Managing Concurrency in Temporal Planning using Planner-Scheduler Interaction

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

Department of Computer and Information Sciences
University of Strathclyde, Richmond Street
Glasgow G1 1XH, United Kingdom

Abstract

Metric temporal planning involves both selecting and organising actions to satisfy the goals and also embedding these actions in time, which is a central concern of scheduling. In PDDL2.1, the widely adopted planning domain description language standard, metric temporal planning problems are described using actions with durations. A large number of planners have been developed to handle this language, but the great majority of them are fundamentally limited in the class of temporal problems they can solve.

In this paper, we review the source of this limitation and present an approach to metric temporal planning that is not so restricted. Our approach links planning and scheduling algorithms into a planner, CRIKEY, that can successfully tackle a wide range of temporal problems. We show how CRIKEY can be simplified to solve a wide and interesting subset of metric temporal problems, while remaining competitive with other less capable planners. We provide empirical data comparing the performance of this planner, CRIKEY$_{SHE}$, our original version, CRIKEY, and a range of other modern temporal planners.

Our contribution is to describe the first competitive planner capable of solving problems that require concurrent actions.

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

Time plays a vital role in the construction of plans. However, a great deal of work in planning has been concerned only with qualitative properties of time, such as the relative ordering of activities. The problem of reasoning with quantitative 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 IxTeT [10], TLPLAN [11], TALPLANNER [12], TGP [13] and ZENO [14]. A brief discussion of these and other relevant planning systems appears in Section 12.

The introduction of quantitative time in PDDL2 was managed through the use of *durative actions*, which are actions that execute over a specified period of time. Although many planners have been developed to manage durative actions, the overwhelming majority of them suffer from a significant weakness. To understand this weakness, it is helpful to observe that time can play significantly different roles in planning problems. These roles were discussed by Fox and Long in the context of their planner, LPGP [6] and also by Halsey, Fox and Long [15]. A more recent treatment by Cushing, Kambhampati, Mausam and Weld [16] provides a thorough formal analysis of the issues, leading to the definition of problems with the property of *required concurrency*. These are problems for which the only solution plans contain concurrent actions. In contrast, inherently sequential problems are those for which all solutions can be expressed as sequential plans. We further consider this analysis in section 2, but emphasise the key finding of Cushing *et al*: almost every planner developed to handle PDDL2 domains has only been capable of handling inherently sequential problems. That is, although the plans they construct might include concurrency, these planners can only manage concurrency as an essentially cosmetic effect, by rescheduling the actions in a sequential plan.

In this paper we introduce an approach to managing planning with quantitative time that is capable of solving problems with required concurrency. We present a planning system, CRIKEY, that is designed to handle a class of these problems. We go on to describe a simplified approach, implemented in CRIKEY$_{SHE}$, that is able to manage a significant class of problems with required concurrency but can also solve inherently sequential problems with only a minimal performance penalty compared with the planners designed to exploit an assumed inherent sequentiality.

We describe the consequences of the assumptions on which the majority of planners for PDDL2 are based and also consider some of the alternative approaches that have been explored for performing quantitative temporal planning. We describe our own approach, which uses a forward state-space search,

using well-known relaxed-plan heuristics for guidance, but combines this with well-understood techniques for performing efficient temporal reasoning in order to achieve an effective treatment of temporal problems with required concurrency. Not only is our planner capable of solving complex temporal problems with required concurrency, but it is also able to solve temporal problems that involve the use of variable duration actions, including those that impact on the consumption of resources. CRIKEY is, as far as we are aware, the only planner currently capable of solving such problems in PDDL2.1. We present empirical data to demonstrate that the techniques we describe are competitive with more limited approaches to temporal planning, giving us the combined benefits of a more capable temporal planner and acceptable performance.

CRIKEY is not a complete planner, since it uses heuristics to determine the causal structure of the plans built by forward search and the heuristic forward search itself is only rendered complete by the somewhat artificial approach of abandoning the heuristic and resorting to a full state-space search if the heuristic arrives in a deadend. Nevertheless, CRIKEY and its simplification, CRIKEY$_{SHE}$, are both capable of solving a range of temporal problems that are beyond the scope of other modern planners.

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 the performance of CRIKEY and CRIKEY$_{SHE}$.

2 The Planning-Scheduling Spectrum

Many authors have previously observed that planning problems, which can be crudely characterised as problems that involve deciding *what* actions to

perform, and scheduling problems, characterised as problems that involve deciding *when* to perform actions, lie at ends of a spectrum [17–19]. In practice, the decisions about what to do and when to it can rarely be cleanly separated. Nevertheless, in the majority of planners designed to tackle problems expressed in PDDL2.1, using durative actions, a simplification is made in which problems are solved first by planning a collection of actions that achieve the goals and, second, by scheduling these actions into an efficient timeline. Cushing *et al* [16] observe that most of the PDDL2.1 benchmark problems, and all of the International Planning Competition problems, can be solved by planners that exploit this decomposition. They also observe that there is a class of problems that require concurrency to solve them. That is, every plan to solve instances of these problems contains concurrent actions.

An example of a domain with required concurrency is the Match domain (see Appendix 13.1), 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, 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 1 shows a valid plan for the problem and illustrates the way in which concurrency arises.

Required concurrency plays a critical role in temporal planning, but it is worth distinguishing two ways in which it can arise: some problems require concurrency because of interactions between the activities in the domain, while other problems require concurrency only because of a deadline that forces activities to be compressed in time. In the problem in figure 2, there are two goals, $G1$ and $G2$, that must be achieved by the deadline indicated. There is only one sequence of actions that achieves $G2$, shown as strand 2, using actions E, F and G. $G1$ can be achieved by two alternative sequences, shown as plan strands 1 and 3. The point of this example is that to satisfy the goals by the deadline, strand 1 must be used to avoid interaction between action H and the end of action F, which deletes P. A correct sequencing of the actions in

```
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]
```

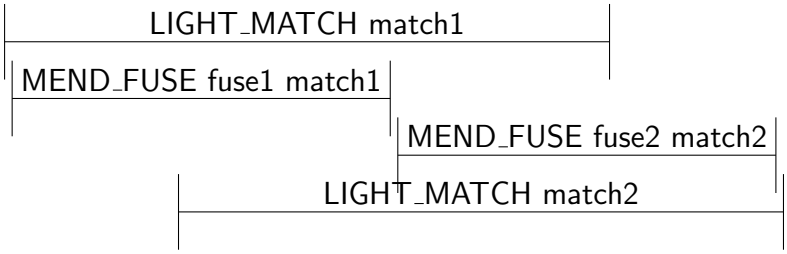


Fig. 1. A valid plan for the Match Problem, assuming that matches may burn concurrently.

strands 2 and 3 will force F and G to be delayed until the completion of H, missing the deadline.

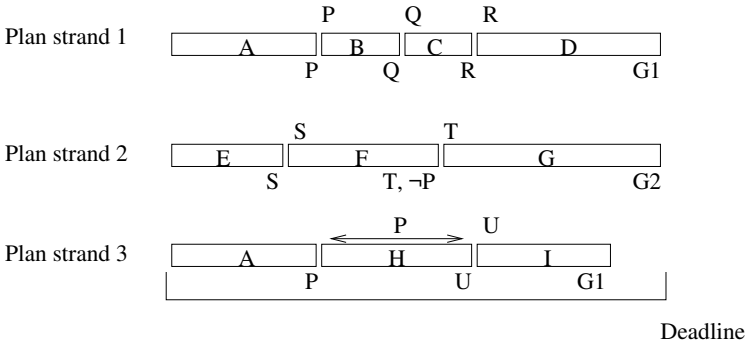


Fig. 2. A problem with required concurrency for a deadline.

Cushing *et al* exclude deadlines from the planning problems they consider, since, as they observe, it forces concurrency in most cases. However, the difference between these forms of required concurrency is important, because it points to two different levels of integration required between planning decisions and scheduling decisions in a solver. For problems that require concurrency only to meet deadlines a relatively loose integration is possible: planning decisions can be made independently of scheduling decisions, with backtracking over poor planning choices only being forced when the selected actions cannot be scheduled within the required deadline. Although the scheduling constraints affect what plans can be constructed, the core relationships between actions remain those that can be treated as inherently sequential. For example, although concurrency is required to solve the problem shown in figure 2, the reasoning it requires can be performed by constructing a sequential plan and then post-processing it to achieve the concurrency required for the deadline. In other words, the requirement for concurrency can be ignored during plan-

ning and need only be considered after the plan is completed. Of course, a planner might generate an initial plan that uses strands 2 and 3, preventing a successful subsequent scheduling of the actions to form a plan that meets the deadline. This example illustrates that ignoring the role of concurrency during planning might be an inefficient way to find a valid plan. Furthermore, if there is no deadline then either plan (strands 1 and 2 or strands 2 and 3) can be scheduled to achieve the goals, although not with equal efficiency.

The complex form of required concurrency that arises in deadline-free problems, as considered by Halsey [15] and Cushing *et al* and appearing in the Match domain, forces a closer integration of planning and scheduling decisions: it is impossible to construct a plan at all without considering how the actions are embedded on the timeline and scheduling them together in order to exploit or avoid their interactions.

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 [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 logical fluents (with the exception of Section 7.5 where we consider interactions with numeric fluents) and to actions with preconditions expressed purely as conjunctions of (positive) propositions. 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.

Most of the more complex features we assume do not arise can be tackled by techniques that are orthogonal to the concerns of this work and it is therefore simpler to ignore them in this presentation of metric temporal planning.

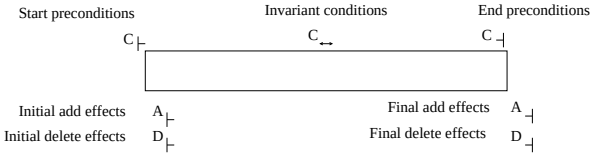


Fig. 3. The Structure of a Standard Durative Action

Most modern planners that attempt to manage temporal domains do so by a 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. 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.

Definition 2 — Compressed Action
A compressed action, $ca = (cond, add, del)$, is a 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* [16], so that planners using this reduced representation cannot solve (or, indeed, properly represent) problems that require 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 propositional variable, `(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)` proposition 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 [20] and SGPLAN [5].

