3]. GenVoca automatically generates implementations for a set of software operations (analogous to plan operators in our domain) by combining

[1]Support for this research is provided by a grant from Digital Equipment Corporation and a contract from the Air Force Office of Scientific Research (F49620-93-1-0239)

software components. In particular, we import the insight from GenVoca that the basic representational unit should not be a software operation (plan operator), but rather a domain feature that may contribute to many operations (operators).

For convenience we present this work using the STRIPS planning formalism, in which operators are represented using precondition, delete and add lists [7, 6]. It is important to note that we are not claiming anything special about the STRIPS representation, and we acknowledge its many limitations. Our approach is not specific to this formalism, and can also be applied to other formalisms where the effects of an action are represented as an individual data structure. Also note that our work concerns the construction of operator sets for planning, not planning algorithms themselves.

Section 2 introduces the problem caused by adding details to a representation, and Section 3 presents our solution. Section 4 describes how a system can plan using representations built from components, and how composition and planning can be interleaved. Finally, Section 5 discusses the compositional approach, and its implications for abstracting and extending domain representations.

2 The Problem

2.1 Planning

For presentation purposes, we use the STRIPS formalism throughout this paper, noting that the same approach can be applied to other representations. In this formalism, the state of the world is represented by a set of ground assertions or **facts**, expressing things that are true in that state (eg. `on(a,b)`). Actions are represented by **plan operators**, consisting of **precondition**, **delete** and **add** lists. Preconditions are the facts that must be true in a state before an action can be carried out, and the delete and add lists are the facts that should be removed and added (respectively) to the current state in order to model the effects of that action. For example:

```
put(Obj,Container) ::
   pcs: holding(Obj)
   del: holding(Obj)
   add: in(Obj,Container)
```

denotes a `put` operator, where the abbreviations `pcs`, `del`, and `add` indicate the preconditions, delete and add lists respectively. We sometimes refer to the pcs/del/add lists as the operator's **implementation**. For notational convenience, we also use projections (denoted '`.`') to express 'chains of clauses'. For example, the chain of clauses used to test whether a container's lid is open:

```
lid(Container,Lid), status(Lid,Status), Status = open
```

can be re-expressed equivalently as

```
Container.lid.status = open
```

Many planners use operators represented with this formalism, or a similar one (eg. [7, 13, 12, 9]). A simple one is shown in Figure 1; given a goal, the planner finds an action to achieve it, (recursively) plans how to achieve the preconditions for that action, and returns, as the final plan, the concatenation of those plans plus the original action. Backtracking is used to remake decisions at choice-points should failures occur, in the normal fashion.

For discussion purposes, consider the operator set for the simple robot domain shown in Figure 2. The robot can move between rooms, pick up and drop objects, and use containers to put and get objects. We can describe these activities using six operators (showing just the first in full):

```
ACHIEVE-GOAL(Goal, ProtectedGoals, State) -> Plan
  IF   Goal already true in State
  THEN RETURN Plan = {}
  ELSE
        1. Find an operator Op that achieves Goal
           (ie. has Goal in its ADD list)
        2. For each precondition PCi of Op:
              ACHIEVE-GOAL(PCi, ProtectedGoals, State) -> Plani
        3. IF   Op doesn't destroy any ProtectedGoals
                (ie. none of its DEL list members are in ProtectedGoals)
           THEN - Simulate doing action Op:
                   MODIFY State using Op's ADD and DEL lists
                - Add Goal to ProtectedGoals
                - RETURN Plan = concatenation of Plani + Op
```

Figure 1: The STRIPS planning algorithm.

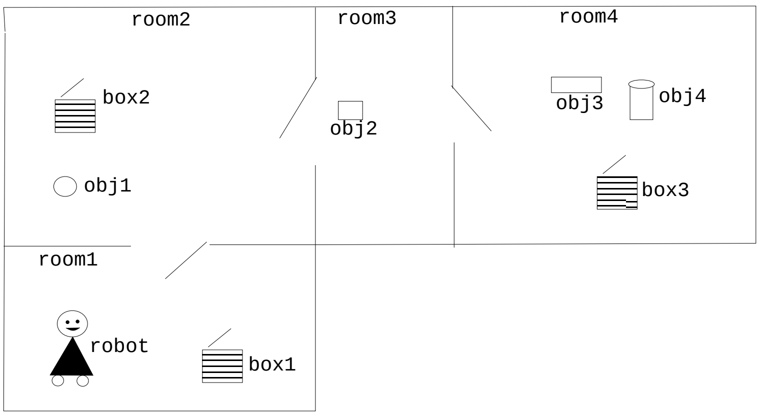

Figure 2: A Simple Robot World.

```
 get(Obj,Container) ::
   pcs: in(Obj,Container), nextto(robot,Container)
   del: in(Obj,Container)
   add: holding(Obj)
```

