Exploiting a Graphplan Framework in Temporal Planning

Derek Long and Maria Fox
University of Durham, UK
D.P.Long@dur.ac.uk,Maria.Fox@dur.ac.uk

Abstract

Graphplan (Blum & Furst 1995) has proved a popular and successful basis for a succession of extensions. An extension to handle temporal planning is a natural one to consider, because of the seductively time-like structure of the layers in the plan graph. TGP (Smith & Weld 1999) and TPSys (Garrido, Onaindía, & Barber 2001; Garrido, Fox, & Long 2002) are both examples of temporal planners that have exploited the Graphplan foundation. However, both of these systems (including both versions of TPSys) exploit the graph to represent a uniform flow of time. In this paper we describe an alternative approach, in which the graph is used to represent the *purely logical* structuring of the plan, with temporal constraints being managed separately (although not independently). The approach uses a linear constraint solver to ensure that temporal durations are correctly respected. The resulting planner offers an interesting alternative to the other approaches, offering an important extension in expressive power.

Introduction

Graphplan (Blum & Furst 1995) has proved an influential planning system, even if its performance has been superceded for many of the classical benchmark problems. Amongst its many adaptations is one of the first domain-independent planners (not using hand-coded control knowledge) to manage temporal planning, TGP (Smith & Weld 1999). In that system, the structure of the graph was exploited to represent the flow of time and durations were attached to *actions*, using a very strong constraint on interactions between concurrent actions. In particular, any proposition that is changed by the action is effectively locked to that action for the interval of its execution. This means that it is impossible to express actions in which the desired effect is something that is true precisely for the duration of the action. For example, if one needs to cross the basement and replace a fuse, then the light to achieve these actions might be provided by striking matches. The light begins when the match is struck and ends after the duration of the match burning action. Potential for this kind of concurrency is captured in PDDL2.1 (Fox & Long 2002). Durative actions in that model use the end points of durative actions to capture the behaviour that occurs in the interval of the durative action, allowing more sophisticated exploitation of concurrent activity, including the consultation of propositions that are changed by durative actions. This model is founded on an underlying principle that duration is attached to state and that transitions between logical values of state-describing propositions should be instantaneous at the level of the model, since otherwise it is necessary to complicate the model of state updates by introducing a third value representing *undefined* which must be used for propositions that are in flux. It was, in effect, to handle the consequences of a conservative model of undefined values that Smith and Weld (Smith & Weld 1999) introduced such a tightly constrained mutual exclusion relation between concurrent actions, preventing any activities from attempting a concurrent access of a proposition that might be undefined because of the transitional activity in a parallel durative action.

In this paper we propose a different approach to the treatment of time in a Graphplan framework, resting on the PDDL2.1 model of durative actions. The principle idea is to use the layers of the graph to represent not the flow of time, but the logical structure of the plan. That is, each layer in the graph will represent the occurrence of interesting activity, and never simply the passage of time. This is an important shift, since TGP is required to model the flow of time in the graph structure in a different way: during graph construction, the lengths of durative actions can take any relative values without causing any difficulties, but during search there is a problem in identifying how far back to step from one goal set to another when using a *noop* to achieve a goal. The solution is to adopt a specific size for the increment represented by a *noop*. The size of this increment can, of course, be selected to minimize the number of goal layers that need to be considered during search (using the GCD of the lengths of the actions in the domain), but this is very likely to prevent a graph from being successfully searched if the longest action, or the required plan length, is a large multiple of this time increment. For example, if the longest action has a duration a thousand times longer than the shortest action, then a plan that involves executing just one instance of each of the longest and shortest actions could require a search through a thousand layers, which is simply impractical for current Graphplan planners. In contrast, the *logical* structure of the plan consists of just three happenings: the start of the two

actions, the end of the shortest action and, finally, the end of the longest action.

The objectives of this work are:

1. to modify Graphplan to perform temporal planning in a language that allows both initial and final effects of a durative action to be exploited in a plan;
2. to construct an architecture that is as simple an extension or modification of Graphplan as possible, since Graphplan is a widely and well-understood foundation, making this approach more accessible;
3. to demonstrate the relationship between mutex relations in Graphplan and the interactions between concurrent durative actions.

We proceed to describe the way in which a PDDL2.1 domain is converted into a form that allows Graphplan planning to be used. Several constraints must be satisfied by the domain for this transformation to be possible and they are described. We then go on to describe the modifications to the Graphplan algorithm that allow correct temporal planning behaviour to be achieved and the mechanisms by which the temporal durations of states are introduced into the planning structure. We present some results and discuss the shortcomings of the approach. Finally, we make some observations about the relationship between the Graphplan model of concurrency and the PDDL2.1 model of concurrency.

Treatment of PDDL2.1 domains

In this section we discuss the preprocessing of a PDDL2.1 durative actions domain model to achieve a form that can be used for Graphplan planning. The core idea is to translate durative actions into standard simple actions that can be used to simulate the original semantics of the durative-actions.

