MSc in Information Technology: Knowledge Based Systems

Intelligent Assembly Systems

A Practical Example - The Soma Cube Assembly Problem

The text of these notes is an edited extract of sections of *Planning and Performing the Robotic Assembly of Soma Cube Constructions*, MSc Dissertation, Chris Malcolm, Edinburgh University, September 1987. For further reading (not required) you may consult this.

The problem

Construct a planner which will plan reliable soma cube assemblies. The planner will be told the shape and position of the parts, and the shape of the intended assembly. It will have a priori knowledge of the capabilities of the robot, gravity, etc.. The plan should execute reliably in the real world.

What to do first?

It will be a behaviour-based system. Thus reliable behaviours must be contrived in the on-line system, in terms of which the plan will be constructed. Therefore the behaviours must be constructed before the planner, since what behaviours turn out to be possible will influence the planner.

But behaviours are always designed within a context of a set of rules which define the subset of the world which the behaviours must tackle. A good choice of rules simplifies the job of the behaviours. So before tackling the behaviours the rules should be set out. So the first thing to do is to decide on the rules. Of course the rules will be liable to revision as the implementation of the behaviours and the planner proceeds. Here are the rules that were used in their final form.

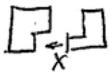
The rules

Picking up the pristines

These are presented in distinct and separate positions without possibility of interference from neighboring acquisition procedures, and with an isolated **cubie** standing up, so that the double grip (grasp, release, rotate z axis 90 degrees, grasp) may be used for best accuracy of acquisition. Only the Zed part can't be presented with a **cubie** up. The acquisition grasp will always be downwards.

Putting the part in the assemblage

Parts will be put into the **assemblage** with a downwards motion of a downward gripped part, except for the last part, which must be pushed in sideways. A sideways push could have been used for other insertions as well, thus increasing the choice of the planner at the expense of complexity - but experiment showed that restriction of this motion to the last part still left plenty of assemblable solutions. Note the important point that while a part might be able to be slid in sideways under another part, this is in general liable to failure due to form and other uncertainties, hence the insistence upon downwards part mating.



Grasping in general

This is always done across a single **cubie** - a limitation of the gripper. To avoid gripper interference problems it is also a rule that the gripped **cubie** must be (one of) the nearest to the gripper along the axis of approach.

Top
Interference if
pick up here
Use
Bot

Finger clearance

It is assumed that a finger requires no more than one **cubit** clearance in order to release a gripped **cubie.**

Gripper clearance

In order to avoid gripper clearance problems requiring detailed reasoning about the gripper size in **cubits** (and the **cubit** is liable to change), gripper clearance is assured by the rule that during part mating no gripped **cubie** will be below the level of another **cubie** of the **assemblage**.

Uncertainty

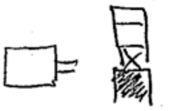
The part acquisition behaviours will always cope with at least half a **cubit** of location uncertainty, within which envelope the rotation uncertainty of the pristine must be less than 45 degrees.

Gross motion without collision

It is assumed that there exists a **cubit**-related safe height within the scope of the robot in which all part translocations can be performed without fear of collision.

Regrasping

Most parts cannot be acquired with a part mating grip, and so must be regrasped. A small table is provide for this regrasping, since the Adept robot suffers from knuckle clearance problems close to the table. This small table is intended to permit grasping of the bottom **cubies** of a part, but will this be possible under all circumstances? Having discovered by experiment that this does not impose a serious limitation on the number of possible assemblies, the rule was made that bottom **cubies** will *not* be grasped.



don't pick bottom cube

The general principles of behaviour design

I learned some of the principles I now know by considering the results of this experiment. Those are given in the Conclusions. The principles I used in the implementation are explained in this section.

There are two distinct aspects to programming behaviours: the action in the real world; and the programming in the robot control language (in this case VAL2). As far as the programming is concerned the principles are the same as for good structured programming, the most important of which is to ensure that each parameter is assigned a value in only one place in the program, so that to change it (and everything that depends on it) requires a change in only one place. Examples of such parameters are the **cubit** the depth of grip of the gripper on a **cubie**, the spring-out of the fingers on opening, the amount of vertical error to allow in part put-downs, and so on. Thus changing the **cubit** should never entail changing more than one number in the on-line program.

