Managing Coordination in Temporal Planning using Planner-Scheduler Interaction

Andrew Coles [a] Maria Fox [a] Keith Halsey [a 1] Derek Long [a]
Amanda Smith [a]

[a] *Department of Computer and Information Sciences,*
University of Strathclyde,
Livingstone Tower,
26, Richmond Street,
Glasgow G1 1XH
United Kingdom

Abstract

Temporal planning contains aspects of both planning and scheduling. Many temporal planners assume a loose coupling between these two sub-problems in the form of "blackbox" durative actions, where the state of the world is not known during the action's execution. This simplifies the temporal planning problem and restricts what can be modelled. In particular, the simplification makes it impossible to model coordination, where actions *must* be executed concurrently to achieve a desired effect.

Coordination results from logical and temporal constraints that must both be met, and for this reason, the planner and scheduler *must* communicate in order to find a valid temporal plan. This work examines where coordination arises in temporal domains, and then uses this information to minimise communication between a planner and scheduler in a temporal planner called Crikey. Results presented show Crikey to be a competitive planner, whilst not making the same simplifying assumptions. The investigation of the interaction between planning and scheduling has revealed a class of problems, which can be modelled using PDDL2.1, that cannot be solved by state-of-the-art planners. Crikey is able to solve temporal problems involving coordination that other planners are unable to solve.

Key words: Temporal planning, PDDL2.1, planning and scheduling

[1] The work presented in this paper has its foundations in Keith Halsey's PhD research.

1 Introduction

Planning with real, rather than relative, time was invigorated by the extension of PDDL to include temporal features [1]. This extension led to the development of a number of planners capable of handling temporal domains, with greater or lesser degrees of competence [2–9]. General temporal planners that preceded PDDL2.1 include TLPLAN [10], TALPLANNER [11], TGP [12] and ZENO [13].

Time can play quite different roles in determining the difficulty of temporal planning problems. For example, it can simply provide the metric used to determine plan quality. In this case, exploiting concurrency can improve the quality of plans, but might not be necessary to find a solution. Alternatively, time can be a critical resource that must be properly handled in order to successfully solve a problem. This case includes problems with deadlines, with fixed windows of availability of some resources and with actions that create windows of availability of resources. The nature of some of these problems was discussed by Fox and Long in the context of their planner, LPGP [6] and also by Halsey, Fox and Long [14]. A more recent treatment by Cushing, Kambhampati, Mausam and Weld [15] has formalised the nature of these distinctions and provided a more thorough catalogue of the kinds of interactions that are possible. A key finding, reported in their paper, is that most benchmark domains do not require concurrency in order for a solution to be constructed. That is, time plays a role only in determining plan quality for most benchmark problems. In fact, most planners cannot handle temporal domains that require concurrency in order to manage time-restricted resources.

In this paper we explore temporal planning problems in which concurrency is required for a solution. We present a planning system, CRIKEY, that is designed to handle a class of these problems. It does so by decomposition, splitting the planning problem into a part that is essentially the selection and ordering of actions (what might be seen as the purely planning aspect of the problem) and the temporal scheduling of those actions. The division of planning problems into planning and scheduling elements has been explored by other researchers [5,16,17], but not in the context of the solution of problems that *require* concurrency for their solution. Cushing *et al* describe a planning approach that could, in principle, handle temporal planning in problems of this kind, but, to the best of our knowledge, their algorithm has not been implemented. LPGP is also capable, in principle, of handling temporal problems of this type, but there are certain difficulties raised by the direction of search in that planner that make it impractical to apply to large problems without further revisions of the algorithm.

Our approach has been implemented in the planner CRIKEY, which works by applying our decomposition-based strategy. Other attempts have been made to decompose planning and scheduling [18–20,16]. A key difficulty in achieving a successful decomposition lies in the handling of communication between the planner and scheduler elements. This balance not only affects the quality of a solution, but can also determine whether a solution can be found at all. In Crikey we strive to minimise this communication, so that the planner and scheduler communicate only where absolutely necessary. Communication is required where the two components are tightly coupled — where actions *must* happen concurrently. In other parts of planning problems, where the components are not closely coupled, it is possible to limit the interactions and reduce the overheads in the communication costs.

The remainder of the paper is organised as follows: we begin by reviewing the coupling between planning and scheduling in planning problems. We then examine how temporal actions are represented in pddl2.1, contrasting this with how most temporal planners in fact reason with the actions, and why the approach most commonly used is neither sound nor complete. In the context of the capabilities of pddl2.1, we then examine the nature of temporal constraints that can arise in domain encodings. From here, we go on to consider how the temporal capabilities of pddl2.1 can better be captured, revisiting the representation used in the planner lpgp, and how this potentially could be adopted for use in a forward-chaining search setting forming an lpgp–ff hybrid. Although flawed, the hybrid sets out the key challenges, which are subsequently tackled in the remainder of the work. In particular, we develop the concept of *envelopes* and their *contents* as structures within temporal domains and we then proceed to explain how these structures are managed in the architecture of Crikey to overcome the problems of the lpgp–ff hybrid. We begin with a general form of the planner, Crikey, and then go on to describe a more specialised and, therefore, more restricted version, Crikey$_{SHE}$. Finally, we present some results showing how Crikey and Crikey$_{SHE}$ perform and then conclude.

2 Coupling of Planning and Scheduling

Figure 1 illustrates the spectrum of coupling between planning and scheduling in temporal planning domains. The coupling increases towards the centre of the diagram and the higher numbered parts of the spectrum. The idea that planning and scheduling problems can be seen as lying on a spectrum between purely planning problems and purely scheduling problems is not new (see, for example, the working notes of the Workshop in Integrating Planning into Scheduling [21], or work by Smith, Frank and Jónsson [22]).

On the left (type 1p) are pure planning problems that contain no scheduling. These include classical planning benchmark domains. On the far right (type

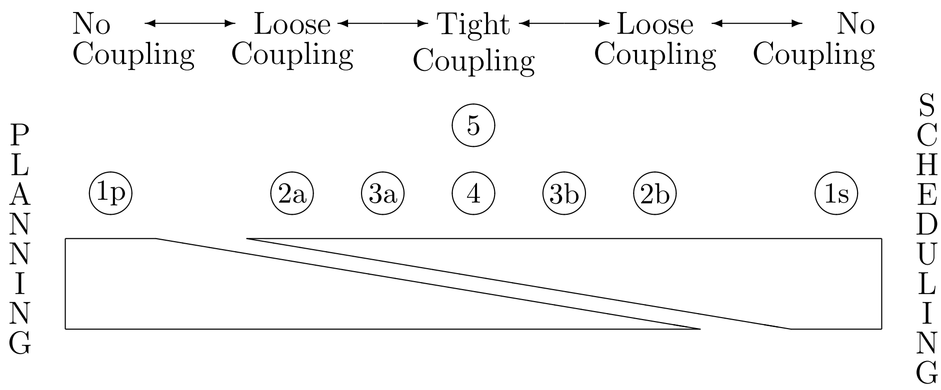


Fig. 1. Coupling Between Planning and Scheduling in Temporal Planning Domains

1s) are the domains that are completely scheduling problems that have been encoded as planning problems. In these problems there is no choice about which actions are required to be executed, only about the order and timing of those actions.

Domains of type 2 represent problems in which there is a core that falls squarely into the planning (2a) or scheduling (2b) part of the spectrum, but where there is a component with the character of the complementary sub-problem. In these domains, this component is more easily solved than the core problem and creates no consequences for either the satisfiability or the quality of the solution. For example, a domain of type 2b will be predominantly a scheduling task with a planning component in which the choice of actions will be easy and have no effect on the quality of, or difficulty in finding, the schedule. Here there is no coupling between the problems.

Domains of type 3 can also be divided into those that are planning-centric (3a) and those that are scheduling-centric. All of these domains exhibit a loose coupling between the planning and scheduling aspects of the problems, where the solution to one part of the problem only affects the quality of the solution to the other, and not the satisfiability. All of the problems in the Third International Planning Competitions [23] (IPC3) were of this type, where the choice of action affected only the quality of the schedule produced. For example, in the ZenoTravel Time domain from IPC3 there are two fly actions: one for flying fast, and the other for flying slowly using less fuel, but taking longer. The choice of action (i.e. whether to fly fast or slowly) affects the quality of the schedule that is produced, but a schedule can always be found whatever choices are made for the actions in the plan. Concurrency in these domains *may* occur in order to produce a better schedule, but it is not *necessary* to find a solution. A consequence of this is that all plans to solve problems in the first three levels of this spectrum can be sequentialised so that there is a total ordering between the actions forming a plan. Problems of this type are *temporally simple* in the terms defined by Cushing *et al* [15].

The tightest coupling happens in domains in the centre of the spectrum (types 4 and 5) where concurrency *must* happen. Type 4 refers to domains where the concurrency is required in order to achieve goals by their deadlines. This can be seen as a slight extension of problems of type 3, in which the "soft" effects of temporal structure on plan quality are replaced by the hard constraint that goals must be met by their deadlines. In domains of type 5, concurrency *must* happen, not only to achieve a goal by a deadline, but to achieve a goal at all. The coupling between planning and scheduling is stronger in domains of type 5 since the concurrent actions interact, whereas in domains of type 4, they do not. This interaction is coordination. The problems that lie in both these parts of the spectrum include those that are defined as *requiring concurrency* by Cushing *et al.* They do not distinguish problems in which the concurrency is required because of deadlines from those in which it is required because of interactions between actions. In fact, they only consider required concurrency implied by the interaction between actions. The definition of *required concurrency* demands that all solutions to a particular problem include concurrent actions. In practice, reasoning about concurrent activity will also be useful in problems of type 3, where the quality of a plan is affected by the use of concurrency.

An example of the requirement for coordination in a domain (hence, a domain of type 5) occurs in the Match domain (see Appendix 12.2), a variant of which was first presented in [6], where the goal is to mend fuses in a dark cellar. To mend a fuse requires the MEND_FUSE action for which there must be light throughout the duration of the action. The only light available is achieved by lighting a match, using the LIGHT_MATCH action, which provides light only whilst it burns (i.e. for the duration of the action). To mend a fuse one must also have a hand free, the impact of which is that one can only fix one fuse at a time.

Where the LIGHT_MATCH action is 8 time units long and the MEND_FUSE action is 5 time units long, it should be obvious that two matches will be needed to provide enough light to fix two fuses, since both fuses cannot be fixed by the light of one match before it burns out. However, if the fuses take less time to fix, the matches burn for longer, or fuses can be fixed concurrently, then a different number of matches may be required. Importantly, the MEND_FUSE actions *must* be executed (and completed) during the execution of the LIGHT_MATCH action. These actions must be co-ordinated (i.e. happen concurrently and in the correct order) so that the goal of fixing the fuse is reached. Figure 2 shows a valid plan for the problem.

Temporal planning domains in the Fourth International Planning Competition [24] (IPC4), other than those in which the Timed Initial Literals introduced in PDDL2.2 [25] were used, were of type 3. Domain variants in which timed initial literals were used were of type 4. Once compiled into PDDL2.1,

```
0.01: (LIGHT_MATCH match1) [8.0]
0.02: (MEND_FUSE fuse1 match1) [5.0]
2.04: (LIGHT_MATCH match2) [8.0]
5.03: (MEND_FUSE fuse2 match2) [5.0]
```

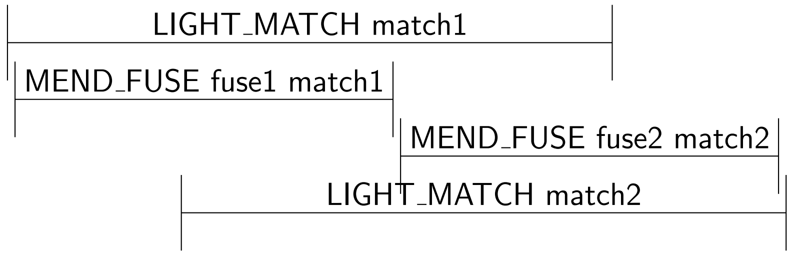


Fig. 2. A Valid Plan for the Match Problem, assuming that matches may burn concurrently.

these variants become domains of type 5 since the dummy actions used to encode the compiled versions require coordination. Similarly, temporal planning domains in the Fifth International Planning Competition [26] (IPC5) also required no coordination, other than if a compilation of timed initial literals was used.

3 Representation of Temporal Actions in PDDL2.1

The most basic action representation in PDDL2.1 is the STRIPS action. This does not encode any temporal information, and is assumed to be an instantaneous action: all of its effects appear immediately upon the action being applied and all of its preconditions have to hold immediately before. The action can be thought of as denoting a single *happening* with conditions and effects attached. Such actions are not sufficient to model a temporal domain accurately, yet they remain important as many planners address temporal planning by a reduction of a more complex model to these actions.

In PDDL2.1, a model of temporal actions was introduced in which an action has two happenings: one at the start, and one at the end. Further, between these points, invariant conditions can be required to hold — conditions which must remain true during the execution of the action. The two happenings marking the start and end of the durative action are separated in time by either a fixed duration, or a duration in a specified range represented by a *durational inequality*. As with STRIPS actions, the conditions can be specified as arbitrarily complex logical formulæ, using all logical connectives, quantification and even arithmetic expressions on number-valued fluents. For the purposes of this work we will restrict our attention to purely logical fluents and to actions with preconditions expressed purely as conjunctions of (positive) propositions. Similarly, we will not consider conditional or quantified effects.

Most planners that attempt to manage temporal domains do so by a simple compilation approach, in which durative actions are simply flattened into STRIPS actions. This compilation creates a *compressed action* with which the planner can reason in a relatively simple way. These compressed actions are referred to as "blackbox" actions. A compressed action has the effect of applying the whole action at once: that is, applying the start effects first followed immediately by the end effects, while still respecting the conditions as far as possible. The preconditions of the compressed action are the start conditions of the durative action and all end conditions and invariants not achieved by the start effects. We assume that the conditions associated with a durative action can be split into a conjunction of literals that must hold at the start of execution, a conjunction that must hold throughout execution and, finally, a conjunction that must hold in order for execution to complete. We further assume that there are no conditional effects attached to durative actions. In fact, both these assumptions are simplifications of PDDL. Our last assumption is that the duration of a durative action (once grounded) is fixed, rather than state-dependent or underconstrained. We capture these assumptions in the following definition of a *simple* durative action.

Definition 1 — Simple Durative Action
A Durative Action operator da is a tuple:

$$da = (C_{\vdash}, C_{\leftrightarrow}, C_{\dashv}, A_{\vdash}, A_{\dashv}, D_{\vdash}, D_{\dashv}, \Delta)$$

where the first three elements are the sets of literals that must be true at the start, throughout and at the end of execution, respectively; the following four elements are the add and delete effects at the start and end of the action and the last element is the action duration.

Definition 2 — Compressed Action
A compressed action, $ca = (cond, add, del)$, is an STRIPS action that has been formed from a simple durative action, $da = (C_{\vdash}, C_{\leftrightarrow}, C_{\dashv}, A_{\vdash}, A_{\dashv}, D_{\vdash}, D_{\dashv}, \Delta)$, where

$$\begin{aligned} cond &= C_{\vdash} \cup ((C_{\dashv} \cup C_{\leftrightarrow}) \setminus A_{\vdash}) \\ add &= (A_{\vdash} \setminus D_{\dashv}) \cup A_{\dashv} \\ del &= (D_{\vdash} \setminus A_{\dashv}) \cup D_{\dashv} \end{aligned}$$

This compression has two key problems: it is neither complete nor sound. Incompleteness follows from the fact that the compression results in a less expressive language, as observed by Cushing *et al* [15], so that planners using this reduced representation cannot solve (or, indeed, properly represent) problems that require coordination (or *required concurrency*) at all. In the Match domain example, the compressed action to light a match would not add the fact that the match is lit! Since this effect is both added at the start and deleted at the end of the action, it would be compiled out of the domain, rendering the (compressed) problem unsolvable.

One can show the compression to be unsound by modifying the Match domain in two ways: first, instead of using a predicate `(light ?l - match)` we use a parameter-less predicate, `(light)` and second, the goal is modified to require that both matches be burnt. With the former modification, two LIGHT_MATCH actions are now mutually exclusive since the end of the action deletes `(light)`, an invariant of the action. However, if we compress this action according to 2 all interactions with the `(light)` predicate are discarded. This allows a planner working with the compressed domain to achieve the goal of having burnt both matches by scheduling two LIGHT_MATCH actions, one for each match, in parallel. Clearly, under the original semantics, this plan is unsound since the two actions are mutually exclusive.

Despite these problems, this compression technique has been widely used in planners that attempt to solve temporal problems. One strength is that as the domain is reduced to an essentially non-temporal planning formulation, one can perform temporal planning using a two-stage approach: find a solution to the problem using compressed actions with a non-temporal planner and then schedule it to account for the durations of the actions. This process is used in MIPS [17] and SGPlan [5].

4 Temporal Constraints in PDDL2.1

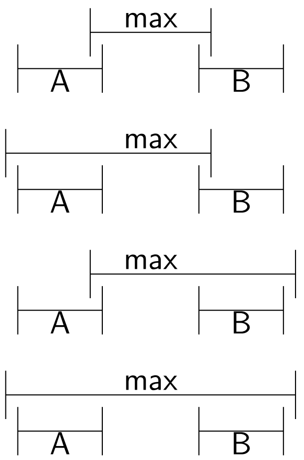
In this section we consider the nature of temporal relationships that can arise in PDDL2.1 planning problems. Our objective is to identify the kinds of constraints that might have to be managed by a planner.

PDDL2.1 is very expressive. It can capture a wide range of temporal relations and constraints, such as Allen's interval relations [27] applied to durative actions, through the use of dummy actions to enforce the conditions that are needed. This idea was explored in some detail in work reported by Fox, Long and Halsey in [28].

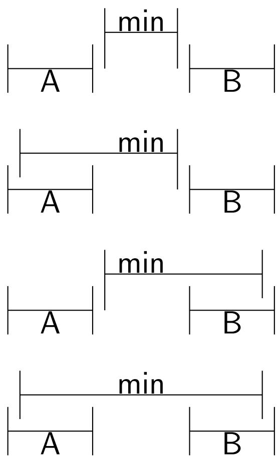
Individual temporal constraints can be reduced to the form: $x - y$ $\{\leq, <, \geq, <\}$ b, where x and y are the actual times of the start or end points of actions, and the difference $(x - y)$ is how far apart in time they are relatively.

b gives the bound on this difference. Note that conjunctions of constraints can capture equality and interval ranges, while disjunctions can express a rich variety of temporal structures. All Allen interval constraints [27] can be captured using temporal constraints of the form described here, using time points representing the ends of the intervals. Deadlines can be represented by setting one time point to be 0, so, for example, if timepoint e must happen by deadline d, this is represented as $e - 0 \leq d$.