3.1 Further examples of domains with required concurrency

It is very easy to conceive of examples of problems in which a single action makes a particular resource available for a limited period of time. Other than the Match example, an action that starts a work shift makes the labour of that employee available for a fixed period of time, an action that takes a pan of melted chocolate off the heat will offer the opportunity to use the liquid chocolate for a limited period until it cools and hardens and an action throwing

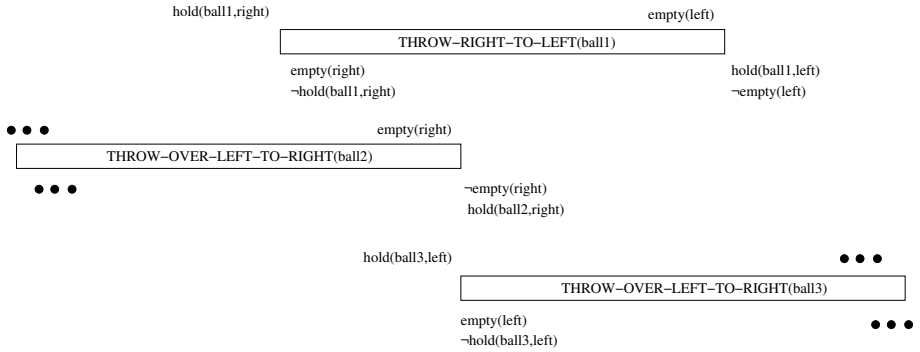


Fig. 4. Interlocking Actions in Part of Juggling Plan

a ball into the air will release the hand to perform another action until the ball must be caught again (one can imagine writing a description of a juggling domain using such constraints).

The relationship between the period of availability of a resource and actions that require that resource can vary. One possible relationship is that the end points of interacting durative actions must lie inside the period of an enclosing durative action. Figure 4 illustrates how part of a juggling plan using ball-throwing actions might be constructed in this way. Each hand must be holding the appropriate ball in preparation for a throw action, and the other hand must be empty when it is required to catch the ball at the end of the throw action. However, because one ball is always in the air, the two hands must each perform either a catch or throw part of an action within the period of flight of the third ball.

The requirement that the fuse mending action lie entirely inside the period of a burning match is another relationship, which is also the condition that will be required of any activity undertaken by an employee if it must fit within their shift.

Consideration of various examples leads us to believe that the examples such as the fuse mending and the shift-worker are a common pattern and that a planner capable of handling this particular form of required concurrency would be a useful extension of the compression-based temporal reasoning of other modern planners. Although CRIKEY is designed to deal with the more general case, including the interactions involved in juggling, we will describe a simplified version, CRIKEY$_{SHE}$, that is intended to manage the more restricted case of interactions like those of the match, shift-worker and chocolate.

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.

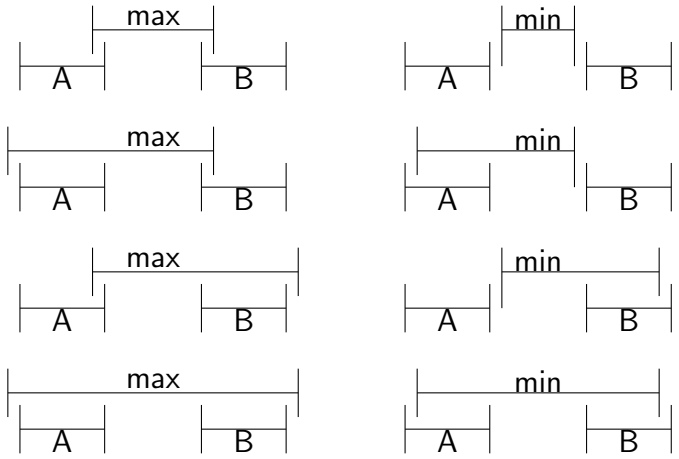
PDDL2.1 is very expressive. It can capture a wide range of temporal relations and constraints, such as Allen's interval relations [21] applied to durative actions, through the use of dummy actions to enforce the conditions that are needed. This idea was explored in some detail in work reported by Fox, Long and Halsey in [22].

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 they are in time. 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 of Allen's interval constraints [21] 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 5. 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.

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 this constraint that are not of interest here. The details of handling these constraints are discussed by Fox *et al* [22].



(a) Maximum Constraints

(b) Minimum Constraints

Fig. 5. Possible Combinations that Represent Similar Constraints

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 5 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 this 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 5 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 5 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 ($A \neq B$) in $\mathscr{A}$, are part of a

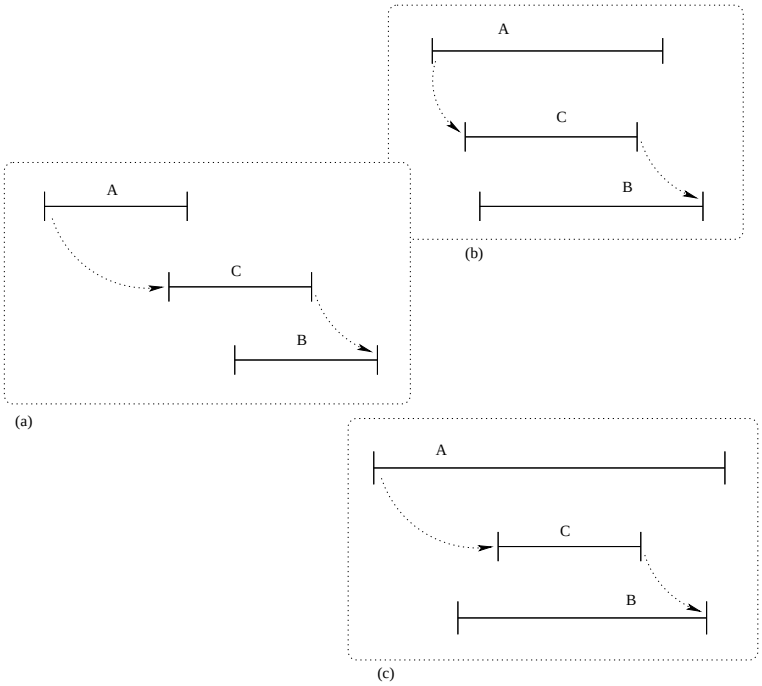


Fig. 6. Configurations of Actions Satisfying $A \prec^{min} B$: (b) Showing Overlapping A and B and (c) Showing Nested Actions

maximum separation relationship, written $A \prec^{max} B$ if $A_{\vdash} \prec B_{\dashv}$ and 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 5 (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 ($A \neq B$) in $\mathscr{A}$, are part of a *minimum separation relationship*, written $A \prec^{min} B$ if $A_{\vdash} \prec B_{\dashv}$ and there exists an action, $C \in \mathscr{A}$, such that $A_{\vdash} \prec C_{\vdash}$ and $C_{\dashv} \prec B_{\dashv}$.

Figure 6 shows an example of a collection of actions satisfying $A \prec^{min} B$. The figure illustrates that the constraints that must hold to achieve this relationship can be satisfied when A and B overlap (or even with B inside A), but the necessary constraints are precisely those that appear in Figure 5 (b).

Maximum separation constraints specify a maximum possible time between

the occurrences of two actions; minimum separation constraints represent a minimum time between two actions. 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 7. 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

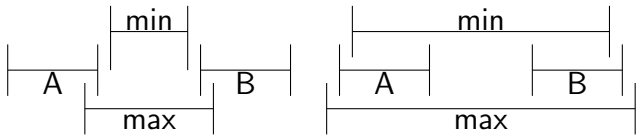


Fig. 7. Expressing both Minimum and Maximum Time Between Actions in PDDL2.1

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

4.1 Required Concurrency and Temporal Constraints

Temporal constraints are highly relevant to the problem of required concurrency, both in the form Cushing *et al* discuss and also in the form that is forced by the need to meet deadlines. In both cases, the constraints that govern the structure of the plan are formed from the temporal constraints discussed above. In particular, the coupling of a maximum separation constraint and a minimum separation constraint leads to problems with required concurrency.

Theorem 1 *Given a plan that contains actions A and B such that $A \prec^{max} B$ then the plan contains concurrent activity.*

Proof:

If $A \prec^{max} B$ then $A_{\vdash} \prec B_{\dashv}$. Suppose that A and B are not concurrent (otherwise the result is trivial). Then it follows that $A_{\dashv} \prec B_{\vdash}$. If $A \prec^{max} B$ then there exists C such that $C_{\vdash} \prec A_{\dashv}$ and $B_{\vdash} \prec C_{\dashv}$, so C runs concurrently with both A and B.

□

Since compression can only be used in problems that do not require concurrency [16], this result demonstrates that the $\prec^{max}$ relationship is critical in

determining whether compression can be used. In section 6.1 we go on to show that the $\prec^{min}$ relationship carries similar force in determining when the scheduling of a plan is actually non-trivial.

5 An LPGP–FF Hybrid

Having considered various constraints that can be expressed in PDDL2.1 and that 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. To avoid ambiguity, in the remainder of this work we will use the word 'action' to refer to a durative action, rather than a classical instantaneous action. We therefore define the term 'instant-action' to distinguish the classical instantaneous actions from durative actions and to allow more convenient reference to the components.

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

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

The propositions P-inv and iP-inv are dummy propositions that do not appear elsewhere in the domain description. $P_{\vdash}$, $P_{\leftrightarrow}$ and $P_{\dashv}$ are called instant-actions.

Where we refer specifically to start instant-actions (end instant-actions) we will, in general, abbreviate to start actions (end actions, respectively).

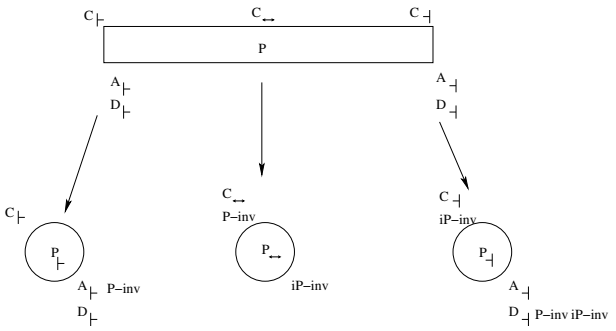


Fig. 8. The Mapping of a Durative Action into Snap-Actions

As can be seen in figure 8 (and in Definition 5), the two dummy propositions, *P-inv* and *iP-inv* act to ensure that the end action cannot be exploited without preceding it with an invariant checking instant-action (to achieve *iP-inv*), which must, in turn, be preceded by the start action (to achieve *P-inv*).

