Planning with Resources and Concurrency A Forward Chaining Approach[*]

Fahiem Bacchus
Dept. Of Computer Science
University Of Toronto
Toronto, Ontario
Canada, M5S 1A4
fbacchus@cs.toronto.edu

Michael Ady
Winter City Software
Edmonton, Alberta
Canada, T5R 2M2
winter.city@v-wave.com

Abstract

Recently tremendous advances have been made in the performance of AI planning systems. However increased performance is only one of the prerequisites for bringing planning into the realm of real applications; advances in the scope of problems that can be represented and solved must also be made. In this paper we address two important representational features, concurrently executable actions with varying durations, and metric quantities like resources, both essential for modeling real applications. We show how the forward chaining approach to planning can be extended to allow it to solve planning problems with these two features. Forward chaining using heuristics or domain specific information to guide search has shown itself to be a very promising approach to planning, and it is sensible to try to build on this success. In our experiments we utilize the TLPLAN approach to planning, in which declaratively represented control knowledge is used to guide search. We show that this extra knowledge can be intuitive and easy to obtain, and that with it impressive planning performance can be achieved.

1 Introduction

For a long time AI planning systems were either capable of solving only trivial problems, or required extensive engineered knowledge to solve problems that were still relatively simple. Recently, however, tremendous performance gains have been made. These gains have come from the development of new approaches to planning, and most recently from the improvement of the old idea of forward chaining. As a result, the fastest planning system in the recent AIPS-2000 planning competition [AIPS, 2000] was a forward chaining planner that was able to generate plans containing 2000 steps in less than 2 seconds. This level of performance was achieved on simple test domains. Nevertheless, if performance within one or two orders of magnitude of this can be achieved in real application domains, tremendous possibilities for the practical application of AI planning will be created.

However, performance is not the only impediment to the practical application of AI planning systems. The scope of problems they can represent and solve is also a problem. The planners in the competition were restricted to problems in which actions could be modeled by a set of simultaneous updates to the predicates describing the world. This is the model of planning inherited from the STRIPS action representation; it is also the model used by the ADL [Pednault, 1989] representation. ADL simply provides more flexibility in specifying the set of predicate updates that an action generates, so that, e.g., this set can be conditional on the current world.

Real world applications require modeling a number of more sophisticated features, including uncertainty, sensing, varying action durations, delayed action effects, concurrently executing actions, and metric quantities. In this paper we address the last four issues. In particular, we present an approach to modeling and solving planning problems containing metric quantities, actions of varying duration, actions with delayed effects, and concurrently executing actions. Our approach is based on extending the forward chaining approach to planning. In our experiments we demonstrate that, in particular, the TLPLAN approach to planning [Bacchus & Kabanza, 2000] can be successfully extended to deal with such problems.

Metric quantities have never been a significant problem for forward chaining planners, and in fact the TLPLAN system has been able to deal with metric quantities since its original 1996 implementation. Hence, the ability to deal with metric quantities is not a contribution of this paper. However, the manner in which TLPLAN deals with metric quantities is unique and has many advantages that help support the other extensions that are new to this paper.

In the sequel we first review and motivate the manner in TLPLAN extends the STRIPS/ADL representation to deal with metric quantities. Then we develop our approach to modeling actions with delayed effects from which the ability to model concurrent actions follows naturally. The approach we present can be used in any forward chaining planner. We compare our approach to some of the other work in this area, and then present some empirical results to demonstrate the potential of our approach and the capabilities of the planner

[*]This research was supported by the Canadian Government through their NSERC and NCE-IRIS programs.

we have developed. Unfortunately it proved to be impossible to run controlled experiments to compare our planner with other systems that have been developed. So we have instead made an effort to present a suite of experiments and provide all of the necessary data sets so that a reusable experimental basis can be established for future work.

2 Functions

We take planning problems as including a fully specified initial state, a goal, and a set of actions for transforming states to new states. A solution to the problem is a sequence of actions that when applied to the initial state yields a sequence of states satisfying the goal. The goal might simply be a condition on the final state of the sequence, or it might place conditions on the entire sequence of states.[1]

A forward chaining planner is one that searches in the space generated by applying to each state s all actions whose preconditions are satisfied by s, starting at the initial state I. A forward chaining planner expands this space as it searches for a sequence of actions that transform I to a state (or sequence of states) satisfying the goal. In other words, forward chaining planners treat the planning problem as a state-based search problem.

The key difference between the planning problem and a generic search problem, however, is that planning assumes a particular representation of states and operators. In planning states are represented as databases of predicate instances, and operators are represented by specifying the set of updates they make to the database (state) to generate a new state (database). In other words, planning uses a factored representation of the state in which each transition updates only a few of the state's components. It is this factored representation that has allowed the development of planning specific notions such as goal-regression.