PDDL2.1 extends PDDL in several important ways. In this paper we consider only the temporal extension. The treatment of numeric values has been explored in the Graphplan framework (Koehler 1998), but we have not considered it further in this work. PDDL2.1 offers the opportunity to use different kinds of durative actions: the simplest are those in which the durations are fixed, possibly as a function of the parameters of the action. Since we are not considering numeric values in any other context, we can only allow durations that are a fixed function of the parameters, defined in the initial state. Thus, for example, a fly action that takes time that is a function of the start and end of the flight is possible, but we cannot capture fuel use or flight time that is a function of how much fuel the plane is carrying. PDDL2.1 represents durative actions by describing the transitions that occur at the end points of the interval of activity, using an essentially classical pre- and post-condition model of these transitions, together with a collection of invariant conditions that must hold over the duration of the action. The invariants are an essential element that bind together the start and end points of the actions into a coherent durative activity. The other element is the duration itself, which governs the separation of the start and end points.

In the following discussion, we are motivated by the intention to modify the underlying Graphplan behaviour as little as possible. This offers several possible advantages, including that the wide array of Graphplan extensions and algorithmic improvements that have been considered could be applicable to the temporal form. There is a trade-off to be considered. On one hand it is possible to use mechanisms in a carefully constructed domain encoding that will force a standard Graphplan system to simulate our intended semantics of durative actions. On the other hand, some mechanisms cause such a deterioration in Graphplan behaviour that it is impractical to avoid making modifications to the algorithm to achieve the intended behaviour.

A straightforward conversion of PDDL2.1 actions into actions that can be used by a standard Graphplan planner is to create the simple actions representing the end points of the durative actions. This is a good starting point, although we shall see that there are some complications that must be addressed. The first of these is that we want to ensure that the start and end point actions are always managed as a pair. To achieve this, we add a new effect to the start action that is required by, and deleted by, the end action. Thus, the end action cannot be executed without also executing the start action. The converse case raises an interesting question about the nature of plans: if a plan contains the start of a durative action, without seeing it through to successful completion, should it be considered a valid plan? This question is discussed in detail elsewhere (Long & Fox 2001), but in the current work we adopt the view that durative actions carry an intention to complete them, and it is therefore inappropriate to construct a plan exploiting the initial effects of a durative action without confirming that the obligations for successful completion can be met. Therefore, if a plan contains the start action it must also contain the end action. This is a constraint that cannot be imposed using pre- and post-conditions of the actions alone. It can be achieved by adding a delete effect to the start action that is reachieved by the end action, and then adding this proposition to the initial state of the problem and to the goal state, for every action instance. Thus, use of the start action will prevent the goal from being reached unless the end action is used to restore the equilibrium. This mechanism is a relatively expensive one to use, since it requires a large number of new artificial facts and also creates a large number of artificial new choice points. The alternative is to prevent start actions from being used to achieve goals unless their corresponding end action has already been inserted into the plan (while searching backwards through the plan graph).

A subtle but deliberate choice in the semantics of PDDL2.1 requires that invariant conditions hold in the *open* interval defined by the end points of a durative action. In fact, the invariant is checked between each pair of happenings executed in the plan within the interval of the durative action. A happening is a collection of (instantaneous) actions (or end points of durative actions) executed at the same time. If we model durative actions with only the pair of end point actions then the invariant is effectively ignored. To correctly account for the invariant we introduce a new action with the invariant as its precondition. In order to force this action to sit between the end points of the durative action from which it is derived, we give the action a precondition

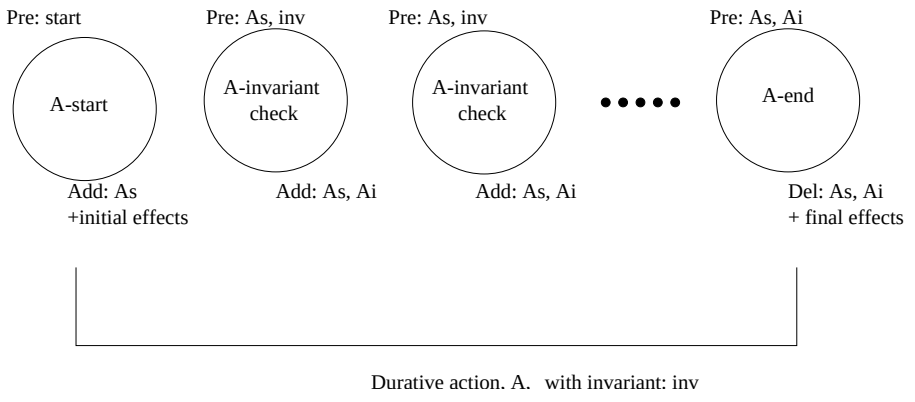

Figure 1: Modelling a durative action with a collection of simple instantaneous actions.

achieved by the start action and an effect required as precondition by the end action. However, there remains a problem: we want it to be possible for multiple happenings to occur in the interval between the end points, and in that case the invariant should be rechecked following each such happening. This means that we must force the invariant checking action to be reapplied at each layer in the graph between the layer containing the start action and the layer containing the corresponding end action. Unfortunately, Graphplan will attempt to exploit *noops* to make the effect of the invariant action persist until the end point at which it is required, or the effect of the start action persist until the invariant action requires it, placing a single instance of the invariant checking action at whichever intermediate layer is least inconvenient. To prevent this we require two mechanisms, one being a modification of the Graphplan machinery itself and the other being an addition to the domain encoding. The latter is the requirement to add an additional effect to the invariant checking action, which is the special proposition achieved by the start action and used as a precondition of the invariant-checking action itself. It can be seen that this action then behaves like a *noop* with additional preconditions — the invariant conditions of the durative action to which it corresponds. The modification in the Graphplan engine is not to generate the standard *noop* for either the special effect of the invariant-checking action or for the effect of the start action that acts as precondition for the invariant-checking action and the end action.

