AN IMPLEMENTATION OF A MULTI-AGENT PLAN SYNCHRONIZER

Christopher Stuart

Department of Computer Science
Monash University
Clayton 3108 Australia

ABSTRACT

A program is described which augments plans with synchronizing primitives to ensure appropriate conflict avoidance and co-operation The plans are particularly suitable for describing the activity of multiple agents which may interfere with each other The interpretation of a plan is given as a non deterministic finite automaton which exchanges messages with an environment for the commencement and conclusion of primitive actions which take place over a period of time The synchronized plan allows any and all execution sequences of the original plan which guarantee correct interaction

1. Introduction

All planning systems operate by combining actions or sub-plans in some way so that the total plan satisfies some constraint usually to achieve a goal state. Knowledge of how actions interact with each other and the world is used to determine the appropriate combinations In NOAH[5], for example, consideration of interactions between sub-plans is explicit in the planning process A plan is a partial order of sub-plans The planning technique is to expand sub-plans into partial orderings of lower level sub-plans, and then look for and resolve ensuing conflicts The resolution may impose fixed orderings This can be unnecessarily restrictive, as in the case where two sub-plans may execute in any order but not at the same time.

This paper considers the use of synchronizing primitives to resolve conflicts and produce a plan which is as unrestrictive as possible The models of plan and action used are appropriate for simple agents or robots engaged in parallel and or repetitive tasks which may be described at a level not using sensory input to the agents. Use could be made of this technique, for example, in automated assembly lines Georgeff [2] has also done related work on planning for multiple agents

2. Underlying theory.

Here we give an informal summary of a fully formalized theory of action and the world, which is described in [6].

2.1. Actions

An agent changes the world by executing *actions* When multiple agents are operating in parallel, it may be possible for two actions to be executing simultaneously, and so actions must have a beginning and an end. We consider an action to be decomposed into discrete transformations of the world, which are called *event??*. An event also has an associated correctness condition, which must be true at the moment it is executed An action will be a set of possible finite sequences of events.

2.2. The environment

The state of the *environment* in which actions are executed consists of a world state, and a set of actions currently being executed. If an agent executes an action, one of the possible sequences of events for that action is selected non-deterministically and added to the environment state The environment may at any time take a currently executing action, pop the next event from the event sequence, check for event failure, and change the world state according to that event.

The environment defines a set of symbols called *operators,* and gives each operator an interpretation as a set of event sequences (action). These operators are the means of interaction between an agent and the environment, and are exchanged as messages

2.3. Agents

The execution of a plan corresponds to some sequence of messages between the environment and an *agent*. Let *A* be the set of operators Then a sequence of messages will be denoted by a string over the alphabet *{begin,end) X A*. For any a€-4, *{begin a)* corresponds to the agent sending a to the environment to cause the associated action to be executed, and *{end)* corresponds to the environment sending a to the agent to indicate that the associated action has completed We refer simply to *strings* and assume them to be over this alphabet.

An *agent* is an acceptor for *string**. The formal model for an agent is similar to a non-deterministic finite automaton. It has a set of nodes (agent states), and a set of arcs defining allowed state transitions with associated messages An agent deadlocks if it is in a state from which there are no possible state transitions involving a *{begin* α) message, and the environment has finished executing all the operators sent by the agent

An agent defines a set of possible strings, and for any string, an environment defines some set of possible world state sequences. The planning problem is to take information about the environment, and find an agent which has some desired elfect on the world, such as the ultimate achieving of a goal world state, no matter what choices the environment makes We say two agents are *equivalent* if in any environment they induce the same set of possible sequences of world states.

An agent is *bounded* if there is a finite upper bound on the number of actions which the environment can be executing at a time. Thus a bounded agent will suffice to represent the concurrent activity of a finite number of multiple real world agents

2.4. Plans

Given three symbol sets *A, M* and *S* being *operators, memory states* and *signals* respectively, *plans* are defined recursively.