Between two actions, A and B, there are four time points (two start times and two end times), which can be related in pairs in lower- and upper-bounded constraints. Ignoring symmetric alternatives, there are eight possible constraints between the pair of actions, shown in Figure 3. The constraints are all imposed through the use of a third action whose length is determined by the durations of A and B and the desired maximum or minimum time between the actions. We call this action an *auxiliary* action.



(a) Maximum Constraints



(b) Minimum Constraints

Fig. 3. Possible Combinations of Representing the Same Constraint

Regardless of the form used, when expressing a maximum time between actions A and B, where A precedes B, the ordering is from the start of the auxiliary action to A, and from B to the end of the auxiliary action. When expressing a minimum time gap, regardless of the form used, the ordering is from A to the start of the auxiliary action and then from the end of the auxiliary action to B. Ordering constraints are achieved simply by adding a dummy propositional effect to the first time point in the ordered pair and the same proposition as a condition in the second time point of the pair. This ensures that the second time can only occur once the first time has occurred. [2]

[2] The semantics of PDDL2.1 imposes separation constraints between time points that are necessarily ordered in this way. There are minor technical consequences of

It can be seen that in maximum constraint orderings there are *no* ordering relations in which an action end precedes an action start, while in minimum constraint orderings, there are *no* ordering relations in which an action start precedes an action end. An examination of the relationships in Figure 3 reveals that ordering constraints between time points can be organised into two types: those that determine an upper bound on the separation of two actions and those that determine a lower bound on the separation.

In practice, the maximum and minimum separation relations between actions in a plan will often be captured through multiple interacting actions, as we discuss in Section 6. Furthermore, the auxiliary actions that support these relations in Figure 3 will usually have direct roles in a planning domain, rather than appearing simply as the mechanism by which the temporal constraints are expressed. For these reasons, the pure separation constraints illustrated in Figure 3 are likely to appear in more complicated forms in real plans. Nevertheless, they form the building blocks of the more complex forms and illustrate the expressive power of PDDL2.1 in capturing temporal constraints.

The *maximum separation relationship*, $\prec^{max}$, specifies the ordering constraints imposed on actions that must be separated by no more than some specified value; the *minimum separation relationship*, $\prec^{min}$, is the equivalent when the actions must be separated by no less than a bounding value. In the following, we use $A_{\vdash}$ and $A_{\dashv}$ to represent the start and end of an action respectively.

Definition 3 — Maximum Separation Relationship
Given a collection of action instances, $\mathscr{A}$, with end points partially ordered by the partial order $\prec$, two actions, A and B in $\mathscr{A}$, are part of a *maximum separation relationship*, written $A \prec^{max} B$ if there exists an action, $C \in \mathscr{A}$, such that $C_{\vdash} \prec A_{\dashv}$ and $B_{\vdash} \prec C_{\dashv}$.

This definition requires that $\prec$ be a partial order, which means that it must be closed under transitivity. Thus, each of the four relationships shown in Figure 3 (a) is an instance of a maximum separation relationship, since in each case the start of *max* is constrained to lie before the end of A and the end of *max* is constrained to lie after the start of B.

Similarly, we make the following definition:

Definition 4 — Minimum Separation Relationship
Given a collection of action instances, $\mathscr{A}$, with end points partially ordered by the partial order $\prec$, two actions, A and B in $\mathscr{A}$, are part of a *minimum*

this constraint that are not of interest here. The details of handling these constraints are discussed by Fox *et al* [28].

separation relationship, written $A \prec^{min} B$ if there exists an action, $C \in \mathscr{A}$, such that $A_{\vdash} \prec C_{\vdash}$ and $C_{\dashv} \prec B_{\dashv}$.

Maximum separation constraints specify a maximum possible time between the occurrences of two actions; minimum separation constraints represent a minimum time between two constraints. With only maximum separation constraints and no minimum constraints, B could happen before A and, of course, with only minimum constraints B could happen arbitrarily far after A without breaking the constraints. Interesting cases occur when both maximum and minimum separation constraints occur for the same pair of actions, two examples of which are shown in Figure 4. In this case, the constraints lead to an equivalent expression of the form $b_1 \leq x - y \leq b_2$, where x corresponds to either the start or end time of A, y corresponds to either the start or end time of B and b_1 and b_2 are the durations of the *min* and *max* actions respectively. Of course, for it to be possible for constraints with both maximum and

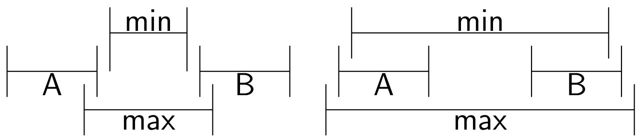


Fig. 4. Expressing both Minimum and Maximum Time Between Actions in PDDL2.1

minimum separations to be satisfied, the duration of min must be less than or equal to the duration of max.

5 An LPGP–FF Hybrid

Having considered various constraints that can be expressed in PDDL2.1 and which can only be properly managed by a planner that does not perform action compression (as described in Definition 2), in this section we will take a step towards developing a planner that can search without using compressed actions. The approach that will be discussed is intuitive, albeit flawed, and serves as a motivating foundation for the general idea we present in subsequent sections. The shortcomings of the approach serve to highlight what problems must be resolved in order to construct a more general solution to temporal planning, as we go on to do.

Recalling the format of a durative action presented in Definition 1, a ground action P can be split, conceptually, into three classical planning actions: a *start* action, an *invariant* action and an *end* action. Two additional dummy facts are required, to ensure the actions have to be executed in the order (start, invariant, end). The three actions used to represent P are instantaneous — duration information is expressed separately as a constraint on the separation of the start and end actions. We emphasise this point by calling them *snap-actions*, formally defined as:

Definition 5 — Snap-actions
Given a ground durative action, P, the start, end and invariant actions, representing P are defined as follows:

- $P_\vdash = (C_\vdash,\ A_\vdash \cup P\text{-}inv,\ D_\vdash)$;
- $P_\leftrightarrow = (C_\leftrightarrow \cup P\text{-}inv,\ P\text{-}inv \cup iP\text{-}inv,\ \emptyset)$;
- $P_\dashv = (C_\dashv \cup P\text{-}inv \cup iP\text{-}inv,\ A_\dashv,\ D_\dashv \cup P\text{-}inv \cup iP\text{-}inv)$

$P_\vdash$, $P_\leftrightarrow$ and $P_\dashv$ are called snap-actions.

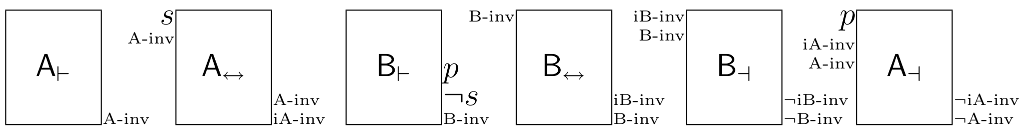
An approach based around this division forms the foundation of LPGP [6], where a variant of GraphPlan search is employed over a planning graph populated with these snap-actions. However, the idea is not restricted to use within a GraphPlan setting: the snap-action domain can be used with alternative planning strategies, such as the forward state-space search used in FF [29]. The resulting system is a hybrid between the action-splitting approach of LPGP, and the forward-chaining state-space search approach of FF. The resulting plan found by FF (in terms of snap-actions) can be post-processed to restore the structure of the temporal constraints between start and end points of durative actions and to resolve the ordering constraints that must be satisfied. A part of this post-processing is to lift a partial order from the sequential plan structure produced by FF. This can be achieved by applying an algorithm due to Veloso, Peréz and Carbonell [30].

Whilst intuitive, three problems arise when using this approach:

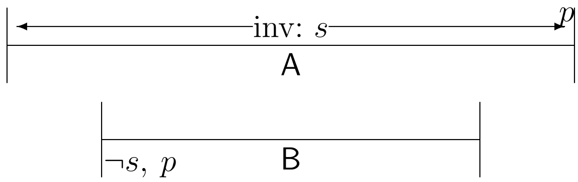
- invariants may not be respected correctly;
- a goal state may be found where some actions have not terminated;
- temporal constraints may not be satisfied.

The first problem is illustrated in Figure 5 where a start ($\mathsf{A}_\vdash$) and invariant action ($\mathsf{A}_\leftrightarrow$) are put into the plan, followed by an action (B) that breaks the invariant condition, s, before the end action ($\mathsf{A}_\dashv$). Even if the invariants are made conditions of the end action, it would still be possible for invariants to be broken and then reachieved. This is because the translation of a durative action treats its invariant as a single point of time, when it is actually a condition across the entire interval between the start and end points of the action. Therefore, in the situation shown in Figure 5, FF produces a "valid" total order classical plan for the translated domain, but when this is passed through the partial order lifter and scheduled, it produces an invalid temporal plan with respect to the original temporal domain, since the invariant, s, of action A

is broken. In LPGP this creates no problems because there is a mechanism employed to force the invariant-checking action to be reapplied alongside all actions chosen between the active start and end points of a selected action.



(a) Valid Total Order Plan



(b) Corresponding Invalid Temporal Plan where the Invariant s is Broken

Fig. 5. Example of a Broken Invariant

The second problem with this hybrid approach arises due to the way in which the dummy propositions operate in the translation. Whilst there cannot be an end action without a start, there could be a start in the plan without its end. This is counter to PDDL2.1 semantics, which requires that all actions must be complete in a goal state, so a post-processing step is needed to ensure that an invariant and end action are added to the plan for each start action. However, this is not suitable if the end action then deletes a goal (as shown in Figure 6). LPGP handles the problem by preventing start actions from being selected unless a corresponding end action has been selected (recall that LPGP searches backwards in the plangraph).

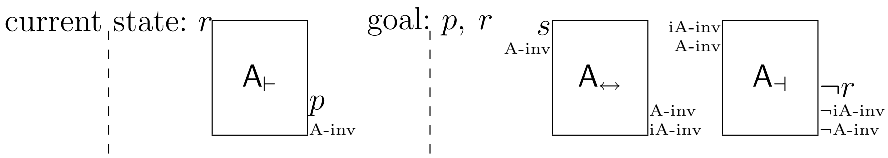


Fig. 6. Example of an End Action Deleting a Goal

The third problem, relating to duration constraints being violated, arises because no notion of 'time elapsed' is maintained during search in FF, as all duration information is discarded when durative actions are split into three snap-actions. Referring again to Figure 5, suppose that the the same problem were used with the proposed hybrid, but with the durations of actions A and B being 8 and 10, respectively. Then FF would still find the plan illustrated, but it is clear that no valid schedule could then be inferred from the plan:

the duration of B, which according to the sequential plan *must* occur entirely between $\mathsf{A}_{\vdash}$ and $\mathsf{A}_{\dashv}$, exceeds the duration of A.

In the remainder of this work, we show how a forward-chaining state-space search planner, Crikey, can be constructed, inspired by the hybrid we have just described, but with appropriate modifications made to overcome the three problems we have highlighted.

6 Envelopes and Contents

Strong coupling between planning and scheduling occurs where actions must happen concurrently. This is most important when it is required in order for the problem to be solved, but it also matters in order to minimise the makespan of the plan. This strong coupling brings with it several problems that impact search for a solution plan, as demonstrated in Section 5.

Conceptually, coupling arises when durative actions create *envelopes* of opportunity in which other actions must start or finish executing or, indeed, both start *and* finish executing. Such envelopes are a corollary of strong coupling between planning and scheduling: if no such envelopes arise, then considering decision epochs between the start and end of actions is unnecessary, and the compressed action representation presented in Definition 2 is sufficient. Formally, an envelope and its *contents* can be defined as a sequence of actions that are logically constrained to be executed concurrently with one another. In the simplest case, an envelope will consist of one or more actions that are subject to maximum separation constraints, while the contents will be one or more actions constrained to lie within the temporal extent defined by the envelope and, typically, constrained with minimum separation constraints. In this way the contents of an envelope exert a counter-tension to the envelope, forcing the envelope actions to separate to make room for the contents. In practice, the interactions between actions that form the contents of an envelope might include constraints that determine maximum separations of actions in the set, so that these actions themselves form an envelope for some other subset of actions that form its contents. Thus, different envelopes do not necessarily form mutually exclusive sets of actions.

Before we provide a formal definition of the concept of an envelope and its contents, we repeat the observation made in Section 4, that the constraints that govern the separation of end points of actions can all be captured as expressions of the form $x - y\ \{\leq, <, \geq, <\}\ b$, where x and y are the times at which the end points of the corresponding actions are executed and b is a bound arising from the constraints on durations of actions, or on separation of actions. A collection of such constraints forms a *simple temporal problem*

(STP), sometimes called a *simple temporal network* (STN) after the graphical representation of the STP [31]. An STN expresses a partial order on the time points it constrains, and we will use $\prec_S$ to denote the partial order defined by the STN S.

Definition 6 — Envelope and Contents
An envelope, E, is a 4-tuple, $(A_\vdash, B_\dashv, C, S)$, where C is a set of snap-actions forming the contents of the envelope, the snap-actions of A and B are in C and S is a simple temporal network expressing the constraints that must hold between the snap-actions in C. $A_\vdash$ is a start action, $B_\dashv$ is an end action and, for every snap-action, x, in C, $A_\vdash \preceq_S x$ or $x \preceq_S B_\dashv$.

Note that the definition of an envelope does not require that the same action provide the start and end points of the envelope (although an action may do so). These points represent the maximal extent of the envelope and all the content actions are constrained with respect to them. The only necessary constraints are that every snap-action in the envelope is constrained to fall after the start of the envelope or before the end of the envelope.

In planning problems that require coordination, concurrent actions are logically, as well as temporally, constrained. Thus, in these cases, one or more of the content actions in an envelope are constrained to execute concurrently with the enclosing actions because the enclosing actions supply resources that must be available to the content actions. This logical constraint implies a further temporal constraint: that the contents must fit between the enclosing actions. The envelope associated with such resources will then include constraints (in its STN) that force the contents to fit between its extreme points.

For the STN of an envelope to be solvable or, equivalently, for the envelope to be schedulable, it is necessary to know whether the minimum amount of time in which the content actions can be executed is less than the maximum amount of time that the envelope actions could take to execute. It stands to reason that if the envelope has an infinitely large maximum time or the content actions have a minimum time of zero, then there will be no problems scheduling, since the content actions will always fit in the envelope. The problem occurs where the inverse is true. An envelope, $(A_\vdash, B_\dashv, C, S)$, will have a finite maximum total execution time when the STN, S, captures a finite upper bound on the gap between $A_\vdash$ and $B_\dashv$. This situation arises when the envelope contains one or more maximum separation constraints, including the actions A and B. The envelope becomes a significant constraint when the contents include one or more minimum separation constraints.

As observed above, content actions can be envelope actions themselves (with other actions being the contents) and so, similarly, envelope actions can also be content actions for other envelope actions. Content and envelope actions cannot be sequentialised with respect to one another and *must* be executed in parallel. In the case of the match domain, the LIGHT_MATCH action is the envelope action, and the MEND_FUSE actions are the content actions. See Figure 7 for examples of envelopes and content actions.

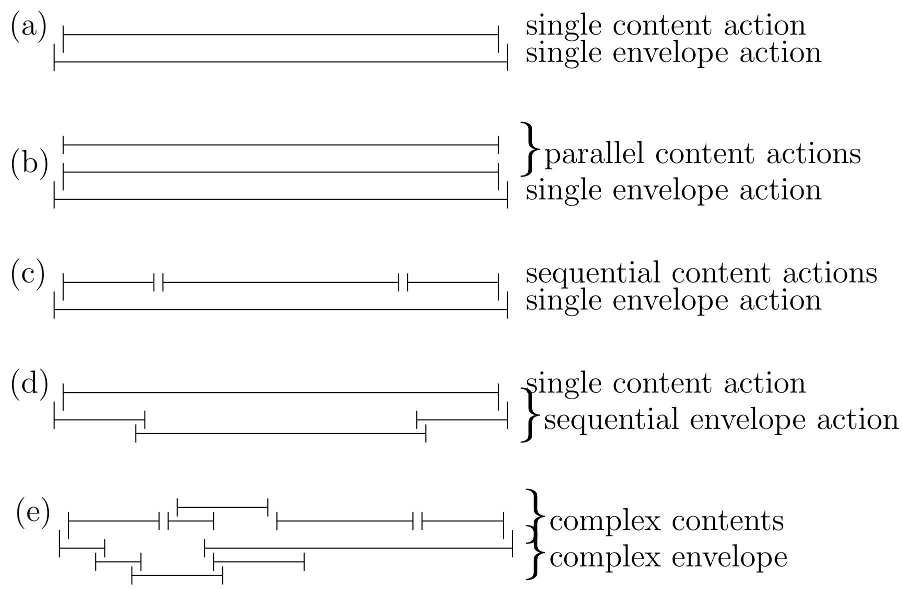


Fig. 7. Envelopes and Contents

7 Crikey

The concept of envelopes embodies the kernel of the necessary coordination between planning and scheduling that occurs when working with temporal planning domain models. In this section, we describe Crikey: an extension of the LPGP–FF hybrid presented in Section 5 which overcomes the problems of that approach by reasoning with envelopes explicitly during search.

On a basic level, Crikey has much in common with the hybrid system, notably:

- the search algorithm used is the same as that used in FF: Enforced Hill Climbing (EHC) followed by Best-First Search (BFS) if EHC fails;
- the relaxed planning graph (RPG) heuristic from FF is used to guide search, with helpful action pruning also being used in the same manner;
- Crikey reasons with the translated version of the domain using start and end actions; rather than the conventional "blackbox" actions compressing all of the effects into a single, instantaneous action.

A simplification of CRIKEY's workings that may aid a conceptual understanding is as follows. Consider planning beginning as a forward-chaining search from the initial state. When any start action is applied CRIKEY creates a new envelope, E. Each time a new action is to be added to the plan that interacts with any of the actions in E, CRIKEY extends the simple temporal network for the envelope to determine whether E can be safely scheduled. An action interacts with the actions in E if it has any enforced ordering relationships with respect to the start, end or content actions of the envelope (as determined by the Partial-Order-Lifting algorithm, described in Section 7.4.1). If the action does not interact with any actions in any of the envelopes that are currently open it can be harmlessly scheduled either alongside, before, or after them once planning has finished, so there is no need to perform any more sophisticated reasoning. Postponing this expensive reasoning where possible and completing it in one step at the end is a major benefit of this approach. What follows is a formal specification of the behaviour of CRIKEY.