The way in which this collection of actions now fits together to model the enactment of a durative action can be seen in Figure 1. An example of the actions generated for a durative action from PDDL2.1 can be seen in Figure 2.

The last requirement to harness Graphplan to our purpose is a means to communicate the durations of durative actions. This is really an implementation detail, but our solution is to separate the duration values out of the initial state and put them into a separate file, while adding a special duration field to each of the start and end point actions. The transformed domain is therefore in a pseudo-PDDL syntax, representing a collection of almost classical STRIPS PDDL actions, but for this additional information. The transformation can be performed automatically from a PDDL2.1 input,

```
(:durative-action debark
 :parameters (?p - person ?a - aircraft ?c - city)
 :duration (= ?duration debarking-time)
 :condition (and (at start (in ?p ?a))
                 (over all (at ?a ?c)))
 :effect (and (at start (not (in ?p ?a)))
              (at end (at ?p ?c))))

(:action debark-start
 :parameters (?p -person ?a -aircraft ?c -city)
 :duration (debarking-time)
 :precondition (in ?p ?a)
 :effect (and (not (in ?p ?a))
              (debarking-inv ?p ?a ?c)))

(:action debark-inv
 :parameters (?p -person ?a -aircraft ?c -city)
 :precondition (and (debarking-inv ?p ?a ?c)
                    (at ?a ?c))
 :effect (and (idebarking-inv ?p ?a ?c)
              (debarking-inv ?p ?a ?c)))

(:action debark-end
 :parameters (?p -person ?a -aircraft ?c -city)
 :duration (debarking-time)
 :precondition (idebarking-inv ?p ?a ?c)
 :effect (and (not (idebarking-inv ?p ?a ?c))
              (not (debarking-inv ?p ?a ?c))
              (at ?p ?c)))
```

Figure 2: The result of converting a PDDL2.1 durative action (at the top) into linked instantaneous actions. Note the introduction of the duration field in both the start and end actions.

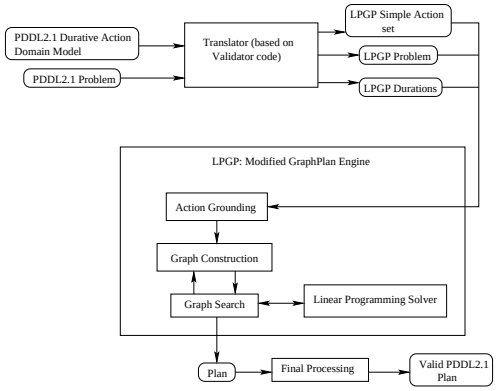

Figure 3: The architecture of the temporal Graphplan variant, LPGP.

so should not be seen as introducing a new language, but simply as a compilation into an internal representation format.

An important question arises in determining the treatment of start actions as possible achievers. When an *end* action is used to achieve a goal the corresponding start action will be forced into the plan in order to satisfy the preconditions of the end action. On the other hand, if a *start* action could satisfy a goal then the corresponding end action *should already have been placed in the plan*. This organisation follows from the backward sweep search that is used to construct a plan in Graphplan. Unfortunately, it is very difficult to return to a previously visited layer in the search and insert additional actions, so using a start action to achieve a goal is very problematic. This problem does not arise in TGP because the action representation precludes durative actions achieving anything at the start of their execution. A further encoding can be used to handle this situation. We add a special effect to the end action that also appears as a goal and in the initial state. This means that the goal can be achieved by *noop*s from the initial state, or it can be achieved using the end action encoding the durative action. The consequence of this construction is to allow the search to insert the end action into the plan at any layer in the search. In addition, we ensure that start actions can only be selected if their corresponding end point is already in the plan. The special proposition acts as a catalyst to allow insertion of the end action when the release of the start action is necessary for the construction of a plan. It is possible to use some analysis to determine when the start effects of an action could be required to achieve the goals, based on the reachability within the plan graph. This allows us to limit use of this construction to cases where it might play a useful role.

The process by which our Graphplan variant, which we call LPGP (Linear Programming and Graph Plan), plans with a PDDL2.1 domain is shown in Figure 3. The durations file produced in the translation conveniently separates the information required for managing the durative actions from the initial state information required for the usual grounding of actions.

Modelling Time Flow in the Planning Graph

In TGP and TPSys, the plan graph is adapted to perform temporal planning by attaching duration to actions, where actions are represented by a single structure as with the original Graphplan. The disadvantage of this approach is that the layers containing actions can then no longer be treated as uniform, since they can contain actions of different durations. As a result, the structure of the graph is complex to construct and to search.