A second principle, partly a corollary of the above, is that the minimum number of positions should require to be taught - or alternatively passed from the planner. Where positions are interdependent, this interdependence should be expressed in the on-line coding. Where interdependence is mutual or cyclic this may involve the need for a matrix (spatial transformation) inversion function in the programming language. There may be even more complex examples inexpressible within languages like VAL2, requiring the power of a spatial inference engine such as forms the basis of the RAPT system [Popplestone et al 1978]. Generally speaking one should import into the on-line system the kind of representations and computations it can handle, no more; and on the other hand, good justification should be found for importing less.

These are the programmable side. The action side is the general requirement that the output uncertainty of a behaviour should be significantly less than the input uncertainty with which it can cope. This is a signpost rather than a literal rule. A literal derivation which must be adhered to is that the input uncertainty of a behaviour must exceed the output uncertainty of the previous behaviour.

Of course it would have been most unwise to rely entirely upon my own ideas of what a behaviour ought to be, so the family development approach was adopted, i.e., test them using a family of related assemblies, with the requirement that they should be general enough to handle the members of the family without change. In this case the family comprised the several versions of the soma cube available.

I decided that I needed four elementary behaviours: part acquisition; part regrasping; part put-down; and cube pat-into-shape. It turned out that I needed two different part acquisitions: pick, which just grabbed the part; and get, which grabbed it twice at right angles. This was because one of the parts could not be presented with a single cubie upwards. Because these two could only handle limited uncertainty I invented another behaviour, patting-into-place, which prefaced the getting of each part.

It turned out that I did not need to take any precautions to adjust out errors in the regrasping behaviour: the regrasping always worked, though there was one extreme case which was close to the limits of possibility. And the scheme of spacing out the assembly turned out to be able to cope with all the errors that were introduced during the regrasping.

The generality of these behaviours was developed by making them work on three different versions of the soma

Extreme case:

regrasping fails.

would be 10% more certain if patting-into-place executed.

cube, using two different plans that I invented myself.

This was the bottom up part of the development finished. The job of the planner was now defined: it had to invent plans just like the one I ended up with when I had got all the behaviours working in a hand-coded assembly. A shortened form of a typical generated plan is reproduced below.

The plan

```
PROGRAM plan()
   CALL sinit()
;
   CALL zjustz(b1.get, 2)
   CALL zjustz(b4.get, 2)
   CALL zjustz(b7.get, 2)
   .....
;
;
   CALL zpcalc(centre, b1.put, -1,0,2,0*gap, 2*gap, drop)
   CALL zpcalc(centre, b4.put, 0,0,2,1*gap, 1*gap, drop)
   CALL zpcalc(centre, b7.put, 1,1,2,2*gap, 2*gap, drop)
   .....
;
;
; ----------- The placing of lell -----------
   CALL zpatget(b1.get, RZ(90), -0.5,1.5, RZ(0), -0.5,0.5)
   CALL zget(b1.get:RZ(0))
; - No regrasping required.
   CALL zput(b1.put:RZ(-270))
;
;
; ----------- The placing of fork3 -----------
   CALL zpatget(b4.get, RZ(90), -1.5,0.5, RZ(0), -1.5,0.5)
   CALL zget(b4.get:RZ(-90))
; - No regrasping required.
   CALL zput(b4.put:RZ(-90):RZ(-90))
;
;
; ----------- The placing of right -----------
   CALL zpatget(b7.get, RZ(90), -1.5,0.5, RZ(0), -0.5,1.5)
   CALL zget(b7.get:RZ(0))
; - Straight case.
   CALL zmanip(table, RZ(-90):RY(-90),0,-1,2,RZ(-90):RZ(0),0,0,2)
   CALL zput(b7.put:RZ(-90):RZ(0))
;
;
; ----------- The placing of zed -----------
   CALL zpatget(b6.get, RZ(90), -0.5,1.5, RZ(0), -0.5,0.5)
   CALL zpick(b6.get:RZ(0))
; - Reversed case.
   CALL zmanip(table, RZ(-180),0,0,2,RZ(0):RY(90),1,0,2)
   CALL zput(b6.put:RZ(0))
;
; ................
;
   CALL zvpatcube(centre)
   RELAX
   MOVES park
END
```

Knowledge representation

PROLOG was the language the planner was going to be written in. The first question to be addressed was the question of knowledge representation. This is always a very important question in any AI program.