The key difference between CRIKEY and FF lies in the definition of the states used in planning and in the temporal scheduling abilities of CRIKEY and the corresponding effects on the search space. Firstly, in FF:

- **vertices** in the search space corresponds to planning states — sets of facts;
- **edges** correspond to ground actions (actions whose parameters are fully specified);
- the **successor rule** is that an edge corresponding to an action A leaves a state F iff F satisfies the preconditions of A and following the edge leads to a state, F', containing the facts from F updated to reflect the effects of A;
- the **goal** of search is to find a path from the vertex denoting the initial state to any state goal state, G. The edges along the path from the initial state to G then represent a solution plan.

The modifications made to each of these will now be considered.

7.1 States with Envelopes

A planning state in CRIKEY (a **vertex** in its search space) comprises a set of facts and a set of *open envelopes*, as described in Definition 7. As can be seen, the facts used in states in FF are augmented with a set of open envelopes, through which temporal coordination can be managed.

Definition 7 — Planning State
A planning state S is

$$S = (F, \xi)$$

where F is the set of true facts and ξ, the set of open envelopes.

The envelopes in a planning state in CRIKEY are "open" in the sense that their contents, and even the end action that defines their extent, can change as the plan develops. Furthermore, the end points of the envelopes have not yet been added to the plan (although, since the start actions have been added, the end actions can be identified and it be known that they will eventually be added to the plan). Nevertheless, the structure of these envelopes is as defined in Definition 6.

The open envelopes in a CRIKEY planning state correspond to the actions for which the start action has been selected and put into the plan, but for which the end action has yet to be added. This definition allows for envelopes that are many actions long.

Definition 8 — Consistency Function
For an envelope, $E = (A_{\vdash}, B_{\dashv}, C, S)$, the function $consistent(E)$ returns *true* when the STN, S, is consistent. In this case, E is said to be a consistent envelope.

An envelope is consistent if the contents fit inside the envelope. Consistency is tested by performing Bellman-Ford's Single Source Shortest Path algorithm from $B_{\dashv}$ (i.e. from the end of the envelope). Any negative cycles for this envelope must involve this end action as this will have a positive edge directed out of it for the maximum time difference from its start action, and then negative edges leading back to it for the minimum duration of the contents.

7.2 Applying Actions to States — Logical Constraints

As in FF (as part of the hybrid), **edges** in the search space correspond to applying actions. However, the **successor rule** is substantially more complex: the conditions under which an action can be applied to a state, and the details of the state reached, are a function of both the facts F in the state, as before, but also of the open envelopes ξ.

Just as in FF, an action can only be applied if its preconditions are satisfied and the facts, F, in the state are updated to reflect the effects of an action. Beyond this, to ensure soundness, it is necessary to also enforce the constraint that an action can only be applied if it respects the invariants of any other action currently in progress. As shown in the LPGP–FF hybrid, because durative actions have been split into snap-actions, without tracking invariants there is a possibility that an invariant could be broken and then reachieved. To ensure this does not occur in CRIKEY, rather than represent invariants as actions in their own right, any snap-action, $a = (cond, add, del)$, selected to add to the plan is required not to delete any invariant of any open envelope in state $s = (F, \xi)$. That is:

$$\forall (A_{\vdash}, B_{\dashv}, C, S) \in \xi \cdot del \cap cond_{\leftrightarrow}(B) = \emptyset$$

where $cond_{\leftrightarrow}(B)$ is the set of invariants of durative action B.

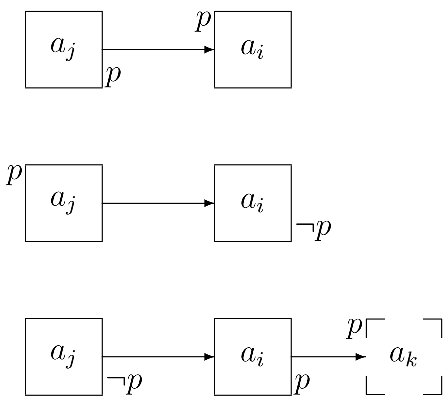
7.3 Applying Actions to States — Temporal Constraints

As well as considering the logical consequences of action selection, it is also necessary to consider the temporal consequences: that is, whether the envelopes ξ remain consistent when updated to reflect an action selection. In FF, a logical formula can ascertain whether an action can be applied (i.e. its preconditions are satisfied) and a distinction is made between simply testing for applicability and actually applying an action. However, when working with temporal constraints such a distinction cannot be made: the act of determining whether an action is applicable is performed by attempting to apply the action and detecting inconsistencies amongst the resulting updated temporal constraints. What follows is a discussion of how the envelopes in Crikey are updated to reflect the attempted application of a snap-action: if the process fails then the action is deemed to be inapplicable, but if it succeeds, the action is applicable and the envelopes in the successor state have been found.

A key facet of envelope maintenance in Crikey is the 'CheckOrder' function, presented in Definition 9. CheckOrder determines whether there is an interesting interaction between two snap-actions based on their preconditions and effects. Its logic is based on the Partial-Order-Lifting algorithm which is discussed fully in Section 7.4.1. That algorithm depends on three reasons for enforcing an action ordering $a_j \prec a_i$, illustrated in Figure 8 and CheckOrder returns true if any of these apply.

Definition 9 — CheckOrder Function
The function CheckOrder(a, b) applied to snap-actions, a and b, returns the value true iff there is an interaction between two actions $a = (cond\text{-}a, add\text{-}a, del\text{-}a)$ and $b = (cond\text{-}b, add\text{-}b, del\text{-}b)$ that indicates a should precede b. The function is defined to return the value of the expression:

[3] This is a standard declobbering technique used in partial order planners. Another is to order $a_k \prec a_j$, however, the Partial-Order-Lifting algorithm does not allow for this as it can only *relax* the total order it starts with, not add new constraints to it. If $a_j \prec a_k$ is in the total order it is not possible to have $a_k \prec a_j$ in the lifted partial order.



(a) a_j achieves a precondition of a_i so $a_j \prec a_i$

(b) a_i deletes a precondition of a_j so $a_j \prec a_i$

(c) a_j deletes a proposition that is an effect of a_i used to achieve a third action, a_k. Here $a_j \prec a_i$. If this ordering is not asserted, then a_j could follow a_i, deleting the precondition of a_k.[3]

Fig. 8. The Three Reasons to Order Actions

$$
\begin{aligned}
&a \text{ and } b \text{ do not start or end the same action} \wedge \\
&\qquad\qquad (\; \textit{add-a} \cap \textit{cond-b} \neq \emptyset \\
&\qquad\qquad \vee\;\; \textit{cond-a} \cap \textit{del-b} \neq \emptyset \\
&\qquad\qquad \vee\;\; \textit{del-a} \cap \textit{add-b} \neq \emptyset \;)
\end{aligned}
$$

7.3.1 Application of Start Actions

Applying a start snap-action $A_\vdash$ is a two-step process. The first step is to check that the action choice is consistent with all the open envelopes in the current state, ξ. Each envelope $e \in \xi$ is updated using the *update* function, defined as follows:

Definition 10 — Update Envelope
Given an envelope $E = (A_\vdash, B_\dashv, C, S)$ and a snap-action a, then $update(E, a)$ is the new envelope $E' = (A_\vdash, B_\dashv, C', S')$, where:

$$
\begin{aligned}
C' &= C \cup \{a\} \\
S' &= S \cup \{\varepsilon \leq a - x \mid x \in C \setminus \{B_\dashv\} \cdot \text{CheckOrder}(x, a)\} \\
&\quad \cup \{\varepsilon \leq B_\dashv - a \mid \text{CheckOrder}(a, B_\dashv)\} \\
&\quad \cup \{da(a)_{dur} \leq da(a)_\dashv - da(a)_\vdash \leq da(a)_{dur}\}
\end{aligned}
$$

if $\exists x \in C. A_{\vdash} \prec_S x \wedge \text{CheckOrder}(x, a) \vee \text{CheckOrder}(a, B_{\dashv})$ and otherwise:

$$\begin{aligned} C' &= C \\ S' &= S \end{aligned}$$

where $da(a)$ is the durative action from which a is derived.

After each envelope E has been updated, the consistency function (Definition 8) is used to ascertain whether E' remains consistent: if it does not, the process terminates and the start action is deemed to be inapplicable.

After having updated the existing envelopes to reflect the action choice (and assuming they remain consistent), producing a set of envelopes ξ', the second step is to create new envelopes. Adding a start action $A_{\vdash}$ always creates at least one envelope: the envelope bounded from $A_{\vdash}$ to the corresponding future end action $A_{\dashv}$. This can be defined formally, as the *emptyenvelope* function, as follows:

Definition 11 — New Empty Envelope
An empty envelope associated with the start action, $A_{\vdash}$ is created by the function $\mathit{emptyenvelope}(A_{\vdash}) = (A_{\vdash}, A_{\dashv}, \emptyset, \emptyset)$.

Additional new envelopes are then created from each $E = (x, y, C, S) \in \xi'$ if $\mathit{CheckOrder}(A_{\vdash}, y)$. Each such new envelope E' is a copy of the original envelope, but with its end action set to the corresponding end of the new action, $A_{\dashv}$. A temporal constraint is also added for the duration of this new action, and also to specify that the new envelope actions are of the maximum separation type. This whole process is defined formally in the *expenv* function, as follows:

Definition 12 — Expand Envelope
Given an envelope, $E = (A_{\vdash}, B_{\dashv}, S, C)$ and a start snap-action, $X_{\vdash}$ such that $\mathit{CheckOrder}(X_{\vdash}, B_{\dashv})$, the result of expanding E by the addition of $X_{\vdash}$ is given by the function $\mathit{expenv}(E,\ X_{\vdash}) = E'$ where:

$$E' = (A_{\vdash}, X_{\dashv}, C \cup \{X_{\vdash}, X_{\dashv}\}, S')$$

and:

$$\begin{aligned} S' = S & \\ & \cup \{\varepsilon \leq X_{\dashv} - x \mid x \in C \cdot x \prec_S B_{\dashv}\} \\ & \cup \{A_{dur} + X_{dur} \leq X_{\dashv} - A_{\vdash} \leq A_{dur} + X_{dur}\} \\ & \cup \{\varepsilon \leq B_{\dashv} - X_{\vdash}\} \end{aligned}$$

Conceptually, the envelopes created in this manner serve to capture the interactions in compound envelopes, rolling out an extended envelope between what previously was the start and the new end point (the end of the action whose start has just been applied). As before, when adding actions to existing envelopes, any newly created envelopes are checked for consistency using the consistency function (Definition 8) and if a new envelope is found to be inconsistent, the process aborts and the start action is deemed to be inconsistent.

7.3.2 Application of End Actions

Applying an end action $A_\dashv$ is a two-step process, but is less involved than the process of applying a start action as there is no need to create new envelopes: clearly, ending an action cannot create a window of opportunity in which something can happen. The first step is to close any envelopes whose end point corresponds to $A_\dashv$. This produces a new set of open envelopes ξ' where $\xi' \subset \xi$. At least one envelope will be erased (that from $A_\vdash$ to $A_\dashv$) so ξ' will certainly be smaller than ξ.

Secondly, the remaining envelopes in ξ' are updated to reflect the fact that $A_\dashv$ has been applied, using the *update* function presented earlier in Definition 10. The *update* function is applied to each envelope $e \in \xi'$ to produce envelopes ξ''. As with start actions, if the consistency function (Definition 8) indicates an envelope has become inconsistent, the process terminates and $A_\dashv$ is deemed to be inapplicable. Otherwise, the envelopes ξ'' correspond to those in the state reached by applying $A_\dashv$.

7.3.3 Goal States

The requirement that end actions must be applied to complete every action that is started in a plan, before the plan is concluded, leads to the need to encode this constraint in the goal state. The **goal** of search in CRIKEY is to find a path from the vertex denoting the initial state to a state $G = (F,\ \xi)$, where F satisfies the goals specified in the planning problem *and* $\xi = \emptyset$.

7.4 Scheduling

During search, CRIKEY only communicates with the scheduler where absolutely necessary and only on that part of the plan where there is danger of producing an unschedulable plan. This communication occurs in the form of the temporal constraints encoded in the STN in each envelope. In this manner,

Input: TO-Plan: A list of actions $\langle a_1, \ldots, a_n \rangle$
Output: PO-Plan: A set of orderings between actions $\{a_i \prec a_j\}$

    for $i = n$ down-to 1 **do**
        (a) **for each** $p \in precond(a_i)$ **do**
            Find *an* action a_j where $p \in add(a_j)$
            Add an ordering $a_j \prec a_i$
        (b) **for each** $d \in del(a_i)$ **do**
            Find *all* actions a_j where $d \in precond(a_j)$
            Add an ordering from all actions $a_j \prec a_i$
        (c) **for each** $p \in primary\_add(a_i)$ (in the goal/sub-goal chain) **do**
            Find *all* actions a_j where $p \in del(a_j)$
            Add an ordering from all actions $a_j \prec a_i$

Fig. 9. The Partial-Order-Lifting Algorithm to Translate Totally Ordered Plans to Partially Ordered Plans

Crikey can deal with all types of envelopes, including those that are many actions in length: if, when putting a content action in the envelope, there is a maximum separation relationship, then a new envelope (many actions long) is created according to Definition 12.

When a goal state G has been found, all the open envelopes are closed and, as in the case with search in FF, the actions along the edges from the initial state to G represent a solution plan. From this totally ordered plan, in terms of action starts and action ends, we wish to find a solution plan in terms of time-stamped durative actions. To do this, a partial order is lifted from the totally ordered plan, and the temporal plan is inferred from this.

7.4.1 The Partial Order Lifter

The Partial Order Lifter takes a totally ordered plan and converts it into a partially ordered plan. It is an implementation of the Partial-Order-Lifting algorithm [30] (sketched out in Figure 9) that takes advantage of the total ordering of the sequential plan by only visiting earlier actions in the plan on each iteration of the algorithm. It removes unnecessary precedence orderings from the total order to produce a partial order. The total order plan *is* a valid partial order plan, due to the consistency enforced by the use of envelopes during search, so in the worst case no precedence orderings will be removed and the original total order is returned. The algorithm finds concurrency where possible.

When reasoning about the split envelope actions the dummy propositions (described in Section 5) ensure that start and end pairs are ordered correctly with respect to each other. The Partial-Order-Lifting algorithm is a greedy polynomial algorithm that does not necessarily find the best (temporally shortest) partial order. The greedy policy of selecting the latest possible achiever in the plan removes the requirement for search at step (a), that would otherwise be required in order to optimise the solution.

7.4.2 The STN

Definition 13 — Conversion of a Partial Order to an STN
A Partial Order $pop = (ia, pr)$ where ia is a set of instantaneous STRIPS Actions and pr is a set of precedence relations between the members of ia, is converted into a set of temporal constraints tc such that

$$\begin{array}{ll}
\text{(a)} & \forall a_i \prec a_j \in pr \cdot \{\varepsilon \le a_j - a_i \le \infty\} \in tc \\
\text{(b)} & \forall a_i \preceq a_j \in pr \cdot \{0 \le a_j - a_i \le \infty\} \in tc \\
\text{(c)} & \forall a_i \in ia \cdot \{\varepsilon \le a_j - X_0 \le \infty\} \in tc \\
\text{(d)} & \forall a_{\vdash} \in ia \cdot \{a_{dur} \le a_{\dashv} - a_{\vdash} \le a_{dur}\} \in tc
\end{array}$$

where $X_0 = 0$ and represents the start of the plan. Part (a) of Definition 13 ensures that timepoints that are in strict precedence must be separated by at least ε (the tolerance value that specifies the minimum separation between mutex happenings — see [1] for a full account of the significance of this value). Timepoints that are not in strict precedence can happen simultaneously (part (b)) — this could happen where an ordering is due to an invariant condition rather than a start or end condition. Part (c) constrains each action to start after the start of the plan (X_0). Each corresponding start and end action must have a constraint, made by part (d), for their duration. These constraints take the model of time from a point-based, back to an interval-based model.

To calculate the earliest possible start time for each action, the shortest distance must be found between the action's start timepoint and X_0 in the network. Floyd-Warshall's All Pairs Shortest Path algorithm is used once and is of complexity $O(n^3)$ (where n is the number of timepoints.) Bellman-Ford's Single Source Shortest Path would have to be used repeatedly — once for each action ($\frac{n}{2}$) — making the complexity $O(\frac{n}{2}) \times O(ne) = O(n^2 e)$ (where e is the number of constraints). Since there are *at least* n constraints (one for each timepoint to make it follow the start — see part (d) of Definition 13) this makes the complexity at least $O(n^2 n) = O(n^3)$. As there will be in fact more constraints from precedence relations and duration constraints, the complexity will be greater than this, and so it is more efficient to use Floyd-Warshall's.

Scheduling needs to be more sophisticated to handle actions with variable durations, especially when such actions can generate a variable amount of a given resource. The resource reasoning is performed with precedence graphs. This is not a novel technology, but rather an new application of it. Precedence graphs are summarised below and described in full in [32]. The rest of this section describes how they are integrated into CRIKEY including the changes to [32] that had to be made, followed by an example of how they operate.

Most resource scheduling approaches reason with the actual timing bounds of actions. However, Precedence Graphs look at their relative positions. Each resource in the plan has its own graph, where the nodes are action end points that contain either a condition relating to that resource, or a resource operator in the effect. Each node is labelled with the minimum and maximum production or consumption of the resource at that node. Edges between the nodes are precedence orderings. These graphs need not be represented explicitly but can be deduced from the STN that holds this information.

The "balance constraint" is calculated for each node in each graph [4]. The basic idea of the balance constraint is to compute a lower and upper bound on the resource level just before and just after each event (i.e. $x \pm \varepsilon$). To calculate an upper bound, all maximum production levels of all events that *could* happen before the event are summed with the minimum consumption levels of all events that *must* happen before the event. In a similar way the other balance constraints are calculated.

In fact, precedence graphs as described in [32] use a slightly different model of resources to PDDL2.1. In that model, all resources have a maximum possible level and a minimum possible level that is always zero. PDDL2.1 does not explicitly model resources, and does not have maximum and minimum possible levels encoded in. Instead, the resources must meet conditions which can change from action to action. This has the effect of changing the minimum and maximum possible levels of the resource throughout the plan.

For example, the model used in [32] would specify a fuel tank to have a minimum level of zero and some constant maximum capacity. In PDDL2.1, this maximum capacity can change during the plan, as can the minimum.

For this reason, some simple changes are made to the reasoning presented [32].

[4] For reservoir resources (as PDDL2.1 fluent variables are), the balance constraint requires the resource to be closed, i.e. there are no more nodes to be added to the graph. This is the case in CRIKEY, since the resource reasoning is performed after the planning is complete.

Instead of calculating balance constraints at every node in the graph, it only calculates them for those nodes that contain conditions. The maximum and minimum levels must then meet these conditions, (and not, as in the model in [32], keep the maximum and minimum between zero and the maximum level). Secondly, when calculating the minimum and maximum values, it only considers nodes that contain resource operators.

