A comparison between two paradigms of intelligent systems—An example

by ABRAHAM WAKSMAN
Temple University
Philadelphia, Pennsylvania

ABSTRACT

Almost all intelligent computer systems of the past decade could be characterized by the General Problem Solver (GPS) paradigm. This paradigm states that the intelligent system activity consists of two distinct elements considered as separate modules. The first module is the generalist, the general problem solver while the second module could be considered as its data base, consisting of facts about the universe of discourse.

Current research indicates that to bridge the gap between simple display of inference making ability and an actual complex world situation requires a shift in philosophical approach. A new approach which promises to overcome the major drawbacks of the old paradigm could be characterized as the Plan-Debug paradigm. Similar to the old paradigm it could also be characterized as consisting of two modules, the plan making module and the debugging module. Conceptually this paradigm states that in order to execute any task or solve a problem we need to start with a plan of action regardless how imperfect. Once we get stuck, we consult a specialist with a lot of knowledge about the particular situation. In this Plan-Debug paradigm the emphasis has shifted from a large and powerful generalist module to a simple plan making module, from a small and simple data base to a large and dynamically structured data base. This shift facilitates the optimization of search processes in a semantically relevant way. It also facilitates the updating without requiring system modifications.

EXAMPLE

Design problems are very often problems in constraint satisfaction. Given a final design as a goal, the problem is to accomplish the goal without violating a set of pre-specified conditions concerning the inter-relationships between parameters and concerning the resources available. The final design is as a rule, a compromise between these factors in a way which optimizes some predefined criteria.

What follows is an example of the constraint satisfaction problem. We relate it to the two approaches to the design of intelligent systems as discussed above.

Consider a design problem, one which allows no compromises, but rather total constraint satisfaction with no duplications. We feel however, that it is not a toy problem in the sense that the solution will carry over to problems solved on more complex systems, while still employing the underlying concepts.

Our example displays the suitability of the new approach to solving problems encountered in computer aided design. We bring forth the difference in the processing load exhibited by the two systems. We show that the ability to perform more direct searches result in a more effective system.

Computer-aided design problems of the constraint-satis-
faction form can be characterized by a query and a set of partial information elements relating to the query. The problem solution corresponds to interrelating the partial information in a nonconflicting way. The response to the query is then directly derivable.

Constraint-satisfaction problems could also be compared to problems of tiling the finite plane with a set of non-regular tiles. The completely tiled plane constitutes the solution. The size of the plane, shape and number of each type of tile used constitute the set of constraints imposed on the solution.

Consider five modules that have to be arranged in a row from left to right, one next to the other. Each module is supposed to perform a specific task different from the tasks of the other modules. To each such task (function), we assign four specific attributes. The problem is to assign one, yet unassigned, attribute to one of the functions, in such a way that a prespecified set of constraints, relating to the modules, functions, and attributes and to the relations between them, will not be violated.

Let us state this problem in terms of the so-called Zebra-problem:

There are five gentlemen who live in a row of five houses.

1. The gentleman that smokes Old-Gold has a snail for a pet.
2. The gentleman that smokes Kool lives in a green house and has a neighbor with a horse for a pet.
3. The gentleman that smokes Chesterfield lives next to the gentleman that has a fox for a pet.
4. The gentleman that smokes Lucky-Strike drinks Orange.
5. The gentleman that smokes Parliament is Japanese.
6. The Spanish gentleman has a dog for a pet.
7. The English gentleman lives in the red house.
8. The gentleman who lives in the green house drinks coffee and is to the right of the ivory house.
9. The Norwegian gentleman who lives in the first house lives next to the blue house.
10. The gentleman in the third house drinks milk.
11. The Ukrainian gentleman drinks tea.

The problem is to find out which of the above five gentlemen owns the zebra, given that a zebra is one of the five pets belonging to the five gentlemen.

Theorem proving systems based on the general problem solving paradigm, when given a problem such as the zebra problem, will proceed in converting the given constraints into a set of axioms. Using the axioms, they will then proceed to state all possible solutions as theorems. These theorems are to be proven by the system true or to be refuted by it.