In planning, the closed world assumption is standard: any predicate instance not in the database is assumed to be false. Under this assumption the state databases become first-order models against which arbitrary first-order formulas can be efficiently evaluated [Halpern & Vardi, 1991]. Given a first-order formula $\phi(\vec{x})$ containing some set of free variables $\vec{x}$, we can efficiently find all tuples of bindings for $\vec{x}$ that make $\phi(\vec{x})$ true in a state: this is the same problem as computing the relation specified by an SQL query in databases.

A natural semantics for operators specified in the STRIPS or ADL notation is to view them as being update queries. Each operator has a precondition $\pi(\vec{x})$ that is a first-order formula containing the free variables $\vec{x}$. Every binding $\vec{c}$ for the variables $\vec{x}$ such that $\pi(\vec{c})$ is true in a state s generates an action that can be applied to s to yield a new state. The STRIPS/ADL action representations have the property that a set of fully instantiated predicates to add and delete from s can be computed simply by evaluating formulas in s. For example, if an ADL operator `(drive ?t ?l ?l')` (drive truck `?t` from location `?l` to `?l'`) contains the conditional update[2]

```
(forall (?o) (in ?o ?t)
  (and (add (at ?o ?l')) (del (at ?o ?l))))
```

(i.e., update the `at` property of all objects `?o` in `?t`), then given a binding for `?t`, `?l`, and `?l'`, i.e., a fixed action instance, by computing the set of bindings for `?o` that satisfy `(in ?o ?t)` in s all instances of `at` that must be changed can be determined. Notice that the specific predicate instances in the update are determined by replacing the terms `?o`, `?l`, and `?l'` by their values. In this case these terms are variables and their values (interpretations) are determined by the current variable bindings.

First-order languages typically include functions. Terms can then be constructed by applying functions to other terms. Thus a *natural* extension to the STRIPS/ADL representation is to remove its function-free restriction. For every function `f` the state can include in its database a relation specifying the value of `f` on its various arguments. First-order formulas can be evaluated just as before: whenever we encounter a term like `(f t1 ... tk)` in a formula we replace it with its value by recursively evaluating each of the `ti` and then looking up the resulting tuple of values in the relation specifying `f`. We specify updates to these function values by asserting equalities that must hold in the next state. Now, e.g., instead of describing the location of objects `?x` with an `(at ?x ?l)` relation, we could describe their location with a `(loc ?x)` function. Then the operator `(drive ?t ?l ?l')` could contain the conditional update

```
(forall (?o) (in ?o ?t) (add (= (loc ?o) ?l)))
```

We use the convention that the function that is the first argument of the equality (`loc`) is the function to be updated, its arguments (the variable `?o`) are evaluated in the current state to determine which arguments of `loc` are to be updated, and the second argument (the term `?l`) is evaluated in the current state to determine the new value.

Functions whose values are numbers, and numeric functions like $+$ can now be accommodated in the same way. Furthermore, the standard numeric functions like $+$ can be computed using existing hardware or software: we do not need to have a table of its values as part of the state's database.

For example, if `(capacity ?t)` is the fuel capacity of truck `?t`, `(fuel ?t)` is its current level of fuel, and `(fuel-used)` is a (0-ary) function whose value in any state is the total amount of fuel used, then we can write an operator like (**`refuel`** `?t`) with the update

```
(and
 (add (= (fuel-used)
         (+ (fuel-used)
         (- (capacity ?t) (fuel ?t)))))
 (add (= (fuel ?t) (capacity ?t))))
```

In the new state truck `?t` will have a full tank and we would have accounted for the amount of fuel put into its tank.

[1]See [Bacchus & Kabanza, 1998] for more about such "temporally-extended" goals. In this paper we will confine our attention to "final-state" goals. However, the extensions we describe here could also be realized in the context of temporally-extended goals.

[2]The update asserts the truth or falsity of a collection of predicate instances in the next state, and due to the closed world assumption these assertions can be realized by adding or deleting these instances from s.