The balance constraints can then be used to discover:

- dead ends
- new precedence relations
- new bounds on resource usage
- new bounds on time variables

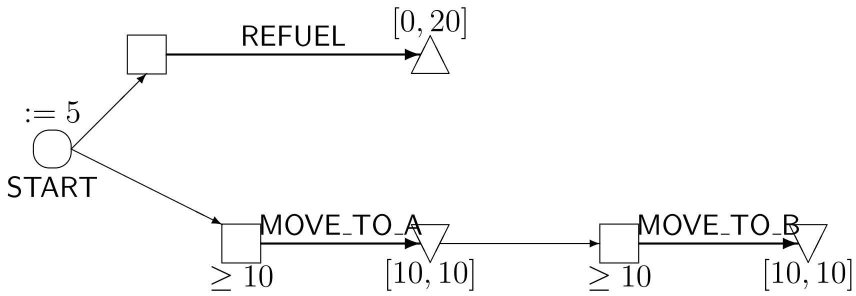
Dead ends (where the conditions cannot be met) are not found in CRIKEY, since it keeps track of metric values during the planning phase to ensure that there is always adequate resource. Resource reasoning is not separated out (unlike the temporal reasoning) so there is no chance of finding an un-schedulable plan due to lack of resources. In the worst case, the precedence graphs will order all the actions identically to the total order plan produced. However, it will find concurrency where possible.

CRIKEY does discover new precedence relations. For each condition, it is made sure that either the maximum and minimum resource levels must meet the condition and if not, precedence relations are put in to ensure that the condition is met (by ordering producers or consumers to occur before the condition).

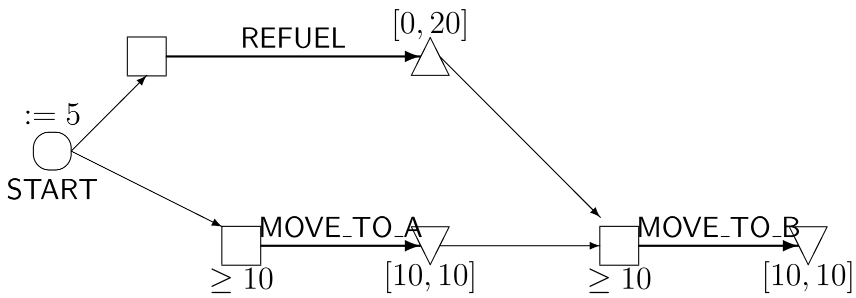
CRIKEY can use the balance constraints to find new bounds on both the time variables (which can be propagated through to the STN) and resource usage variables. This only occurs where there are duration inequalities in the domain, as this is the only case where operators in the plan can produce or consume variable amounts of resource with actions of variable duration.

An example precedence graph is given in Figure 10(a) for the fuel level of a car. There are two move actions, both of which consume between 10 units of fuel. There is also a refuel action (not presently ordered with respect to the move actions) that can produce between 0 and 20 units of fuel (depending on the length of the action).

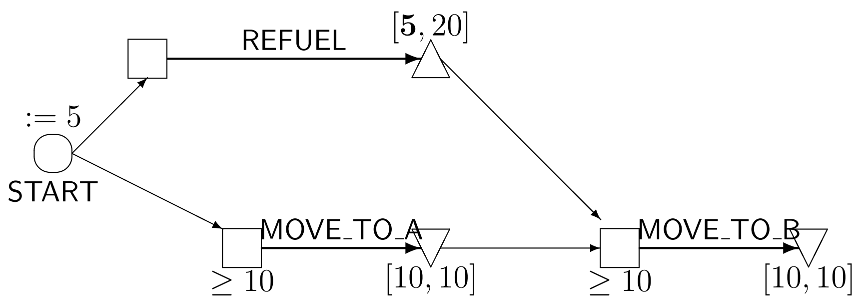
Firstly, in Figure 10(b), the precedence graph is able to reason that the REFUEL action must happen before the second MOVE_TO_B action and so the appropriate precedence relationship is added. This is turn allows reasoning for the resource bounds of the REFUEL action, as it must now produce a minimum of 5 units. The refuel action must now be of sufficient length to supply the 5 units, and this information can be propagated up to the STN.



(a) Precedence Graph for the Fuel Level of a Car



(b) A Precedence Relationship is Added



(c) The Resource Bounds change

Fig. 10. Example Precedence Graph

7.5.1 Duration Inequalities

PDDL2.1 allows the specification of duration inequalities. Rather than fixing the duration of a durative action, these allow bounds to be put on the duration. These bounds can be a function of other metric values (for example, one cannot drive for longer than the amount of fuel available). However, resource change can also be dependent on the duration of an action (for example, the longer one heats water, the hotter it becomes). The duration of an action now effectively becomes a hidden parameter of the action. This allows resource change to be decided by the planner. For example, it is possible to decide how long to fill the tank up (the duration of the the refuel action) and so therefore how full the tank is at the end of the action. The possible combinations are summed up in Table 1.

Specification	**Example**	**Notes**
Durations		
(a) Fixed	`(= ?duration 5)`	The duration of the action is always known and does not change.
(b) Function	`(= ?duration (fuel ?t))`	The duration of the action will depend on the state.
(c) Condition	(≤ `?duration (fuel ?t))`	The duration is a choice of the planner.
Resource Conditions and Operators		
(d) Fixed	(≥ `(fuel ?t) 0)` `(increase (fuel ?t) 3)`	The value of the operator or condition is always known and does not change.
(e) Function	(≥ `(fuel ?t) (fuel_required ?t))` `(decrease (fuel ?t) (fuel_used ?t))`	The value of the operator or condition is dependent on the state.
(f) Function of Duration	`(increase (fuel ?t) (* (refuel_rate) ?duration))`	The resource change is dependent on the duration.
Combinations		
(f) & (b)		equivalent to (e)
(f) & (c)		The resource change is a choice of the planner

Table 1
Possible Specifications of Durations and Resource Conditions and Operators

The (c) and (f) cases then present resource scheduling problems where it would intuitively seem illogical to decide exactly how long an action should be and exactly how much resource should be produced or consumed until after the

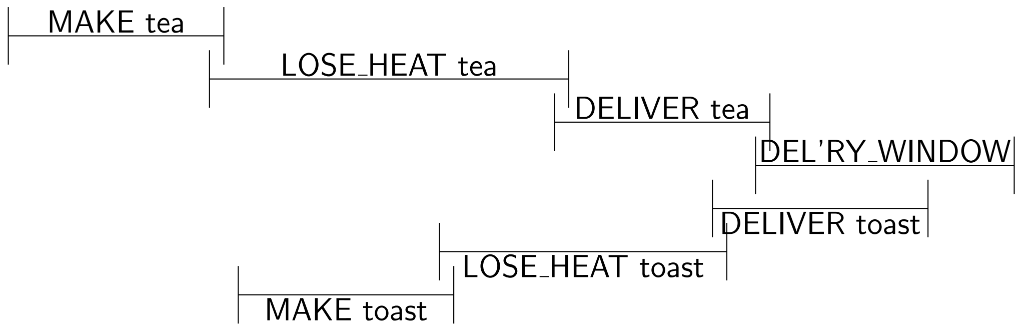


Fig. 11. A Partial Order for the Café Domain

plan is produced (i.e. the problems should be separated out). CRIKEY provides the ideal architecture for this since both the STN and the precedence graphs handle upper and lower bounds on both resource production and consumption and also on time. Through these, contents can be made to fit exactly in envelopes, and resources can be maximised and minimised. For example, in the match domain, if the duration of the match is set to `:duration (<= ?duration 8)` it would be possible to "blow out" the match once the fuse is fixed.

CRIKEY reads the quality metric in the PDDL2.1 problem file to decide what to maximise or minimise in the precedence graphs. This could be a resource or the total time. If it is a resource that is to be maximised, then that precedence graph is selected and the producers maximised and the consumers minimised (by changing the duration of their corresponding actions). If it is to be minimised, then the converse happens. After calculating this, CRIKEY propagates the results through to the STN and the other precedence graphs. If it is the total-time to be minimised, then the duration of each durative action is set to its minimum. The default behaviour is to minimise the total-time and the resource levels.

An example of this is the Café Domain (see Appendix 12.2) where the object is to deliver breakfast to a table in a café, as drawn diagrammatically in Figure 11[5]. However, due to there only being one electrical socket in the kitchen, the toast and the tea cannot be made simultaneously. Once either is made, it starts to cool, until delivered to the table. Whilst it is preferable to have them as hot as possible when delivered, it is also preferable to deliver them at the same time (or as close to each other as possible). There are three possible metrics, one is to minimise the heat lost by each item whilst it is in the kitchen, another is to have them delivered as close as possible together (i.e. minimising the delivery window), and finally simply to minimise the total-time of the whole plan.

[5] This domain contains maximum orderings (the LOSING_HEAT and DELIVERY_WINDOW actions) and so also requires coordination.

(:metric minimize (total_delivery_window))	(:metric minimize (total_heat_lost))
0.01: (MAKE_TEA tea1 socket1) [1.00]	0.01: (MAKE_TEA tea1 socket1) [1.00]
1.00: (LOSING_HEAT tea1) [2.04]	1.00: (LOSING_HEAT tea1) [0.03]
1.02: (MAKE_TOAST toast1 socket1) [2.00]	1.01: (DELIVERY_WINDOW table1) [4.03]
3.01: (LOSING_HEAT toast1) [0.03]	1.02: (DELIVER tea1 table1) [2.00]
3.02: (DELIVERY_WINDOW table1) [2.02]	1.02: (MAKE_TOAST toast1 socket1) [2.00]
3.03: (DELIVER tea1 table1) [2.00]	3.01: (LOSING_HEAT toast1) [0.03]
3.03: (DELIVER toast1 table1) [2.00]	3.03: (DELIVER toast1 table1) [2.00]
Total Delivery-Window: 2.02	Total Delivery-Window: 4.03
Total Heat-Lost: 2.07	Total Heat-Lost: 0.06

Fig. 12. Two Plans with Identical Goals but Different Metrics

For each metric the same partial order plan is lifted, with the same bounds on both the resource levels and the action times. However, if the first metric is chosen, then the LOSING_HEAT actions are minimised. This has the effect of delivering the tea and toast as soon as they are made. This is propagated through to the precedence graph with the DELIVERY_WINDOW, which will mean this can no longer be as short as it could have been. Then, by default the DELIVERY_WINDOW is minimised and then the total-time. If the second metric is chosen, first the DELIVERY_WINDOW action is minimised (resulting in the tea waiting and cooling whilst the toast is prepared) and then the LOSING_HEAT actions are minimised. Finally, if the total time is to be minimised, the precedence graphs are ignored, the actions' duration minimised, and then the earliest start times chosen for each action. Figure 12 shows two plans. One where the heat lost is minimised, and one where the delivery window is minimised.

Some assumptions were made in the implementation of the precedence graphs that limit what can be expressed in the problem. Firstly an operator affecting a resource cannot cause a change that is a function of another resource that is also a dependent on the duration of some activity. This means that once a change has been made in a precedence graph (i.e. a new resource bound found or a new limit on the duration of an action), it will only propagate up to the STN and will not affect any other resource changes in other precedence graphs. There is no reason why Crikey cannot be extended to relax this assumption, meaning that propagation would also be required between precedence graphs, but we leave this as future work. Secondly, resource change that is a function of the duration cannot be a binary function of the duration. Once again, this condition could be relaxed, but has been exploited for ease of implementation. Finally, the metrics in PDDL2.1 allow functions of resources to be optimised, but this implementation only allows for a single resource to be optimised. In practice, few planners are capable of multi-objective optimisation.

8 Crikey$_{SHE}$

Crikey implements a general solution to the problem of managing temporal actions in envelopes, but it relies on costly reasoning to manage envelopes, even in cases where they are not actually required. In fact, many domains do not require the kind of coordination that Crikey is intended to support and, even amongst examples that do, the envelopes that arise are much simpler than Crikey's machinery is designed to manage. For this reason, a reduced version of Crikey has also been developed, capable of handling a specific type of envelope that arises most frequently in domains we have considered.

8.1 Restricted Envelopes

The vast majority of action interactions in planning domains arise when an add effect of one action achieves a start condition of another, so must precede it. It is much rarer to find examples of envelopes that impose constraints on content actions (such as occurs in the LIGHT_MATCH action) involving start effects and end conditions (as already noted — none appear in benchmark domains). In fact, the most significant envelope for encoding time-limited resource availability, including deadlines, is a simple single action that adds the resource at its start and removes it again at its conclusion. This motivates the following definition:

Definition 14 — Single Envelope
An envelope $E = (A_{\vdash}, B_{\dashv}, C, S)$ is a Single Envelope iff $A = B$.

The structure of single envelopes and their contents are shown as examples (a), (b) and (c) in Figure 7. Longer envelopes, such as those shown in in examples (d) and (e) in Figure 7, are more complex and cannot be captured by single envelopes.

As Crikey develops a sequential plan, some envelopes are created that correspond to situations in which one set of actions produce time-limited resources for a concurrent collection of (content) actions. In these cases, all the content actions are constrained to fall inside the limits of the envelope. In other cases, constraints place some of the snap-actions in an envelope after the start of the envelope and others before the end of the envelope, but not necessarily both. We distinguish the following case:

Definition 15 — Hard Envelopes

An envelope, $E = (A_{\vdash}, B_{\dashv}, C, S)$ is a Hard Envelope if:

$$\forall x \in C \cdot A_{\vdash} \prec_S x \prec_S B_{\dashv}$$

When hard envelopes are required in the solution of the planning problem, coordination *must* occur in order for the planning problem to be solvable. It is important to note that when envelopes are not hard, the contents cannot simply "slip" out of the envelope. There must be an ordering between the end points of the envelope and a content action. However, when envelopes are not hard there will be a branching point leading to alternative states in the search space: one with the content inside the envelope and others with the content partially or totally outside the envelope. In the case of hard envelopes, there will only be one accessible state in the search space: that where the content action is in the envelope.

We have developed a more efficient version of CRIKEY, CRIKEY$_{SHE}$, that handles a specific detectable envelope type, the single hard envelope (SHE). Detection of this type of envelope allows the planner to reason efficiently with "blackbox" actions for most of the planning process. The planner only splits actions into start and end actions, according to the LPGP translation, when necessary. This presents a compromise: reasoning about the most likely envelopes whilst maintaining greater efficiency. A single hard envelope arises when there is a time-limited-resource-producing durative action, creating the resource as its initial effect and removing it as its end effect.

Definition 16 — Single Hard Envelope
A durative action, A, generates a Single Hard Envelope iff

$$\emptyset \neq add_{A_{\vdash}} \subseteq del_{A_{\dashv}}$$

The Single Hard Envelope associated with a durative action will indeed lead to the generation of a single envelope and, since it provides a resource that is available only for the duration of the action it will be a hard envelope.

There is a good reason to select this particular envelope type as the basis of specialised treatment. This is because it models a unary resource that is *only* available over a time window. It is common to want to model this. In the case of the match domain, the resource is light which is *only* available during the LIGHT_MATCH action. The *handfree* proposition also models a unary resource. However, the difference between the resources is that the *handfree* resource is always available, *except* during the MEND_FUSE action.

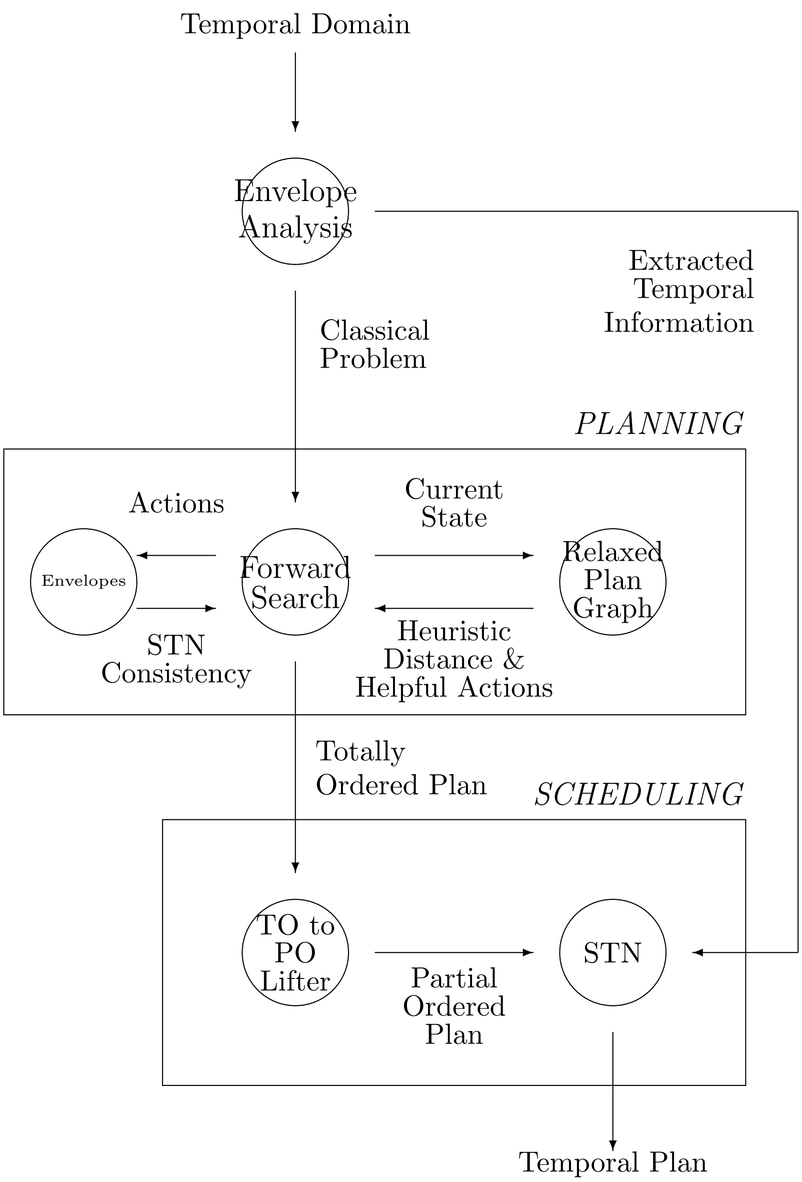


Fig. 13. Architecture Overview of CRIKEY$_{SHE}$

A simple domain analysis step can detect durative actions that generate single hard envelopes in a problem. The next section describes a temporal planner, CRIKEY$_{SHE}$, that can use this analysis to ensure that a valid plan is found, and so solve the match domain problem and other cases where coordination is present in the problem.

Figure 13 shows the overall architecture of CRIKEY$_{SHE}$. It uses the same three-phase action-splitting–planning–scheduling architecture as CRIKEY. However, in the action-splitting phase, only those actions recognised as single hard envelopes (SHEs), according to Definition 16, are split into snap-actions.