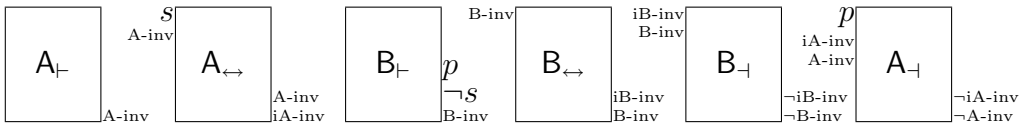
An approach based around this division forms the foundation of LPGP [6], where a variant of GraphPlan search is employed over a planning graph populated with instant-actions. However, the idea is not restricted to use within a GraphPlan setting: the instant-action domain can be used with alternative planning strategies, such as the forward state-space search used in FF [23]. 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 instant-actions) can be post-processed to restore the structure of the temporal constraints between start and end points of 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 [24].

Whilst intuitive, three problems arise when using this approach:

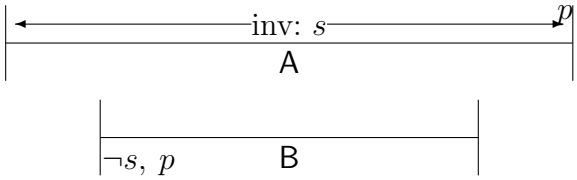
- invariants might not be respected correctly;
- a goal state might be found where some actions have not terminated;
- temporal constraints might not be satisfied.

The first problem is illustrated in Figure 9 where a start ($\mathsf{A}_{\vdash}$) and invariant instant-action ($\mathsf{A}_{\leftrightarrow}$) are put into the plan, followed by a instant-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 an 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 9, 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 instant-action to be reapplied alongside all instant-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. 9. 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 instant-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 instant-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 10). LPGP handles the problem by preventing start instant-actions from being selected unless a corresponding end action has been selected (recall that LPGP searches backwards in the plangraph).

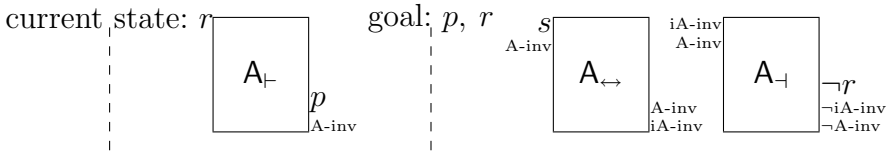


Fig. 10. Example of an End Action Deleting a Goal

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 actions are split into three instant-actions. Referring again to Figure 9, 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. Informally, an envelope and its *contents* can be defined as a collection 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 (Definition 3), 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 (Definition 4). 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 another subset of actions. 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 [25]. 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 instant-actions forming the contents of the envelope, the instant-actions of A and B are in C and S is a simple temporal network expressing the constraints that must hold between the instant-actions in C. $A_{\vdash}$ is a start action, $B_{\dashv}$ is an end action and, for every instant-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 at least one of them. The only necessary constraints are that every instant-action in the envelope is constrained to fall after the start of the envelope or before the end of the envelope. It is important to observe that the content instant-actions of an envelope are not each necessarily constrained to lie between *both* ends of the envelope: it is only necessary that each content instant-action *could* lie between the two end points. The end points of an envelope are part of a partial order and each content instant-action is connected to the partial order at one end or the other, but the entire partial order need not be forced to lie between the end points of the envelope. An envelope is intended to capture a collection of elements of a plan that are constrained with respect to one another and could require concurrency in resolving their embedding on a timeline.

In planning problems that require concurrency, actions are interlocked by logical, as well as temporal, constraints. 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 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 is obvious that if

an envelope has an infinitely large maximum time or the content actions have a minimum duration of zero, then there will be no problem in scheduling the envelope, 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 (Definition 3), including the actions A and B. The envelope becomes a significant constraint when the contents include one or more minimum separation constraints (Definition 4).

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 11 for examples of envelopes and content actions.

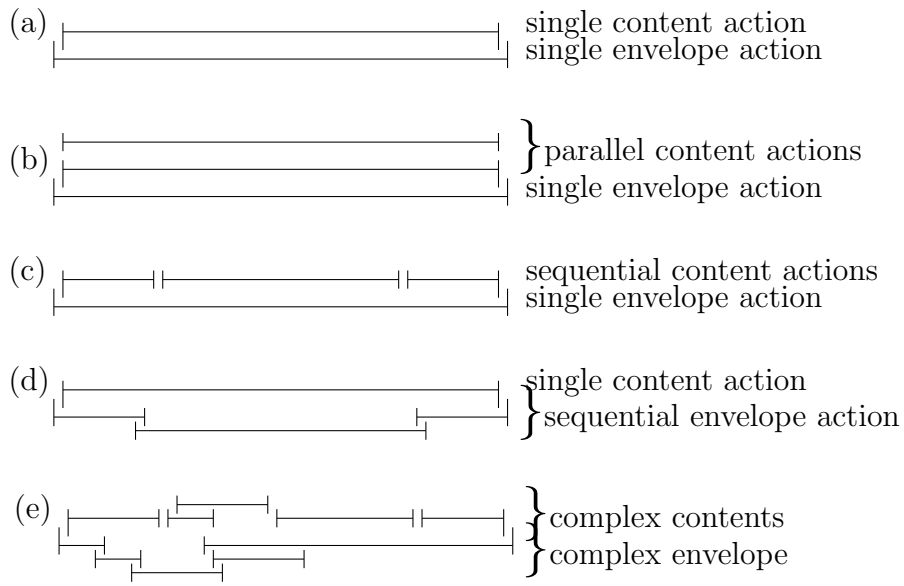


Fig. 11. Envelopes and Contents

6.1 Consistency Checking and Envelopes

We now introduce a result, using the following definition, that shows how we can limit the need to check the consistency of the contents of an envelope.

Definition 7 — Consistent Temporal Embedding
Given a partially ordered $(\prec)$ collection of instant-actions, $\mathscr{S}$, $f : \mathscr{S} \to \mathbb{R}$ is

a *consistent temporal embedding* of $\mathscr{S}$ if for every $a, b(a \neq b) \in \mathscr{S}$, $a \prec b \rightarrow f(a) < f(b)$ and for all actions A, with duration δ_A, such that $A_\vdash, A_\dashv \in \mathscr{S}$, $f(A_\dashv) - f(A_\vdash) = \delta_A$.

Theorem 2 *Suppose $\mathscr{S}$ is a partially ordered ($\prec$) set of instant-actions such that every total ordering consistent with $\prec$ yields an executable sequence (and there is at least one such sequence). If there is no consistent temporal embedding of $\mathscr{S}$ then there is a pair of actions, A and B, such that $A_\vdash, A_\dashv, B_\vdash$ and $B_\dashv$ are in $\mathscr{S}$ and $A \prec^{max} B$ and $A \prec^{min} B$.*

Proof:

Given that the partially ordered set can be totally ordered and executed, for there to be no consistent temporal embedding the inconsistency in each candidate temporal embedding must arise from the temporal constraints. This implies that the STN that represents the constraints on the elements in $\mathscr{S}$ must contain a negative cycle.

The STN will contain only three kinds of edges: those corresponding to partial ordering constraints ($a \prec b$), which will have weight 0, directed from b to a [3], edges of weight Δ_A from $A_\vdash$ to $A_\dashv$, where Δ_A is the duration of A, and edges of weight $-\Delta_A$ from $A_\dashv$ to $A_\vdash$. A negative cycle must include one of the negative weighted edges. However, such edges are always directed backwards through the partial order, as are all the edges denoting the partial ordering constraints. To create a cycle, the end points of such an edge must be connected using a path that contains at least one edge directed forwards, which must be a positively weighted edge between the start and end actions of some durative action. Thus, the cycle will have the general form shown in figure 12, and $\Delta_B > \Delta_A$, so $A \neq B$.

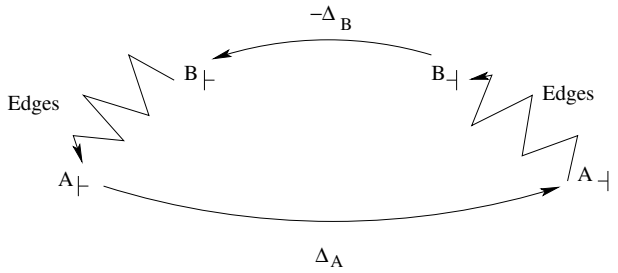


Fig. 12. A Negative Cycle in the STN

Therefore, $A_\vdash \prec B_\vdash$ and $B_\dashv \prec B_\dashv$, so $A \prec^{min} B$. Similarly, $A_\vdash \prec A_\dashv$ and $B_\vdash \prec A_\dashv$, so $A \prec^{max} B$.

[3] In fact, to ensure that non-mutex actions are not placed simultaneously, the corresponding edge will have to have weight $-\epsilon$, but we can assume ϵ is small enough not to be the source of any problems in the temporal embedding.

□

The implication of this result is that we need only be concerned about the consistency of an envelope once there is at least one maximum separation and one minimum separation constraint in it.

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.

CRIKEY has much in common with the hybrid system described in section 5. In particular:

- 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 compressed actions in which all of the effects are treated as though achieved by a single, instantaneous action.

An outline of the process CRIKEY follows in solving a problem is as follows. Planning begins 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 instant-action is to be added to the plan that interacts with any of the instant-actions in E, CRIKEY extends the simple temporal network for the envelope to determine whether E can be safely scheduled. A instant-action interacts with those in E if it has any enforced ordering relationships with respect to the start, end or content instant-actions of the envelope (as determined by the Partial-Order-Lifting algorithm, described in Section 7.4.1). If the instant-action does not interact with any of those 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 the approach. What follows is a formal specification of the behaviour of CRIKEY. This process is illustrated in figure 13, which is labelled with the section parts that describe the various components.