The eleven statements of the zebra problem will become then the set of axioms. The theorems to be proven true will become:

Th1. The zebra belongs to the Japanese.
Th2. The zebra belongs to the Englishman.
Th3. The Ukrainian owns the zebra.
Th4. The Norwegian owns the zebra.

In order to gain some insight into the processing load imposed by the theorem proving system, we proceed to develop a search path for the proof of Th3.

Theorem: Uk.—zebra
Then: Uk.—not (O.G., Par., L.S.) Fm. (1,4,5)
Then: Uk.—or(Kool, Ch.)

Lemma1: Uk.—Kool.
Then: Uk.—not(1st, 2nd, 3rd, 4th) Fm. (9,2,7,8)
Then: Uk.—5th.
Then: Ivy.—4th.
Green.—3rd. Fm. (2,8,12)
Then: Green.—Milk Fm. (10)
But: Green.—Coffee Fm. (8)

Contradiction. (Lemma1)
Then: Uk.—not (Kool, O.G., Par., L.S.)
Then: Uk.—Chesterfield.
Then: Uk.—not (1st, 4th)
Then: Uk.—or(2nd, 3rd, 5th.)

Lemma2: Uk.—2nd.
Then: Uk.—Blue Fm. (12)
Red.—or(3rd, 5th.)
Green.—or(3rd, 4th.)
Ivy.—or(4th, 5th.)
Yellow.—1st.

Then: Uk.—Horse Fm. (2)
But: Uk.—Zebra Fm. (Th3)

Contradiction. (Lemma2)
Then: Uk.—not(1st, 2nd, 4th.)
Then: Uk.—or(3rd, 5th.)

Lemma3: Uk.—3rd.
Then: Uk.—Milk Fm. (10)
But: Uk.—Tea Fm. (11)

Contradiction. (Lemma2)
Then: Uk.—not(1st, 2nd, 3rd, 4th.)
Then: Uk.—5th.

4th.—Ivy Fm. (8)
3rd.—Green Fm. (8)
2nd.—Blue Fm. (12)
1st.—Yellow Fm. (2)

Then: Uk.—Red.
But: Eng.—Red.
Contradiction. (Th3)

We have thus disproved that the Ukrainian gentleman is the owner of the zebra. We still have to tackle the other three theorems. Even when we discover a proof for the validity of one of the theorems, we cannot stop since there might be more then one solution. The constraints might be satisfied in more than one way.

Inherent in a solution as the above is the inability to make use of prior deductions to aid in the solution process. It is, however, possible to derive and save axioms which are needed more then once. In our case, the fact that the
second house is blue (Axiom 12) is such as axiom, deduced from the fact that the 1st house has a blue house for a neighbor (Ax. 9).

A somewhat longer chain of deduction is needed to conclude that the Norwegian lives in the yellow house (9, 8, 12, 7). An automatic system such as a theorem prover cannot have, however, prior knowledge as to the utility of a given deduction that results in a new axioms derived from old axioms.

The danger in giving a theorem prover freedom to derive new axioms is that proliferation of axioms can very easily get out of hand causing the solution process to bog down for any moderate size problem.

In contrast to the theorem prover, the plan-debug system approach to the solution of constraint-satisfaction problems consists of relegating the pre-processing activity to a set of specialized procedures which have the responsibility to do the domain-specific processing before the main algorithm is getting activated and when it is confronted with a conflict. We can divide the plan-debug system structure into three main modules as follows:

1. A data base that consists, initially, of the problem-constraints as its data entities.

2. A set of "trigger-functions" or demons. These demons could be considered axiom-activated update functions. Each demon is associated with an axiom or a set of axioms which activate (invoke) it whenever they enter into new relation with other axioms. The demons’ task is to insure that no side effect of any newly formed association is left un inspected, unrecorded or reported if need be. The demons might add new axioms to the data base; they might introduce data base elements into new relations; and will declare conflicts as a consequence of improper or illegal update.