Adding functions to the action representation in this way, motivated directly by viewing operators as database updates, provides all the flexibility needed to model complex resource usage. For example, the following operators model FIFO access to a fixed resource and also track the number of times the resource is used. `(qhead)` and `(qtail)` are 0 in the initial state, `(queue i)` is a function whose value is the `i`'th request in the queue, and `(serve ?x)` is a predicate true of `?x` if `?x` is currently being served. `(qtail)` will always be the total number of times the resource is used, and `(qhead)` − `(qtail)` is always the number of items currently in the queue.

```
(def-adl-operator (enqueue-access ?x)
  (and (add (= (queue (qtail)) ?x))
       (add (= (qtail) (+ (qtail) 1)))))
(def-adl-operator (dequeue-and-serve)
  (pre (> 0 (- (qtail) (qhead))))
  (and
    (del (serve (queue (qhead))))
    (add (serve (queue (+ (qhead) 1))))
    (add (= (qhead) (+ (qhead 1))))))
```

In the literature addressing metric quantities, specialized notation has been developed for expressing resources (e.g., [Wolfman & Weld, 1999; Kvarnström, Doherty, & Haslum, 2000]). Our argument is that such notation is not required. The natural extension of making functions first-class citizens along with standard operator preconditions provides a better solution.[3] What we have just described is the manner in which TLPLAN has implemented functions since its original 1996 version. Functions as first-class citizens were also present in the original ADL formalism [Pednault, 1989], and [Geffner, 2000] provides some other arguments in support of using functions.

3 Modeling Concurrent Actions

Forward chaining has proved itself to be a very fruitful basis for implementing high-performance planners. For example, the two fastest planners in the recent AIPS-2000 planning competition (TALPlanner [Doherty & Kvarnström, 1999], a planner that uses the TLPLAN approach, and Fast-Forward [Hoffmann, 2000] a planner using domain independent heuristics to guide its search) were both forward chaining planners. However, there is at least one aspect of forward chaining planners that seems to be problematic: they explore totally ordered sequences of actions. This is where they get their power: such sequences provide complete information about the current state and that information can provide powerful guidance for search. But modeling concurrent actions with linear sequences seems to be problematic. However it turns out that there is a surprisingly simple way of modeling concurrency with linear actions sequences.

We associate with every state a time stamp, starting with a fixed start time in the initial state. The time stamp denotes the actual time the state will occur during the execution of a plan. In a linear sequence of states a number of successive states may have the same time stamp. Intuitively this means that the transitions between these states occur instantaneously, so the intermediate states are never physically realized. Their existence is simply a convenient computational fiction.

[3]Computing plans in the presence of metric quantities might require restrictions on how these quantities can be updated. But such restrictions should be imposed on a general representation, not used to determine the representation.

Additionally, each state has an event queue. The event queue contains a set of updates (events) each scheduled to occur at some time in the future (of the state's time stamp). Along any fixed sequence of states generated by a sequence of actions, each state inherits the pending events of its parent state. It might also queue up some additional events to be passed to its children. Thus if we arrive at the same state via two different actions sequences we could generate two different event queues. We regard two states as being equal only if they have both the same database and the same event queue. Thus, when the planner backtracks it backtracks to a state with a prior event queue, in effect backtracking the state of the event queue.

As before, an action a can be executed in a state s only if its preconditions are satisfied by s. Applying a to s generates a new successor state s^+. Standard actions do not advance the world clock, so s^+ will have the same time stamp as s. Typically, it will have a different event queue (i.e., what will happen in the future has changed), and a different database (i.e., what is true "now" has changed). Updates to s's database are used to model a's instantaneous effects, and updates to the event queue are used to model a's delayed effects.

For example, consider the action of driving a truck `?t` from `?l` to location `?l'`:

```
(def-adl-operator (drive ?t ?l ?l')
  (pre (?t)  (truck ?t)
       (?l)  (loc ?l)
       (?l') (loc ?l')
     (at ?t ?l))
  (del (at ?t ?l))
  (delayed-effect
    (/ (dist ?l ?l') (speed ?t))
    (arrived-driving ?t ?l ?l')
    (add (at ?t ?l'))))
```

We can execute this action in s if `?t` is a truck, both `?l` and `?l'` are locations, and `?t` is at location `?l` in s. The instantaneous effect of the action is to delete the current location of the truck, and the delayed effect is to add the new location of the truck. The delayed effect is realized by adding an item to the event queue. The first argument of the **`delayed-effect`** specification is the event's time delta, the number of time units from the current time the event is scheduled to occur. This time delta is a term that will be evaluated in the current state. In this case it is the distance between the two locations divided by the speed of the vehicle. The next argument is simply a label for the event (designed to make the final plan more readable). The delayed effects are the subsequent arguments. In this case it is the addition of the new location of the truck. In general, delayed effects can be any kind of effect allowed in a normal action, including, e.g., conditional effects.