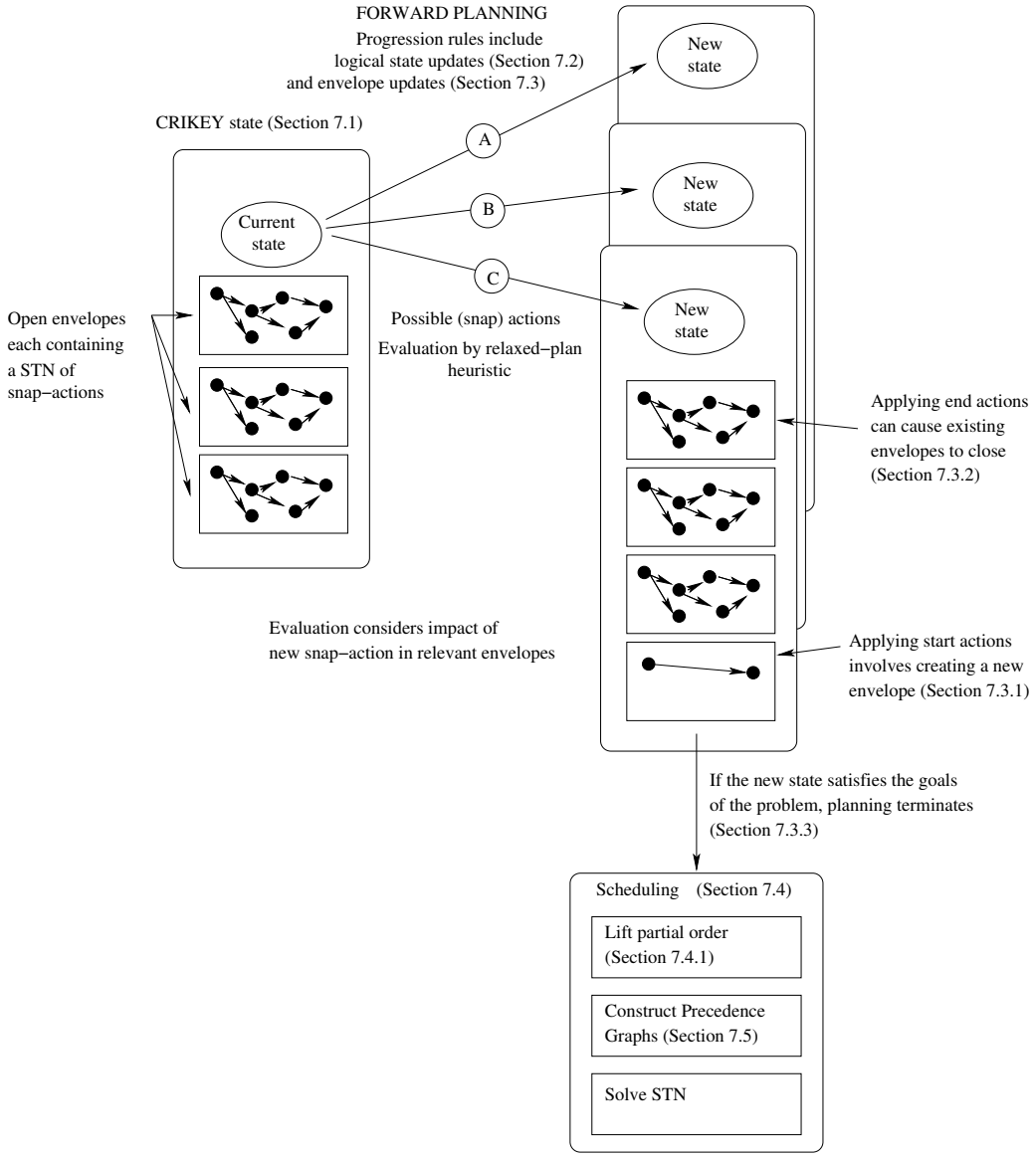


Fig. 13. The CRIKEY Planning Process

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 8. As can be seen, the facts used in states in FF are augmented with a set of open envelopes, through which required concurrency can be managed.

Definition 8 — 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 is known that they will eventually be added to the plan). 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 9 — 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 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 instant-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 instant-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. It is straightforward to check whether the logical precondition of an action is satisfied and a distinction is made between simply testing for applicability and actually applying an action. However, the same is not true when checking the temporal constraints on an candidate action: 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. If there are inconsistencies, the action must be rejected and the temporal constraints restored to their prior state. What follows is a discussion of how the envelopes in CRIKEY are updated to reflect the attempted application of a instant-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 are those constructed in the process of testing applicability.

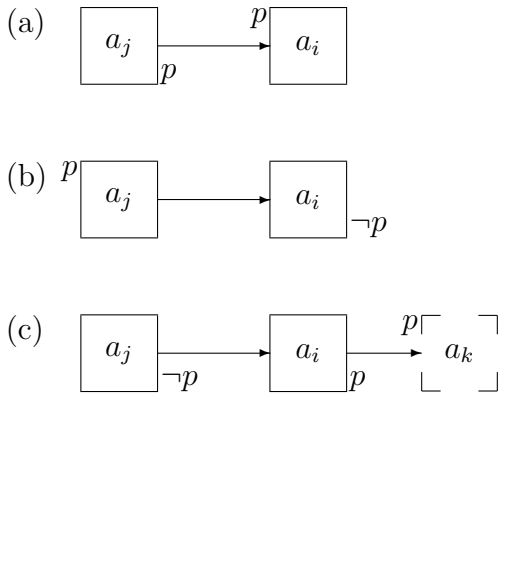


Fig. 14. The Three Reasons to Order Actions

Central to envelope maintenance in CRIKEY is the 'CheckOrder' function, presented in Definition 10. CheckOrder determines whether there is an interesting interaction between two instant-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 14 and CheckOrder returns true if any of these apply. The situation shown in Figure 14 (c) exploits a standard *declobbering* technique used in partial order planners. An alternative technique 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.

Definition 10 — CheckOrder Function
The function CheckOrder(a, b) applied to instant-actions, a and b, returns the value true iff there is an interaction between two actions a = (*cond-a*, *add-a*, *del-a*) and b = (*cond-b*, *add-b*, *del-b*) that indicates a should precede b. The function is defined to return the value of the expression:

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

7.3.1 Application of Start Actions

Applying a start action $A_{\vdash}$ is a two-step process. The first step is to check that the 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 11 — Update Envelope
Given an envelope $E = (A_{\vdash}, B_{\dashv}, C, S)$ and a instant-action a, associated with durative action da with duration Δ and start and end actions $a_{\vdash}$ and $a_{\dashv}$, then $update(E,\ a)$ is the new envelope $E' = (A_{\vdash}, B_{\dashv}, C', S')$, where:

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

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

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

After each envelope E has been updated, the consistency function (Definition 9) 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 is formally defined as follows:

Definition 12 — New Empty Envelope
An empty envelope associated with the start action, $A_{\vdash}$ is created by the function $emptyenvelope(A_{\vdash}) = (A_{\vdash}, A_{\dashv}, \emptyset, \emptyset)$.

Additional new envelopes are then created from each $E = (x, y, C, S) \in \xi'$ if $CheckOrder(A_{\vdash}, y)$. 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 13 — Expand Envelope
Given an envelope, $E = (A_{\vdash}, B_{\dashv}, S, C)$ and a start instant-action, $X_{\vdash}$ such that *CheckOrder*$(X_{\vdash}, B_{\dashv})$, the result of expanding E by the addition of $X_{\vdash}$ is given by the function *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 9) and if a new envelope is found to be inconsistent, the process aborts and the start action is deemed to be inconsistent.

The process we have described could lead to a quadratic number of envelopes being open as a function of the size of the plan. In practice, for this to occur there must be potential interactions between many of the pre- or post-conditions of pairs of actions, while it appears that in practice interactions are more localised than this.

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. This is because ending an action cannot initiate a time-limited window of opportunity. Time-limited windows are always enclosed within durative actions and start when a start action is applied. Therefore, 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 11. The *update* function is applied to each envelope $e \in \xi'$ to produce envelopes ξ''. As with start actions, if the consistency function (Definition 9) 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, 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 13.

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 start and end actions, 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 actions are scheduled according to the partial order. This process is a similar post-processing of plans as is performed in several other temporal planners, such as MIPS [4], although it uses a different mechanism to achieve it.

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 described in [24] (sketched out in Figure 15) 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

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

Initialise $primary\_add(a_i)$ empty for all $i = 1...n$
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 $a_j \prec a_i$
Add p to $primary\_add(a_j)$
(b) **for each** $d \in del(a_i)$ **do**
Find *all* actions a_j where $d \in precond(a_j)$
Add an ordering for each a_j found: $a_j \prec a_i$
(c) **for each** $p \in primary\_add(a_i)$ **do**
Find *all* actions a_j where $p \in del(a_j)$
Add an ordering for each a_j found: $a_j \prec a_i$

Fig. 15. The Partial-Order-Lifting Algorithm to Translate Totally Ordered Plans to Partially Ordered Plans

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.

Alternative approaches to extracting the causal-link structure of the plan from the total order are possible. An interesting possibility is the approach exploited by Laborie and Godard [26], which might offer a more efficient algorithm.

7.4.2 The STN

The partial order that is lifted from the plan is captured as a simple temporal network, making it straightforward to solve the problem of scheduling the actions in the plan into an efficient temporal structure.

Definition 14 — 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

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

where $X_0 = 0$ and represents the start of the plan. Part (a) of Definition 14 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.

The STN is solved using a standard solver [25], in order to identify the earliest point of application of each action.

7.5 Precedence Graphs

A significant capability implemented in CRIKEY is the management of metric resources within the temporal context. This means that plans can be constructed using variable durations and using behaviours that can generate or consume metric resources according to their duration. In this implementation of CRIKEY the management of such resources is only applied in the post-processing scheduling of plans, so is used in order to attempt optimisation of the plan quality metric where it depends on the use of resources.

The scheduling needs to be more sophisticated than simply solving an STN in order to handle actions with variable durations when such actions can generate or consume a variable amount of a given resource. The resource reasoning is performed with precedence graphs. This is not a novel technology, but rather a new application of it. Precedence graphs are summarised below and introduced by Laborie in [27]. The rest of this section describes how they are integrated into CRIKEY including the changes to [27] 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 [5]. 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 [27] 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 distinguish resources and maximum and minimum resource values appear implicitly in action preconditions — conditions which can change from action to action. This has the effect of there being possibly varying bounds on the amount of the resource throughout the plan.