The knowledge to be represented is the shape of the parts, of the **assemblage**, and of the empty space. The kind of questions that need to be answered from this knowledge are: does this part fit into this empty space; is the **cubicle** a technical term meaning a **cubie** space in the soma cube world. to the left (up, under, etc) of this one occupied?

The two candidate types of representation which occurred to me were: a direct representation of space occupancy, such as a 3D array or list of triples, from which could be deduced the relationships of **cubicles** and faces to one another; or a constructive representation of the parts and spaces, in terms of a tree of *up, left, under,* etc. types of relationships, from which could be deduced space occupancy.

Let us examine these alternatives in the light of the problem of fitting a part into the empty space remaining at some point in the assembly. It is most important to be able to answer this question quickly. There are seven parts, and there are about 100 different ways of fitting each part into the empty soma cube. If this seems surprising, note that there are 24 possible rectangular rotations (6 faces of the cube, and 4 rotations of each face), times the translations of each orientation within the cube shape. The figure of 100 is in fact a rough average of the results obtained after throwing out symmetries of the parts. There are about 100 billion (100^7) possibilities here. If one were examined every microsecond, it would take three years to examine them all. Under these terrifying circumstances the crucial question is which representation will permit this fitting operation to be done as rapidly as possible.

Consider the spatial relationship representation. The overwhelming requirement of speed for the part fitting loop means that nothing should be computed within that loop which could be precomputed beforehand. In other words, relationships of adjacency should be fully enumerated. While the soma4 parts are simple enough to be represented as trees even in a fully redundant manner, this is not true of the shape of the **assemblage**, and the remaining space. This means that the fitting question is one of fitting a tree into a net, which doesn't sound very tractable. It is also hard (i.e. I couldn't do it) to devise a unique normal form of tree expression so that, for example, rotational symmetries can be identified by equivalence of expression.

Consider a spatial occupancy representation. Suppose each **cubicle** is represented as a triple [x,y,z], being its coordinates. A part, or the hole left, or the **assemblage**, can be represented as lists of triples. The question of fitting becomes a question of whether each of the **cubies** of the part is also a member of the hole list. This can be done in a single pass if the lists are kept sorted into order. This seems nicely tractable so far. Translations of the part are easily handled just by vector addition throughout the list. It turns out that rotations are also easily accomplished by combinations of axis swopping and negation. It would be wasteful to repeat these translations and rotations every time the part was considered for a fit, so these computations could be removed from the fitting loop by doing them all at the start, and keeping a large list of all possible translations and rotations at the start. Thus the fitting operation reduces to selecting one member from each list of part rotations and translations which fits into the remaining hole, backtracking until they all fit.

That sounds like a reasonably simple and rapid computation, except of course for the size of the solution space. Since the translations and rotations are pre-computed once and for all, there is little extra overhead in being clever about the possibilities, so as to reduce the number. For example, many of the parts have rotational symmetries. These can be checked for and thrown out of the list. All possible translations can be taken to be all possible translations which fit inside the assembly shape - other shapes are not as simple as the cube.

When spatial relationships are required, their computation is relatively simple. For example, the **cubicle** xwards of [x,y,z] is [x+1,y,z].

I have discussed this spatial representation in terms of lists of triples, rather than a 3 axis array, since that is the simplest and most efficient representation in PROLOG.

The planner

The program consists of a number of sequentially run sections. The first section, the parts expansion, generates an enormous record of all possibilities. The general solution finder boils this down to a small record representing a general solution. Subsequent sections then proceed to decorate this record with extra information until all the information necessary to instruct the robot how to enact the assembly has been discovered. Finally, this is translated into a sequence of parameterised behaviours, a sequence of VAL2 **call** instructions.