`put(Obj,Container) ::...`	(obj becomes in container)
`goto(Obj,Room) ::...`	(if in same room, robot becomes next to obj)
`gothru(Door,Room1,Room2) ::...`	(if rooms connected, robot location becomes room2)
`take(Obj) ::...`	(if next to obj, robot becomes holding obj)
`drop(Obj) ::...`	(robot stops holding obj)

Using this representation, a planner can generate simple plans adequate for this microworld. However, the operator set is a relatively naive representation of real-world domains (eg. a warehouse) in which a rich variety of different types of objects are manipulated. The operators ignore many aspects of the real world (eg. whether an object will fit in a container, or whether the robot is strong enough to lift a particular object). For the planner to take these aspects into account, we need to add more detail to the domain representation.

2.2 More Detailed Models of the World

To illustrate the scaling problem caused by adding detail, consider the features of the domain that the above formalism omits:

- size of objects (will an object fit through a door?)
- capacity of containers (will an object fit in a box?)
- robot strength (is the robot strong enough to move an object?)
- container strength (is a container strong enough to hold an object?)
- lids (is the lid of a container, or the door of a room, open?)
- liquid containment (is the container water-tight?)
- ...

We can, of course, elaborate our planning operators to include these features. Consider, for example, an operator for putting a heavy object into a container:

```
put(Obj,Container) ::
 pcs: nextto(robot,Container)                   % proximity
      Container.freespace > Obj.volume          % volume
      Container.portal.lid.state = open         % container-lid
      strong(robot)                             % robot-power
      Container.hold_strength > Obj.weight      % strength
      holding(Obj)                              % containment
 del: holding(Obj)                              % containment
      Container.freespace = X                   % volume
 add: in(Obj,Container)                         % containment
      Container.freespace = X-Obj.volume        % volume
```

Here, we have added various conditions to model some additional domain features, named in the comments beside the operator. `Container.portal.lid.state = open`, for example, takes the lid of the container into consideration, and ensures that it is in the correct state.

This elaboration, however, has led to a problem: The operator no longer describes a generic `put` operation, but instead describes a specific kind of `put`, namely putting a heavy, rigid object into a lidded, rigid container. Other kinds of put actions require different representations, depending on the properties of the object and container involved. For example, if the container is elastic then the constraints on container volume should be modified, or if the object is a liquid then a precondition that the container is non-porous might be needed. Operators become more idiosyncractic as details are added, and it becomes laborious or impossible to enumerate the complete list of variants.

However, at the same time there is much repeated structure within these variants. For example `Container.portal.lid.state = open` applies to all operators involving transfer through a lidded portal. The model of operator composition we present is based on isolating and encapsulating these fragments of structure, as we now describe.

3 A Compositional Approach to Representing Planning Operators

In this section, we present an alternative approach to representing planning operators in which a set of operators is specified as a *composition* of components. Our goal is to show how the potentially large operator set in a detailed representation can be specified as compositions, rather than enumerated manually. In Section 4 we show that while the complete operator set could then be synthesized automatically before planning, in practice this is unnecessary; the operator set can be synthesized on demand by interleaving composition with planning.

3.1 Components Encapsulate Domain Features

The fundamental insight that we import from the GenVoca model is that the basic unit of representation should be a *domain feature*, not a plan operator. A domain feature is a single aspect of the domain we wish to model, such as (from Section 2.2) container volume and robot strength. A domain feature (henceforth, 'feature') may contribute information to several plan operators, and hence can be thought of as orthogonal to plan operators. Each feature is represented by one *component*, specifying that feature's contribution to a set of plan operators.