For example, the model used in [27] 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 [27]. 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 [27], 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

[5] 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.

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).

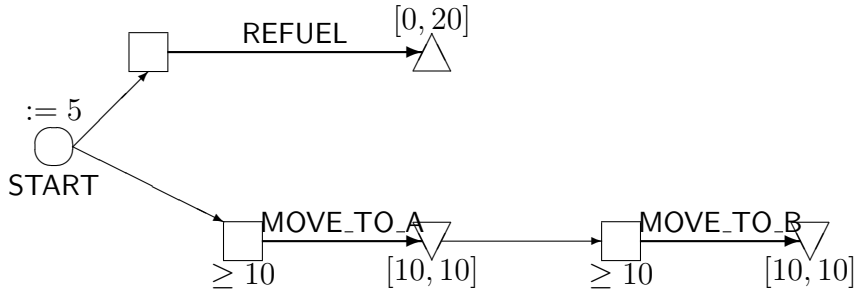
CRIKEY can use the balance constraints to find new bounds on both the time variables (which can be propagated through to the STN) and resource usage variables. This only occurs where there are duration inequalities in the domain, as this is the only case where operators in the plan can produce or consume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

An example precedence graph is given in Figure 16(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).

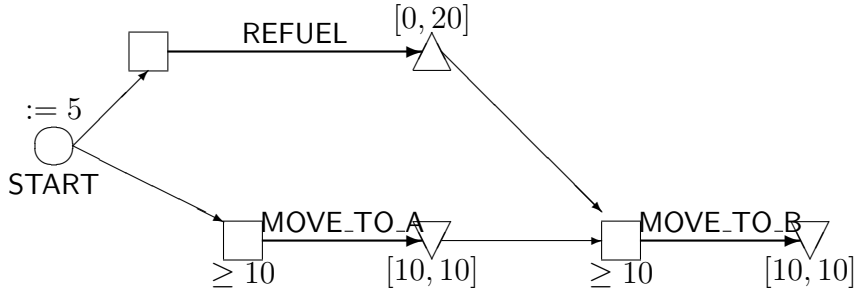
Firstly, in Figure 16(b), the precedence graph is able to reason that the REFUEL action must happen before the second MOVE_TO_B action and so the appropriate precedence relationship is added. This is turn allows reasoning for the resource bounds of the REFUEL action, as it must now produce a minimum 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.

7.5.1 Duration Inequalities

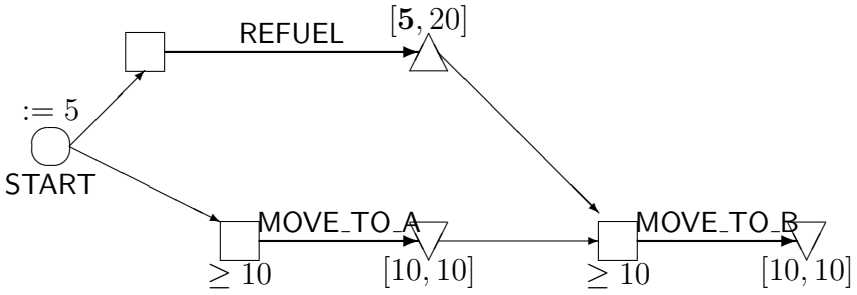
PDDL2.1 allows the specification of duration inequalities. Rather than fixing the duration of a durative action, these allow bounds to be put on the duration. These bounds can be a function of other metric values (for example, one cannot drive for longer than the amount of fuel available). However, resource 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 determined by the planner. For example, it is possible to decide how long to fill a tank (the duration of a refuel action) and, therefore, how full the tank will be at the end of the action. The possible combinations are summed up in Table 1.



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



(b) A Precedence Relationship is Added



(c) The Resource Bounds change

Fig. 16. Example Precedence Graph

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 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

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

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

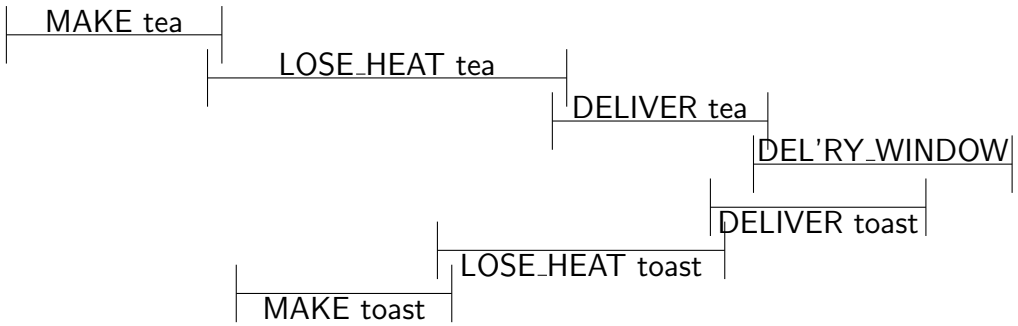


Fig. 17. A Partial Order for the Café Domain

(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 13.1) where the object is to deliver breakfast to a table in a café, as drawn diagrammatically in Figure 17[6]. 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.

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. By default the DELIVERY_WINDOW is minimised, with corresponding impact on the makespan for the plan. 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 18 shows two plans. One where the heat lost is minimised, and one

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

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

```
0.01: (MAKE_TEA tea1 socket1) [1.00]
1.00: (LOSING_HEAT tea1) [2.04]
1.02: (MAKE_TOAST toast1 socket1) [2.00]
3.01: (LOSING_HEAT toast1) [0.03]
3.02: (DELIVERY_WINDOW table1) [2.02]
3.03: (DELIVER tea1 table1) [2.00]
3.03: (DELIVER toast1 table1) [2.00]

Total Delivery-Window: 2.02
Total Heat-Lost: 2.07
```

```
(:metric minimize (total_heat_lost))
```

```
0.01: (MAKE_TEA tea1 socket1) [1.00]
1.00: (LOSING_HEAT tea1) [0.03]
1.01: (DELIVERY_WINDOW table1) [4.03]
1.02: (DELIVER tea1 table1) [2.00]
1.02: (MAKE_TOAST toast1 socket1) [2.00]
3.01: (LOSING_HEAT toast1) [0.03]
3.03: (DELIVER toast1 table1) [2.00]

Total Delivery-Window: 4.03
Total Heat-Lost: 0.06
```

Fig. 18. Two Plans with Identical Goals but Different Metrics

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 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 affected by other metric variables. 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.

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 temporal relationships 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.

8.1 Restricted Envelopes

The simplest interaction between actions is where one action achieves a condition required in order to execute another action. This is the most important

relationship that planners are designed to reason about and is precisely the problem tackled in relaxed planning when delete effects are ignored. Forming this relationship requires the establishment of an ordering between the related actions. It is less common 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 15 — 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 11. Longer envelopes, such as those shown in examples (d) and (e) in Figure 11, 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 instant-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 16 — 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, concurrency *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, in general, there will be a branching point leading to alternative states in the search space: one with the action inside the envelope and others with the action partially or totally outside the envelope. In the case of hard envelopes, there will only be one accessible state in the search space: that in which the 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 compressed 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 — as occurs with the LIGHT_MATCH action.

Definition 17 — Single Hard Envelope
A durative action, A, generates a Single Hard Envelope iff

$$|add_{A_\vdash} \cap del_{A_\dashv}| = 1$$

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.

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 required concurrency is present in the problem.

8.2 Overview of the Architecture of CRIKEY$_{SHE}$

Figure 19 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 17, are split into instant-actions.

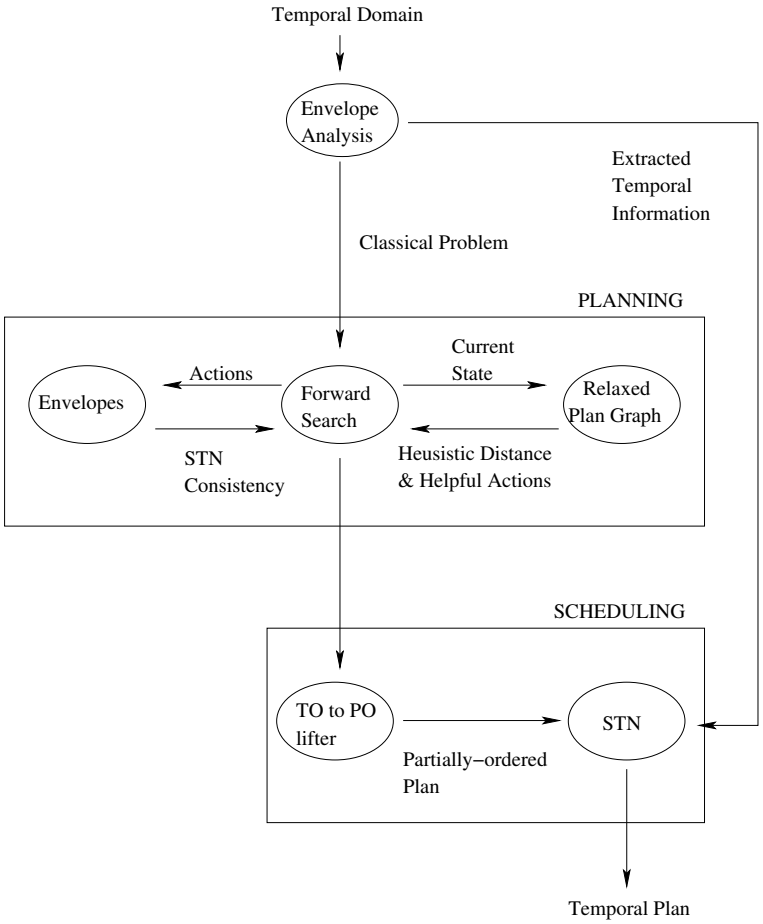


Fig. 19. Architectural Overview of CRIKEY$_{SHE}$

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 20 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 13, 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 8. 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 instant-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 15), used to identify the necessary temporal constraints. The function, *ordering* is defined as follows:

Definition 18 — Ordering Function
The ordering function, *ordering*$(E,\ a)$, returns a set of temporal constraints tc between a instant-action a and an open envelope $E = (A_{\vdash}, A_{\dashv}, C, S)$ where tc is defined to be:

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

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 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 19 — 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 20 — Update Envelope (CRIKEY$_{SHE}$)
Given an envelope, $E = (A_{\vdash}, A_{\dashv}, C, S)$ and a instant-action b, associated with action B having start and end actions $B_{\vdash}$ and $B_{\dashv}$ and duration Δ_B, to add to E, the envelope is modified according to the function: $update_{SHE}(E,\ a) = E'$ where E' is defined as:

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

8.2.1 Number-valued Fluents

CRIKEY$_{SHE}$ has a simpler mechanism for handling PDDL2.1 numeric fluents 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 numeric aspects have been omitted from the reasoning and definitions for simplicity in the presentation. There are two areas of note when considering numeric fluents in CRIKEY$_{SHE}$. The first is in the compression and splitting of durative actions. Numeric fluents 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 condition attached to the compressed action or start action becomes that the resource should be less than 8.