Finding a general solution

The goal here is simply to find a way in which the parts fit together inside the shape of the assembly, without bothering about the details of how they might get into those positions. This stage corresponds to the design stage of an exploded assembly diagram in manufacturing. The strategy chosen is based on the simple idea of putting parts into the assembly one after another until they all fit in, or until one fails to fit in. If it fails in one position, then it is tried in another. If it fails in all positions, then it is tried in another orientation. If all its orientations fail, then another position of the previous part is tried. Once all those have failed, another orientation of the previous part is tried. And so on. This kind of backtracking through the tree of possibilities is supplied for free with PROLOG - one of the reasons for choosing the language.

That method is guaranteed to find a solution if one exists, but it is very inefficient. The first inefficiency is because it recalculates the different rotations and translations of a part within the assembly again and again, every time it needs them. So the first optimisation was to precalculate all the possible rotations and translations of a part which fitted inside the assembly. Thus getting the next translation or rotation was simplified to getting the next item from a list. The second optimisation was to notice that most of the parts had symmetries, i.e., some of the rotations were equivalent to others. These equivalences were thrown away.

This third optimisation came from noticing how stupid this solution finder was compared to me: for example, it would go on trying to fit the last three parts in all possible combinations, even though it had already, with the first four parts, walled off one single cubicle, so that a solution was clearly impossible. Some way of recognising this kind of situation would help to prune the solution space a lot. A method was devised of analysing the remaining space in the assemblage into separate walled-off sets. Since six of the parts contained four cubies, and only one contained three cubies, then putting the three cubie part in first meant that each one of these walled-off sets had to contain a number of cubicles exactly divisible by four, otherwise a solution was impossible. This was checked after each part was put into the assemblage.

The fourth optimisation came from noticing that, after rejecting the symmetries, some parts had a lot more possibilities than others. It helped to minimise the amount of deep backtracking required to try out the ones with least alternatives, and thus the greatest likelihood of being correct, first.

These four optimisations speeded up the solution search a great deal. The final optimisation did not speed up the solution search, but it made it more likely that the solution found would be assemblable. It so happens that the most important reason why a solution is not assemblable is because of finger interference, i.e., the robot can't put the part down because other cubies get in the way of its fingers. This is less likely to happen if the part has a single cubie sticking up in the air. So after all the rotations had been generated, and the symmetries thrown out, they were sorted into an order with sticking-up-single-cubies first. This meant that the first solutions found (there happen to be 240 x 24 solutions) are more likely to be assemblable than the last ones.

Although the actual symmetries are thrown out, the sets of different rotations which were found to be equivalent are noted, because this proves useful later: if a part needs to be rotated in a certain way, which proves impossible, then one of the equivalent rotations may be possible.

2 Finding a gravitationally stable order

Having found out how the parts are disposed in the final assembly, the first constraint is to select a gravitationally stable ordering, so that every stage of the **assemblage** is stable under gravity. In general manufacturing assembly the sequence of assembly would be decided, at least in general terms, during the design process. This general sequencing involves knowledge of what kind of things can be expected to be done without difficulty in the assembly stage. So this stage can be seen as the validation or rejection of suggested assembly sequences. In general terms making the sequence of assembly gravitationally stable implies that the necessary base to support a part stably (uncertainty near boundary cases makes this sometimes more than one **cubie**) must be a subset of the **cubies** of the part which are actually supported. But there are practical considerations which simplify this.

It is a rule that all parts will be introduced into the **assemblage** from above. The implication of this is that when each part is sitting down in its final place, there must be no empty space underneath it - else how would that space be filled?

So the general stability requirement simplifies to the requirement that every **cubie** which *could be supported, must* be supported; and a **cubie** which could be supported is simply a **cubie** which does not have another **cubie** of the part beneath it.

It turns out that there are often several dozen of stable orderings of a solution. It takes a few seconds to compute a stable ordering. No steps were taken to optimise this in a look-ahead fashion, but, as explained in the next section, it proved to be most advisable to implement failure-directed backtracking into this stable ordering from the put down grasp planner.