In addition to the standard actions there is one special action that advances the world clock: the **`unqueue-event`**

```
Plan(<s,Q>,Goal)
  if (s ⊨ Goal and Q = {})
    return(s)
  else
    s+ := s
    s+.prev := s
    Q+ := Q
  choice a ∈ {act : s ⊨ pre(act)}
    s+.action := a
    if a != unqueue-event
      s+ := ApplyInstantaneousUpdates(s,a)
      Q+ := AddDelayedEvents(Q,s,a)
    else
         newTime := eventTime(front(Q))
         s+.time := newTime
         while eventTime(front(Q+)) == newTime
           e := removeFront(Q+)
           s+ := ApplyEffect(s+,e)
    Plan(s+,Q+)
```

Figure 1: Forward Chaining Search

action.[4] This action moves time forward to the next scheduled event, removes all events scheduled for that new time and uses them to update the state's database. This realizes various delayed effects of previous actions. For example, eventually the arrival of `?t` at `?l'` will reach the front of the queue and will be dequeued. This will cause a transition to a new state in which the fact `(at ?t ?l')` is added and the time is updated. If a set of events have been scheduled for the same time, they will all be dequeued and applied sequentially in FIFO order.

Figure 1 specifies more precisely forward chaining search in this enhanced search space. The non-deterministic ***choice*** operator is realized by search. `AddDelayedEvents` examines the action, and for each **`delayed-effect`** in `a` evaluates the term specifying the delay of that effect. Adding the time of the current state, `s.time`, gives the absolute time of the effect, and the effect is merged into the queue so as to keep the queue in time sorted order. The current variable bindings for the free variables in the effect are also stored along with the effect. If the chosen action is **`unqueue-event`** time is moved forward, and all effects scheduled for that time are removed from the queue and applied sequentially to the state. A goal state is a state whose database satisfies the goal and that has an empty event queue; we can find the sequence of actions leading to that state by following the state's `prev` pointers.

Search for a plan is started by calling **Plan** on the initial state. Note that the queue need not initially be empty. Instead it could contain some set of events that are going to occur in the future. The planner will then have to find a plan that negotiates around these future events. This also facilitates replanning where some previous actions cannot be canceled. This feature is similar to the ability of temporal refinement planners like IxTeT [Ghallab & Laruelle, 1994] and RAX [Jónsson *et al.*, 2000] to flesh out an initial set of temporal constraints.

[4]Dead time can be inserted into the plan by including a "wait" action in the domain. This action would have no instantaneous effects and would enqueue a null delayed effect, delayed by the wait period, into the event queue. The presence of such an event on the queue would allow **`unqueue-event`** to advance the world time by the wait period.

The choice of which action to try next is where heuristic or domain specific control comes into play. In the TLPLAN approach we restrict the set of possible action choices by requiring that the next state s^+ satisfy the temporal control formula (see [Bacchus & Kabanza, 2000] for details).

With delayed effects, concurrent actions are automatic. When an action is executed it generates a successor state in which its immediate effects have been made. This state "marks" the start of the action, and since it has the same time stamp as the previous state the action can be viewed as starting at the current time. After some stream of delayed effects have been executed the final delayed effect generates a state that "marks" the end of the action. Depending on how we interleave the **`unqueue-event`** action with the ordinary actions we can start a whole series of actions at the same time, these actions can execute concurrently and some can end before others. Thus at any particular time any number of actions can be executing concurrently. If we only choose **`unqueue-event`** when there is no other action available, we will maximize the number of concurrent actions at each stage: each ordinary action whose precondition is satisfied will be started before the world clock is advanced. Or we can achieve finer control over the degree of concurrency by controlling (via, e.g., a temporal control formula) when **`unqueue-event`** is chosen.

In our approach all concurrency control is handled by action preconditions. Typically the instantaneous effects of an action are used to modify the state so as to achieve concurrency control, while the delayed effects of an action are used to model physical achievements in the world. This is a low level but very powerful approach to concurrency control. For example, in the previous **`drive`** operator, the current location of the truck is immediately deleted. Since this is also a precondition of **`drive`**, any attempt to concurrently drive the same truck to another location is blocked. More sophisticated situations are also quite straightforward to model, in part because of our general approach to functions and numeric computations.

