Building Action Descriptions from Components: Working Note 4

Peter Clark and Bruce Porter
Dept. CS, UT Austin
{pclark,porter}@cs.utexas.edu

1 Introduction

In the earlier Working Notes, we described the process of building concept descriptions from components. Components (unlike frame-slot-value triples or frames[1]) encapsulate a single statement about the world, and are the 'basic units' from which concept descriptions are built. By including or omitting components, we can selectively represent different aspects of concepts, as required by the particular problem-solving task at hand (eg. explanation, diagnosis). They also provide the 'units of retraction' for fixing buggy concept descriptions (Working Note 3).

This Working Note extends these ideas to the STRIPS-like representation of actions, ie. actions considered as discrete events with preconditions, add, and delete lists [Fikes et al., 1981]. As with concepts in general, we have to decide what to model and what to ignore when describing actions. Our goal here is to similarly encapsulate each modellable feature of the dynamics of the world as a different action component, and then build action rules as a composition of components. As with concepts in general, this allows us to selectively represent different aspects of actions as the current task demands, by including or excluding action components. For example in diagnosis of computing systems, a reported timeout error would suggest including timing aspects in the model of the computing system used for diagnosis, whereas a reported security error would not. The end result of this approach is that we can construct models of systems on the fly, specifically tailored to the particular problem at hand, rather than work uniformly with a single model which may include irrelevances in some places and be too simple in others.

Before continuing, we make a few prefacing notes: First, action representations allows us to construct 'executable' models of the world, ie. ones in which we can analyse and simulate behaviour. These types of models are fundamental to many kinds of problem-solving, including prediction, planning, diagnosis, and design, and hence are important to consider. Second, we are not claiming anything special about STRIPS-like representations. We are this formalism for simplicity (it seems to be largely adequate for our purposes), and acknowledge its many limitations. We hope the general compositional approach could be applied to other styles of representation also. Third, we note that representing actions takes us beyond standard first-order logic, as action rules make

[1] A frame-slot-value triple can be considered as the simplest type of component. However, in general components may involve relationships between several slot-values, as described in the ealier Working Notes. A frame can be considered as a opaque composition of several components.

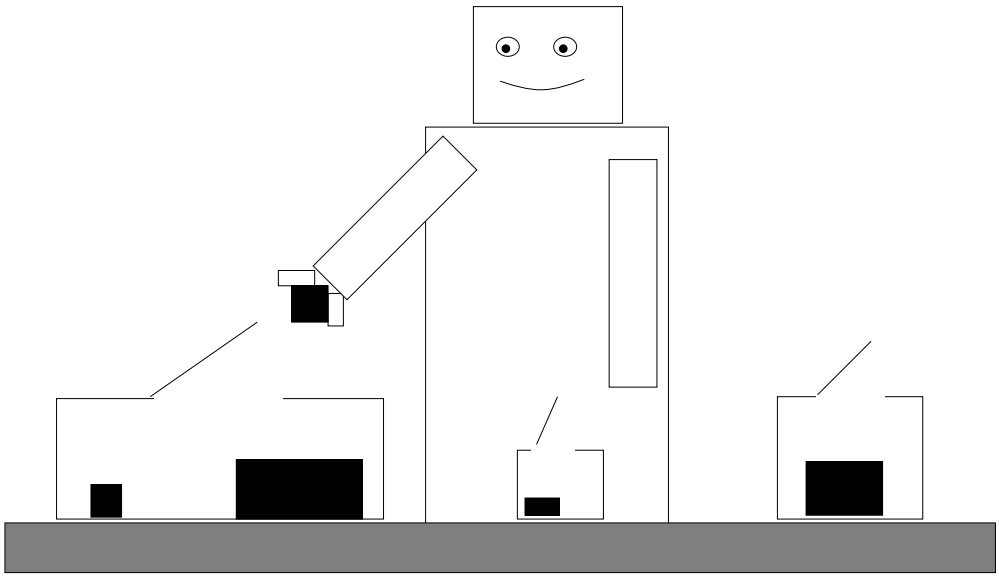


Figure 1: A simple robot world.

statements about statements (ie. statements about which statements become true/false if the action is carried out).