The second area that numeric fluents complicate is the lifting of the partial order. Any precedence relationship in the total order between two actions that either test or change the same resource has to be preserved in the partial order, since it is otherwise hard to ensure that the resource is correctly managed across the plan.

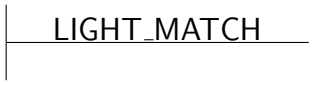
Numeric fluents are incorporated into the heuristic in a similar way to Metric-FF [28]. At each fact layer of the relaxed planning graph, the maximum and minimum possible levels of each resource are calculated based on the values at the previous fact layer and the actions available in the previous action

layer. For an action to be applicable in the relaxed planning graph, either the maximum or minimum level must meet the metric condition.

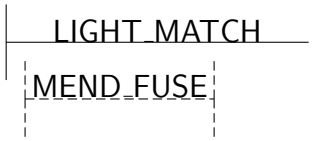
8.2.2 A Worked Example: The Match Domain

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 replaced, and the burning of a single match does not allow sufficient time for both fuses to be replaced sequentially.

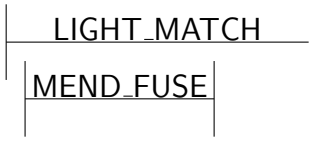
CRIKEY$_{SHE}$ performs envelope analysis, discovering that the LIGHT_MATCH action is potentially a single hard envelope action: the effect `have_light` is added by the start action and deleted by the end action. Following this all other actions are compressed to single STRIPS actions, and the LIGHT_MATCH 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.



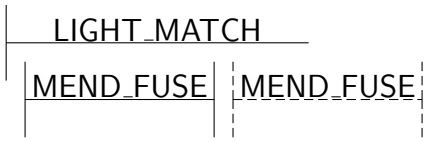
When heuristic evaluation suggests the start action to the LIGHT_MATCH action (a single hard envelope), it will create a new open envelope.



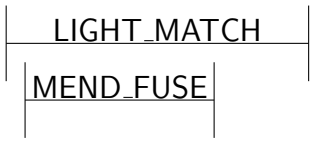
Heuristic search then finds the MEND_FUSE action should be applied next and CRIKEY$_{SHE}$ will then test to see if a MEND_FUSE action needs to go in this envelope, and if so, if it is consistent.



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$_{SHE}$ will eventually proceed to light a second match (and so start a new envelope). 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. However, if 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 instant-action translation only for single hard envelopes (detected in a preprocessing stage): actions that might have to be applied concurrently to solve the problem. All other actions are translated using the compressed 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 compressed 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 and can solve a larger set of complex temporal problems that other competing planners.

9 Results

Having described CRIKEY, a general system for solving temporal planning problems involving required concurrency, 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. CRIKEY and CRIKEY$_{SHE}$ are specifically designed to plan in domains requiring concurrency, but none of the IPC4 domains have this property. Therefore, we present several new domains that do require concurrency and evaluate planner performance on these. For all comparisons, the planners are run on the same machine with the same resources.

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 concurrency:

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	✓	*Partial*	✗	✗

Table 2
Capabilities of Temporal Planners

MIPS, LPG and TP4 cannot. To the best of our knowledge, the only other PDDL2.1 planner, capable of handling required concurrency, that has been discussed in the literature is the TEMPO planner discussed by Cushing *et al* [16], but it is not implemented. The planners that cannot find plans in these cases rely on a compressed 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 required concurrency where there are single hard envelopes, but in practice it cannot. The first, trivial, reason for this is that 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. The more significant reason is that when Sapa first finds a plan it does not advance time when it applies multiple start or end actions at the same time point, even if they are mutually exclusive. This is because to separate them would require to advance time by a small amount

(ε) instead of advancing to the next event in the event-queue. This would have an important impact on the structure of the search space explored by Sapa. Instead of handling this separation in the planning phase, Sapa applies a post-processing to the plan to introduce separation between successive actions. However, this post-processing does not account for start effects (even though Sapa does consider them whilst planning), and can therefore place the content actions outside the single hard envelopes that should contain them.

9.2 The Fourth International Planning Competition

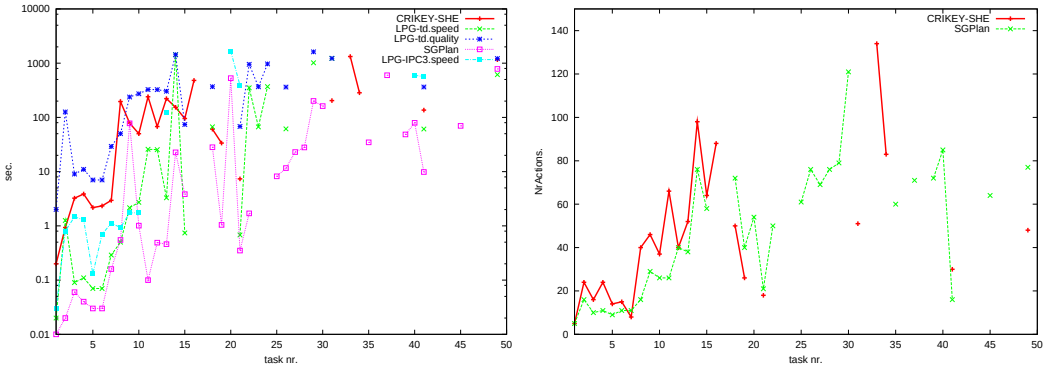
CRIKEY$_{SHE}$ was a participant in the Fourth International Planning Competition (IPC4). The competition was run over a period of approximately three months during which time competitors ran their planners on a series of problems using a Linux PC with two CPUs running at 3GHz. For each problem, planners 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 [29]. Each of these have a variety of different formulations including STRIPS only, ADL, numeric fluents, durative actions and combinations of numeric fluents with durative actions. 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 concurrency 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 aspects of the domains are further limited since there are no state dependent durations. Planners are compared in terms of both the time taken to solve problems and the solution quality (in the case of CRIKEY$_{SHE}$, this is the number of actions in the plan).

A selection of results from the competition is discussed here to demonstrate that CRIKEY$_{SHE}$ is competitive in general benchmark domains. This shows that the additional reasoning being performed in CRIKEY$_{SHE}$ does not hinder its performance on domains in which the reasoning is not necessary.

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



(a) Performance (b) Plan Quality

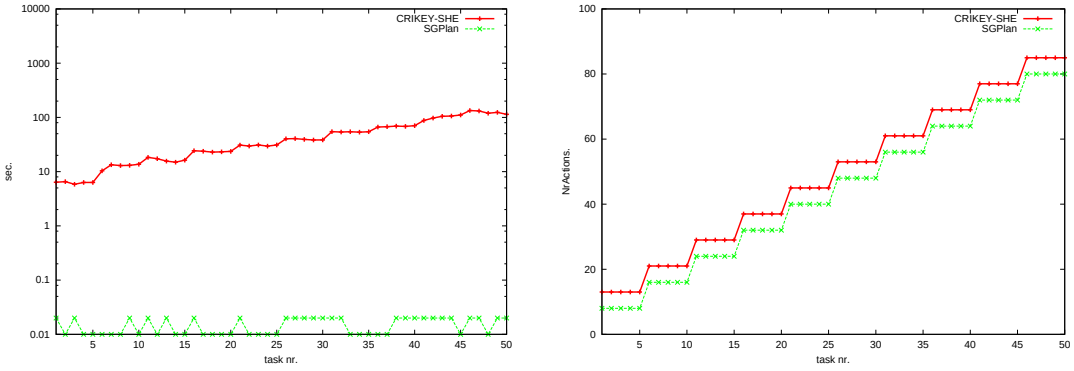
Fig. 20. Temporal Tankage Pipesworld Domain Results from IPC4

analysis of the competition results considering all planners on all domains refer to the IPC4 results paper [29].

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 both temporal and metric planning problems. As shown in figure, 20(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



(a) Performance (b) Plan Quality

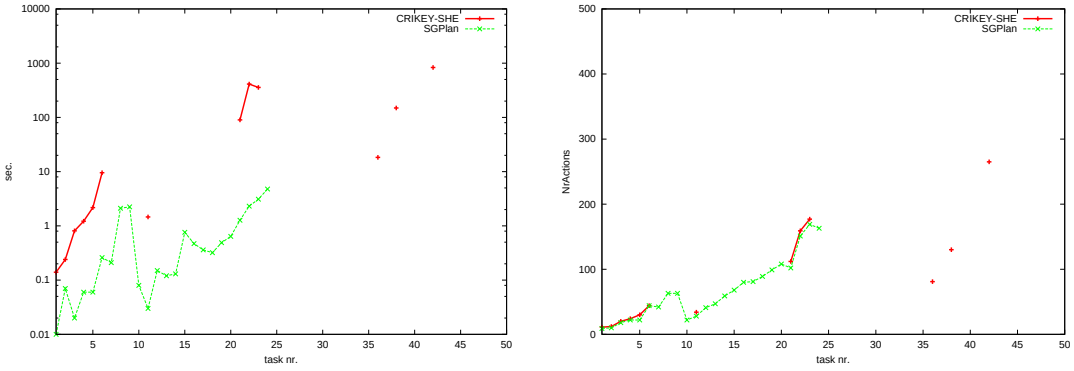
Fig. 21. Temporal UMTS Domain with compiled Time Windows

makespan of the plan. If this objective is ignored then the planning is trivial. CRIKEY$_{SHE}$ competed in three formulations of this domain: two variants of a temporal version 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[7]. Its performance scales similarly to other planners that competed in these domains, although it is generally slower. Since it is implemented in Java and not heavily optimised it is possible that the differences might be reduced by more careful implementation. Only two planners competed with the time windows compiled into PDDL2.1: CRIKEY$_{SHE}$ and SGPLAN (see Figure 21). CRIKEY$_{SHE}$ and SGPLAN find the same plans, although SGPLAN is clearly rather faster. The reason for the difference in the reported plan lengths is that CRIKEY$_{SHE}$ includes the dummy actions used to encode the time windows in the total action count, whereas the figures for SGPLAN do not (we have reported here exactly the data reported for IPC4). These results show that CRIKEY$_{SHE}$ can successfully and competitively handle required concurrency in the limited form of timed initial literals compiled into PDDL2.1.