I sometimes refer to this stage of assembly planning as *ghostly assembly,* since it is a manipulatorless assembly plan, suitable for psychokinesis. The general solution can be regarded as an assembly plan which requires direct materialisation of the parts in the final positions.

The put grasp planner

Implement boredom to give up on a plan if not found solution after 10 mins. Don't loose very much

Given a stable ordering of the parts, it is not always possible to devise a grip with which they can be put in place. The main problem is allowing for finger clearance, i.e., there must be empty **cubicles** on either side of the gripped **cubie** to accommodate the fingers. This is a sufficiently serious constraint that it is possible for only a minority of solutions.

The assemblable solutions behave rather like number factoring problems, where impossible assemblies are represented by prime numbers. Most possible assemblies fall out very quickly and simply, and could be done in a variety of ways; whereas impossible assemblies require a great deal of thinking to be quite sure that they are impossible. And there are always some few solutions which are nearly impossible. This suggests that only a few solutions would be missed by implementing boredom - skipping to the next solution as soon as the search became tedious.

There are two minor extra problems to take into account. The first is gripper clearance, as opposed to finger clearance. This demands that nearby cubies do not stand above the level of the gripped **cubie.** Of course this is a question of the relative sizes of gripper and **cubie**, but it proves to be a very minor restriction, so the simple and general rule is adopted of no **cubie** anywhere in the **assemblage** standing proud of the gripped **cubie.**

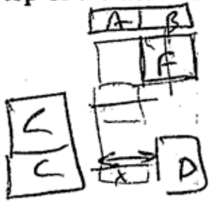
The second problem is that it will sometimes be impossible to find finger clearance for the last part - always in the case of the cube. This is most easily seen if you consider the problem of the finished cube: where is the **cubie** which could be gripped so as to remove a part? So the last part is permitted an alternative method of assembly - it can be put down in an offset position so that the offset allows the necessary finger clearance, and then, once the fingers have been withdrawn, the part can be pushed sideways into place. This means that the last part has an extra freedom as far as gripping is concerned: it need only have space for one finger. This involves an extra constraint as far as motion is concerned: not only must it be possible for it to move into place vertically downwards, but it must also be capable of moving into place sideways along the finger axis.

Now if this offset put down and nudging into place is allowed for the last part, then why not make use of it for other difficult situations as well? Since this would add complications to the planning process beyond those needed just for the last part, the answer to this question depends on how many assemblable solutions can be found without making this extension. Since the answer is about half of the total general solutions, I did not implement this extension. The last part is the only one for which this offset put down is permitted.

Failure directed backtracking from put grasp failure

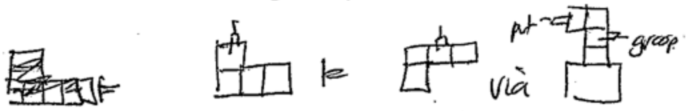
It must always be the case that if finger clearance does not permit a part to be gripped, then it must be an earlier part, already in the **assemblage**, which is in the way. Note that if two parts forbid finger clearance on either side, then the important culprit is the older one (the one which was placed in the **assemblage** first). Backtracking to find the next possible stable ordering will not succeed in avoiding this obstacle until it has succeeded in re-ordering the culprit. Given the large number of stable orderings, i.e., the bushiness of the tree, a great deal of fruitless re-ordering and grasp planning can be avoided if the stable ordering clause can be directed to backtrack back to the point necessary to re-order the culprit. The culprit is the youngest of all the parts in the way of the fingers.

The other reason for failure of a put grasp is gripper clearance, i.e., this part requires to be gripped at a level below the top of the **assemblage.**



can't grip at x so clear c the 'older'
allowable grips are —, must remove youngest of --
these to backtrack.

In cases where gripper and finger interference co-exist, the culprit (i.e. the backtracking level) is taken to be the older, because either obstacle damns the assembly.