In the LPGP planner we invert the way in which time is attached to states and actions and we do not use the graph layers to measure time in uniform increments. In TGP and TPSys states are instantaneous, while time flow is attached to the actions. Actions can span several layers of the graph between the point at which their preconditions must be achieved and the end point at which they have their effects. TGP actions do not have initial effects and actions are mutex with any other actions that might attempt to access the propositions used or changed by them. This is a strong mutex relationship, and prevents any attempt to model, for example, executing an action to wash ones hands while a sink is being filled — the sink-filling action must end before the water is accessible.

During search, TGP must attach a length with *noop* actions. To ensure that an optimal length plan is found, this must be set to be the GCD of the lengths of actions in the domain, which can result in an expensive search process. However, this technique has an important benefit which is that the optimality of the plan length in terms of graph layers searched is equivalent to the optimality in terms of execution time. However, the price is very high: any plan that has a long execution time relative to the lengths of any of its actions will require a very large graph structure to be searched, even if the plan requires relatively few actions.

In LPGP we attach duration to states, so each fact layer is associated with a duration. The layers are used only to capture the points at which events occur within the execution trace of the plan, rather than uniform passage of time. By separating the graph structure from the flow of time in this way we gain the benefit that plans with few events only require short graph structures. However, it is not always true that a plan that requires fewest distinct points of activity will be the shortest in duration. For example, if two goals can be achieved by the parallel execution of actions A and B, with durations 3 and 5 time units respectively, or by the single action C with duration 100 units, the plan in which A and B are used will require more distinct levels of activity (the simultaneous start of A and B, the end of A and then the end of B) than the plan using C alone. In fact, this example is slightly simplified because of the need to insert the special actions to check invariants. The complete plan structure is illustrated in Figure 4.

With this interpretation of the relationship between the plan graph and the flow of time in mind, we now consider the modifications to the underlying Graphplan algorithms to support temporal planning in LPGP.

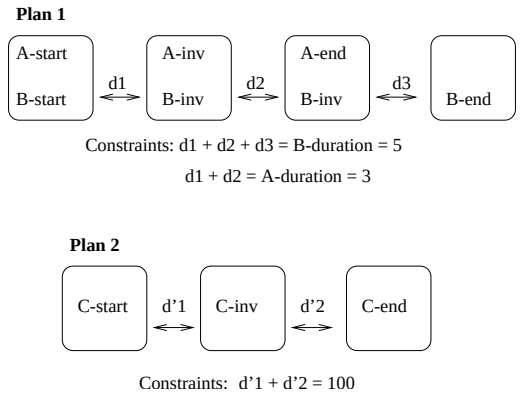

Figure 4: Two alternative plan structures showing how a temporally longer plan can have a simpler activity structure, being represented in fewer plan graph layers.

Modification to Graphplan

The architecture of Graphplan can be divided into three components: the grounding of actions, the construction of the plan graph and the subsequent search for a plan. In practice, of course, graph construction and search are interleaved (leading to an iterated depth-first search), but it is convenient to consider these three components separately in order to explain how LPGP modifies the original Graphplan algorithm.

Action Grounding

During grounding of actions we prune out start and end actions if their duration field is undefined in the durations file. These actions are not legally applicable. Also during the instantiation phase we attach the correct duration values to the instances of the start and end point actions, looked up from the values to be found in the initial state. The final modification in the instantiation phase is to mark actions with their type (derived from an examination of the suffix in their names): start, end and invariant-checking actions. Only one instance of each of these actions is required (including the invariant-checking action), even though multiple instances might be applied in a plan. Where a durative action spans multiple layers, requiring several instances of the invariant-checking action to be applied, the instances are all copies of the same action instantiation, each attached to its own layer of the graph.

Thus, the only modification to a standard Graphplan algorithm required during the grounding phase is the addition of extra information to the ground action structures that can be inferred from the names of the actions.

Graph Construction

The graph construction phase is modified in two ways. Firstly, a stronger mutex relation is enforced between actions than the classical Graphplan mutex. In Graphplan, two actions are permanently mutex if the delete effects of one intersect with the preconditions or add effects of the other. In PDDL2.1 there is a stronger requirement to ensure that actions do not interfere with one another, called the "no moving targets" rule. This insists that two actions cannot be executed concurrently if they add the same effect or delete the same effect, in addition to the possible sources of interference identified in Graphplan. We discuss the implications of this in the context of a temporal plan in Section . The second modification is that no *noops* are constructed for the facts that have an `-inv` suffix. This forces the invariant checking actions to be used to propagate these facts between layers, ensuring that the invariants are checked as the propagation is carried out.

Graph search

This phase of the Graphplan algorithm is the one most affected by the introduction of time. The original Graphplan search algorithm is shown in Figure 5, annotated with the modifications required to support LPGP. Underlying the modification is the introduction of a linear programming problem, with constraints derived from the durations of actions. This problem can be maintained during the search so that, at all times, a solution to the linear constraints will provide a consistent allocation of durations to the fragment of the plan that has so far been constructed. If the problem is ever unsolvable then the plan is invalid and search must backtrack.