A color-demon, for example, might have the responsibility to see to it that:

A. No color is used in more than one house.

B. No house can have two colors.

C. The next-to data type has the following properties;
   a. IF Loc.—5th. Then Next-to—4th.
   b. IF Loc.—1st. Then Next-to—2nd.
   c. IF Loc.—x. Then Next—to—or(x—1), (x+).

Also, IF Loc.—x. Then To-The—Right-of—(x—1).

3. A general purpose search algorithm, the planner, which follows simple guidelines for the initial search. As the search progresses, the data base upon which the planner acts gets modified by the demons. Demons that remain active for any length of time become temporarily part of the planner and remain active under its control. This is, in effect, a form of parallel processing in the sense that when update occurs, the planner is considering the total data base terms of a solution rather then one part of it at a time. Each update will, in effect, generate a modified version of the data base together with a list of modifications that occurred since the start of the solution process.

We can characterize the activity of the planner as follows:

1. Consider every constraint as an incomplete tuple ranging over all the domains in the data base. Thus the constraint includes, besides the original set of domain-value pairs, domain-value pairs with unspecified value.

2. The logical deduction process has the effect of introducing the appropriate values to fields with unspecified values.

3. Whenever a value is entered in an incomplete tuple’s field, the planner checks to see if some other incomplete tuple has such a value under a similar domain. In that case a join is performed, that is, the two incomplete tuples are combined into one resulting in less unspecified fields.

4. A solution exists whenever there are no more incomplete tuples in the data base. That is, when all the tuples ranging over all the available domains consist of domain value pairs and all the values are uniquely specified. This will also result in the smallest number of tuples in this final version of the data base.

Let us consider the solution process in a plan-debug system as described above:

The color-demon establishes at the start of the computations the following fact:

(Blue—2nd.) and (Green—to-the-right-of Ivory)

Ivory—(4th., 5th.)

Lemma1: Ivory—4th.

Then: Yellow—1st.

Blue—2nd.

Green—3rd.

Red—5th.

Then: Coffee—3rd.

But: Milk—3rd.

Contradiction. (Lemma1)

Then: Ivory—5th.

Then: Uk.—or(2nd., 5th.)

Lemma2: Uk.—2nd.

Then: (since there are no contradictions to lemma2) we proceed to represent the five complete tuples in a table form. In actuality, such a table will result from the collapsing of the data base with the incomplete tuples as tuples continue to be joined.

<table>
<thead>
<tr>
<th>Cig.</th>
<th>Pet</th>
<th>Color</th>
<th>Loc.</th>
<th>Nat.</th>
<th>Drink</th>
<th>(Pet)</th>
<th>Next</th>
<th>(Color)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kool</td>
<td>Fox</td>
<td>Yellow</td>
<td>1st.</td>
<td>Nor.</td>
<td>—</td>
<td>Horse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Che.</td>
<td>Horse</td>
<td>Blue</td>
<td>2nd.</td>
<td>Uk.</td>
<td>Tea</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.S.</td>
<td>Dog</td>
<td>Ivory</td>
<td>5th.</td>
<td>Sp.</td>
<td>Orange</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the process of generating a duplicate data base for the case of (Uk.— — — - 5th.) we discover a contradiction as fol-
Lemma 3: Uk. - - - 5th.
But: Orange—Lucky Strikes
Contradiction. (Lemma 3)

Thus, the above table represents the only possible solution to the zebra problem. The zebra belongs to the Japanese and the data base consists of five tuples ranging over nine domains each. This is the only combination of parameters which will allow for a solution without a conflict.

CONCLUSION

The dramatic improvement in the way that the plan-debug system handles the zebra problem as against the theorem prover is due fundamentally to the ability of the set of demons to capture in a procedural way the semantics of the data base (the set of constraints). This knowledge, which is specific to the problem at hand, need not be encapsulated in a more general way in the main algorithm.

The main algorithm, the planner, uses general deduction to add values to fields in incomplete tuples. It also performs the joins.

The debug facility introduces special deduction initially as well as when the general deduction is not adequate.

REFERENCES