4 Regrasp planning

generate + test

In some carefree cases it so happens that a part can be picked up with the required grip to put it down. This happens in those cases where the part only rotates between initial and final configuration about the vertical axis, or where symmetry permits an equivalent vertical rotation. In all other cases the part must be regrasped. A little table is provided for this purpose, to make regrasping from all angles as simple as possible.

The grasp planner can provide alternative grasps. If finds the grasps nearest the centre of gravity first. The regrasp problem is to find a gravitationally stable orientation of the part on the regrasp table in which one of the get grasps and one of the put grasps are both possible - suitably rotated. This was solved by **generation and test.** First would be discovered a rotation which the robot was capable of making from the put down grasp attitude. This would be failed if the rotated part was not gravitationally stable. Once a stable and reachable attitude had been discovered a get grip would then be selected. All possible rotations of the adept from this get grip would be generated and checked for conformity with the orientation of the part on the regrasp table.

If this process failed to find a solution, for any of the rotations possible from any of the put grasps, then the planner would repeat the process through the list of axial rotation equivalences discovered in the initial symmetry discarding phase.

next symmetry (different rotations)

This process covered a lot of ground, and could take several seconds in a bad case. A most interesting problem arose if a part turned out to be impossible to regrasp. Was this part impossible to regrasp because local finger interference prevented a successful grasp, or was it that this orientation of the part was impossible to regrasp under any circumstances?

This planning of regrasping, with its associated failure directed backtracking, not only had the largest and most logically complex clauses of any part of the problem, but also it was extremely difficult to understand. I had to write a simplified regrasp planner first, and get it working, before I was capable of understanding the general case. This kind of experimental development is typical of AI programming.

Failure directed backtracking from the regrasp planner

In order to discover whether a regrasp failure was general or not, the analysis would be repeated, but omitting from consideration the rest of the **assemblage**. If it proved to be possible under this circumstance, then the regrasp failure was due to finger clearance problems, and treated as such.

If on the other hand it was impossible even without the rest of the **assemblage**, then this part was unregraspable in this orientation, and backtracking had to be forced back into the general solution finder to pull out a new orientation of this particular part.

These various forms of failure directed backtracking, from finger and gripper failure into the stable orderings, and from regrasp back into stable orderings or rotation selection, greatly improved the fruitfulness of the search for assemblable solutions, by leaping over dead areas in the solution spaces. The cases where the planner still seemed to behave stupidly were where it got locked into cycles involving three parts. The planner would backtrack to one of them, whereupon all three would change places, thus fooling it that a new situation had arisen, wheres in reality it was running round in a circle until it had enumerated its way out of all possible combinations of ordering of the parts younger than the threesome.

I didn't bother to attempt detection of threesome cycling. The computationally much simpler implementation of boredom would also have handled this problem.

Spacing out the assembly to contain the uncertainties

The rule is that there must exist, between every pair of horizontally mating faces (i.e., coming together along a horizontal axis), a small space. This is known as a pad. A pad is a behaviourally-determined unit (i.e., the size is not the concern of the planner), in practice about 1/8th of a **cubit**. The purpose is the isolated containment of the errors of positioning of the parts, as they are brought into the **assemblage**. One could suppose each part to repel its neighbours to a constant distance. The problem is that in some cases the *space must be larger, consisting of more* than one pad, due to other chains of faces pushing this one further apart than its own repulsion dictated.

The information required by the planner is not the location and number of these pads, but the offsets, in terms of unit pads in the x and y directions, of the parts from their nominal positions in the **assemblage**.

Since there there is no interference between the x and y axial pads, each axis can be considered independently of the other.

A horizontal adjacency between the faces of two parts requires that one part must be offset from the other; the offset of one must be at least one pad unit *greater* than the other, or, the other must be at least one pad *less* than the one. Since each relationship implies the other, we may choose to consider only one of these relationships (either greater or lesser), without loss of information. Thus the problem can be simplified to considering one relationship, in one dimension at a time. Trying to consider both "<" and ">" relationships makes the problem tediously and unnecessarily complex - it turns it into a net, whereas with only one sense of relationship it is a tree.