- For any αE *A* : α is a plan for executing an single action
- For any mE M, *s£S* (set m), (send *s)* and (guard *m s)* are synchronizing primitives.
- If p_l and p_2 are two plans, then p_1p_2 is the plan to execute them in sequence, $P_1||p_2$ is to execute them in parallel, $p_1\backslash p_2$ is to execute one or the other by non-deterministic selection, and p_1 executes p, an arbitrary number of times

The semantics for plans is given as a mapping from *plans* to *agents,* which is described in |(6) Intuitively, the agent for the simple plan α is an automaton accepting only the string *{(begin α),(end* α)} The plan operators build automata in the conventional manner. The synchronizing primitives correspond to arcs in the automaton which have a side effect on plan execution without exchanging any messages with the environment *Set* changes the *memory statt* of the plan, and *guard* and *send* can only be executed simultaneously, and then only when the memory is in the specified state and the signals match. This particular form of primitive is an adaptation of synchronization in the parallel programming language CSP[3], which uses guards which may be a combination of an input/output operation and a normal conditional

The following two results, given without proof, assert that there is a one to one correspondence between plans and bounded agents.

- Any agent which is given as the semantics of a plan is bounded.
- For any arbitrary bounded agent, there is a plan which has an interpretation equivalent to that agent.

3. The interaction problem

The problem addressed here is that of ensuring that a plan does not deadlock, or allow any event to fail. Correctness conditions on events or actions can be used to represent many types of plan correctness If a plan should achieve some goal state given some initial condition, this is ensured by beginning the plan with an action that, asserts the initial condition, and terminating it with an action that will always fail in the absence of the goal condition. A condition which must be maintained during a plan can be enforced with an action in parallel that will fail in the absence of the maintained condition.

A program has been developed which takes a plan and a description of an environment, and generates a revised plan which allows all and only the sequences of communication, acts of the first plan that cannot c ause failure, and also that will never deadlock

3.1. Preventing event failure

For this program we use a very simple form of event, corresponding to the operators of the STRIPS planner|lj. The world is modeled as a set of propositions. Events are constrained to add or delete propositions without reference to the current world state Also, the correctness condition is a conjunction of propositions or negated propositions Thus an event is four sets of propositions; a *require true* set, a *require false* set, an *add* set and a *delete* set

It turns out in this case that to prevent event failure an action is completely defined by five sets of atomic formulae (possibly negated propositions). The five properties are defined by considering the execution of the action in isolation from other actions An atomic formula is

- *asserted* if it is inevitably true after the execution
- *retracted* if it could possibly become false after the execution
- *conflicted* if it could become false at some stage during the execution.
- *a precondition* if it must be true immediately before the action begins to ensure that no event will fail.
- *a during condition* if it must be true for some event in the action.

The necessary and sufficient rules for ensuring no event failure are:

- An action which has a during condition may not run in parallel with an action that conflicts that during condition.
- An action $_{op}$ which has a precondition may not begin until some action $_{od}$ which asserts that condition has completed, and also no action $_{or}$ which retracts that condition may be running from when α_a begins until α_p ends.

3.2. Synchronising Plant

We synchronize a plan by inserting send operations, and running it in parallel with a *synchronization skeleton* consisting only of *guard* and *set* operations The set of possible strings for the resulting plan is a subset of those for the original plan. Manna and Wolper describe an algorithm for generating such a skeleton from propositional temporal logic (PTL) formulae used to express constraints on execution sequences of a plan.[4]

PTL is a logic for reasoning about sequences of states. The interpretation of a PTL formula is the set of sequences for which it is *true*. States give truth values to propositions and hence to non modal formulae, and the temporal connectives of PTL *(always, eventually, until* and *next)* have truth values depending on the successors of a state. We also use a *regular expression* operator equivalent in expressiveness to the grammar operators of Wolper.[7] A regular expression with the basic elements being non modal formulae translates into a set of sequences of non modal formulae, and then to a set of sequences of states.