For example, consider a gas station with 6 refueling bays and a limited amount of fuel shared among these bays. Let `(station-fuel)` be the current amount of fuel at the station, `(bays-free)` the number of bays currently free, `(capacity ?v)` the fuel capacity of a vehicle, and `(fuel ?v)` the fuel in the vehicle. Then the following actions model resource bounded concurrent access to the gas station:

```
(def-adl-operator (refuel ?v ?amount)
 (pre
   (?v)       (vehicle ?v)
   (?amount)  (= ?amount (- (capacity ?v)
                            (fuel ?v)))
  (and (> 0 (bays-free))
       (> (station-fuel) ?amount)))
 (add (= (station-fuel)
         (- (station-fuel) ?amount)))
 (add (= (bays-free) (- (bays-free) 1)))
 (delayed-effect 10
  (fueled ?v ?amount)
(and (add (= (bays-free)
             (+ (bays-free) 1)))
     (add (= (fuel ?v) (capacity ?v))))))
```

The operator also demonstrates our system's ability to use functions to bind operator arguments. In this case `?amount` is bound to a function value computed from the binding of `?v`.

Suppose in a state s there are 12 vehicles and that various sets of these vehicles require more fuel that the station has. From s a number of different sequences of concurrent **refuel** actions can be initiated. But in each of these sequences no more than 6 vehicles will be concurrently fueled, due to the instantaneous update of the `(bays-free)` resource, and no set of vehicles needing more fuel than available will be concurrently fueled, due to the instantaneous update of the `(station-fuel)` resource. In fact, it can be that after the first batch of 6 vehicles is concurrently fueled an additional batch of vehicles enter the station after 10 units of time have elapsed and the bays have become free.[5] Exactly which sequence of concurrent refueling appears in the plan will be determined by what sequences allow the goal to be achieved and the search strategy.

The final plan will be a linear sequence of actions grouped into subsequences of actions each with the same time stamp. However, the linear sequencing is not a limitation of our approach. For example, a simple post analysis of these subsequences can be used to determine if the actions have to be started in the supplied order or if some other ordering can be used. For example, if no action in the subsequence affects the preconditions of another then they can be started in any order, or simultaneously. However, if starting the actions is near instantaneous in practice then there will be little to gain from such a post analysis. For example, say that one subsequence of actions to be executed at the same time is `(drive truck1 locA locB)` followed `(drive truck2 locC locD)` (start driving two trucks concurrently) then it typically will make very little difference if we tell `truck1` to start driving before telling `truck2`—the command to start driving takes negligible time in comparison to the actual drive.

Often what is required in these kinds of planning problems are goals that specify conditions over time. That is, specifying conditions on the final state is not sufficient. Temporal refinement planners like IxTeT [Ghallab & Laruelle, 1994] allow one to, e.g., enforce that a predicate holds without interruption over a particular interval of time. In our approach if one specifies only a condition on the final state, there is no way of stopping the planner from inserting actions produce undesired intermittent effects—there is no way of telling the planner that these intermittent effects are undesirable. Although we have not implemented it, there is no conceptual difficulty with combining our approach with our previous work on specifying temporally-extended goals [Bacchus & Kabanza, 1998]. With such a combination, an extremely rich set of extended conditions can be enforced on the final plan, including the typical conditions supported by temporal refinement planners.

[5]It would be easy have a more complex model of the time required to complete the fueling.

4 Empirical Results

We have implemented the above event-queue mechanism as an extension of the TLPLAN system, and tested our implementation using different versions of the metric logistics domain developed by [Wolfman & Weld, 1999].

Using this domain allows us to make some empirical comparisons with previous work. Unfortunately, it proved to be impossible to run controlled experiments with other planning systems (the systems and problems sets were not readily available, or the systems were not easily ported to our machine). Therefore, the results we report are simply to demonstrate that our approach can efficiently solve large planning problems containing the features we are concerned with. However, we are reporting a range of results, and making the test sets available [Bacchus, 2001] so as to provide an experimental base for future work.

In the logistic domain there are a collection of packages that need to be transported to their final destination, trucks for moving packages between points in a city, and planes for moving packages between airports located in different cities. Packages can be loaded and unloaded from vehicles and the vehicles can be moved between compatible locations.

[Wolfman & Weld, 1999] added a fuel capacity for each vehicle, and a refueling action that fills up a vehicle given that the vehicle is located at a depot. In addition, each `drive-truck` operator consumes a fixed amount of fuel, each `fly-airplane` operator consumes an amount of fuel based on the (fixed) fuel efficiency of planes and the distance between the two airports, and one is not allowed to move a vehicle to a location unless it contains enough fuel to get there.

We take the TLPLAN approach to planning in which domain specific information is declaratively encoded in a temporal logic. Unlike standard heuristics which try to measure the worth of a state, TLPLAN typically uses negative information that tells it that certain kinds of action sequences are flawed [Kibler & Morris, 1981]. This information is checked against the sequences generated during forward chaining, and any sequence satisfying a bad property is pruned from the search space. This approach to planning has proved to be extremely successful: in yields a level of planning performance that is an order of magnitude better than any other approach, and the approach has been applied to a wide range of different domains. In [Bacchus & Kabanza, 2000] an extensive set of examples and empirical results are presented to demonstrate both the performance of this approach and the fact that the requisite knowledge for many different planning domains is easily obtained and represented in the formalism.