2 Decomposing Actions into Components

2.1 An Example of Alternative Representations

As a toy illustrative example, consider representing a world where a robot can get and put objects from boxes in front of it, as depicted in (Figure 1). One simple representation of the possible actions the robot can take might be:

```
              REPRESENTATION I
              ================

put(Obj,Box) ::
   pcs: holding(Obj)
   del: holding(Obj)
   add: in(Obj,Box)

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

This particular representation ignores certain aspects of the world, eg. whether the box's lid is open, whether the Obj will fit in the box, whether the robot is strong enough, etc. An alternative, where (say) we also care about whether the box's lid is open or not, might be:

```
              REPRESENTATION II
              =================

put(Obj,Box) ::
```

```
   pcs: holding(Obj), opening(Box,Portal), lid(Portal,Lid), open(Lid)
   del: holding(Obj)
   add: in(Obj,Box)

get(Obj,Box) ::
   pcs: in(Obj,Box), opening(Box,Portal), lid(Portal,Lid), open(Lid)
   del: in(Obj,Box)
   add: holding(Obj)

open(Lid) ::
   pcs: not open(Lid)
   del:
   add: open(Lid)

close(Lid) ::
   pcs: open(Lid)
   del: open(Lid)
   add:
```

A second alternative, where we care about whether the object will fit in the box might be that shown below:

```
            REPRESENTATION III
            ==================

put(Obj,Box) ::
   pcs: holding(Obj), size(Obj,Size), free_space(Box,FSpace), FSpace >= Size
   del: holding(Obj), free_space(Box,FSpace)
   add: in(Obj,Box), free_space(Box,FSpace-Size)

get(Obj,Box) ::
   pcs: in(Obj,Box), size(Obj,Size), free_space(Box,FSpace)
   del: in(Obj,Box), free_space(Box,FSpace)
   add: holding(Obj), free_space(Box,FSpace+Size)
```

In this second variant, the predicate `free_space(Box,Space)` has been introduced to denote the current free space in a box, and the action rules used to update this 'counter' appropriately.

2.2 Action Components

2.2.1 Content

As always, we follow the s/w reuse maxim of "identify the part which varies and encapsulate it" to build our action components. Our goal is that each of these two transformations (from representation I to II, and from representation I to III) should each correspond to the composition of a single component into the representation.

To summarize the syntactic structure of these two transformations: In the first example (checking the lid is open), we have added the preconditions `opening(Box,Portal)`, `lid(Portal,Lid)`, and `open(Lid)` to the `put` and `get` actions, and also added two new actions `open` and `close`. In the second example (checking for sufficient space), we have modified the `put` and `get` actions to include additional members of the preconditions, add and delete lists. Our components for `lid` and `space` should correspondingly encapsulate these changes.

2.2.2 An Action Component

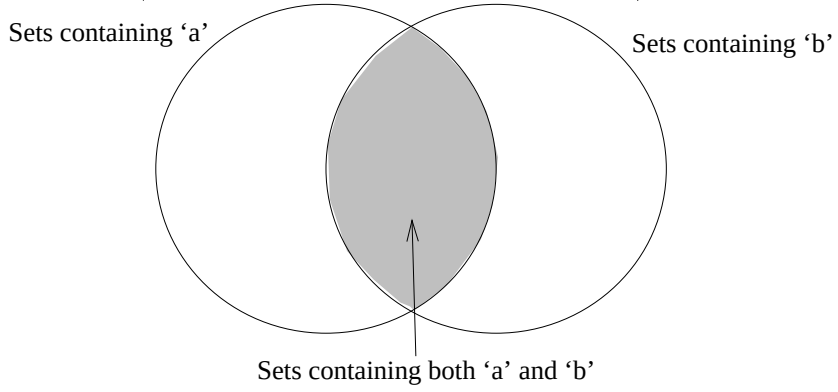
The simplest way of representing the open-lid component is to just write down the differences between representations I and II:

```
            OPEN-PORTAL COMPONENT
            =====================

put(Obj,Box) ::
   pcs: opening(Box,Portal), lid(Portal,Lid), open(Lid)
   del:
   add:

get(Obj,Box) ::
   pcs: opening(Box,Portal), lid(Portal,Lid), open(Lid)
   del:
   add:

open(Lid) ::
   pcs: not open(Lid)
   del:
   add: open(Lid)