3.3. The algorithm

The plan synchronizer has three phases First, PTL formulae are generated Correctness constraints, being the two rules defined above, are expressed in standard PTL, using propositions to represent relevant stages in plan execution A *regular expression* formula corresponding to a simplification of the plan is used to express the constraints imposed by the plan syntax on the order of the relevant stages. The formulae are simplified or ignored depending on orderings already enforced by the plan syntax This reduces the time spent in the second phase without altering the interpretation of the conjunction of all the PTL formulae generated

Second, a tableau method of theorem proving is applied Formulae are decomposed into non modal constraints on the first state of a sequence, and general constraints on the remainder A graph is constructed, with arcs corresponding to states in plan execution, and nodes labeled with PTL formulae This graph is pruned to enforce eventuality constraints Every interpretation for the original PTL formula is a path through the final graph, and every finite path is the prefix of an interpretation

Finally, the graph is converted directly into a synchronization skeleton, and *send* operations are inserted into the plan for every proposition used The set of memory states used corresponds to the set of nodes in the graph, aud each arc is represented as a guarded command to alter the memory. The resulting plan allows all possible execution sequences of the original which do not permit an event to fail, and which always allow for plan termination

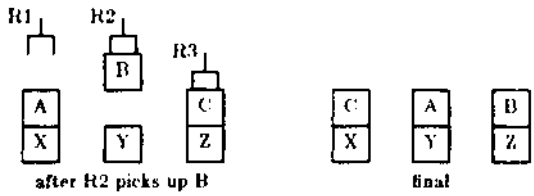
The current version of the program only handles restricted dasses of plans and actions (no loops, no selection, and actions consisting of a single event sequence), but is being extended at the moment to include these

4. An example

Consider the problem of three robots all trying to pickup a block and move it clockwise to a location which another robot will clear as it moves, represented by the following unsynchroni/.ed plan

```
( (START (RI R2 R3) ((A X) (BY) (C Z)))
  (PARALLEL ( (PICKUP RI AX) (PUTDOWN RI A V))
            ( (PICKUP R2 B V) (PUTDOWN R2 BZ))
            ( (PICKUP R3 C Z) (PUTDOWN R3 C X))))
```

The start action sets up the initial conditions, and then each robot in parallel executes a pickup and putdown. Clearly collisions might result. The problem is represented pictorially in the following diagram:



The synchronized plan produced by the program is

```
( (PARALLEL
    ( (SEND (BEGIN 1))
     (START (R1 R2 R3) ((A X) (B Y) (C Z)))
     (PARALLEL
        ( (PICKUP R1 A X) (SEND (END 2 1 1))
          (SEND (BEGIN 2 1 2)) (PUTDOWN R1 A Y))
        ( (PICKUP R2 B Y) (SEND (END 2 2 1))
          (SEND (BEGIN 2 2 2)) (PUTDOWN R2 B Z))
        ( (PICKUP R3 C Z) (SEND (END 2 3 1))
          (SEND (BEGIN 2 3 2)) (PUTDOWN R3 C X))))

    ( (RECV (BEGIN 1))
     (SETQ N 2)
     (WHILE (NOT (EQ N 13))
        (SELECT-ONE OF
           (IF (AND (EQ N 2) (RECV (END 2 3 1)))
              THEN (SETQ N 5))
           (IF (AND (EQ N 2) (RECV (END 2 2 1)))
              THEN (SETQ N 4))

                  (IF (AND (EQ N 5) (RECV (END 2 1 1)))
                     THEN (SETQ N 6)))))))
```

The syntax is close to CSP, and can be translated directly into plans as we have defined them. A large section of the synchronization skeleton has been removed in the example, since it contains 42 guarded commands - one for each arc in the model for the PTL formulae. The final plan has the desired result of holding any putdown until the appropriate pickup has completed. Each *pickup* is followed by a *send* which indicates a block is clear, and each *putdown* is preceded by a *send* which is delayed until the appropriate destination block is clear.

5. Conclusion and Future Work

There are some optimizations possible in the general method by pruning the graph, which in the above example would have reduced the size of the synchronization skeleton, and removed redundant references to (DEGIN 1). It is also worthy of investigation to consider how synchronization primitives could be inserted in the main plan without adding a new parallel branch with the synchronization skeleton, or how the synchronization skeleton could be made more modular, with distinct components to handle particular constraints.