When an end action is selected to act as the achiever for a goal fact we introduce a temporal constraint. This constraint will assert that the total duration of the fact layers between the start and end actions of the durative action must equal the duration of the action. However, when the end action is first introduced we cannot yet know when the start action will appear. Therefore, the constraint is initially an assertion that the layers between the current layer and the layer containing the end action must have total duration less than the duration of the associated action. The two forms of constraints, then, are simply linear constraints on the durations of the fact layers and the equations must be solved for these durations, minimizing the total duration of the plan. This means that it is possible to use a linear programming algorithm (such as the simplex algorithm) to solve the equations. The form of the constraints for such a solver is best given as a matrix of the coefficients for the linear combinations of the variables (which are the durations attached to the fact layers). The matrix contains as many columns as there are fact layers in the graph and as many rows as there are durative actions in the (current) plan. New columns are added to the matrix as the graph is extended, prior to searching from each new layer. As an end action is introduced into the plan a new row is added to the matrix. When an invariant-checking action is added, the column denoting the fact layer succeeding the action layer containing the invariant check is set to 1 in the row corresponding to the end action coupled to this invariant check. When a start action is introduced, the correct entry is set to 1, just as for the invariant check, but also the constraint is switched from an inequality to an equality. Backtracking through the choice of any of these action types causes the exact reversal of these activities, resetting matrix entries to 0 where they were set to 1. The matrix associated with a simple example developing plan structure is shown in Figure 6.

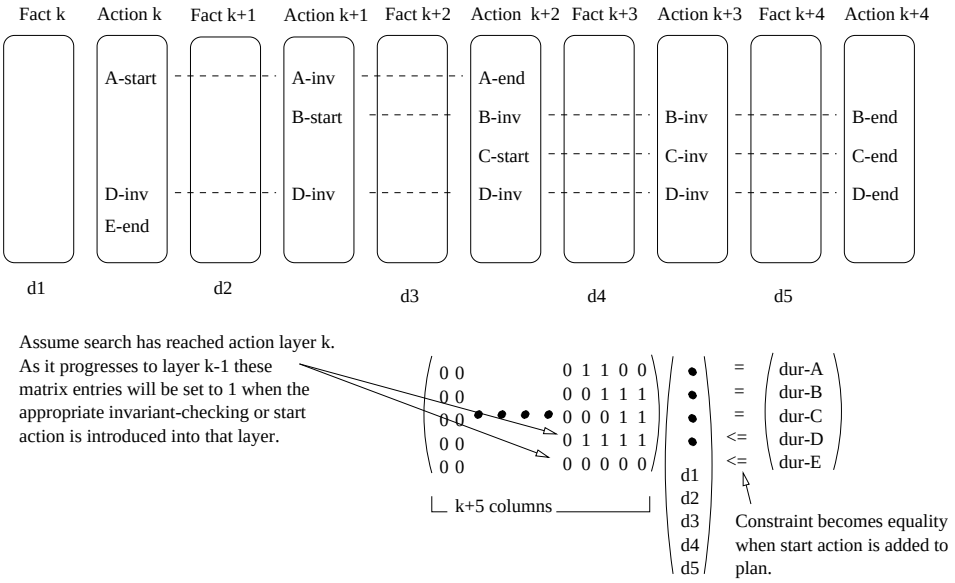

Figure 6: The matrix of constraints associated with an example partially complete graph search.

An important decision is when to check the equations. One possibility would be to check them whenever the matrix is modified. However, the inequality constraints are typically less difficult to satisfy than the equality constraints, so we choose to carry out checks only when start actions are added to the plan, which convert inequality constraints to equality constraints. This has the benefit of reducing the number of calls to the linear constraint solver, but the cost of not always discovering that the equations are unsolvable until several choices after the point of failure. Other schemes would be possible, such as checking the constraints at each layer in order to avoid developing bad choices into the next layer.

This approach is similar to that taken in Zeno (Penberthy & Weld 1994), but in that planner the underlying architecture was a partial order planner. In the Graphplan framework we gain all of the benefits that have been associated with Graphplan in comparison with partial order planners (and all of the weaknesses), and we are able to construct a complete collection of constraints at all points in the planning process. In contrast, Zeno was unable to invoke the numeric constraint solver until the constraints were properly instantiated, preventing it from identifying flawed plans as early as might be hoped. Of course, Zeno was handling a richer language, including numeric effects, presenting a harder problem than a treatment of duration constraints alone.