close(Lid) ::
   pcs: open(Lid)
   del: open(Lid)
   add:
```

The machinery for composition – adding a component into a representation – is then simply to take the union of the pcs/del/add lists for actions which correspond, and add any additional actions introduced by the component into the representation. In fact, this is still a unification process (consistent with the previous Working Notes), if we consider the objects being unified as partially specified sets:

> if {a} denotes a set which contains at least a
> and {b} denotes a set which contains at least b
> then the unification of {a} and {b} is the set {a,b} (ie. a set which contains at least a and b)

This is consistent with the standard definition of unification as the greatest lower-bound of two sets of objects (in this case, each 'object' is itself a set):



The important point here is that we are not introducing a new composition mechanism for merging action components, but instead still using the standard unification mechanism advocated earlier.

Note that a component typically does more than simply add one, extra precondition of an action rule[2]. Also note that the component may affect the 'add' and 'delete' lists, as well as the preconditions list, and also introduce extra actions.

2.2.3 Adding Hierarchical Structure

In fact, this formulation of the `open-portal` component isn't ideal. In particular, we've duplicated the extra preconditions list {`opening(Box,Portal)`, `lid(Portal,Lid)` , `open(Lid)`} for both `put` and `get`, whereas really they constitute a single aspect of a more general notion – moving through a portal. It would be better to make this more general concept explicit, and have these preconditions 'passed down' to specific actions which involve moving through a portal.

We can do this by introducing an action-like `move_thru_portal` concept, and then use that to describe the open-portal component:

```
            OPEN-PORTAL COMPONENT
            =====================

move_thru_portal(Obj,Portal) ::
   pcs: lid(Portal,Lid), open(Lid)
   del:
   add:

open(Lid) ::
   pcs: not open(Lid)
   del:
   add: open(Lid)

close(Lid) ::
   pcs: open(Lid)
   del: open(Lid)
   add:
```

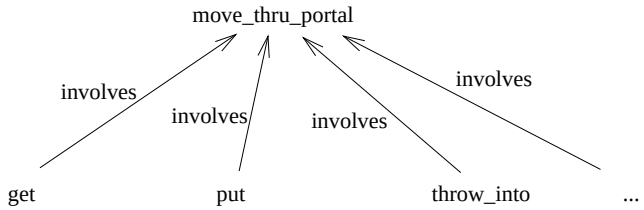
We additionally have to no specify the relationship between `put` and `move_thru_portal`:

```
put(Obj,Box) involves move_thru_portal(Obj,Portal) where opening(Box,Portal)
get(Obj,Box) involves move_thru_portal(Obj,Portal) where opening(Box,Portal)
```

This is considerably better than the open-portal component in Section 2.2.2, as we have made the concept we are actually describing (moving through a portal) explicit. In addition, if other actions are introduced which involve moving through a portal (eg. `drop_into`, `throw_into`) they will automatically acquire the `open(Lid)` constraint from this more general component. Finally, we can describe other components of the move-through-portal concept in a similar way (eg. whether the portal is big enough). As for the open-portal component, their content will automatically propogate down to all the action rules involving moving through portals through the `involves` hierarchy. In software engineering terms, this component has an 'elastic interface' as we don't explicitly enumerate all the actions ('operations') which it modifies.

[2]Though a single, extra precondition is the simplest type of action component, analagously to a frame-slot-value triple being the simplest type of component for concept descriptions.

Above we introduced an `involves` hierarchy to specify the hierarchical relationship between action concepts:



The `involves` hierarchy is almost, but not quite, an `isa` hierarchy: if `A involves B`, then `A` inherits the preconditions, add and delete lists of `B` but *not* other properties. This restriction is important, as other inheritable properties may conflict (with multiple inheritance). `buy`, for example, involves both a `give` of a purchase (in which the `donor` is the `seller` and the `recipient` is the `buyer`) and a `give` of money in which the `donor` is the `buyer` and the `recipient` is the `seller`). We clearly don't wish `donor` to be inherited from both the (specialized) `give` superconcepts to `buy` simultaneously as this leads to a conflict.

The hierarchical organization of action concepts (with action rules as the leaf nodes), and the fact action components can contribute to any node in the hierarchy, is important – it means that a single action component can affect a number of action rules simultaneously. Similarly, a particular action rule can 'inherit' information from a number of action components through the `involves` hierarchy.

3 The Power of Compositionality for Complex Worlds

In the previous Section, we stated that getting something from a container involved making sure that the container's lid was open. However, this really only makes sense if the container has a lid in the first place. Similarly, there are other components which we may or may not with to include depending on the properties of the objects involved. For example, if the object is heavy we should check the container is strong enough, or if the object is a liquid then we should check the container is water-tight, or if the object is heavy *and* a liquid then we should check the container is both strong enough and water-tight.

There would be a combinatorial explosion if we tried to manually write down all such variants of action rules. However, by describing actions as compositions of components, and rules about when a component applies, we can build appropriate action rules on the fly when reasoning about particular activities.

As an example of this, we just consider a `put` action and just consider preconditions for the action rule. Some possible components are:

```
UNLOCKED-CONTAINER COMPONENT:
        put_obj_in_lockable_container(Obj,Container) ::
                pcs: not locked(Container).