The forward-chaining planning phase in CRIKEY$_{SHE}$ proceeds similarly to CRIKEY, searching through a space populated with vertices containing propositional facts and open envelopes. The application of actions to states updates the existing envelopes as before (although see Definition 19 for the modified procedure used in CRIKEY$_{SHE}$), while applying the start of a SHE action creates a new empty envelope. The difference arises in that only *single* hard envelopes are being considered: the expansion of existing envelopes to represent compound envelopes, as described in Definition 12, is never performed. The result of these changes is that the search space size is reduced, and whilst completeness is lost in comparison to CRIKEY, the remaining capabilities are sufficient to reason with SHEs.

The definition of planning states remains as in Definition 7. In contrast to CRIKEY, however, CRIKEY$_{SHE}$ restricts the open envelopes to those in which the start and end points of the envelope correspond to snap-actions derived from the same durative action (in other words, to single envelopes).

The consistency function also remains the same. This check still verifies that actions all fit within the envelope. CRIKEY$_{SHE}$ only needs to consider three types of temporal constraint: (a) the start of the envelope must be ordered before all content actions, (b) all content actions must be ordered before the end of the envelope and, finally, (c) any dependencies between the content actions themselves must be respected through enforcing orderings between actions. To achieve this it relies on a function based on the analysis performed by the Partial-Order-Lifting algorithm (Figure 9), used to identify the necessary temporal constraints. The function, *ordering* is defined as follows:

Definition 17 — Ordering Function
The ordering function, *ordering*(E, a), returns a set of temporal constraints tc between a snap-action a and an open envelope $E = (A_\vdash, A_\dashv, C, S)$ where tc is defined to be:

$$
\begin{aligned}
tc \quad &= \{A_\vdash \prec a \mid \text{CheckOrder}(A_\vdash, a)\} \\
&\quad\; \{a \prec A_\dashv \mid \text{CheckOrder}(a, A_\dashv)\} \\
&\quad \{x \prec a \mid x \in C.\text{CheckOrder}(x, a)\}
\end{aligned}
$$

The same conditions must be imposed to maintain invariants in CRIKEY$_{SHE}$ as in CRIKEY, for all durative actions that are not compressed into a single STRIPS action (i.e. single hard envelope actions). The definition of action applicability changes only to reflect the different way in which the consistency function is now used, still requiring the conditions that (a) the preconditions of the action are satisfied, (b) the action does not delete any currently maintained invariants and (c) the action fits inside the envelope (or does not interact with it).

Definition 18 — Applicability
An action a is applicable in state s if

$$
\begin{aligned}
&\text{(a) } cond \subseteq F \\
\wedge\ &\text{(b) } \forall (A_{\vdash}, A_{\dashv}, C, S) \in \xi \cdot del \cap cond_{\leftrightarrow}(A) = \emptyset \\
\wedge\ &\text{(c) } \forall (A_{\vdash}, A_{\dashv}, C, S) \in \xi \cdot consistent(S \cup ordering(E,\ a))
\end{aligned}
$$

The update envelope function now reflects the simpler definition of an open envelope If there is no ordering between the action and the end of the envelope then the action can safely be scheduled after planning is complete. If, however, the action must occur before the end of the envelope then the constraints this implies must be checked and the envelope must be updated to contain the action.

Definition 19 — Update Envelope (CRIKEY$_{SHE}$)
Given an envelope, $E = (A_{\vdash}, A_{\dashv}, C, S)$ and a snap-action a to add to E, the envelope is modified according to the function: $update_{SHE}(E,\ a) = E'$ where E' is defined as:

$$
\begin{aligned}
E' &= E && \leftarrow ordering(E,\ a) = \emptyset \\
&= (A_{\vdash}, A_{\dashv}, C \cup \{da(a)_{\vdash},\ da(a)_{\dashv}\}, \\
&\qquad S \cup ordering(E,\ a) \\
&\qquad \cup \{da(a)_{dur} \leq da(a)_{\dashv} - da(a)_{\vdash} \leq da(a)_{dur}\}) && \leftarrow \text{otherwise}
\end{aligned}
$$

where $da(a)$ is the corresponding durative action for a

8.2.1 Metrics

CRIKEY$_{SHE}$ has a simpler mechanism for handling PDDL2.1 fluents (metrics) than that used in CRIKEY. Each state keeps a record of the current resource levels. These are changed by the operators in the effects of actions, and tested by conditional statements in the conditions.

The metric aspects have been omitted from the reasoning and definitions for simplicity in the presentation. There are two areas of note when considering metrics in CRIKEY$_{SHE}$. The first is in the compression and splitting of durative actions. Metrics involved in both the start effects and invariants of an action must be treated in a similar way to invariant propositions that are achieved by a start effect and do not become conditions of the compressed or start action (Definitions 5 and 2). For example, if an action has a start effect to increase a resource by 2 and an invariant requiring that the resource be less than 10, then the conditions of the compressed action or start action becomes that the resource should be less than 8.

The second area that metrics complicate is in the lifting of the partial order. Any precedence relationship in the total order between two actions that either test or change the same resource is kept in the partial order.

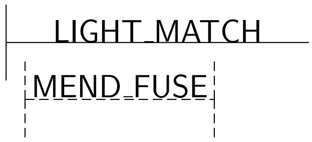
Metrics are incorporated into the heuristic in a similar way to Metric-FF [33]. At each fact layer of the relaxed planning graph, the maximum and minimum possible levels of each resource is calculated based on the values at the previous fact layer and the actions available in the previous action layer. For an action to be applicable in the relaxed planning graph, either the maximum or minimum level must meet the metric condition.

8.2.2 A Worked Example: The Match Domain

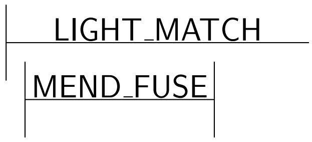
We will now consider a simple problem in the match domain in order to demonstrate how the planning process works in CRIKEY$_{SHE}$. The example concerned has two fuses to be mended, and the burning of a single match does not allow sufficient time for both fuses to be mended sequentially.

CRIKEY$_{SHE}$ performs envelope analysis, discovering that the LIGHT_MATCH action is potentially a single hard envelope action: the effect `have_light` is added by the start action and deleted by the end action. Following this all other actions are compressed to single STRIPS actions, and the LIGHT_MATCH action is compiled into two instantaneous actions: one representing the start of the action and one representing the end. EHC search then begins ignoring temporal information.

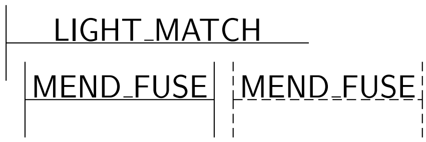
When heuristic evaluation suggests the start action to the LIGHT_MATCH action (a single hard envelope), it will create a new open envelope.



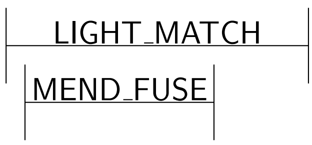
Heuristic search then finds the MEND_FUSE action should be applied next and Crikey$_{SHE}$ will then test to see if a MEND_FUSE action need go in this envelope, and if so, if it is consistent.



Indeed, it fits, so the action is applicable and selected for the plan.



Heuristic search will then suggest the second MEND_FUSE action. This is not consistent with the envelope (there is not enough time left to fix it before the match burns out), so cannot be inserted in the plan. (If the fuses could be fixed in parallel, then this second action would be consistent).



The end of the light action could then be selected and the envelope closed. Crikey$_{SHE}$ would then proceed to either light a second match (and so start a new envelope) or solve another part of the problem. In this way a schedulable plan is produced.

Note that if this match problem were embedded inside a larger problem with other activities, the correct MEND_FUSE would not necessarily be immediately suggested following the LIGHT_MATCH action. If, however, other unrelated actions were selected first, they would not affect the currently open envelope, so when the MEND_FUSE action is eventually chosen it will be able to be added to the envelope.

8.3 Scheduling

Scheduling is performed in the same way as the initial version of CRIKEY: a partial order is lifted using the Partial-Order-Lifting algorithm and the plan is scheduled using an STN. The precedence graph reasoning is not done in CRIKEY$_{SHE}$ as it does not handle durational inequalities.

8.4 Summary of CRIKEY$_{SHE}$

CRIKEY$_{SHE}$ is a much simpler, and more efficient version of CRIKEY that handles the most commonly occurring type of envelope, the single hard envelope. CRIKEY$_{SHE}$ plans in an FF style using the snap-action translation only for single hard envelopes (detected in a preprocessing stage): actions with which concurrency may be *required* to solve the problem. All other actions are translated using the "blackbox" STRIPS action translation as in many other temporal planners. Scheduling and temporal reasoning is done in a post processing phase.

The strengths of this version are its roots in using existing well known planning technology together with the increased efficiency gained by many planners in using the "blackbox" action translation to avoid extra reasoning. The compromise made to achieve these advantages is a loss in completeness compared to CRIKEY. CRIKEY$_{SHE}$ does, however, adhere to the semantics of PDDL2.1 more accurately than most other temporal planners and can still perform competitively.

9 Results

Having described CRIKEY, a general system for solving temporal planning problems involving coordination, and a specialisation of this, CRIKEY$_{SHE}$, that trades some coverage for increased performance, we now evaluate each of them. First, we will perform an evaluation in terms of *capabilities*, comparing the two variants of CRIKEY to a selection of state-of-the-art temporal planners. This is followed by results from the Fourth International Planning Competition (IPC4) in which CRIKEY$_{SHE}$ competed. Finally, since CRIKEY and CRIKEY$_{SHE}$ are specifically designed to plan in domains requiring coordination, but the IPC4 domains require none, we present new domains that do and evaluate planner performance on them. For all comparisons, the planners are run on the same machine with the same resources.

Temporal Planner	PDDL2.2 Timed Initial Literals (TIL)	TIL compiled to PDDL2.1	Single Hard Envelopes	Complex Multiple Envelopes
CRIKEY$_{SHE}$	✗	✓	✓	✗
CRIKEY	✗	✓	✓	✓
Sapa	✗	✗	✓	✗
MIPS	✓	✗	✗	✗
LPGP	✗	✓	✓	✓
LPG	✓	✗	✗	✗
TP4	✗	✗	✗	✗
VHPOP	✗	✓	✓	✓
SGPlan	✓	?	✗	✗

Table 2
Temporal Planner Concurrency Capabilities

Just as CRIKEY and CRIKEY$_{SHE}$ are designed around the idea of coordination between planning and scheduling, so other planners are developed with different motivations. These different specific goals affect the performance of planners across general problems.

9.1 Capabilities

We will now compare the capabilities of a range of state-of-the-art temporal planners against both versions of CRIKEY. Only original planners are used (i.e. not extensions to planners that explore some non-temporal aspect of planning). Also, only planners where there is sufficient documentation or the source code is available are included. The documentation and previously published results are used to determine the capabilities, alongside testing the planners on a simple set of domains with the characteristics under comparison.

Table 2 compares the capabilities of different planners with regard to the complexity of concurrency that they can handle. Only CRIKEY, CRIKEY$_{SHE}$, Sapa, VHPOP, LPGP and SGPLAN can handle domains requiring coordination: MIPS, LPG and TP4 cannot. It is not thought that there are any other temporal planners that are able to (including the SAT-based planners). The planners that cannot find plans in these cases assume a blackbox durative action model, and fail to take into account start effects and end conditions. SGPLAN has limited capabilities for handling TILs compiled into PDDL2.1, which appear to be restricted to the variants used in IPC4.

Sapa uses a slightly different model of durative action to PDDL2.1. Effects can happen at any time during the duration of the action (and so the end effects of PDDL2.1 can be easily translated into Sapa's language). Conditions and invariants can hold for any arbitrary length of time but must start from the beginning of the action. This makes it impossible to correctly translate the end conditions which are not invariants. For this reason, Sapa is marked as not being able to solve envelopes many actions long since this often requires the use of end conditions.

Sapa should be able to plan with coordination where there are single hard envelopes, but in practice it cannot. The reason for this is twofold. When Sapa first finds a plan it does not respect the tolerance value correctly. This is partly because the "advance-time" action would then only take the time forward by ε rather than to the next event in its queue. It post-processes the plan to optimise it and separate the actions by ε. However, this post-processing does not account for start effects (even though Sapa does whilst planning), and so wrongly places the content actions outside the envelope actions. Secondly, Sapa contains a bug which means that, when it first searches for a plan, it can fail to check that an invariant of an action is not deleted by an action already in the queue.

Theoretically, LPGP should also be able to find plans in domains requiring coordination. However, a few modifications to the planner are needed. Often in domains involving coordination, it is the start effects of an envelope action that are required and not the ends. Since LPGP searches backwards in finding a plan, it must choose to place the (unwanted) end action in before it realises that it needs the start of the envelope, and so fails to find a plan.

9.2 The Fourth International Planning Competition

CRIKEY$_{SHE}$ was a participant in the Fourth International Planning Competition (IPC4). The competition was run over a period of approximately three months during which time competitors ran their planners on a series of problems using a Linux PC with two CPUs running at 3GHz. For each problem, planners were limited to 1GB of memory and 30 minutes of CPU time. During the competition, competitors were allowed to modify their planners to correct bugs and optimise them for the competition domains.

There are 7 domains: Airport, Pipesworld, Promela, PSR, Satellite, Settlers and UMTS. These are described below and in more detail in [24]. Each of these have a variety of different formulations such as STRIPS only, fluents, durative actions etc. CRIKEY$_{SHE}$ was able to compete in all but two of the domains, Satellite and Settlers: these two domains were only available in an ADL formulation, which CRIKEY$_{SHE}$ does not support. There is no requirement for coordination in any of the competition domains, except for where PDDL2.2 timed initial literals are compiled into PDDL2.1 domains. In these cases, the dummy actions involved in the compilation require envelopes. The temporal aspect of the domains are further limited since there are no state dependent durations. Planners are compared both in terms of time taken to solve problems and solution quality (in the case of CRIKEY$_{SHE}$ the number of actions in the plan).

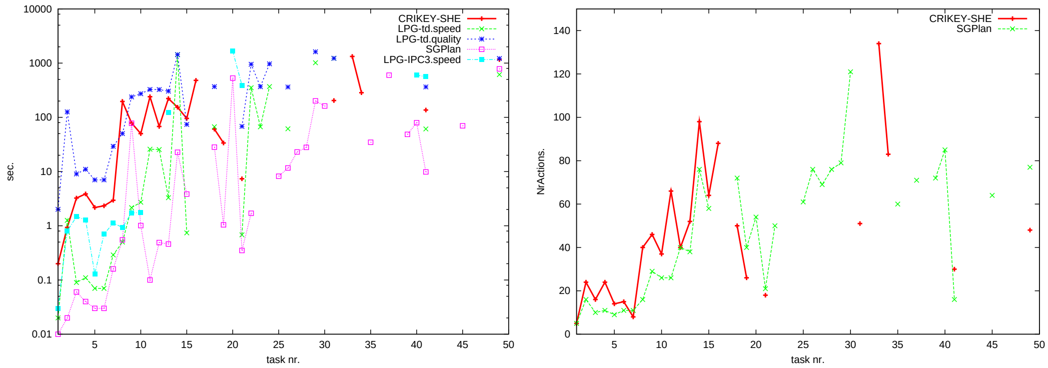
A selection of results from the competition are discussed here to demonstrate that CRIKEY$_{SHE}$ is competitive in general benchmark domains. This shows that the additional reasoning being performed in CRIKEY$_{SHE}$ is not hindering its performance on domains in which the reasoning is not necessary. The performance of CRIKEY$_{SHE}$is made more creditable by the fact that CRIKEY$_{SHE}$was not at all optimised during the competition to fit the competition domains. As competing in these standard domains, with restricted expressivity, CRIKEY$_{SHE}$ represents the integration of well-known and well understood planning and scheduling technologies.

9.2.1 Performance in IPC4

This section highlights interesting aspects of CRIKEY$_{SHE}$'s performance on the IPC4 domains with reference to the other competing planners. For a full analysis of the competition results considering all planners on all domains refer to the IPC4 results paper [24].

The first domain to be considered, PSR, is a domain having only non-temporal formulations; in such domains CRIKEY$_{SHE}$ performs as FF. Overall in this domain there is little difference between the competing planners. No planner performs consistently better than any other, and, with the exception of LPG, the quality is comparable for all planners. The domain shows CRIKEY$_{SHE}$ performing competitively against state-of-the-art planners in classical propositional planning and solving 29 of the 50 problems. The Promela domain is another domain that is available only in non-temporal formulations. In this domain the scalability of CRIKEY$_{SHE}$ was similar to that of Macro-FF and P-MEP but did not equal that of the faster planners; such as FAP, SGPLAN and YAHSP.

The Pipesworld domain requires the planner to control the flow of oil derivatives through a pipeline network, obeying various constraints such as product compatibility and tankage restrictions. CRIKEY$_{SHE}$ competed in four domains, two without resources (no-tankage) and two with resources (tankage). Of these domains, one was non-temporal, the other was temporal. In all these formulations CRIKEY$_{SHE}$ performed competitively showing that it can compete in



(a) Performance

(b) Plan Quality

Fig. 14. Temporal Tankage Pipesworld Domain Results from IPC4

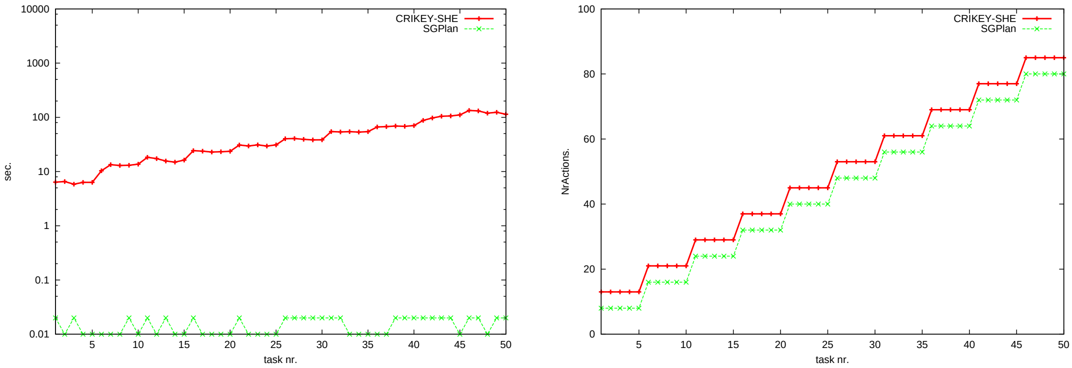
both temporal and metric planning problems. As shown in figure, 14(a), in the temporal metric formulation, the most expressive version of the domain, CRIKEY$_{SHE}$ solves problems that no other planner does. This shows that the decomposition of temporal planning into planning and scheduling is a viable solution.