To illustrate the encapsulation of a feature in a component, we compare two alternative representations of a trivial domain in which a robot can only move objects between containers. A simple representation of actions in this domain might consist of just two operators:

```
            REPRESENTATION I
            ================
put(Obj,Cont) ::
   pcs: holding(Obj)
   del: holding(Obj)
   add: in(Obj,Cont)

get(Obj,Cont) ::
   pcs: in(Obj,Cont)
   del: in(Obj,Cont)
   add: holding(Obj)
```

This representation ignores many features, eg. whether the box's lid is open, whether the object will fit in the box, whether the robot is strong enough, etc. An alternative, where (say) we also model the box's lid, might be this set of four operators:

```
              REPRESENTATION II
              =================
put(Obj,Cont) ::
   pcs: holding(Obj), Cont.portal.lid.status=open
   del: holding(Obj)
   add: in(Obj,Cont)

get(Obj,Cont) ::
   pcs: in(Obj,Cont), Cont.portal.lid.status=open
   del: in(Obj,Cont)
   add: holding(Obj)

open(Portal) ::
   pcs: not Portal.lid.status=open
   del:
   add: Portal.lid.status=open

close(Portal) ::
   pcs: Portal.lid.status=open
   del: Portal.lid.status=open
   add:
```

In representation II, an extra precondition has been added to the `get` and `put` operators, and two new operators (`open` and `close`) have been added. In fact, it only makes sense to apply these changes to representation I in their entirity, or not at all. They constitute a *single* abstract notion (activities involving lidded containers) and should be encapsulated as a single unit or *component*. Writing down just these changes, the component looks:

```
            CONTAINER-LID COMPONENT
            =======================
put(Obj,Cont) ::
   pcs: Cont.portal.lid.status=open
   del:
   add:

get(Obj,Cont) ::
   pcs: Cont.portal.lid.status=open
   del:
   add:

open(Portal) ::
   pcs: not Portal.lid.status=open
   del:
   add: Portal.lid.status=open

close(Portal) ::
   pcs: Portal.lid.status=open
   del: Portal.lid.status=open
   add:
```

This component thus describes just those aspects of the representation related to container lids; it encapsulates a particular feature that we may or may not wish to model.

In fact, we have undesirably had to repeat a single constraint (namely that a portal's lid must be open in order to move through it) in the preconditions of both the `put` and `get` operators (`Cont.portal.lid.status=open`). We can avoid this by instead introducing an **operator taxonomy**, and then creating a `move_thru_portal` operator as one generalization of `get` and `put`. We can then use the generalized operator to describe effects on more specific operators that involve moving through a portal. A revision of the CONTAINER-LID component in this way would look:

```
              CONTAINER-LID COMPONENT
              =======================
move_thru_portal(Obj,Portal) ::
   pcs: Portal.lid.status=open
   del:
   add:

open(Portal) ::
   pcs: not Portal.lid.status=open
   del:
   add: Portal.lid.status=open

close(Portal) ::
   pcs: Portal.lid.status=open
   del: Portal.lid.status=open
   add:
```

We additionally have to specify the `isa` relationship between `put` and `move_thru_portal`:

```
put(Obj,Cont) isa move_thru_portal(Obj,Portal) where portal(Cont,Portal)
get(Obj,Cont) isa move_thru_portal(Obj,Portal) where portal(Cont,Portal)
```

We also add a flag to `move_thru_portal` indicating it is a generalized operator, and hence should not be used by the planner as a primitive action in plans.

This improves on the original component's representation by making the concept we are actually describing (moving through a portal) explicit and removed the redundancy in the original description. In addition, the component is now more flexible: If other actions are introduced that involve moving through a portal (eg. `drop_into`, `throw_into`) they will automatically acquire the `Portal.lid.status=open` constraint from this more general component simply by declaring them as specializations of `move_thru_portal`, and not requiring the CONTAINER-LID component to be modified.

We can similarly construct other components encapsulating other features. A component *library* contains a set of components for a domain. For example, a small library might contain the components:

`CONTAINMENT(Obj,Container)`	(whether obj is in container or not)
`CONTAINER-LID(Obj,Container)`	(open/close/check container's lid)
`VOLUME(Obj,Container)`	(track and check sufficient free volume)
`STRENGTH(Obj,Container)`	(is container strong enough)
`STABILITY(Obj,Container)`	(is container positioned to avoid tipping)
`WATER-TIGHTNESS(Obj,Container)`	(only water-tight containers hold liquids)
`SIZE(Obj,Container)`	(can obj fit through portal)
`HEAT-PROOF(Obj,Container)`	(check container can hold hot objs)

Note that we parameterize components by the objects they involve. We describe a use of these parameters in Section 3.3.