The definition of actions and environments given here enables very strong properties to be given to the synchronized plans: in particular that *all and only* the correct executions of the initial plan are permitted. This is in contrast to previous means of synchronizing plans which prohibit some execution sequences that would succeed.

By extending the definition of actions to include general state transformations in events, a similar algorithm could generate a plan which is still less restrictive than that produced by previous plan modifying techniques, but might still disallow certain correct executions. There is no simple action description capturing all the essential properties in the same way as can be done in the simple case with five sets of atomic formulae. Also, the PTL formulae might need to reference propositions reflecting world state as well as the stages of plan execution. This problem could be considered in more detail.

There is also the problem of types of non-determinism. The current selection operator corresponds to the case where a plan may proceed in one of two directions, and the synchronizer is permitted to chose one over the other. This is *angelic* non-determinism. However, it may be the case for some plans that the choice is critical, but made at execution time, in which case the synchronizer must allow both cases or none at all. This is *demonic* non-determinism, and implies some additional structure to a plan which restricts the ways in which it may be synchronized. For added complexity, the decision may be based on the state of the world model, and so the synchronizer can determine the choices it must leave open, depending on the possible world models it derives for the moment of choice.

Loops often have a termination condition which is a function of all the activity in the loop, and yet may not easily be derived from the given information. Such a termination condition could be specified if a plan segment were treated as a single hierarchical action, and given properties similar to those for individual actions. Consider a loop of an action that removes a single item from a box until none are left. To represent this in the formalism given here, the entire loop would be given an assert condition that the box become empty. To guarantee termination, the entire loop could be given a during condition that no one places anything in the box.

A version of the program is being designed which will take as input an arbitrary plan as defined above, and will also handle both types of non-determinism, and conditions attached to sub-plans as hierarchical actions. The theoretical justification is being pursued concurrently.

ACKNOWLEDGEMENTS

Much of this investigation was conducted at the AI center at SRI International. Thanks are due to the center and especially to Michael GeorgeiT for helpful discussion; and to Monash University for financial assistance, and where the work is proceeding.

REFERENCES

[J] Fikes, R.E., Nilsson, N.J. "STRIPS: A new approach to the application of theorem proving in problem solving." *Artificial Intelligence* 2 (1971), pp 189-208.

[2] Georgeff, MP. "Communication and Interaction in Multi-Agent Planning." In P*roc. AAAISS,* (1983) ppl25-129.

[3] Hoare, CAR. "Communicating Sequential Processes." In *Communications of the ACM* 21:8 (1978), pp 666-677.

[4] Manna, Z.; Wolper, P. "Synthesis of Communicating Processes from Temporal Logic Specifications." Report STAN-CS-81-872, Stanford University Computer Science Dept, September 1981.

[5] Sacerdoti, ED. "A Structure for Plans and Behaviour." Tech Note 109, SRI AI Center, Menlo Park, CA 1975.

[6] Stuart, C.J. "An Implementation of a Multi-Agent Plan Synchronizer Using a Temporal Logic Theorem Prover." Report under development, SRI AI Center, Menlo Park, CA, 1985.

[7J Wolper, P. "Temporal Logic Can Be More Expressive." In *Proc. of the 22nd Symposium on Foundations of Computer Science.* Nashville, TN, (1981)

that A* is optimal, in terms of number of nodes generated, over the class of admissible best-first searches with *monotone* heuristics A monotone heuristic function is one which never decreases along a path This is not a serious restriction since given an admissible heuristic, we can trivially construct a monotone one by taking the maximum value along the path so far as the value of each succeeding node on the path [Mero 84] Therefore, IDA* is asymptotically optimal in terms of time over the class of admissible best-first tree searches

Finally, we consider the space used by IDA* Since IDA* at any point is engaged in a depth-first search, it need only store a stack of nodes which represents the branch of the tree it is expanding Since it finds a solution of optimal cost, the maximum depth of this stack is *d,* ana hence the maximum amount of space is *O(d)*