In the UMTS domain, the task is to set up applications for mobile terminals. The objective is to minimise the time needed for the set up, i.e. to minimise the makespan of the plan. If this objective is ignored then the planning is trivial. CRIKEY$_{SHE}$ competed in three formulations of this domain: a temporal domain, a flawed temporal domain and a temporal domain with time windows which had been compiled down to PDDL2.1.

CRIKEY$_{SHE}$ successfully solved almost all the problems in this domain[6]. Whilst it is not as fast as other planners competing in the standard temporal formulation, its performance degrades at a similar rate and so this could be due to implementation differences. Only two planners competed with the time windows compiled into PDDL2.1: CRIKEY$_{SHE}$ and SGPLAN (see Figure 15). Whilst CRIKEY$_{SHE}$ solved all problems in reasonable time (less than 100 seconds), SGPLAN is faster still. In fact, CRIKEY$_{SHE}$ and SGPLAN were finding the same plans. The reason for the difference in the number of actions is that CRIKEY$_{SHE}$ includes the dummy actions in the total action count, whereas SGPLAN does not. This domain shows that CRIKEY$_{SHE}$ can handle coordination when it is in the form of time initial literals compiled into PDDL2.1.

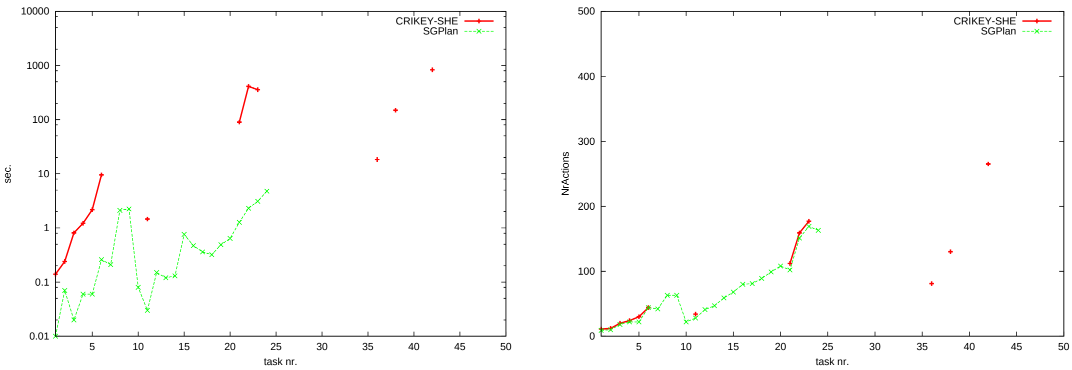
The reasoning being performed by SGPLAN to handle the compiled time win-

[6] Those it did not solve were discovered later to be due to a bug in detecting repeated visited states.



(a) Performance (b) Plan Quality

Fig. 15. Temporal UMTS Domain with compiled Time Windows



(a) Performance (b) Plan Quality

Fig. 16. Temporal Airport Domain with Time Windows

dows domains, involving coordination, is undocumented. Experiments performed using SGPLAN on all other domains involving coordination result in the predictable problem, that the goal cannot be reached as the required effect, present during the envelope action, is compiled out: the action both adds and deletes the required predicate. Our experiments, combined with the facts that SGPLAN discounts the timed initial literal dummy actions from the action count and that renaming the dummy actions causes SGPLAN to produce invalid plans, suggest that SGPLAN recognises the idiomatic compilation as a special case, detecting the timed initial literals and reasoning with them explicitly, in the same way it handles the non-compiled domain. As a consequence, SGPLAN does not need to reason about coordination to solve this problem.

The final domain in which CRIKEY$_{SHE}$ competed is the Airport domain. The purpose of this domain is to control ground traffic in airports, moving planes between gates and runways safely. The largest instances (problem numbers 21–

50) in the test suites are realistic encodings of Munich airport. CRIKEY$_{SHE}$ competed in the non-temporal formulation, the temporal formulation and the temporal formulation with deadlines complied into PDDL2.1.

Again, CRIKEY$_{SHE}$ performs competitively in all formulations of this domain. Where there is a requirement for coordination in the compiled time windows formulations, CRIKEY$_{SHE}$ finds solutions that other planners do not, as shown in Figure 16.

9.3 Analysis Overview of IPC4 Domains

These competition results show that CRIKEY$_{SHE}$ is a temporal planner that performs reasonably well in propositional, metric and temporal benchmark domains. CRIKEY$_{SHE}$'s implementation is not optimised and its design and algorithm are not intended to be outstanding. The requirement for coordination is the focus of CRIKEY$_{SHE}$, not the specific demands of the benchmark problems. However, CRIKEY$_{SHE}$ is more expressive than the other competing planners (as shown in Section 9.1 at the beginning of this section), with the possible exception of the poorly performing P-MEP. By limiting the expressive power of the problems, assumptions are made as to the nature of the problems. These assumption lead to a decrease in the computation necessary which in turn leads to better performance. CRIKEY without the SHE restriction does not make these assumptions, and so whilst it can plan for more domains (as demonstrated in the next section) it pays for it in its performance on some domains.

9.4 Coordination

Having demonstrated that the performance of CRIKEY$_{SHE}$ is comparable to that of recent state-of-the-art planners we now evaluate the envelope reasoning presented in this paper. As none of the standard benchmark domains contain such envelopes we present several new domains to demonstrate the use of such concepts. Many real-world situations require the use of envelope; for example drivers in a logistics problem will not work every hour of the day, but will instead work a finite but moveable shift.

The only planner fully able to handle coordination, producing valid plans and fully obeying the semantics of the language is the partial order planner VHPOP [3]. Sapa can attempt to solve these problems, in so far as a solution is returned, but often the plan it finds is invalid. Therefore, VHPOP is used for comparison with CRIKEY.

The results in this section are produced using the IPC3 competition machine: a Linux PC running at 1400Mhz. The planners had 500MB of memory and a time limit of twenty minutes. This is a significant cut in resources compared to IPC4. To compensate for this the problem instance sizes are smaller. As well as limitations on availability, the reason for the reduction in resources available is that the difficulty in the problems does not come from the size of the instance but from the interaction between the planning and scheduling (i.e. the coordination) and it is of more interest to know whether the planners easily find the solution in the search space (if at all) and not actually how long the planners take.

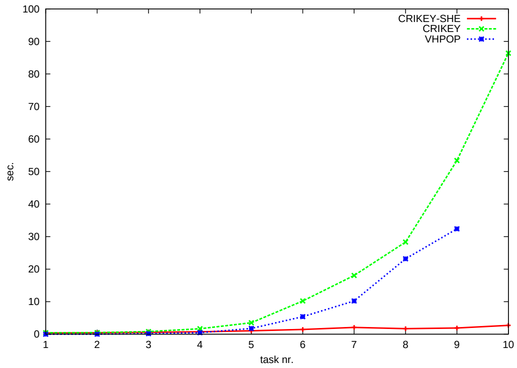
The performance graphs are now on a linear scale (not logarithmic), and the quality is no longer calculated by the number of actions in the plan but by the makespan (temporal length) of the plan. The domains contain required concurrency (in the form of coordination) and so the temporal length of the plan is quite separate from the number of actions in the plan. In effect, content actions are not counted, as it is the envelope actions that account for the length of the plan. It is these actions that a good planner will want to minimise. By comparing the temporal length, it is the scheduler, planner and their interaction which is really being tested; whereas when comparing the number of actions, the scheduler has no impact on the quality. It is also important to compare the temporal length of the plan where there are duration inequalities, since the number of actions will remain the same regardless of their duration.

9.4.1 The Match Domain Revisited

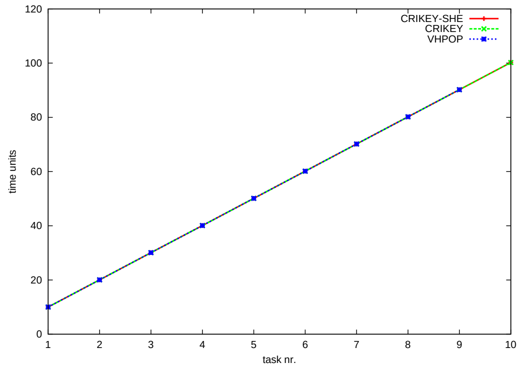
CRIKEY, CRIKEY$_{SHE}$, VHPOP and Sapa were all tested on 4 variations of the match domain, based on the domain initially presented in Section 2 and in full in Appendix 12.2. However, VHPOP and Sapa do not obey PDDL2.1 semantics quite as closely as either of the versions of CRIKEY, and where an invariant of an action is achieved by the start effects of an action (as in the LIGHT_MATCH action), these planners report that no plan can be found. In order to relax the problem specification to allow other planners to be used for comparison the invariants are removed for the purposes of these tests. This does not affect the structure of the domain in terms of coordination, as the fuses must still be fixed within the burning of the match.

Figure 17 presents results for the standard match domain. In this domain it takes five time units to fix a fuse, and a match burns for eight time units. The number of matches and fuses in the instance is twice the task number (e.g. problem number 5 has 10 matches and 10 fuses to fix).

As discussed earlier, Sapa fails to find valid plans. It realises that more than one match is required, but produces plans where two fuses are fixed by the



(a) Performance



(b) Plan Quality

Fig. 17. Standard Match Domain

light of one match, resulting in plans with half the number of matches required.

VHPOP can use a multitude of search strategies, flaw selection preferences and heuristic guidance. Some experimentation was performed to find out which combination work best in the match domain and it was found that A* search with the ADD heuristic and preferring plans with few open conditions is the best overall configuration.

As can be seen, CRIKEY$_{SHE}$ performs considerably better than CRIKEY. This is because in CRIKEY a plateau is reached in the search space when one fuse has been fixed by the light of one match and the remaining goal is to fix another fuse. The heuristic in this case does not guide the planner to close the envelope and light another match. Instead, a small amount of search is performed to discover this, including checking that all of the unfixed fuses are not able to be fixed using the rest of the available light. CRIKEY splits all durative actions into two snap-actions, whereas CRIKEY$_{SHE}$ uses a compressed single-action representation of the FIX_FUSE. For this reason, the size of the search at every plateau where a new match is needed is twice as large in CRIKEY than in CRIKEY$_{SHE}$, and thus finding a solution plan takes longer.

Table 3 shows the percentage of time that both versions of CRIKEY spend on parsing and instantiation, planning, and scheduling. The reason that CRIKEY spends proportionally longer planning than CRIKEY$_{SHE}$ is again attributed to the larger search space. Both versions spend most of their time planning as this is the harder problem to solve. The scheduler is not complex, finding a quick greedy solution.

All planners find the same (optimal) solution.

	CRIKEY$_{SHE}$			CRIKEY		
	Parsing & Grounding	Planning	Scheduling	Parsing & Grounding	Planning	Scheduling
1	33.33%	33.33%	33.33%	33.33%	33.33%	33.33%
2	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%
3	0.00%	100.00%	0.00%	0.00%	66.67%	33.33%
4	0.00%	66.67%	33.33%	12.50%	62.50%	25.00%
5	0.00%	50.00%	50.00%	5.56%	66.67%	27.78%
6	0.00%	37.50%	62.50%	2.94%	76.47%	20.59%
7	8.33%	33.33%	58.33%	0.17%	97.97%	1.86%
8	6.25%	31.25%	62.50%	0.79%	84.25%	14.96%
9	5.00%	30.00%	65.00%	0.37%	83.52%	16.10%
10	4.00%	32.00%	64.00%	0.36%	91.55%	8.09%

Table 3
Percentage of Time Spent in Temporal Planning by CRIKEY and CRIKEY$_{SHE}$ in the Match Domain

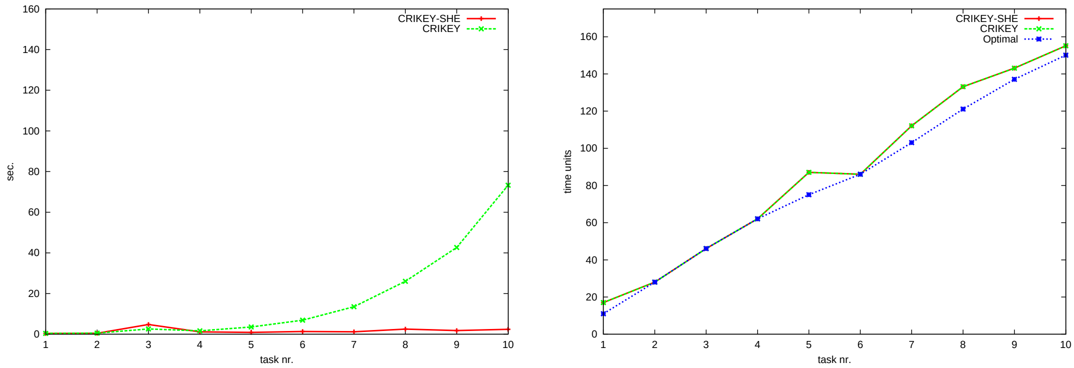
9.4.2 Match Domain with Variable Durations

Figure 18 show the results for a variant of the standard match domain. In this domain, the fuses take different times to mend and different matches also burn for different durations. A fuse that takes a long time to fix must be fixed by the light of a match that burns for a sufficient amount of time. This match must therefore not be wasted on another shorter fuse.

The light match action is changed[7] so that only one match can be alight at any one time. This makes it advantageous to fix as many fuses as possible by the light of one match, in order to minimise the temporal length of the plan. This variant is effectively a bin packing problem.

This variant uses fluents to model the burning time of the match and also the mending time of the fuse. VHPOP cannot handle fluents and so could not be

[7] The proposition light no longer takes the match providing it as a parameter. This prevents two LIGHT_MATCH actions executing concurrently, as one would delete the "light" from the other when they burnt out. Whilst this may not be what is intended, it does mean that the FIX_FUSE action does not need to specify where the light comes from to fix the fuse. Ideally a combination of these two models is needed: two matches could burn at once and not delete each other's light at the end of the action, whilst also not specifying where the light comes from for the fix fuse action. To model this, conditional effects are needed which neither CRIKEY nor the other planners are able to handle.



(a) Performance (b) Plan Quality

Fig. 18. Variable Time Match Domain

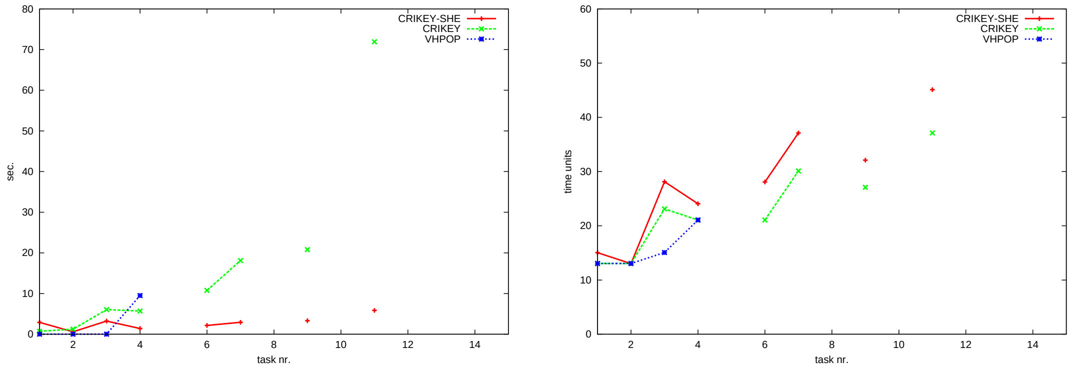
tested on this domain. Again, Sapa produces invalid plans, but is plotted to give an approximate comparison of time.

Once again, CRIKEY and CRIKEY$_{SHE}$ find plateaux in the search space, and this again is the reason why CRIKEY performs worse than CRIKEY$_{SHE}$ since it has a bigger search space to explore at this point.

Figure 18(b) shows the quality of the solutions produced, including the optimal quality achievable. Both versions of CRIKEY produce the same solutions. In some problems, this is the optimal solution. This is usually where the problem is highly constrained and the optimal solution is the *only* solution. (On other problems it may have found the optimal solution purely by accident.) In cases where the problem is highly constrained (i.e. some fuses must be fixed by one of the matches), the planner must perform best-first search in order to find a solution as EHC fails. This is because the heuristic ignores the temporal information and EHC greedily pairs the wrong match with the wrong fuse. In these cases, it takes longer to find a solution.

9.4.3 The Lift-Match Domain

So far, the match domains have only contained coordination. It is more likely that a "real-life" domain with coordination will also have some actions that are not coordinated (i.e. need not happen concurrently). The next variant of the match domain, the 'Lift-Match' Domain reflects this. As before, electricians must fix fuses by the light of matches. The fuses however, are distributed about rooms in a building which the electricians must navigate around using the corridors and lifts. This navigation is not coordinated. Since there is now more than the one electrician, more fuses can be fixed concurrently by the light of one match, so long as the fuses, light and electricians are all present in the same room. For a full description of the domain, see Appendix 12.2.



(a) Performance

(b) Plan Quality

Fig. 19. The Lift Match Domain

Figure 19 shows the results from this domain.

This is a much more complex domain and the planners do not fare so well on it. Again, failure occurs most where the problems are highly constrained and there are fewer matches than fuses. In this case, both electricians must be in the same room at the same time to fix fuses by the light of only one match. In the previous match domains there have only been two operators (LIGHT_MATCH and MEND_FUSE), two types (match and fuse) and four predicates (mended, light, handfree and unused). In this domain there are seven operators, six object types and eleven predicates. This makes the search space bigger and so the problems take longer to solve.

As in the previous match domains, the relaxed plan heuristic is of little help since it ignores delete effects, but the LIGHT_MATCH action deleting the light is critical to plan. As a consequence, Crikey and Crikey$_{SHE}$ often fail in EHC and instead must resort to best-first search. This is a poor search strategy when the problem is big. A more informed heuristic is needed.

In an attempt to reduce the size of the domain, the match objects are turned into a numeric value where only the number of matches unburnt is recorded. Since all matches are functionally symmetrical, this reduces the symmetry in the problem and hence the size of the search space. The LIGHT_MATCH action reduces the number of unused matches by one and has a condition that there is at least one match left (see Appendix 12.2). Figure 20 shows how this reduces the time needed to solve the problems.