3.2 Composition

Components represented in this fashion are simple to compose: form the union of the pcs/del/add lists for corresponding operators, and add any additional operators, introduced by any component, to the operator set. (This includes using the operator taxonomy, so that generalized operators contribute pcs/del/add conditions to more specific operators). Each component can be viewed as specifying *constraints* on the final operator set, by interpreting the pcs/del/add lists as *partial sets*. A partial set is an incompletely specified set, constrained to contain certain named members. For example (writing partial sets using {} braces), the partial set {`a`} denotes a set that contains at least `a`. If a set `S` is both a partial set `PSet1` and a partial set `PSet2`, then `S` must contain all the known members of `PSet1` and `PSet2`; hence `S` must also be a partial set containing the union of `PSet1`'s and `PSet2`'s known members. We thus combine or *unify* partial sets by unioning their known members. For example {`a`} and {`b`} combine to {`a,b`}, as a set containing at least {`a,b`} is both a set containing at least `a` and a set containing at least `b`.

In our case, each component specifies partial sets of pcs/del/add conditions, and their composition is the unification of corresponding partial sets. The composition is thus the most general operator set satisfying the constraints of all its components.

Using the symbol $\oplus$ to denote this composition operator, we can write *type equations*[2] to specify compositions. For example:

$$\texttt{OperatorSet = containment} \oplus \texttt{volume} \oplus \texttt{container-lid}$$

denotes an operator set that is the composition of three components. Figure 3 shows an implementation of these components and their composition.

3.3 Applicability Rules for Components

The above type equation specifies an operator set appropriate only for **lidded, rigid** containers and **rigid** objects. Different components will apply in different situations, depending on the nature of the items involved in actions. Thus, the type equation to use for constructing a particular operator depends on the types of objects that operator is manipulating.

To decide whether a component should apply for a particular operator and its arguments, we write a set of **applicability rules** that test the component's parameters. These parameters refer (ie. are variables shared with) the arguments of operators in the component, as illustrated in Figure 3. Thus, given bindings for a particular operator's arguments, a system can determine from these rules if a component should apply to that instantiated operator or not. An example set of applicability rules is:

```
CONTAINMENT(Obj,Container)     (always)
CONTAINER-LID(Obj,Container)   IF lidded(Container)
VOLUME(Obj,Container)          IF rigid(Container)
STRENGTH(Obj,Container)        IF heavy(Obj)
STABILITY(Obj,Container)       IF heavy(Obj) and narrow_base(Container)
WATER-TIGHTNESS(Obj,Container) IF liquid(Obj)
SIZE(Obj,Container)            IF not liquid(Obj)
HEAT-PROOF(Obj,Container)      IF hot(Obj)
```

[2] named after their equivalent in GenVoca

```
CONTAINMENT(Obj,Cont) ⊕   VOLUME(Obj,Cont)       ⊕        CONTAINER-LID(Obj,Cont)

get(Obj,Cont) ::        get(Obj,Cont) ::                 get(Obj,Cont) ::
 pcs: in(Obj,Cont)       pcs:                             pcs: Cont.portal.lid.status=open
 del: in(Obj,Cont)       del: freespace(Cont,V)           del:
 add: holding(Obj)       add: freespace(Cont,V+Obj.size)  add:

put(Obj,Cont) ::        put(Obj,Cont) ::                 put(Obj,Cont)
 pcs: holding(Obj)       pcs: Cont.freespace >= Obj.size  pcs: Cont.portal.lid.status=open
 del: holding(Obj)       del: freespace(Cont,V)           del:
 add: in(Obj,Cont)       add: freespace(Cont,V-Obj.size)  add:

                                                         open(Portal) ::
                                                          pcs: not Portal.lid.status=open
                                                          del:
                                                          add: Portal.lid.status=open

                                                         close(Portal) ::
                                                          pcs: Portal.lid.status=open
                                                          del: Portal.lid.status=open
                                                          add:

                       = OPERATOR SET

        get(Obj,Cont) ::
         pcs: in(Obj,Cont), Cont.portal.lid.status=open
         del: in(Obj,Cont), freespace(Cont,V)
         add: holding(Obj), freespace(Cont,V+Obj.size)

        put(Obj,Cont) ::
         pcs: holding(Obj), Cont.portal.lid.status=open, Cont.freespace >= Obj.size
         del: holding(Obj), freespace(Cont,V)
         add: in(Obj,Cont), freespace(Cont,V-Obj.size)

        open(Portal) ::
         pcs: not Portal.lid.status=open
         del:
         add: Portal.lid.status=open

        close(Portal) ::
         pcs: Portal.lid.status=open
         del: Portal.lid.status=open
         add:
```