In the standard logistic world the control information needed is very simple.

1. Don't move a vehicle to a location unless it needs to go there to pickup or drop off a package.
2. Don't move a vehicle from a location while it still contains a package that needs to be dropped off at that location.
3. A package needs to be picked up by a truck if it needs to be moved to another destination in the same city.

4. A package needs to be picked up by a plane if it needs to be moved to another city.
5. A package needs to be dropped off from a truck if the truck is at its goal destination or if the truck is at an airport and its goal destination is in another city.
6. A package needs to be dropped off from a plane if the plane is in the package's destination city.

Each of these assertions can be easily encoded as a temporal logic formula [Bacchus & Kabanza, 2000], and with this collection of assertions the planner finds plans very efficiently. Furthermore, this control knowledge is of such a simple form that it becomes possible, by looking at each action's effects, to predict whether or not an action will extend the current plan in such a way as to violate one of these assertions. As a result one can systematically convert the control rules into extra action preconditions [Bacchus & Ady, 1999]. This has the effect of blocking an action first, rather than executing it, generating the plan extension, and then determining that the extension is invalid. Due to the extremely high branching factor in larger logistic problems (over a 1,000 applicable actions in each state on the harder problems), this "compiled one-step" look ahead improves planning performance by a couple orders of magnitude. In our experiments, we used a precondition encoding of these control rules.

We used two sets of test problems. A set of four small problems, loga, logb, logc, and logd, utilized by Wolfman in testing his LPSAT system, and a collection of 30 much larger problems used in the AIPS98 planning competition.[6] All of the experiments were run on a 500MHz PIII machine with 512MB of memory. All times are reported in CPU seconds.

In Table 1 the first set of columns gives the time it takes the current version of TLPLAN to solve the original version of these problems, and the number of steps in the resulting plan. We then encoded Wolfman's metric version, with fuel consumption, and ran the problems again. We used Wolfman's loga–logd problems directly, and for the AIPS98 suite we added distances between the locations, fuel consumption rates and fuel tank capacities for the planes and trucks. We found that TLPLAN could solve these metric logistics problems very efficiently with the same control knowledge as used in the standard logistics world, along with the extra information

1. Allow a vehicle to move to a depot if it needs fuel.
2. Don't refuel a vehicle unless it needs more fuel to make a pickup or drop off.
3. Don't move a truck or plane to a location in order to pickup an object if there is already exists a similar vehicle at that location with sufficient fuel capacity to take it to its destination.

The times required to solve the fuel version of the logistic problems are shown in the second set of columns of the table. The data shows that our approach finds the metric problems not that much more difficult than the standard problems.

[6]This test suite is not to be confused with the 30 problem ATT logistics suite which are much easier.

[Wolfman & Weld, 1999] present a SAT encoding approach to solving these metric logistic problems. Their planner utilizes a combination of SAT solving and linear programming: the SAT solver finds the plan while the linear program solver ensures that the plan satisfies the (linear) metric constraints. The solution times they report are approximately 10 sec. for loga, 300 sec. for logb, 500 sec. for logc and 5000 sec. for logd (they also point out that their approach was faster than previous approaches). These times indicate that their approach scales poorly. Interestingly, in examining the plans their system generated[7] it was found that these plans used much more fuel, e.g., 6426.67 units for the loga solution, and also contained many unnecessary moves, e.g., moving a truck back and forth without using it to transport any packages. Their system does not use domain specific control knowledge but some of the knowledge used here might be useful in improving the performance of their system.

Then, we took the metric logistics domain and made it concurrent, so that the domain contained both metric quantities and concurrent actions. In the concurrent version loads and unloads take 1 unit of time to complete, and multiple concurrent loads/unloads into the same vehicle are allowed (the same object cannot be manipulated concurrently). Refueling a truck takes 1 unit of time, and an airplane takes 10 units. Finally, driving a truck takes 5 units of time, and flying an airplane takes time that is dependent on the distance between the two locations. To the above control knowledge we added the extra information

1. A vehicle can be moved to a location if there is an object en route to that location (in a different type of vehicle) that can be transported by the vehicle.