STRONG-ENOUGH-CONTAINER COMPONENT:
        put_heavy_obj_in_container(Obj,Container) ::
                pcs: strong(Container).

WATER-TIGHT-CONTAINER COMPONENT:
        put_liquid_in_container(Obj,Container) ::
                pcs: water_tight(Container).
```

```
BIG-ENOUGH-PORTAL COMPONENT:
        put_rigid_obj_in_container(Obj,Container) ::
                pcs: portal(Container,P), width(P,W), width(Obj,W2), W2 < W

BIG-ENOUGH-CONTAINER COMPONENT:
        put(Obj,Container) ::
                pcs: capacity(Container,C), size(Obj,S), S < C
```

We similarly have to add rules about which types of `put` action are members of these other action components:

```
put(O,C) involves put_obj_in_lockable_container(O,C) if lockable(C).
put(O,C) involves put_heavy_obj_in_container(O,C) if heavy(O).
put(O,C) involves put_liquid_in_container(O,C) if liquid(O).
put(O,C) involves put_rigid_obj_in_container(O,C) if not liquid(O).
```

By asking about particular `put` actions, we can construct the appropriate action rule on the fly, then check its preconditions.

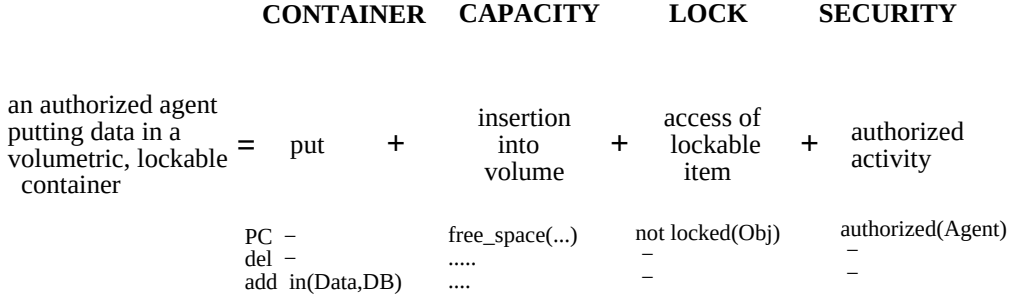
```
Put what into what? put(book,room).

  Testing preconditions:
     not locked(room)
     portal(room,door)
     width(door,100)
     width(book,20)
     20<100
     capacity(room,1000)
     size(book,0.5)
     0.5<1000
  Okay!
----------------------------------------
Put what into what? put(gallon_of_water,paper_bag).

  Testing preconditions:
     strong(paper_bag)                  <=== FAILS!
     water_tight(paper_bag)             <=== FAILS!
     capacity(paper_bag,0.5)
     size(gallon_of_water,1)
     1<0.5                              <=== FAILS!
  Can't do that!
----------------------------------------
Put what into what? put(piano,rucksack).

  Testing preconditions:
     strong(rucksack)
     portal(rucksack,rucksack_top)
     width(rucksack_top,30)
     width(piano,99)
     99<30                              <=== FAILS!
     capacity(rucksack,2)
     size(piano,99)
     99<2*1.5                           <=== FAILS!
  Can't do that!
----------------------------------------
```

The important point to note here is that the preconditions of each specific action rule differ, as different components have been selected based on the properties of the object and container involved. A nice explanation facility to explain the failures would be straightforward to add.

Similar compositions are required for our DCE knowledge base. For example, we may wish to describe an agent adding data to a database, and model aspects of authorization, data protection and database capacity. Composing appropriate components together may produce a rule such as:

```
put(Agent,Data,DB) ::                    % SOURCE:
    PC  authorized(Agent, DB)            % authorization
        not locked(DB)                   % lockability
        free_space(DB, FreeSpace)        % volume
        size(Data, Size)                 % volume
        Size =< FreeSpace                % volume
    DEL free_space(DB, FreeSpace)        % volume
    ADD free_space(DB, FreeSpace-Size)   % volume
        in(Data,DB)                      % container
```

constructed from components as illustrated below.



References

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