The method chosen is to discover all of the (say) xward adjacencies of a part, and to pursue all the similar adjacencies of the adjacent part(s), until no more are discovered. This is a tree structure. No more are discovered when a part is arrived at which has no xward adjacencies, and consequently has no need to be offset in a -x direction. This part is assigned a zero offset along the axis under consideration. The number of nodes in the tree which must be traversed before arriving at such a zero offset leaf will vary. The largest number of nodes traversed before finding a zero leaf is the offset required for the initial part required in order to accommodate all of its relationships.

Translating the final solution record into behaviour calls

This should have been largely a question of a transformation of representation: from a PROLOG list structure to a sequence of VAL2 parameterised behaviour calls. In general fairly straightforward, the foreseen complication was the need to translate between the chosen rotation representation in the planner - lists of absolute axial rotations - and that required by VAL2 - lists of relative rotations, with a couple of strange bugs in the implementation. Spatial reasoning masochists can find the details of these in appendix 2.

The unforeseen complication was that the output of the planner was not complete. This was due to a confusion between deciding the feasibility of the plan, and providing all the details necessary for its enacting. All the rotations necessary at every stage of the part's journey into the final assembly had been elaborated, but I had forgotten that the rotations of the robot were not programmed explicitly, but deduced by its controller from the difference between the source and destination position sextuples.

While the difference defined the translocation of the part, the sextuples themselves defined the orientation of the grips, since each motion took place between two grips. This was not a problem when one of the grips was unambiguous, but downward grips were ambiguous - the planner had not bothered to distinguish between 180 degree rotations of the gripper - and so motions terminated at both ends by downward grips *were* ambiguous. Unfortunately, although left ambiguous in the plan, only one of these alternatives was actually possible.

So this final stage of planning - dividing the A to B rotation spec into a location A rotation and a location B rotation, incorporating the gripper orientations - was performed at the VAL translation stage. Of course it should have been put into the grasp and regrasp planners, but since PROLOG lacks general record specifications, that would have meant a great deal of error-prone record format editing.

Conclusions

There are three main conclusions to be drawn:

* Behaviours have so far fulfilled their promise.
* There is great benefit in implementing a complete planner: despite my best intentions to the contrary, every stage of development (with the notable exception of the last) discovered omissions, sometimes serious, in the work of previous stages; thus a complete planner provides a more robust perspective on the whole problem than any amount of "paper" speculation.
* The soma world is an excellent toy domain for isolating the complexities of the assembly problem.

CM/10/Feb/1989

Testing resourceful programs

I did not at first appreciate the special problem of testing a resourceful program, that it was capable of coping with its own bugs just as well as the constraints of the problem.

The problems of large and irregular solution spaces

The solution space is so fiendishly irregular that I was often tempted to conclusions based on what later turned out to be an unrepresentative set. This usually took the form of deciding, after discovering a problem, and scrutinising the cases to hand, that a fully general avoidance strategy was not needed. Such decisions were nearly always wrong.

The general conclusion that this experience has led me towards is that it is worth while trying the most sophisticated strategies conceivable. They have nearly always turned out to run even more slowly than I had feared, but nevertheless always to be effective.

The triple generality of good behaviours

The behaviours were implemented as VAL2 programs. These were general in three distinct ways. They were general in terms of program structure, in the conventional computational sense. They were general in terms of the assembly operations which were a side effect of the programmed motions. And they were general in terms of the variations of form and position of the component parts which they could cope with. These are distinct generalities in the sense that any one can be achieved without the others.

It was this triple generality which made these VAL2 programs into behaviours. The computational generality meant that they could easily be combined. The operational generality meant that this computationally simple interface could be simply related to suitable terms in which to reason about the planning of the assembly. And the generality over variations of form and position meant that the planner did not have to think about uncertainty. This both greatly simplified the planning task, and ensured that the plan would work reliably in the real world.

---------- O ----------

Extra: DAI Res. Paper 417
Programming Robotic Ass.. In Terms of Task Ach.. Beh.. Modules.

DAI RP 410
: First Expt. Results.

DAI RP 420
Symbol Grounding via a hybrid arch in an autonomous assembly system.