An interesting additional factor in our treatment is connected to an important consequence of the mutex relationships that we use to govern the validity of plans in PDDL2.1. The "no moving targets" rule forbids actions from being executed simultaneously if they could possibly interfere with one another's pre- or post-conditions. Therefore, actions that do interact must be separated by a small, but non-zero, interval. Typically, the only constraint we have to satisfy is that the duration of separation must be positive. This gives rise to the need to introduce very small values into a plan. In the validation of plans, as we discuss elsewhere (Long & Fox 2001), we introduce a small constant that dictates the minimum degree of separation allowed between actions, in order to avoid the problem that one plan might be judged better than another simply because it used smaller separations than the second, possibly otherwise identical, plan. This bound must be introduced into the equations we construct as a lower bound on the values of the variables (the fact layer durations). It will be observed that the first fact layer can never be constrained by any constraint other than this lower bound, since it can never appear *between* a start and end action. This leads the constraint solver to assign the minimum duration to the first fact layer in every case, which is a direct reflection of the decision in the semantics of PDDL2.1 to begin the initial state at time 0, while insisting that states are always associated with intervals that are half-open on the right. That decision prevents the first actions in a plan from being executed at 0 and forces them to begin at a small, non-zero time after 0. The value of the small, non-zero time that is used is selected by the programmer in the current implementation (we set it at 0.001), but it would be easy to set if from the command line, or, as we discuss in (Long & Fox 2001), from a value communicated in a problem description.

Results

We implemented the architecture as shown in Figure 3. The Graphplan implementation we used as our foundation is an older version of STAN. We did not use the newer implementations because they introduce other mechanisms that interact with the temporal planning machinery in ways that have yet to be explored. The implementation was designed to be as convenient a modification of the Graphplan system as possible, rather than working for a highly optimised implementation and we do not make any claims for great effi-

```
boolean find-plan(GoalSet G,Level n)
// Finish when reach initial state.
if(n == 0) return true;
if(G is empty)
then Set H = preconditions of active actions at level n;
// Recursively satisfy goals at next level
     extend M with an extra column;
     return find-plan(H,n-1);
else select goal g in G;
// There are no noops for inv conditions
     if(g has a noop at level n)
     then make noop active at level n;
          remove g from G;
          if(find-plan(G,n)) return true;
          deactivate noop for g at level n;
     for each achiever, a, of g at level n do;
         if(a is not mutex with active actions at n)
         then if(a is an end action)
              then add a new row to M;
                   set row bound to duration of a;
                   set row constraint to ≤;
                   set M[a,n] = 1;
              if(a is an invariant action)
              then set M[a,n] = 1;
              if(a is a start action)
              then set M[a,n] = 1;
                   set row constraint for a to be =;
                   if(not solve(M))
                   then deactivate a;
                        set constraint row a to ≤;
                        restart loop with next a;
              activate a at level n;
              if(find-plan(G,n)) return true;
              deactivate a at level n;
              if(a is start, invariant or end)
              then reverse modifications to M;
     loop;
return false;
```

Figure 5: The original Graphplan search algorithm and *in italics* the modifications required for LPGP. M is the matrix of LP constraints. We use M[a,n] for the entry in M at the row for the durative action from which a is derived and the column for layer n.

ciency.

To solve the linear constraints we used the `lp_solve` library originally developed by Michel Berkelaar (Berkelaar 2000). This we connected as a library to the LPGP code, and used its API to manage the constraint matrix. This solver attempts to solve modified problems from the same basis that solved the problem before the modification, which is an excellent strategy in the context of our exploitation: most often new constraints do not have a dramatic impact on the constraint solution, since most durative actions span few fact layers.

We compared the performance of LPGP with planners competing in the 3rd International Planning Competition. The results of most interest are summarised in Figure 7.

The results suggest the following interpretation: a basic Graphplan architecture is, as is now well-known, not competitive with more recent technologies on larger problems. On the other hand, the quality of the plans produced is competitive with those produced by other planners on these domains, including hand-coded planners. We found this particularly interesting, since, as discussed previously, the algorithm optimises the number of distinct happenings rather than makespan. It is not clear whether the implication is that these domains yield good correlation between numbers of happenings and makespans, or whether the results of the other planners are significantly sub-optimal. There clearly remains considerable work to be done to make this approach competitive across a wider set of larger problems.

A final and important result is the demonstration that LPGP can indeed work with actions that have initial effects. We constructed a domain in which an executive must fix a fuse in a basement, using matches to provide light to cross the basement and to fix the fuse. The problem is simple, but we noted in informal tests that we could not manage to produce a plan for the domain using LPG (Gerevini & Serina 2002) or MIPS (Edelkamp 2001). VHPOP (Younes & Simmons 2002) was able to work with actions of this kind, producing equivalent plans to those constructed by LPGP. It is interesting to observe that VHPOP uses a partial order planning strategy as its underlying mechanism and this is well-suited to adaptation to the initial-and-final effects durative actions model of PDDL2.1.

Concurrency and Graphplan Mutex

Graphplan has long been associated with the construction of parallel plans. This suggests that Graphplan embodies some notion of concurrency. In fact, Graphplan offers concurrency not as a central feature of its design but as a by-product of the reasoning that it encapsulates in the plan graph. The mutex relationship between actions is less concerned with the opportunities for concurrency than with the correctness of the construction and search process. Nevertheless, as the Graphplan authors observe (Blum & Furst 1995), the parallel activities in a layer can be interpreted as some form of concurrency. In the transition to temporal planning concurrency becomes a far more central concern. It is natural to speculate on the extent to which the Graphplan mutex plays a useful role in managing interactions between durative actions.