This allows the planner to get the vehicles moving so that they can make progress towards the pickup location concurrently with the object they are to pickup. For example, the planner can start flying a plane to an airport while a truck is transporting an object to the airport that needs to be transported to another city. This decreases the duration of the plan. The last set of columns shows our results for this domain. In this case we show the duration of the plan as well as the length of the plan (number of actions). Since the actions take at least 1 unit of time, it can be seen that highly concurrent plans are being found. The time to find a solution has also climbed, but not by much, and so has fuel consumption. This makes sense, as the concurrent plan will try to utilize more vehicles in order to maximize concurrency, and more vehicles means more fuel.

Another planning system that is capable of dealing with concurrent actions of different durations is the TGP system [Smith & Weld, 1999] that is based on GraphPlan. However, its underlying algorithms are considerably more complex than the approach we suggest here, and it cannot deal with metric quantities. We were able to run the TGP system on some simpler logistic problems involving varying action durations but without fuel consumption. We found that TGP could not solve any of loga–logd problems (when we removed the fuel consumption component) even when given an hour of CPU time. These results and the performance of Wolfman's LPSAT system, demonstrate that adding domain

[7]Thanks to Steve Wolfman for supplying us with these solutions.

Problem	Standard		Metric Fuel			Fuel+Concurrent			
	CPU	Len.	CPU	Len.	Fuel	CPU	Dur.	Len.	Fuel
loga	0.02	51	0.06	60	2558	0.06	35.00	84	2518
logb	0.10	42	0.06	49	1392	0.08	35.25	90	2425
logc	0.02	51	0.08	60	3158	0.09	42.75	92	3158
logd	0.07	70	0.15	80	4384	0.19	67.25	154	4960
x-1	0.01	26	0.03	26	846	0.07	24.50	53	1994
x-2	0.03	33	0.11	33	2005	0.18	18.00	62	2187
x-3	0.15	55	0.39	55	3364	0.54	26.75	94	3963
x-4	0.22	59	0.53	59	3694	1.17	28.50	114	4562
x-5	0.02	22	0.03	22	1442	0.06	23.00	39	1825
x-6	0.33	72	0.83	74	5549	1.30	38.50	118	5886
x-7	0.04	34	0.22	34	2356	0.31	21.75	64	2855
x-8	0.16	41	0.77	41	3941	1.23	34.25	85	5571
x-9	0.41	85	1.17	85	3923	2.14	28.50	152	5595
x-10	0.50	105	1.77	106	11285	1.69	60.50	184	14554
x-11	0.05	31	0.12	32	859	0.20	16.75	49	1062
x-12	0.35	41	1.06	41	3415	3.16	25.50	73	3528
x-13	0.68	67	3.20	68	6736	2.95	27.25	110	8235
x-14	0.37	94	2.27	94	4641	1.87	24.00	142	5774
x-15	0.12	94	0.48	97	1406	0.43	47.75	151	1786
x-16	0.26	58	1.07	58	1937	2.01	29.00	93	2319
x-17	0.08	45	0.39	45	945	0.72	21.75	76	1497
x-18	3.23	170	10.24	174	15914	7.12	43.50	275	24111
x-19	2.24	153	5.17	159	4373	7.40	48.25	270	7229
x-20	2.34	150	7.35	156	7527	7.02	56.00	226	7999
x-21	1.52	104	4.03	105	4621	9.05	37.00	184	5803
x-22	17.95	296	34.75	305	25121	33.87	79.00	498	35278
x-23	0.25	115	1.20	115	3796	1.08	28.75	183	5728
x-24	0.30	41	1.40	41	1356	7.99	26.00	77	1759
x-25	7.66	190	16.77	196	14209	16.65	50.00	284	19203
x-26	4.89	194	11.65	203	39616	17.40	241.50	361	44291
x-27	2.90	149	16.44	155	7041	14.03	54.00	263	11293
x-28	26.10	274	63.86	283	13855	197.38	58.75	478	21899
x-29	20.36	330	43.08	339	26236	22.03	58.75	531	44308
x-30	4.60	136	10.57	139	10648	63.97	55.00	265	15265

Table 1: Test results on versions of Logistics

specific control knowledge and utilizing our forward chaining approach allows us to move to another level of planning performance.

There are two other planning systems that are quite similar to ours, [Pirri & Reiter, 2000] and [Kvarnström, Doherty, & Haslum, 2000]. Both of these planner utilize the TLPLAN approach and display good performance. However, both of these systems utilize a rich logical representation for actions and states, whereas the approach we present here can be utilized by any forward chaining planner with the much simpler STRIPS/ADL action representation.