The reasoning being performed by SGPLAN to handle the compiled time windows domains is undocumented. Experiments performed using SGPLAN on all other domains involving required concurrency lead to the report that the goal cannot be reached. This appears to be the standard consequence of the compression compilation, since the required effect, present during the envelope action, is compiled away: the action both adds and deletes the required predicate. Our experiments 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

[7] Those it did not solve at the time were later discovered to be the result of a bug in detecting repeated visited states.



(a) Performance (b) Plan Quality

Fig. 22. Temporal Airport Domain with Time Windows

a consequence, SGPlan does not need to reason about required concurrency 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 problems set in 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 concurrency in the compiled time windows formulations, Crikey$_{SHE}$ finds solutions that other planners do not, as shown in Figure 22.

9.3 Analysis Overview of IPC4 Domains

These competition results show that Crikey$_{SHE}$ is a temporal planner that performs competitively in propositional, metric and temporal benchmark domains. Crikey$_{SHE}$ is capable of handling domains that exploit richer temporal relations than the other competing planners can manage (as noted in Section 9.1). By making limiting the forms of interaction that a planner must handle, the difficulty of the problem the planner solves is reduced and this supports simplifying assumptions that can enhance the general performance of the system.

9.4 Required Concurrency

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 require concurrency we present several new domains to demonstrate the use of the concept.

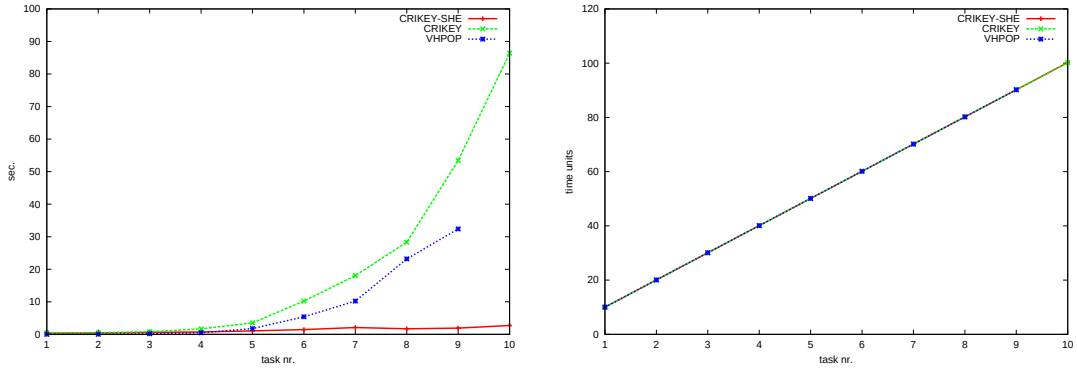
The only planner fully able to handle required concurrency producing valid plans and fully obeying the semantics of the language is the partial order planner VHPOP [3]. Sapa can attempt to solve these problems, and frequently returns a solution, but often the plan it finds is invalid. Therefore, VHPOP is used for comparison with CRIKEY. TEMPO [16] would be a natural candidate for comparison, but it is not implemented at the time of writing.

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. Another reason for considering smaller problems is that the difficulty in the problems does not come from the size of the instance but from the required concurrency and the reasoning about interactions between actions that this requires. It is of most interest to know whether the planners easily find the solution in the search space (if at all) rather than how long the planners take on larger problems.

The performance graphs are shown on a linear scale (not logarithmic) and the quality is no shown as the makespan (temporal length) of the plans. The domains contain required concurrency and so the temporal length of the plan is quite different from the number of actions in the plan. In effect, content actions are not counted, as it is the enclosing 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 the quality of the temporal reasoning performed by the planners is compared. 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 given in full in Appendix 13.1. However, VHPOP and Sapa do not fully observe PDDL2.1 semantics: where an invariant of an action is achieved by the start effects of an action (as in the LIGHT_MATCH action), these planners report



(a) Performance (b) Plan Quality

Fig. 23. Standard Match Domain

that no plan can be found. In order to relax the problem specification to allow these planners to be used for comparison, the invariants are removed for the purposes of these tests. This does not affect the need for concurrency in solutions, as the fuses must still be fixed within the period of the burning of the match.

Figure 23 presents results for the Match domain. In this variant it takes five time units to fix a fuse, and a match burns for eight time units. The number of matches and fuses in each instance is double the instance number (e.g. instance 5 has 10 matches and 10 fuses to fix).

As discussed earlier, Sapa fails to find valid plans. It apparently realises that more than one match is required, since it includes multiple LIGHT_MATCH actions in the plan, but it attempts to fix two fuses by the light of the same match, resulting in plans with half the number of matches required.

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

As can be seen, CRIKEY$_{SHE}$ performs considerably better than CRIKEY. Both planners arrive at a plateau in the search space when one fuse has been fixed by the light of one match and the goal to fix another fuse is considered. In this case the relaxed plan heuristic does not guide the planner to close the envelope and light another match. Instead, a small amount of search is required to discover this, including checking that all of the unfixed fuses are not able to be fixed using the rest of the available light. In CRIKEY, all durative actions are split into start and end actions, whereas in CRIKEY$_{SHE}$ uses a compressed single-action representation for the FIX_FUSE action. For this reason, the size

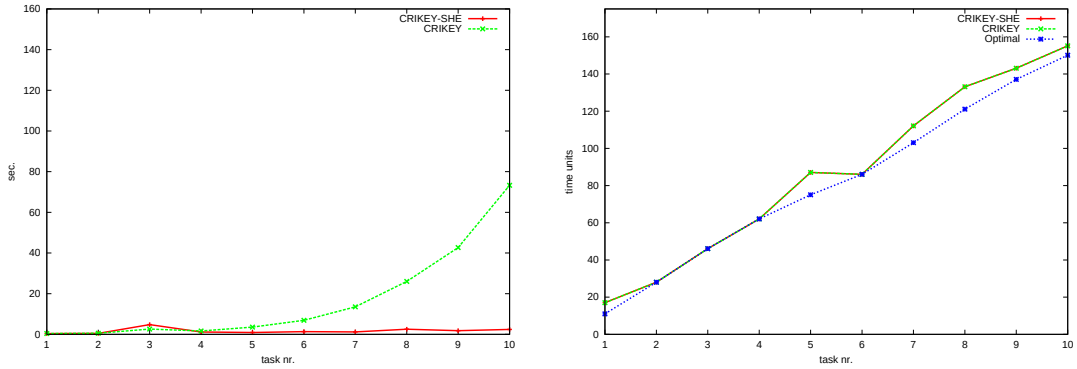
	CRIKEY$_{SHE}$			**CRIKEY**		
	Parsing & Grounding	Planning	Scheduling	Parsing & Grounding	Planning	Scheduling
1	33%	33%	33%	33%	33%	33%
2	0%	100%	0%	0%	100%	0%
3	0%	100%	0%	0%	67%	33%
4	0%	67%	33%	13%	63%	25%
5	0%	50%	50%	6%	67%	28%
6	0%	38%	63%	3%	76%	21%
7	8%	33%	58%	0%	98%	2%
8	6%	31%	63%	1%	84%	15%
9	5%	30%	65%	0%	84%	16%
10	4%	32%	64%	0%	92%	8%

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

of the search at every plateau where a new match is needed is twice as large in CRIKEY than in CRIKEY$_{SHE}$, so finding a solution plan takes longer.

Table 3 shows the percentage of time that both versions of CRIKEY spend on parsing, instantiation, planning, and scheduling. These figures are based on approximate values extracted from the Java implementation and should not be considered exact, especially for instances that take only a few seconds to solve — the difficulty in extracting precise computation statistics from Java accounts for the variability in the results. The reason that CRIKEY spends proportionally longer planning than CRIKEY$_{SHE}$ is again attributed to the larger search space. The results show that the effort spent in scheduling of plans is a relatively small part of the overall effort for CRIKEY, while for CRIKEY$_{SHE}$ the absolute time spent solving the problems is small enough that the proportions are dominated by set up overheads.

All planners find the same (optimal) solution.



(a) Performance (b) Plan Quality

Fig. 24. Variable Time Match Domain

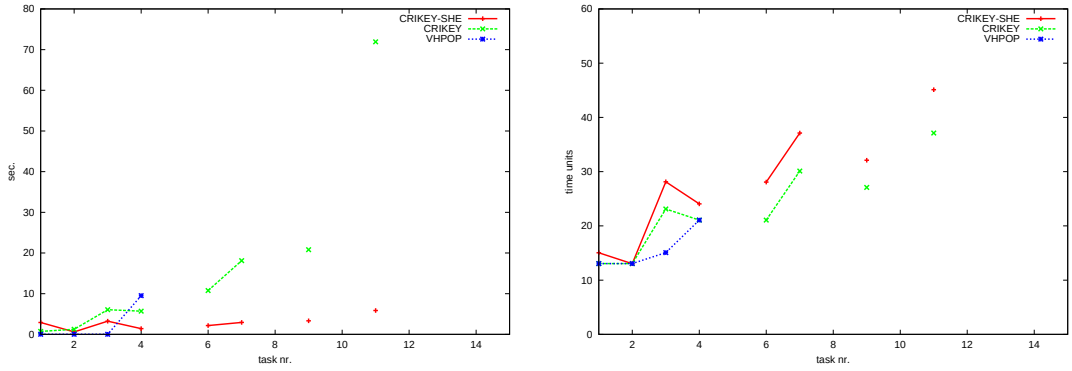
9.4.2 Match Domain with Variable Durations

Figure 24 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.