Figure 3: Composition of Three Components to form an Operator Set.

Now, given a particular operator name and its arguments (eg. `put(book,waste_basket)`), the system can **construct** and **evaluate** the appropriate type equation to find that operator's implementation (ie. pcs/del/add lists). The type equation is constructed by first identifying components mentioning that operator, and then selecting only those whose applicability rules deem that component appropriate. The type equation is then evaluated by unifying the constituent components as described in Section 3.2. For example, from `put(book,waste_basket)`, a system can derive the type equation:

$$\texttt{OperatorSet = containment} \oplus \texttt{strength} \oplus \texttt{size} \oplus \texttt{volume}$$

Evaluating it (just for the `put` operator) produces:

```
put(Obj,Cont) ::
  pcs: holding(Obj), Obj.width < Cont.portal.width, strong(Cont)
  del: holding(Obj), freespace(Cont,V)
  add: in(Obj,Cont), freespace(Cont,V-Obj.size)
```

Similarly, for `put(gallon_of_water,paper_bag)` (say), we automatically derive:

$$\texttt{OperatorSet = containment} \oplus \texttt{strength} \oplus \texttt{water-tightness} \oplus \texttt{volume}$$

that evaluates to (for the `put` operator):

```
put(Obj,Cont) ::
  pcs: holding(Obj), strong(Cont), water_tight(Cont)
  del: holding(Obj), freespace(Cont,V)
  add: in(Obj,Cont), freespace(Cont,V-Obj.size)
```

In this way, a system can generate automatically the many variations of operators in a complex domain, and thus handle the representational combinatorics that a manual enumeration of operators would otherwise present. By scanning the applicability rules' conditions, the system can identify all the different permutations of situations (and hence the different type equations) that might arise, and hence enumerate the entire operator set for the domain. In fact, as we show in Section 4, this exhaustive enumeration (even though it is now automatic) can be avoided by interleaving composition with planning, generating specific operators only on demand.

4 Planning with Composed Action Rules

We have described how we can express knowledge about actions in the form of components rather than a set of operators, and then assemble operator representations by constructing and evaluating type equations.

One method for using this approach in AI planning would be to automatically enumerate the entire suite of possible operators[3], and then use them with a standard planning algorithm. However, this approach is impractical, as the enumeration could be large. In this Section, we describe an alternative that avoids enumerating the operator set. By interleaving planning with generation, operators can be generated on demand, and hence only as needed.

[3]by evaluating all possible type equations, and then adding preconditions to the generated operators to ensure the appropriate variant was selected for a given target action

4.1 Interleaving Planning and Operator Generation

We illustrate this modification for the STRIPS planning algorithm in Figure 1. Again, this modification is not specific to STRIPS – other algorithms can be modified in a similar way. The two basic steps in this algorithm are to select an operator that can achieve a particular goal (Step 1), and then find and satisfy that operator's preconditions (Step 2). If the set of operators were explicitly enumerated, then finding an operator and its preconditions requires a simple scan of the operator set. However, given just a set of components and their applicability rules, we must make a simple modification to these steps, as follows:

1. To find an operator that can achieve some `Goal`, scan the components for operators that have `Goal` in their add list.

2. To find the preconditions (and the delete and add conditions) for that operator, construct and evaluate its type equation (Section 3.3).