Finally, temporal refinement planners are an alternate approach to planning with concurrent actions. The IxTeT planner is an impressive system capable of dealing with resources and concurrent actions [Ghallab & Laruelle, 1994]. NASA's remote agent project RAX also utilized a refinement planner. Temporal refinement planners operate by taking an initial plan that typically specifies the initial state and various goal conditions, and refining that plan by adding additional actions to achieve open conditions or constraints to protect other conditions. A key component of these planners is the use of constraint propagation to maintain the temporal constraints imposed on the plan during the refinement process. Thus these planner search in a space of partially specified plans using constraint propagation to detect deadends, rather than in a space of fully specified worlds as in our approach. The RAX planner also utilized extensive search control knowledge in order to achieve its good level of performance. However, the control knowledge was at a much lower level and was more procedural in style than that utilized by our approach. Many of the standard concurrency control paradigms are easier to specify with a temporal refinement planner than in our approach. However, it should be possible to develop macros for our approach to encapsulate many of these paradigms thus easing the specification problem.

5 Conclusion

Forward chaining's ability to deal with metric quantities was already documented, but in this paper we have demonstrated that there is also a simple way of extending forward chaining to deal with concurrent actions. These two features can then be combined with other ideas like search control knowledge to yield a powerful approach to planning in the presence of concurrent actions of differing durations and resources.

Our empirical results show that complex and lengthy plans can be generated with our approach, and serve to demonstrate the potential of our approach. Further verifying that potential is the subject of future work.

Other items of future work include (a) higher level constructs for concurrency control implemented as macros that can are expanded to the very general lower level constructs already supported by our system, and (b) access by actions to the event queue. This last is worth further explanation. Consider a situation where a truck is being driven from location A to location B. At the start of the drive its arrival at location B is entered into the event queue. Now it could be that location B only has capacity for one truck, thus a subsequent action should not be scheduled that would cause another truck to arrive at location B at the same time. Checking this "precondition" involves querying the event queue. A different example is when via some other action or event the truck gets a flat tire en route. This will delay its arrival time. Thus the "flat-tire" event must not only change the current state of the world but it must also alter events in the event queue. Simply put actions must be able to treat the event queue just like the state's database: they must be able to query and update it. With this ability it becomes easier to model on-going processes that can be interrupted and restarted, something that is cumbersome in our current model.

Acknowledgments The referees provided some very useful suggestions that helped improve the paper.

References

[AIPS, 2000] AIPS 2000. Artificial Intelligence Planning & Scheduling 2000 planning competition. http://www.cs.toronto.edu/aips2000/

[Bacchus & Ady, 1999] Bacchus, F., and Ady, M. 1999. Precondition control. available at http://www.cs.toronto.edu/~fbacchus/on-line.html

[Bacchus & Kabanza, 1998] Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annuals of Mathematics and Artificial Intelligence* 22:5–27.

[Bacchus & Kabanza, 2000] Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

[Bacchus, 2001] Bacchus, F. 2001. On line experimental data sets. http://www.cs.toronto.edu/~fbacchus/tlplan.html

[Doherty & Kvarnström, 1999] Doherty, P., and Kvarnström, J. 1999. Talplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *Proceedings of TIME '99, IEEE Computer Society*, 47–54.

[Geffner, 2000] Geffner, H. 2000. Functional strips: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. in press.

[Ghallab & Laruelle, 1994] Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proceedings of the International Conference on Artificial Intelligence Planning*, 61–67. AAAI Press.

[Halpern & Vardi, 1991] Halpern, J. Y., and Vardi, M. Y. 1991. Model checking vs. theorem proving: a manifesto. In Allen, J. A.; Fikes, R.; and Sandewall, E., eds., *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. San Mateo, CA: Morgan Kaufmann, San Mateo, California. 325–334.

[Hoffmann, 2000] Hoffmann, J. 2000. Fast-forward. http://www.informatik.uni-freiburg.de/~hoffmann/ff.html.

[Jónsson *et al.*, 2000] Jónsson, A. K.; Morris, P. H.; Muschettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of the International Conference on Artificial Intelligence Planning*, 177–186. AAAI Press.

[Kibler & Morris, 1981] Kibler, D., and Morris, P. 1981. Don't be stupid. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 345–347.

[Kvarnström, Doherty, & Haslum, 2000] Kvarnström, J.; Doherty, P.; and Haslum, P. 2000. Extending TALplanner with concurrency and resources. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*.

[Pednault, 1989] Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 324–332.

[Pirri & Reiter, 2000] Pirri, F., and Reiter, R. 2000. Planning with natural actions in the situation calculus. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Press. in press.

[Smith & Weld, 1999] Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 326–337.

[Wolfman & Weld, 1999] Wolfman, S. A., and Weld, D. S. 1999. The lpsat engine and its application to resource planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 310–317.