To show that this is optimal, we note that any algorithm which uses *f(n)* time must, use at least, *klog f(n)* space for some constant *k* [Hopcroft 79] The reason is that the algorithm must proceed through *f(n)* distinct states before looping or terminating, and hence must be able to store that many states Since storing *f(n)* states requires *log f(n)* bits, and *log* b^a is *d log b,* any brute-force algorithm must use *kd* space, for some constant *k* Q.ED

As mentioned above, most heuristics in practice exhibit at least constant relative error Thus, we can conclude that heuristic depth-first iterative-deepening is asymptotically optimal for most best-first tree searches winch occur in practice An additional benefit of IDA* over A* is that it is simpler to implement since there are no open or closed lists to be managed A simple recursion performs the depth-first search, and an outer loop handles the iterations

As an empirical test of the practicality of this algorithm, both IDA* and A* were implemented for the Fifteen Puzzle The implementations were in Pascal and were run on a DEC 2060 The heuristic function used for both was the admissible Manhattan distance heuristic for each movable tile, the number of grid units between the current position of the tile and its goal position are computed, and these values are summed for all tiles The two algorithms were tested against 100 randomly generated, solvable initial states IDA* solved all instances with a median time of 30 CPU minutes, generating over 1.5 million nodes per minute The average solution length was 53 moves and the maximum was 66 moves A* solved none of the instances since it ran out of space after about 30,000 nodes were stored As far as we know, this is the first algorithm to find optimal solutions to randomly generated instances of the Fifteen Puzzle within practical resource limits An additional observation is that in experiments with the Eight Puzzle, even though IDA* generated more nodes than A*, it actually ran faster than A* on the same problem instances, due to the reduced overhead per node The actual data from these experiments are reported in [Korf 85]

4. Conclusions

The best known admissible search algorithm, A*, is severely limited by its exponential space requirement An iterative-deepening version of A* uses only linear space, and is asymptotically optimal in terms of cost of solution, running time, and space required for exponential tree searches Since almost all heuristic searches have exponential complexity, Iterative-deepemng-A* is an optimal admissible tree search in practice. In addition, it was easier to implement and ran faster than A* in our experiments

Acknowledgments

1 would like to acknowledge the helpful comments of Mike Lebowitz, Andy Mayer, and Mike Townsend who read an earlier draft of this paper, and several helpful discussions with Hans Berliner and Judea Pearl concerning this research In addition, Andy Mayer implemented the A* version that was compared with IDA*

References

[Berliner 84] Berliner, Hans, and Gordon Goetsch A quantitative study of search methods and the effect of constraint satisfaction technical report CMU-CS-84 147 Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa , July, 1984

[Dechter 83] Dechter, Rina, and Judea Pearl The optimality of A* revisited Proceedings of the National Conference on Artificial Intelligence, Washington. D C , August, 1983, pp 95-99

Hart 68] Hart, Peter E , Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths " *IEEE Transactions on Systems Science and Cybernetics SSC-4, 2* (19f)8)

[Hopcroft 79] Hopcroft, John E, and Jeffrey D Ullman *Introduction to Automata Theory, Languages, and Computation* Addison-Wesley, Reading, Mass , 1979

[Korf 85] Korf, Richard E "Depth-first iterative-deepening An optimal admissible tree search " *Artificial Intelligence to appear* (1985)

[Mero 84] Mero, Laszlo "A heuristic search algorithm with modifiable estimate" *Artificial Intelligence 28* (1984)

[Pearl 84] Pearl, Judea *Heuristics.* Addison-Wesley, Reading, Mass , 1984

Slate 771 Slate, David J , and Lawrence R Atkin -HESS 4 *> - The Northwestern University chess program In Frey, Peter W , Ed , *Chess Skill in Man and Machine,* Springer-Verlag, New York, 1977

[Stiekel 85] Stickel, Mark E , and W Mabry Tyson An analysis of consecutively bounded depth-first search with applications in automated deduction Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85), Los Angeles, Ca , August, 1985