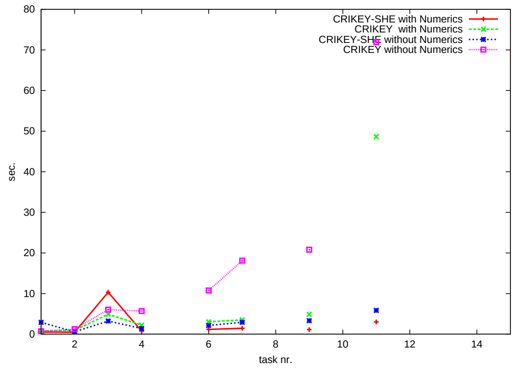
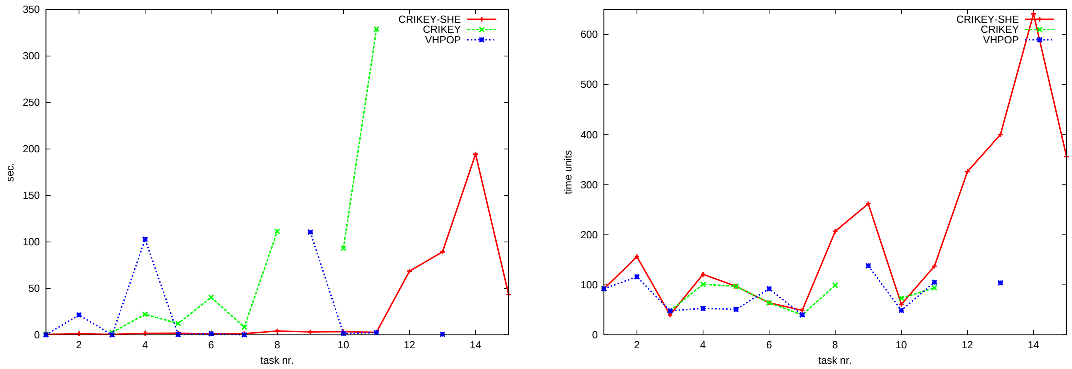


Fig. 20. Performance of CRIKEY and CRIKEY$_{SHE}$ with and without matches encoded using fluents



(a) Performance

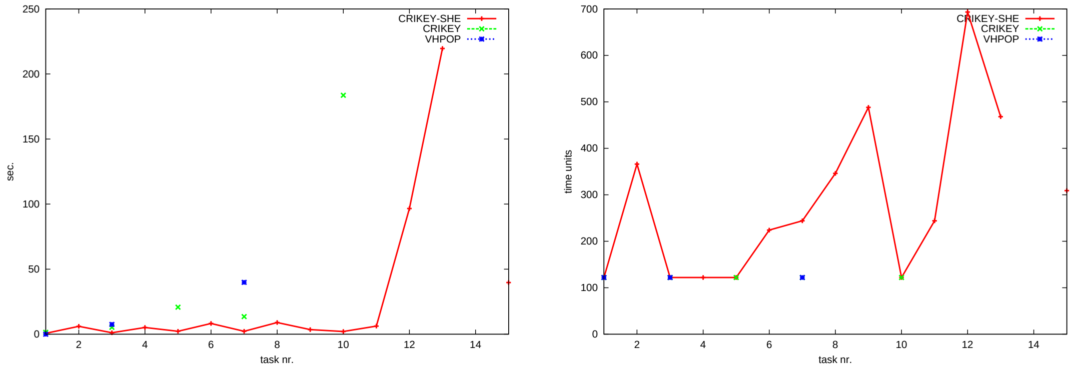
(b) Plan Quality

Fig. 21. Standard DriverLog domain as used in IPC3

10 DriverLog Shift

The Driver Logistics domain (DriverLog, for short) was used at IPC3. It involves moving packages around cities using trucks and drivers to transport them. This domain, with the problems used in the competition, has been transformed to the DriverLog Shift domain (Appendix 12.2), where drivers can only work for a certain amount of time before they must take a break and have a rest. This involves coordination, as the shift action for a driver is an envelope, into which the contents of driving and walking pertaining to the driver in question must fit. This better represents a genuine logistics problem where legislation prevents drivers from driving continually without a break, and also represents that drivers are not available on shift at all times: shift planning is indeed part of the problem.

Figure 21 shows the performance of VHPOP, CRIKEY and CRIKEY$_{SHE}$ on the original first fifteen problems of the 'Simple Time' DriverLog formulation. Figure 22 shows the performance of the planners on the same problems converted



(a) Performance (b) Plan Quality

Fig. 22. DriverLog 'Simple Time' Formulation Converted to use Shifts

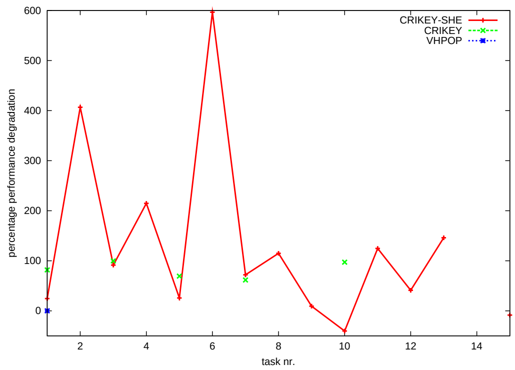


Fig. 23. Degradation of Performance when DriverLog Domain Converted to use Shifts

into shift problems. Figure 23 shows how the performance of the planner deteriorates. This shows how much harder the problems become once the need for coordination is introduced. VHPOP in particular performs much worse even though it is using the flags that worked best on this domain in IPC'03[8].

The search in both variants of CRIKEY must sometimes take arbitrary decisions on which branch of the search tree to explore first where they have the same heuristic value. Problems (e.g. problem 10) where better performance was seen on the shift domain is thought to be due to luckily choosing the correct path at this point.

Since the temporal length of the plan is dictated by the shift envelope action, all planners find the same quality of plan; except in Problem 7 where VHPOP finds a plan that can use one shift fewer.

Table 4 shows the proportion of time spent by CRIKEY and CRIKEY$_{SHE}$ in

[8] VHPOP used grounded actions with A* search, the ADDR heuristic and the MC-Loc-Conf flaw selection criteria.

	CRIKEY$_{SHE}$			CRIKEY		
	Parsing & Grounding	Planning	Scheduling	Parsing & Grounding	Planning	Scheduling
1	33.33%	33.33%	33.33%	50.00%	50.00%	0.00%
2	14.29%	28.57%	57.14%			
3	25.00%	25.00%	50.00%	5.56%	88.89%	5.56%
4	14.29%	28.57%	57.14%	0.88%	97.37%	1.75%
5	10.00%	40.00%	50.00%	2.08%	93.75%	4.17%
6	12.50%	37.50%	50.00%	0.55%	98.90%	0.55%
7	20.00%	20.00%	60.00%	2.56%	94.87%	2.56%
8	5.26%	42.11%	52.63%	0.16%	99.20%	0.64%
9	3.23%	45.16%	51.61%			
10	15.38%	15.38%	69.23%	1.23%	97.95%	0.82%

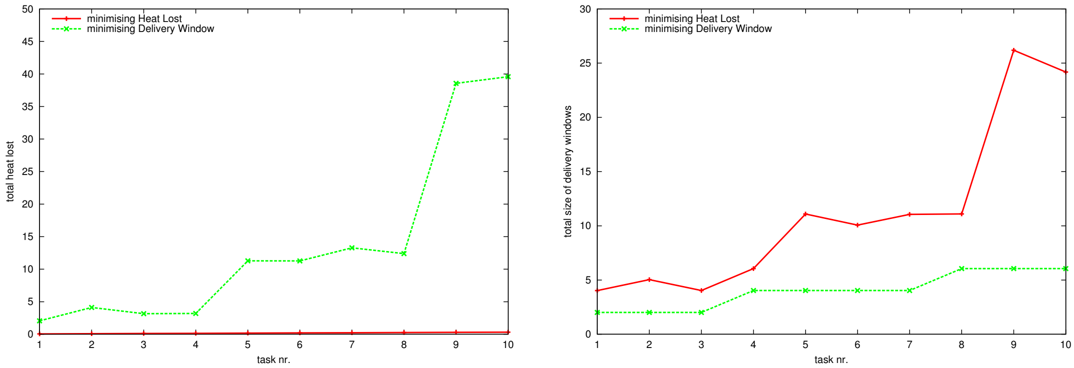
Table 4
Percentage of Time Spent in Temporal Planning by Crikey and Crikey$_{SHE}$ in the Driverlog Shift Domain

the planning and scheduling phases. Again, the Crikey spends proportionally more of its time planning when compared to Crikey$_{SHE}$. This is again due to the increased search space in planning. They are both performing exactly the same task for the scheduling so this would be expected to be equivalent.

Neither the match domain nor the DriverLog Shift domains can be handled by any of the planners in Table 2 that cannot plan with either the single hard envelopes or the complex multiple envelopes. A different variation of the Driverlog Shift domain, originally presented in [34], is one in which the times of the shifts are fixed and cannot not be moved (which is possible in the variation presented here). This fixed shift variation can be encoded using pddl2.2 Timed Initial Literals (TILs) and so any planner able to handle these could tackle that domain. The problem presented here cannot be represented using TILs since the times of the shifts are not known in the initial state: they are choice made by the planner.

11 Using the Metric

Crikey does not hold a queue (or schedule) indicating exactly when future events happen, which makes it possible to extend to use Precedence Graphs and handle domains with duration inequalities. Crikey looks at the given



(a) Minimising Heat Lost (b) Minimising Delivery Windows

Fig. 24. Plan Quality in the Café Domain with Crikey

quality metric to decide on the duration of actions. As with temporal information, this metric is ignored during the planning phase and used only in scheduling, so no guarantee of quality can be given.

Few planners take into account the plan metric (only LPG is claimed to use it in IPC4), but this could be because many temporal domains specify minimising the temporal length of the plan. Only MIPS is known to handle duration inequalities (but, as previously observed, it cannot handle domains with co-ordination).

The goal in the Café domain (as introduced in Section 7.5.1 and in full in Appendix 12.2) is to deliver breakfast (tea, toast and a cooked breakfast) to tables in a café. The plans are constrained in the number of electrical sockets and chefs available in the kitchen. Two possible metrics for this domain include minimising the heat lost by the breakfast items before they are delivered to the table, and minimising the total time window over which items are delivered to a table. The activities in this domain force heat to be lost during delays in preparing the breakfast. This is achieved through the LOOSE_HEAT action, the application of which is forced through dummy conditions in the preparation activities.

Figure 24 shows plan quality with respect to a metric for ten problems in the Café domain. Both graphs show results from exactly the same problems, however in Figure 24(a), the metric is set to minimise the heat lost, and in Figure 24(b), the metric is set to minimise the delivery window. Crikey is trying to minimise the delivery window or heat-loss, as shown in the two different curves.

As can be observed, Crikey finds a better plan with respect to the metric when it considers that metric in the scheduling (as should be expected). In each case (for the four lines) the planner produces the same totally ordered actions, but makes different choices in the scheduling phase when it comes to deciding on the duration of actions where duration inequalities are present.

Again, this domain contains temporal constraints, represented using duration inequalities that cannot be encoded using timed initial literals (since heat loss and delivery windows can occur at any time).

12 Conclusion

Temporal planning is made up of two components: planning and scheduling. Several temporal planners decompose the problem into these two sub-problems. Where these two sub-problems interact, the separate solvers must communicate and this can be expensive, both in terms of CPU time and memory. Many planners, even if they use a PDDL2.1 model of time, assume "blackbox" durative actions where the internal state of an action is not known. This greatly simplifies how the problems can be coupled and does not permit the modelling of coordination, rendering such planners incomplete under the full semantics of PDDL2.1. Those temporal planners that do plan with more expressive durative actions resort back to solving both components at once.

This paper has presented an analysis of where in temporal planning the planning and scheduling components interact. In cases in which this kind of coordination is required, not only is the quality of the schedule affected by the plan, but the possibility of finding a schedule at all. Such difficulties occur where content actions must execute within the duration of envelope actions. It is in these situations, with both logical and temporal constraints, that the sub-solvers must communicate, and this theory is demonstrated in the temporal planner, Crikey.

Crikey does not assume "blackbox" durative actions, but still decomposes the temporal planning problem. Communication between the two sub-solvers is minimised and occurs only where strictly necessary and checking only the part of the plan that contains the coordination. Two versions of Crikey have been presented: the first, Crikey itself, reasons about types of envelope; the second, Crikey$_{SHE}$, reasons only about the most common type, the single hard envelope, but is more efficient.

12.1 Critique of Crikey

Crikey$_{SHE}$ performed competitively in IPC4, while more accurately respecting the temporal semantics of PDDL2.1 than any of the other competing planners[9]. CRIKEY$_{SHE}$ does not depend on the assumption that problems will have no required coordination. In assuming "blackbox" durative actions, other planners are effectively tackling a reduced (and easier) problem, offering opportunities for enhanced performance over both CRIKEY and CRIKEY$_{SHE}$.

CRIKEY has, of course, several weaknesses. Plan quality is one problem: when separating the problems of planning and scheduling the process of solving one problem does not directly feed back in to the solving of the other. This means that plan quality can be poor.

An important limitation of CRIKEY (and CRIKEY$_{SHE}$) is that it cannot solve problems where an action A necessarily needs to occur alongside *itself*: the dummy predicates used to enforce the constraint that $A_{\dashv}$ can only be applied after $A_{\vdash}$ also preclude another $A_{\dashv}$ from being applied again until the termination of A with $A_{\dashv}$. In effect, A has to finish executing before A can start again. In the terminology of recent work due to Rintanen [35], CRIKEY only has one 'counter' for each ground action to mark its execution. However, recent analysis by Fox and Long [36] indicates that the situations in which this limitation applies are restricted and can be managed in alternative ways.

A second limitation of CRIKEY is that the *CheckOrder* function (described in Definition 9) makes a greedy decision about whether an action should be added to the contents of an envelope. As mentioned in Section 8.1, when dealing with soft envelopes where an action does not necessarily *have* to go inside an envelope, there is a branching point over the choices of whether to put it in or not. *CheckOrder* makes a greedy selection over these, adding an action to the contents of an envelope if it achieves a precondition or deletes an add effect of the end of the envelope, on the basis that this will allow the greatest possible number of facts to hold. Due to this, a well-crafted pathological case can cause CRIKEY to find no solution where one exists. Such a case is artificial, however, and relies on peculiar action formulations that do not occur in any of the domains presented in this paper or any of the standard benchmark domains. The greedy selection and the pruning it introduces could be removed from CRIKEY, at a cost to performance, introducing branching as needed.

In the light of these two limitations, it is clear that CRIKEY is incomplete. It does, however, cover substantially more complex temporal interactions than other temporal planners and is able to handle a large, interesting and meaningful subset of temporal planning problems that can be expressed in PDDL2.1.

[9] With the possible exception of tilSapa. There is little documentation of this system but it is based on Sapa. Whilst Sapa should, in theory, be able to solve such problems, in reality it fails.

CRIKEY is still not as efficient as we would like and work is underway investigating combining the approach to temporal planning adopted by CRIKEY with the problem decomposition framework used in TSGP [37]. The idea of using a decomposition approach to planning was highlighted by the participation of SGPLAN in the last two IPCs, where it was demonstrated that a decomposition approach with FF as a subsolver was an efficient means of solving planning problems. By using CRIKEY as a subsolver within TSGP, the aim is to produce a system with the same temporal reasoning capabilities as CRIKEY but with better overall performance.

A further branch for future work centres around the idea of handing continuous numerical effects, occurring across an action's duration, by creating envelopes in which the resources are in certain ranges, or are increasing/decreasing. The behaviour of resources often creates envelopes of opportunity, such as the availability of sufficient solar power for a given task only between two time points, and resource envelopes serve to capture the coordination induced by this behaviour.

References

[1] M. Fox, D. Long, PDDL2.1: An extension of PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 61–124.

[2] A. Gerevini, I. Serina, LPG: A Planner based on Local Search for Planning Graphs, in: Proceedings of the 6th International Conference of Artificial Intelligence Planning and Scheduling (AIPS'02), AAAI Press, Menlo Park, CA, 2002.

[3] H. L. S. Younes, R. G. Simmons, VHPOP: Versatile heuristic partial order planner, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 405–430.

[4] S. Edelkamp, Taming Numbers and Durations in the Model Checking Integrated Planning System 20 (2003) 195–238.

[5] Y. Chen, B. W. Wah, C. Hsu, Temporal planning using subgoal partitioning and resolution in sgplan, Journal of Artificial Intelligence Research (JAIR) 26 (2006) 323–369.

[6] D. Long, M. Fox, Exploiting a Graphplan Framework in Temporal Planning, in: Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS), 2003, pp. 52–61.

[7] A. Garrido, M. Fox, D. Long, A Temporal Planning System to Manage Level 3 Durative Actions of PDDL2.1, in: Proceedings of the 20th UK Planning and Scheduling Special Interest Group (PlanSIG), 2001, pp. 127–138.

[8] M. B. Do, S. Kambhampati, Sapa: a Domain-Independent Heuristic Metric Temporal Planner, in: Proceedings from the 6th European Conference of Planning (ECP), 2001, pp. 82–91.

[9] P. Haslum, H. Geffner, Heuristic Planning with Time and Resources, in: Proceedings of the 6th European Conference on Planning (ECP), 2001, pp. 121–132.

[10] F. Bacchus, F. Kabanza, Using Temporal Logics to express Search Control Knowledge for Planning, Artificial Intelligence 116 (1-2) (2000) 123–191.

[11] P. Doherty, J. Kvarnstrom, TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner, in: Proceedings of the 6th International Workshop on the Temporal Representation and Reasoning, Orlando, Fl. (TIME'99), 1999.

[12] D. Smith, D. S. Weld, Temporal Planning with Mutual Exclusion Reasoning, in: Proceedings of the 16th International Joint Conference on Artificial Intelliegence (IJCAI), 1999, pp. 326–337.

[13] J. Penberthy, D. Weld, Temporal Planning with Continuous Change, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI), 1994.

[14] K. Halsey, D. Long, M. Fox, CRIKEY: A Temporal Planner Looking at the Integration of Scheduling and Planning, in: Proceedings of Workshop on Integrating Planning into Scheduling (WIPIS), 2004.

[15] W. Cushing, S. Kambhampati, Mausam, D. Weld, When is temporal planning *really* temporal planning?, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2007, pp. 1852–1859.

[16] M. D. Rodriguez-Moreno, A. Oddi, D. Borrajo, A. Cesta, IPSS: A Hybrid Approach to Planning and Scheduling Integration, IEEE Transactions on Knowledge and Data Engineering 18 (12) (2006) 1681–1695.

[17] S. Edelkamp, M. Helmert, On the Implementation of MIPS, in: AIPS-2000 Workshop on Decision-Theoretic Planning, AAAI-Press, 2000, pp. 18–25.

[18] B. Srivastava, RealPlan: Decoupling Causal and Resource Reasoning in Planning, in: Proceedings from the 12th Innovative Applications of Artificial Intelligence Conference on Artificial Intelligence (IAAI), 2000, pp. 812–818.