There is, in fact, a complication in step 2: it may not be possible to tell whether a component applies to an operator if that operator is incompletely specified (ie. has variables in its arguments). Consider, for example, the goal `holding(book1)`. By scanning the set of components for the domain (eg. consider those in Figure 3), a planner can identify one operation that might achieve that goal: `get(book1,Container)`[4], where `Container` is some container with currently unknown value. The planner now needs the preconditions for this operator, but has a problem: as the value of `Container` is currently unknown, it cannot tell whether components conditional on this value (eg. CONTAINER-LID) should apply, and hence cannot tell whether they should be in the type equation for this operator. If `Container` turns out to be `waste_basket`, for example, CONTAINER-LID would not apply but if it was `briefcase` then it would.

This problem can be handled by the planner first subgoaling on preconditions that *are* known to apply (ie. from components whose applicability conditions are satisfied), and deferring reasoning about components with unknown applicability until later. This corresponds to first working with those parts of the type equation that are known. In the course of generating a (sub)plan to achieve the known preconditions, variables may become bound and hence the applicability of other components become decided, thus allowing other preconditions to be identified and achieved. For instance, in the example above, CONTAINMENT is known to apply to `get(book1,Container)`, and contributes the precondition `in(book1,Container)`. By achieving the subgoal `in(book1,Container)` (simply a lookup), `Container` becomes bound to `book1`'s current container. Hence, the applicability of CONTAINER-LID can be determined, and planning can proceed. This modified algorithm is shown in Figure 4.

If a component's applicability is still undetermined after all known preconditions have been achieved, then the planner simply chooses values for all unbound variables in order to proceed. This is unavoidable when the planner is given (partially) uninstantiated goals. For example, the goal "put box1 somewhere" necessarily requires the planner to choose a "somewhere" – no amount of reasoning will deduce the location. In the simple algorithm in Figure 4, values are chosen at random; a more sophisticated planner could make this choice more sensibly.

5 Combinatorics, Abstraction and Extensibility

Our approach to composing plan operators offers three important advantages: it handles the combinatorics of building detailed representations, it allows abstract versions of a representation

[4] It can conclude this as `holding(Obj)` is in the `get` operator's add list in the CONTAINMENT component.

```
ACHIEVE-GOAL(Goal, ProtectedGoals, State) -> Plan
  IF   Goal already true in State
  THEN RETURN Plan = {}
  ELSE
        1. Find, in some component, an operator Op that achieves Goal
           (ie. has Goal in its ADD list in that component)
        2a. For each component C whose applicability rules succeed for Op:
              For each precondition PCi of Op within C:
                ACHIEVE-GOAL(PCi, ProtectedGoals, State) -> Plani
        2b. IF   some components still have unknown applicability (due to
                 having variables in their parameters)
            THEN randomly select a value for a variable and goto 2a.
        3.  IF   Op doesn't destroy any ProtectedGoals
                 (ie. none of its DEL list members are in ProtectedGoals)
            THEN - Simulate doing action Op:
                   MODIFY State using Op's ADD and DEL lists
                 - ADD Goal to ProtectedGoals
                 - RETURN Plan = concatenation of Plani + Op
```

Figure 4: The STRIPS planning algorithm, modified to interleave composition and planning.

to be generated automatically, and it permits domain representations to be easily extended. We review each of these in turn.

5.1 Combinatorics

Enumerating an operator set can be combinatorially explosive, and the compositional approach can avoid this expense. In Section 2.1 we considered a simple representation of operators for a robot world, where the robot could move objects and place them in containers (Figure 2). The representation consisted of six operators, but ignored many features of the domain. Now suppose we wished to model other aspects of the world (eg. whether doors were open or closed, the robot's strength), and to make the world richer by introducing a variety of objects (eg. heavy/light, large/small) and containers (eg. lidded ones, flexible ones). This expands the operator set substantially. Considering just the actions `get` and `put`, if we model different domain features captured in the eight components in Section 3.3, there are $2^6 = 64$ variants[5] of each action, depending on the objects involved, yielding 128 operators. Similarly, other actions in the domain will be manifest in many different forms. Clearly, it is unrealistic to manually enumerate this large set. The compositional approach offers an alternative allowing these domain features to be modeled without requiring such an enumeration.