Problem	LPGP			Rank	Best quality	Planner	Time	
	Quality	Time (ms)	Plan steps				(msec)	Avg. Quality
Zeno-1	180	2667	1	5/10	173	TALPlanner	40	177.4
Zeno-2	633	5498	6	7/10	592	TP4/LPG	210/720	618.6
Zeno-3	430	13233	9	6/9	280	TP4/LPG	180/10	416.8
Zeno-4	740	65319	11	5/8	522	LPG	1780	750.6
Zeno-5	583	43830	14	5/9	400	TP4/LPG	26940/640	609.9
Zeno-6	350	57612	12	2/9	323	TP4/LPG	2140/21120	482.3
Drivers-1	91	330	8	=1/9	91	LPG	30	92.5
Satellite-1	41	166	9	1/9	42	TLPlan	10	51.3
Satellite-2	65	24146	13	1/8	70	LPG	20	72.5
Satellite-3	29	62221	13	1/9	34	TP4	310	42.5
Rovers-1	55	303	10	4/9	53	LPG	50	57.2
Rovers-2	44	243	8	=3/9	43	LPG	20	45.1
Rovers-3	58	442	12	5/9	53	LPG	80	69.7
Rovers-4	47	399	8	6/9	45	LPG	60	47.6

Figure 7: Results for problems drawn from the 3rd IPC problem set. Comparison drawn with other systems completing these problems. In each case we report the actual quality (makespan), time and size of the LPGP plan, the ranking in quality of the LPGP plan across all the systems completing plans for those problems (including planners using hand-coded rules) and the best plan produced for the problem. We also report the average quality of plans produced. Planners reporting results on these problems include IxTeT, TPSYS, TP4, VHPOP, MIPS, LPG, TLPlan, SHOP2 and TALPlanner. In the batches that these problems are drawn from, IxTeT reported 8 results in total, TPSYS 10, TP4 precisely the 14 problems listed, VHPOP 54, MIPS 57 and the other planners solved all 102. Note: The LPG results for Rovers were generated after the competition.

```
(:durative-action transmitData
 :parameters (?x - transmitter ?d - data)
 :duration (= ?duration 3)
 :condition (and (at start (calibrated ?x))
                 (over all (calibrated ?x))
                 (at start (holding ?d)))
 :effect (and (at end (sent ?d))
              (at end (not (holding ?d)))
              (at end (not (calibrated ?x))))
)
```

Figure 8: A transmission action.

As we have described, in LPGP the standard Graphplan mutex is extended to a stronger form, in which even actions that agree on the deletion or addition of propositions are considered mutex. We now consider an example which highlights why this is an appropriate modification. Consider a durative action `transmitData` which requires that the transmitter that is being used should be calibrated at the start and, with an invariant, throughout the transmission, but at the end of the action the transmitter loses its calibration status (see Figure 8). If the executive must transmit two blocks of data then it is possible, using the standard Graphplan mutex relationship, to construct a plan in which the transmissions are concurrent, provided they end at exactly the same instant. This is because the delete effects do not then interfere with the invariants. It is counter-intuitive to suppose that it would be plausible to actually synchronise the activities in such a way that they ended at precisely the same moment. The correctness of the plan depends absolutely on the end points being simultaneous. We propose that a more robust plan is one in which the transmissions are sequentialised, recalibrating the transmitter between transmissions. To ensure this the two end points must be considered mutex, even though they agree about the deletion of the same condition. It is examples such as this that motivate our extension of the Graphplan mutex relationship.

Conclusions and further work

This paper has described a successful attempt to exploit the Graphplan architecture to construct temporal plans, but using a different approach to that used in TGP or TPSys (in either of its versions (Garrido, Onaindía, & Barber 2001; Garrido, Fox, & Long 2002)). Where those systems use the graph itself to represent the flow of time, and to solve the associated constraints on the ways in which the durations of actions must interlock in a successful plan, we use the graph to capture only the distinct points of activity and the logical relationships between them, while handling the duration constraints in a separate linear constraint solver. This offers the significant benefit of reducing the necessary graph size during search for most problems. It has the disadvantage that optimisation of temporal duration is then separated from the optimisation of graph length and this prevents the planner from claiming temporal-optimality. Despite this, the results we have managed to produce suggest that the performance is, in practice, not significantly harmed by the potential sub-optimality. Nevertheless, we are continuing to explore possible enhancements of the search to favour better quality plans.

A benefit of our treatment is that it provides an accurate representation of the PDDL2.1 semantics, including an explicit representation of the separations between mutex actions and the impact that they on the structure of plans as well as the timing of the actions they include. This is in marked contrast to the TPSys approach (Garrido, Fox, & Long 2002), where the problem of separation of action end

points represents a much greater burden. This is because the end points of actions are assumed to abut (as in TGP), and then the separation is handled in post-processing.

Although the Graphplan framework offers a convenient one in which to manage time, it is rather less convenient for the management of other numeric values. Some work has explored the introduction of resource management into a Graphplan framework (Koehler 1998), but it is not easy to see how to generalise these ideas to handling more complex numeric expressions. In contrast, the heuristic forward state-space search planners have already been shown to offer some opportunity for extension to handle temporal planning with numeric expressions (Kambhampati 2001; Haslum & Geffner 2001; Edelkamp 2001). It remains an opportunity for further work to determine whether the Graphplan framework can be extended to handle these richer expressive languages.