[19] B. Srivastava, S. Kambhampati, Scaling up Planning by Teasing out Resource Scheduling, in: Proceedings of the 5th European Conference on Planning (ECP), 2000, pp. 172–186.

[20] D. Long, M. Fox, Multi-Processor Scheduling Problems in Planning, in: Proceedings of the 2001 International Conference on Artificial Intelligence (IC-AI), 2001.

[21] M. Boddy, A. Cesta, S. Smith (Eds.), Proceedings of Workshop on Integrating Planning into Scheduling (WIPIS), 2004.

[22] D. Smith, J. Frank, A. Jónsson, Bridging the gap between Planning and Scheduling, Knowledge Engineering Review 15 (2000) 61–94.

[23] M. Fox, D. Long, The Third International Planning Competition: Temporal and Metric Planning, in: Proceedings from the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS), 2002, pp. 115–117.

[24] J. Hoffmann, S. Edelkamp, The Deterministic Part of IPC-4: An Overview, Journal of Artificial Intelligence Research (JAIR) 24 (2005) 519–579.

[25] S. Edelkamp, J. Hoffmann, PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition, Tech. rep., Fachbereich Informatik, Germany and Institut für Informatik, Germany (2003).
URL
`www.informatik.uni-freiburg.de/ hoffmann/ipc-4/DOCS/pddl2.2.ps.gz`

[26] A. Gerevini, Y. Dimopoulos, P. Haslum, A. Saetti, Benchmark Domains and Problems of IPC-5, http://zeus.ing.unibs.it/ipc-5/domains.html (2006).

[27] J. F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM (1983) 832–843.

[28] M. Fox, D. Long, K. Halsey, An Investigation into the Expressive Power of PDDL2.1, in: Proceedings of the 16th European Conference of Artificial Intelligence (ECAI), 2004.

[29] J. Hoffmann, B. Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, Journal of Artificial Intelligence Research (JAIR) 14 (2001) 253–302.

[30] M. M. Veloso, M. A. Pérez, J. G. Carbonell, Nonlinear Planning with Parallel Resource Allocation, in: Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, Morgan Kaufmann, San Diego, CA, 1990, pp. 207–212.

[31] R. Dechter, I. Meiri, J. Pearl, Temporal Constraint Networks, in: Proceedings from Principles of Knowledge Representation and Reasoning, 1989, pp. 83–93.

[32] P. Laborie, Algorithms for Propagating Resource Constraints in AI planning and Scheduling: Existing Approaches and New Results, Artificial Intelligence 143 (2) (2003) 151–188.

[33] J. Hoffmann, Extending FF to Numerical State Variables, in: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI), 2002, pp. 571–575.

[34] S. Cresswell, A. Coddington, Planning with Timed Literals and Deadlines, in: J. Porteous (Ed.), Proceedings of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group, University of Strathclyde, 2003, pp. 22–35, iSSN 1368-5708.

[35] J. Rintanen, Complexity of Concurrent Temporal Planning, in: Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS), 2007, pp. 280–287.

[36] M. Fox, D. Long, A Note on Concurrency and Complexity in Temporal Planning, in: Proceedings of the 26th UK Planning and Scheduling Special Interest Group (PlanSIG), 2007.

[37] A. I. Coles, M. Fox, D. Long, A. J. Smith, Planning with Respect to an Existing Schedule of Events, in: Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS), 2007, pp. 81–88.

Appendix A. — The Match Domain

```
(define (domain matchcellar)
     (:requirements :typing :durative-actions)
     (:types match fuse)
     (:predicates
          (light ?match)
          (handfree)
          (unused ?match - match)
          (mended ?fuse - fuse))

     (:durative-action LIGHT_MATCH
          :parameters (?match - match)
          :duration (= ?duration 8)
          :condition (and
               (at start (unused ?match))
               (over all (light ?match)))
          :effect (and
               (at start (not (unused ?match)))
               (at start (light ?match))
               (at end (not (light ?match)))))

     (:durative-action MEND_FUSE
          :parameters (?fuse - fuse ?match - match)
          :duration (= ?duration 5)
          :condition (and
               (at start (handfree))
               (over all (light ?match)))
          :effect (and
               (at start (not (handfree)))
               (at end (mended ?fuse))
               (at end (handfree)))))
```

A problem instance

```
(define (problem fixfuse)
     (:domain matchcellar)
     (:objects
          match1 match2 - match
          fuse1 fuse2 - fuse)
     (:init
          (unused match1)
          (unused match2)
          (handfree))
     (:goal (and
          (mended fuse1)
          (mended fuse2)))
     (:metric minimize (total-time)))
```

Appendix B. — Café Domain

```
(define (domain CafeDomain)
     (:requirements :typing :fluents :durative-actions :duration-inequalities)
     (:types table chef socket - object tea toast cooked_breaky - item)
     (:predicates
          (delivered ?i - item ?t - table)
          (d_w_available ?t - table)
          (d_w_open ?t - table)
          (ready ?i - item)
          (loosing_heat ?i - item)
          (started_delivery ?i - item)
          (chef_free ?c - chef)
          (socket_free ?s - socket)
          (started_cooking ?i - item))
     (:functions
          (total_delivery_window)
          (total_heat_lost))

     (:durative-action DELIVERY_WINDOW
          :parameters (?t - table)
          :duration (<= ?duration 10000000)
          :condition (and
               (at start (d_w_available ?t)))
          :effect (and
               (at start (not (d_w_available ?t)))
               (at start (d_w_open ?t))
               (at end (not (d_w_open ?t)))
               (at end (increase (total_delivery_window) ?duration))))

     (:durative-action DELIVER
```

```
     :parameters (?i - item ?t - table)
     :duration (= ?duration 2)
     :condition (and
          (at end (d_w_open ?t))
          (over all(d_w_open ?t))
          (at start (ready ?i)))
     :effect (and
          (at start (started_delivery ?i))
          (at end (not (started_delivery ?i)))
          (at end (delivered ?i ?t))
          (at end (not (ready ?i)))))

(:durative-action LOSING_HEAT
     :parameters (?i - item)
     :duration (<= ?duration 1000)
     :condition (and
          (at start (started_cooking ?i))
          (at end (started_delivery ?i)))
     :effect (and
          (at start (loosing_heat ?i))
          (at end (not (loosing_heat ?i)))
          (at end (increase (total_heat_lost) ?duration))))

(:durative-action MAKE_TEA
     :parameters (?i - tea ?s - socket)
     :duration (= ?duration 1)
     :condition (and
          (at start (socket_free ?s))
          (at end (loosing_heat ?i)))
     :effect (and
          (at start (not (socket_free ?s)))
          (at start (started_cooking ?i))
          (at end (socket_free ?s))
          (at end (ready ?i))))

(:durative-action MAKE_TOAST
     :parameters (?i - toast ?s - socket)
     :duration (= ?duration 2)
     :condition (and
          (at start (socket_free ?s))
          (at end (loosing_heat ?i)))
     :effect (and
          (at start (not (socket_free ?s)))
          (at start (started_cooking ?i))
          (at end (socket_free ?s))
          (at end (ready ?i))))
```

```
    (:durative-action MAKE_COOKED_BREAKY
        :parameters (?i - cooked_breaky ?c - chef)
        :duration (= ?duration 4)
        :condition (and
            (at start (chef_free ?c))
            (at end (loosing_heat ?i)))
        :effect (and
            (at start (not (chef_free ?c)))
            (at start (started_cooking ?i))
            (at end (chef_free ?c))
            (at end (ready ?i)))))

(define (problem CafeProblem1)
    (:domain CafeDomain)
    (:objects
        table1 - table
        tea1 - tea
        toast1 - toast
        chef1 - chef
        socket1 - socket)
    (:init
        (d_w_available table1)
        (chef_free chef1)
        (socket_free socket1)
        (= (total_delivery_window) 0)
        (= (total_heat_lost) 0))
    (:goal (and
        (delivered tea1 table1)
        (delivered toast1 table1)))
    (:metric minimize (total_heat_lost)))
```

An alternative metric could be:

```
    (:metric minimize (total_delivery_window))
```

Appendix C. — The Lift-Match Domain

```
(define (domain matchlift)
    (:requirements :durative-actions :typing)
    (:types fuse match lift electrician floor room - object)
    (:predicates
        (light ?match - match ?room - room)
        (handfree ?elec - electrician)
        (unused ?match - match)
        (mended ?fuse - fuse)
```

```
    (onfloor ?elec - electrician ?floor - floor)
    (inlift ?elec - electrician ?lift - lift)
    (roomonfloor ?room - room ?floor - floor)
    (liftonfloor ?lift - lift ?floor - floor)
    (inroom ?elec - electrician ?room - room)
    (fuseinroom ?fuse - fuse ?room - room)
    (connectedfloors ?floor1 ?floor2 - floor))

(:durative-action LIGHT_MATCH
      :parameters (?match - match
          ?elec - electrician
          ?room - room)
       :duration (= ?duration 8)
       :condition (and
          (at start (unused ?match))
          (over all (inroom ?elec ?room))
          (over all (light ?match ?room)))
     :effect (and
          (at start (not (unused ?match)))
          (at start (light ?match ?room))
          (at end (not (light ?match ?room)))))

(:durative-action MEND_FUSE
     :parameters (?fuse - fuse
          ?match - match
          ?room - room
          ?elec - electrician)
      :duration (= ?duration 5)
     :condition (and
          (at start (inroom ?elec ?room))
          (over all (inroom ?elec ?room))
          (at start (fuseinroom ?fuse ?room))
          (at start (handfree ?elec))
          (at start (light ?match ?room))
          (over all (light ?match ?room)))
     :effect (and
          (at start (not (handfree ?elec)))
          (at end (mended ?fuse))
          (at end (handfree ?elec))))

(:durative-action ENTER_ROOM
     :parameters (?floor - floor
          ?room - room
          ?elec - electrician)
     :duration (= ?duration 1)
     :condition (and
          (at start (onfloor ?elec ?floor))
```

```
          (at start (roomonfloor ?room ?floor)))
     :effect  (and
          (at end (inroom ?elec ?room))
          (at end (not (onfloor ?elec ?floor)))))

(:durative-action EXIT_ROOM
     :parameters (?floor - floor
          ?room - room
          ?elec - electrician)
     :duration (= ?duration 1)
     :condition (and
          (at start (inroom ?elec ?room))
          (at start (roomonfloor ?room ?floor)))
     :effect  (and
          (at end (not (inroom ?elec ?room)))
          (at end (onfloor ?elec ?floor))))

(:durative-action ENTER_LIFT
     :parameters (?floor - floor
          ?lift - lift
          ?elec - electrician)
     :duration (= ?duration 1)
     :condition (and
          (at start (onfloor ?elec ?floor))
          (at start (liftonfloor ?lift ?floor))
          (over all (liftonfloor ?lift ?floor)))
     :effect  (and
          (at end (inlift ?elec ?lift))
          (at end (not (onfloor ?elec ?floor)))))

(:durative-action EXIT_LIFT
     :parameters (?floor - floor
          ?lift - lift
          ?elec - electrician)
     :duration (= ?duration 1)
     :condition (and
          (at start (inlift ?elec ?lift))
          (at start (liftonfloor ?lift ?floor))
          (over all (liftonfloor ?lift ?floor)))
     :effect  (and
          (at end (not (inlift ?elec ?lift)))
          (at end (onfloor ?elec ?floor))))

(:durative-action MOVE_LIFT
     :parameters (?floorfrom ?floorto - floor
          ?lift - lift)
     :duration (= ?duration 2)
```

```
        :condition (and
             (at start (connectedfloors ?floorfrom ?floorto))
             (at start (liftonfloor ?lift ?floorfrom)))
        :effect  (and
             (at start (not (liftonfloor ?lift ?floorfrom)))
             (at end (liftonfloor ?lift ?floorto))))
```

Problem File 1

```
(define (problem matchliftproblem01)
     (:domain matchlift)
     (:objects match1 match2 - match
          fuse1 fuse2 - fuse
          lift1 - lift
          elec1 elec2 - electrician
          floor1 floor2 - floor
          room1a room1b room2a room2b - room)
     (:init
          (unused match1)
          (unused match2)
          (handfree elec1)
          (handfree elec2)
          (onfloor elec1 floor1)
          (onfloor elec2 floor1)
          (roomonfloor room1a floor1)
          (roomonfloor room1b floor1)
          (roomonfloor room2a floor2)
          (roomonfloor room2b floor2)
          (liftonfloor lift1 floor1)
          (fuseinroom fuse1 room1a)
          (fuseinroom fuse2 room2b)
          (connectedfloors floor1 floor2)
          (connectedfloors floor2 floor1))
     (:goal (and
          (mended fuse1)
          (mended fuse2)))
     (:metric minimize (total-time)))
```

Lift-Match Numeric Domain (sketch)

Domain header and LIGHT_MATCH action:

```
(define (domain matchCellarComplexNumeric)
     (:requirements :durative-actions :typing :fluents)
     (:types fuse match lift electrician floor room - object)
     (:predicates
```

```
        (light ?room - room)
        (handfree ?elec - electrician)
        (mended ?fuse - fuse)
        (onfloor ?elec - electrician ?floor - floor)
        (inlift ?elec - electrician ?lift - lift)
        (roomonfloor ?room - room ?floor - floor)
        (liftonfloor ?lift - lift ?floor - floor)
        (inroom ?elec - electrician ?room - room)
        (fuseinroom ?fuse - fuse ?room - room)
        (connectedfloors ?floor1 ?floor2 - floor))
    (:functions
        (matchesleft))


    (:durative-action LIGHT-MATCH
        :parameters
            (?elec - electrician
            ?room - room)
        :duration (= ?duration 8)
        :condition (and
            (at start (> (matchesleft) 0))
            (over all (inroom ?elec ?room))
            (over all (light ?room)))
        :effect (and
            (at start (decrease (matchesleft) 1))
            (at start (light ?room))
            (at end (not (light ?room)))))

...
```

Appendix D. — The DriverlogShift Domain

```
(define (domain driverlogshift)
    (:requirements :typing :durative-actions)
    (:types
        location locatable - object
        driver truck obj - locatable)
    (:predicates
        (at ?obj - locatable ?loc - location)
        (in ?obj1 - obj ?obj - truck)
        (driving ?d - driver ?v - truck)
        (link ?x ?y - location)
        (path ?x ?y - location)
        (empty ?v - truck)
        (working ?d - driver)
```

```
     (resting ?d - driver)
     (rested ?d - driver)
     (tired ?d - driver))
(:functions
     (capacity ?t - truck)
     (weight ?t - truck)
)

(:durative-action WORK
     :parameters
          (?driver - driver)
     :duration (= ?duration 102)
     :condition (and
          (at start (rested ?driver)))
     :effect (and (at start (working ?driver))
          (at end (not (working ?driver)))
          (at start (not (rested ?driver)))
          (at start (not (resting ?driver)))
          (at end (tired ?driver))))

(:durative-action REST
     :parameters
          (?driver - driver)
     :duration (= ?duration 20)
     :condition (and
          (at start (tired ?driver)))
     :effect (and
          (at start (resting ?driver))
          (at end (not (resting ?driver)))
          (at start (not (working ?driver)))
          (at start (not (tired ?driver)))
          (at end (rested ?driver))))

(:durative-action LOAD-TRUCK
     :parameters
          (?obj - obj
          ?truck - truck
          ?loc - location)
     :duration (= ?duration 2)
     :condition (and
          (over all (at ?truck ?loc))
          (at start (at ?obj ?loc)))
     :effect (and
          (at start (not (at ?obj ?loc)))
          (at end (in ?obj ?truck))))

(:durative-action UNLOAD-TRUCK
```

```
    :parameters
        (?obj - obj
        ?truck - truck
        ?loc - location)
    :duration (= ?duration 2)
    :condition (and
        (over all (at ?truck ?loc))
        (at start (in ?obj ?truck)))
    :effect (and
        (at start (not (in ?obj ?truck)))
        (at end (at ?obj ?loc))))

(:durative-action BOARD-TRUCK
    :parameters
        (?driver - driver
        ?truck - truck
        ?loc - location)
    :duration (= ?duration 1)
    :condition (and
        (over all (at ?truck ?loc))
        (at start (at ?driver ?loc))
        (at start (empty ?truck)))
    :effect (and
        (at start (not (at ?driver ?loc)))
        (at end (driving ?driver ?truck))
        (at start (not (empty ?truck)))))

(:durative-action DISEMBARK-TRUCK
    :parameters
        (?driver - driver
        ?truck - truck
        ?loc - location)
    :duration (= ?duration 1)
    :condition (and
        (over all (at ?truck ?loc))
        (at start (driving ?driver ?truck)))
    :effect (and
        (at start (not (driving ?driver ?truck)))
        (at end (at ?driver ?loc))
        (at end (empty ?truck))))

(:durative-action DRIVE-TRUCK
    :parameters
        (?truck - truck
        ?loc-from - location
        ?loc-to - location
        ?driver - driver)
```

```
        :duration (= ?duration 10)
        :condition (and
             (at start (at ?truck ?loc-from))
             (over all (driving ?driver ?truck))
             (at start (link ?loc-from ?loc-to))
             (over all (working ?driver)))
        :effect (and
             (at start (not (at ?truck ?loc-from)))
             (at end (at ?truck ?loc-to))))

   (:durative-action WALK
        :parameters
             (?driver - driver
             ?loc-from - location
             ?loc-to - location)
        :duration (= ?duration 20)
        :condition (and
             (at start (at ?driver ?loc-from))
             (at start (path ?loc-from ?loc-to))
             (over all (working ?driver)))
        :effect (and
             (at start (not (at ?driver ?loc-from)))
             (at end (at ?driver ?loc-to)))))
```

Problem File 1

```
(define (problem DLOG-2-2-2)
     (:domain driverlogshift)
     (:objects
          driver1 driver2 - driver
          truck1 truck2 - truck
          package1 package2 - obj
          s0 s1 s2 p1-0 p1-2 - location)
     (:init
          (at driver1 s2)
          (rested driver1)
          (at driver2 s2)
          (rested driver2)
          (at truck1 s0)
          (empty truck1)
          (at truck2 s0)
          (empty truck2)
          (at package1 s0)
          (at package2 s0)
          (path s1 p1-0)
          (path p1-0 s1)
          (path s0 p1-0)
```

```
     (path p1-0 s0)
     (path s1 p1-2)
     (path p1-2 s1)
     (path s2 p1-2)
     (path p1-2 s2)
     (link s0 s1)
     (link s1 s0)
     (link s0 s2)
     (link s2 s0)
     (link s2 s1)
     (link s1 s2))
(:goal (and
     (at driver1 s1)
     (rested driver1)
     (at truck1 s1)
     (at package1 s0)
     (at package2 s0)))
(:metric minimize (total-time)))
```