5.2 Generating Abstract Representations

A second advantage of the compositional approach is the ability to generate abstract versions of a representation. Even if we could enumerate and debug a large operator set, or encode it in some alternative formalism allowing conditional preconditions and effects, it would still constitute a fixed representation of the domain based on particular assumptions about what to model and

[5]Of the eight components, CONTAINMENT is always applicable and SIZE and WATER-TIGHTNESS are mutually exclusive, leaving six degrees of freedom hence 2^6.

what to ignore. In contrast, a compositional approach provides a simple means for varying the assumptions underlying an operator set, simply by adding or removing components. For example, if we wished to ignore the volume of objects (say), we would remove the VOLUME component, thus removing its contributions to the operator set. The compositional approach enables us to easily generate abstractions of a representation, as it appropriately factors a representation into its independent features. Components provide the 'seams' by which a representation can be appropriately disassembled[6]. This ability to generate and work with more abstract domain models is often desirable, both for efficiency and explainability to a user [5, 10].

Work on *compositional modeling* achieves this advantage in a similar way (eg. [5, 8, 10]). In compositional modeling, a component (called a 'model fragment') contributes a set of constraints[7] to a representation of the domain, and composition involves collecting constraints from a suitably chosen set of model fragments. By controlling which model fragments are included in a representation, compositional modeling techniques allow the final representation's level of detail to be automatically tailored to one most suitable for solving a particular problem. Given that our compositional approach allows a representation's level of detail to be varied (by including/excluding components from consideration), it may be possible to use similar methods to automatically control this level of detail and tailor it to particular planning problems.

5.3 Extensibility

A third advantage of the compositional approach is the ability to easily extend a representation. Rather than manually edit a large operator set to incorporate a new domain feature, a representation can be extended by simply adding a new component to the library, and placing any new operators introduced appropriately in the isa-hierarchy of operators (Section 3.1). Because components express their effects on *classes* of operators (eg. in Section 3.1, the `Portal.lid.status=open` precondition is added to any operator involving moving through lidded portals), existing components will automatically contribute to new operators that are declared members of those classes. This considerably eases the task of developing representations in new domains.

6 Summary

We have presented a method for concisely representing a set of complex, domain-specific planning operators using components, rather than enumerating them manually. A component encapsulates a domain feature, not a plan operator, and may contribute to many operators. Detailed, domain-specific representations can then be constructed by composing a small number of such components, addressing the problem of adding detail without also substantially increasing the complexity of the representation. We have shown how this compositional approach can be used in AI planning algorithms. By interleaving composition and planning, operators to be constructed on demand. In addition to constructing representations, the compositional approach allows representations to be easily abstracted and extended, both important requirements for applying AI in real-world domains.

[6] Another early pioneering attempt at this was Abstrips [11]. However, Abstrips did not factor a representation into components, but instead abstracted representations by ignoring 'low priority' preconditions. It thus assumed fixed layers of detail, and had no notion of encapsulating domain features as independent units.

[7] sometimes called its 'relations' [8] or 'behavior conditions' [5]

References

[1] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct 1992.

[2] D. Batory, V. Singhal, and M. Sirkin. Implementing a domain model for data structures. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(3):375–402, Sept 1992.

[3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings ACM SIGSOFT'93 (Symposium on the Foundations of Software Engineering)*, Dec 1993.

[4] H. L. Dreyfus. From micro-worlds to knowledge representation: Ai at an impasse. In J. Haugeland, editor, *Mind Design*, pages 161–204. MIT Press, Cambridge, MA, 1981. (originally in "What Computers Can't Do" (NY: Harper and Row, 1979)).

[5] B. Falkenhainer and K. Forbus. Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

[6] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. In B. Webber and N. Nilsson, editors, *Readings in AI*, pages 231–249. Tioga, Palo Alto, CA, 1981.

[7] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem-proving to problem-solving. *AI*, 2:189–208, 1971.

[8] A. Y. Levy. Irrelevance reasoning in knowledge-based systems. Tech report STAN-CS-93-1482 (also KSL-93-58), Dept CS, Stanford Univ., CA, July 1993. (Chapter 6).

[9] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *AAAI-91*, volume 2, pages 634–639, CA, 1991. AAAI Press.

[10] J. W. Rickel. *Automated Modeling of Complex Systems to Answer Prediction Questions*. PhD thesis, Dept CS, Univ. Texas at Austin, 1995. (Also available as Tech Rept AI-95-234).

[11] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[12] A. Tate. Generating project networks. In *IJCAI-77*, pages 888–893, 1977.

[13] R. Waldinger. Achieving several goals simultaneously. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 250–271. Tioga, CA, 1981.