Many extensions and modifications to Graphplan have been explored, including extensions to ADL features (Nebel, Dimopoulos, & Koehler 1997), filtering irrelevancies (Koehler *et al.* 1997), efficient search beyond the fix-point (Long & Fox 1999), exploitation of symmetry (Fox & Long 1999) and handling sensory actions (Anderson & Weld 1998). The modifications we have explored in LPGP seem to be orthogonal to many of those extensions, since the underlying Graphplan behaviour is largely unchanged. It remains for future work to explore which of these extensions could be successfully integrated with the mechanisms discussed in this paper. We have observed that the EBL/DDB modification proposed in (Kambhampati 1999) cannot be easily integrated with our extension. The key problem is in constructing a conflict set at a layer. This is the set of goals that cannot be solved altogether in a given layer of the graph. In the original algorithm a goal is identified to "blame" for the failure of an action being considered as an achiever for a current goal. This works well when the reason for rejecting an action can be identified with a mutex relationship between it and some other action. In the extended algorithm we have described this is no longer possible: sometimes an action will fail because it causes a violation of the temporal constraints, but this does not lead to identification of a single point of blame in the current layer. In fact, the blame might lie with a choice that was made long before — the choice of the end action, coupled with the failing action, as achiever for a goal in an earlier layer of the search. If the temporal constraints cannot be solved it is potentially expensive to go back through them, determining which is the most recent constraint that could be modified to make the temporal constraints satisfiable. Furthermore, since we check the constraints only when a start action is added, the violation might be the consequence of choices made at a preceding layer in the search. The safest decision is to treat the entire collection of goals so far considered at the current layer as the conflict set. This effectively disables the DDB behaviour when failure results from violation of temporal constraints, but we believe this behaviour can be improved.

A critical extension to the PDDL language introduced in PDDL2.1 is the ability to express plan metrics. Most real planning problems require solutions to be judged by qualities other than simply the number of steps or even their temporal makespan. One of the most significant challenges for the Graphplan architecture, if it is to remain relevant to future developments of planning, is to find ways to modify the search to take into account such plan metric information. The first step in addressing this challenge is to find a convincing means by which to combine efficient behaviour with, at least heuristically, minimising the temporal makespan of the plan. Clearly it would be possible to implement an anytime behaviour in which plans continue to be generated after the first has been found, using the best plan so far as a bound on the continued search. However, we are also concerned with achieving a more efficient performance and we are exploring alternative search strategies as a way to improve the underlying Graphplan search employed in LPGP.

References

Anderson, C., and Weld, D. 1998. Conditional effects in Graphplan. In *AIPS-98*, 44–53.

Berkelaar, M. 2000. Lp-solve. Technical report, ftp://ftp.es.ele.tue.nl/pub/lp_solve/.

Blum, A., and Furst, M. 1995. Fast Planning through Plangraph Analysis. In *IJCAI*.

Edelkamp, S. 2001. First solutions to PDDL+ planning problems. In *Proc. 20th UK Planning and Scheduling SIG*.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Fox, M., and Long, D. 2002. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical Report Dept. CS, 20/02, Durham University, UK. Available at `www.dur.ac.uk/d.p.long/competition.html`.

Garrido, A.; Fox, M.; and Long, D. 2002. Temporal planning with PDDL2.1. In *Proceedings of ECAI'02*.

Garrido, A.; Onaindía, E.; and Barber, F. 2001. Time-optimal planning in temporal problems. In *Proc. European Conference on Planning (ECP-01)*.

Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs. In *Proc. of 6th International Conference on AI Planning Systems (AIPS'02)*. AAAI Press.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. of European Conf. on Planning, Toledo*.

Kambhampati, S. 1999. Improving Graphplan's search with EBL and DDB techniques. In *Proceedings of IJCAI'99*.

Kambhampati, S. 2001. Sapa: a domain independent heuristic metric temporal planner. In *Proc. of the European Conference on Planning, Toledo*.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y.

1997. Extending planning graphs to an ADL subset. In *ECP-97*, 273–285.

Koehler, J. 1998. Planning under resource constraints. In *Proc. of 15th ECAI*.

Long, D., and Fox, M. 1999. The efficient implementation of the plan-graph in STAN. *JAIR* 10.

Long, D., and Fox, M. 2001. Encoding temporal planning domans and validating temporal plans. In *Proc. 20th UK Planning and Scheduling SIG*.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *ECP-97*, 338–350.

Penberthy, J., and Weld, D. 1994. Temporal planning with continuous change. In *Proc. of the 12th National Conference on AI*.

Smith, D., and Weld, D. 1999. Temporal Graphplan with mutual exclusion reasoning. In *Proceedings of IJCAI-99, Stockholm*.

Younes, H., and Simmons, R. 2002. On the role of ground actions in refinement planning. In *Proc. 6th International AIPS Conf.*, 90–97.