In this version, the light match action is changed so that only one match can be alight at any one time. The proposition light no longer takes the burning match as a parameter. This prevents two LIGHT_MATCH actions executing concurrently, as one would delete the "light" from the other when they burnt out. Thus, the FIX_FUSE action does not need to specify which match is burning when the fuse is fixed. In this model it is 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. An alternative, and perhaps more realistic, model in which two matches can burn at once and not delete each other's light at the end of the action, whilst also not specifying which match provides the light for the fix fuse action, requires conditional effects: neither CRIKEY nor the other planners are able to handle these.

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



(a) Performance (b) Plan Quality

Fig. 25. The Lift Match Domain

Figure 24(b) shows the quality of the solutions produced, including the optimal quality achievable. Both versions of CRIKEY produce the same solutions. In some problems, but not all, it is the optimal solution. This is usually where the problem is highly constrained and the optimal solution is the *only* solution. In other problems the optimal solution is found by good fortune. In cases where the problem is highly constrained (specific pairings of match and fuse must be used to solve the problem) the planner must perform best-first search in order to find a solution, since 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 the core elements that require concurrency. A more typical domain with required concurrency will also have some actions that do not interact with the parts of the problem that require concurrency. The next variant of the match domain, the 'Lift-Match' Domain illustrates such a domain. As before, electricians must fix fuses by the light of matches. The fuses however, are distributed about rooms in a building that the electricians must navigate between using the corridors and lifts. This navigation does not require concurrency. 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 13.1. Figure 25 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 often where the problems are highly constrained and there are fewer matches than fuses. In this case, more than one electrician must be in the same room at the same time to fix fuses by the light of only one match.

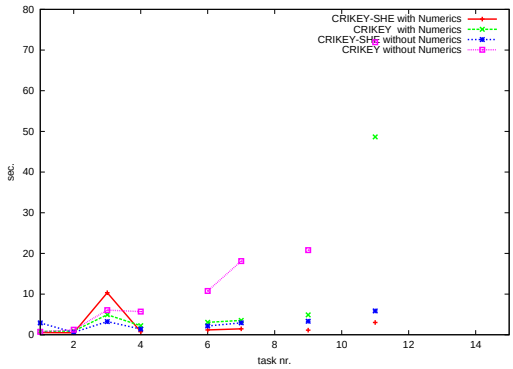


Fig. 26. Performance of CRIKEY and CRIKEY$_{SHE}$ with and without matches encoded using fluents

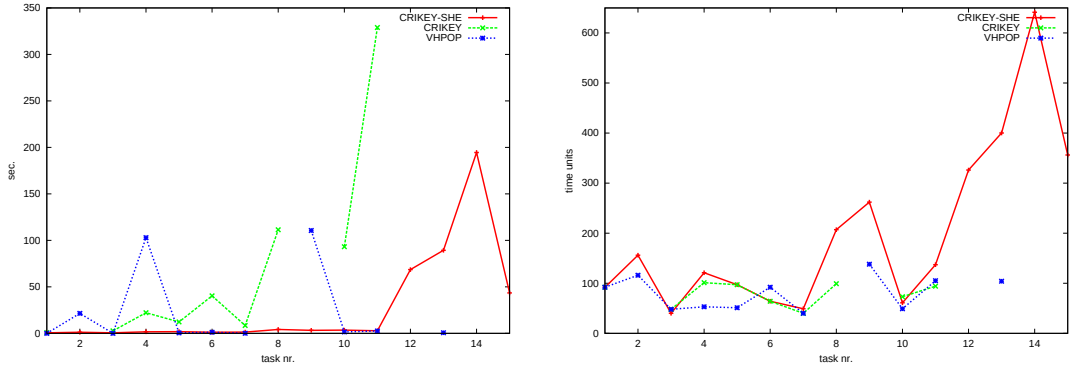
In the previous match domains there are only two operators (LIGHT_MATCH and MEND_FUSE), two types (match and fuse) and four predicates (mended, light, handfree and unused). In this domain there are seven operators, six object types and eleven predicates. This makes the search space bigger and so the 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 fact that the LIGHT_MATCH action ultimately deletes the light is critical in constructing the plan. As a consequence, CRIKEY and CRIKEY$_{SHE}$ often fail when using EHC and must, instead, resort to best-first search. This is a poor search strategy when the problem is big. It is clear that a more informed heuristic is needed to improve on this performance.

In an attempt to reduce the size of the domain, the match objects are encoded as 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 13.1). Figure 26 shows how this reduces the time needed to solve the problems.

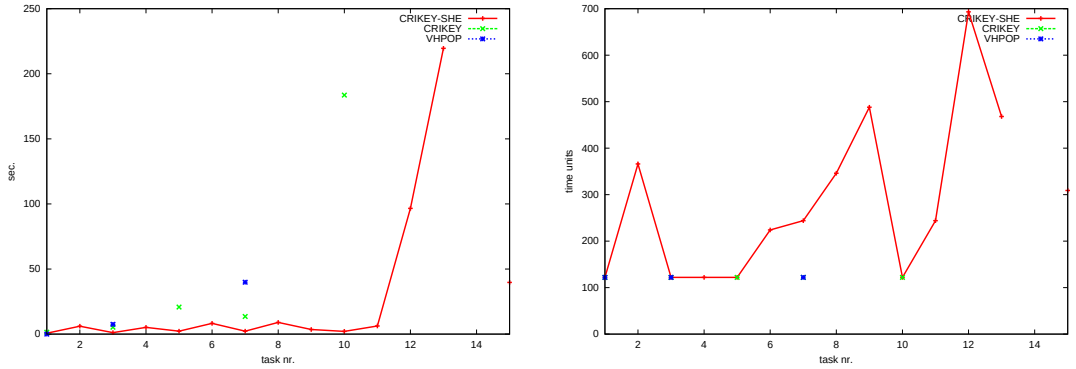
10 DriverLog Shift

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 13.1), where drivers can only work for a certain amount of time before they must take a break and have a rest. This involves required concurrency, as the shift action for a



(a) Performance (b) Plan Quality

Fig. 27. Standard DriverLog domain as used in IPC3



(a) Performance (b) Plan Quality

Fig. 28. DriverLog 'Simple Time' Formulation Converted to use Shifts

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 27 shows the performance of VHPOP, CRIKEY and CRIKEY$_{SHE}$ on the original first fifteen problems of the 'Simple Time' DriverLog formulation. Figure 28 shows the performance of the planners on the same problems converted into shift problems. Figure 29 shows how the performance of the planner deteriorates. This shows how much harder the problems become once the need for concurrency 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].

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

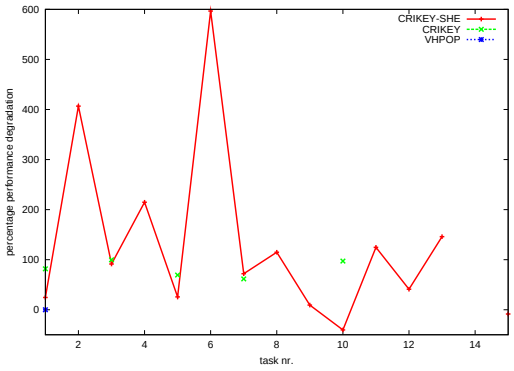


Fig. 29. Degradation of Performance when DriverLog Domain Converted to use Shifts

The search in both variants of CRIKEY must take arbitrary decisions over which branch of the search tree to explore first when they have the same heuristic value. In Shift domain problems that produce better performance (e.g. problem 10) it is likely that there is some good fortune in the search choices.

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 the planning and scheduling phases. Again, CRIKEY spends more of its time planning compared to CRIKEY$_{SHE}$. This is due to the larger search space explored during planning: they are both performing exactly the same task for the scheduling.

Neither the Match domain nor the DriverLog Shift domains can be handled by any of the planners in Table 2 that are unable to handle single hard envelopes or complex multiple envelopes. A different variant of the Driverlog Shift domain, originally presented in [30], is one in which the times of the shifts are fixed and cannot be moved. This fixed-shift variant can be encoded using PDDL2.2 Timed Initial Literals (TILs), so any planner able to handle these can tackle that domain. The variant of the problem considered here cannot be represented using TILs since the times of the shifts are not known in the initial state: they are selected 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 use Precedence Graphs and handle

	CRIKEY$_{SHE}$			**CRIKEY**		
	Parsing & Grounding	Planning	Scheduling	Parsing & Grounding	Planning	Scheduling
1	33%	33%	33%	50%	50%	0%
2	14%	29%	57%			
3	25%	25%	50%	6%	89%	6%
4	14%	29%	57%	1%	97%	2%
5	10%	40%	50%	2%	94%	4%
6	13%	38%	50%	1%	99%	1%
7	20%	20%	60%	3%	95%	3%
8	5%	42%	53%	0%	99%	1%
9	3%	45%	52%			
10	15%	15%	69%	1%	98%	1%

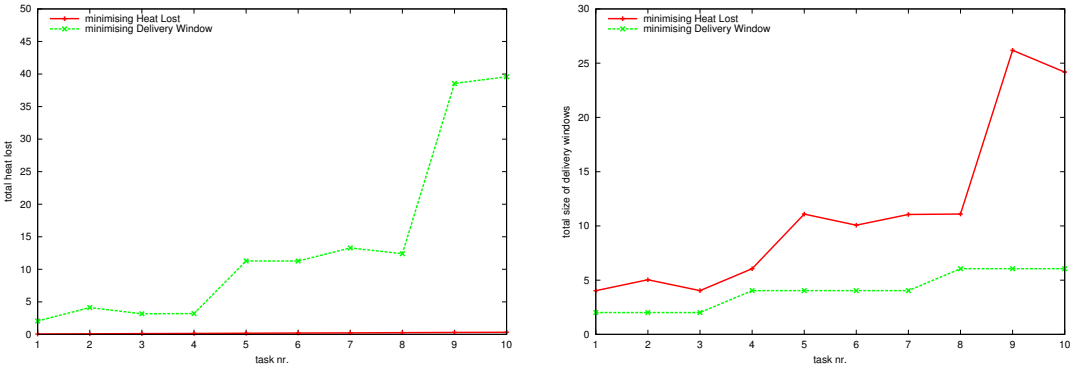
Table 4
Percentage of Time Spent in Temporal Planning by CRIKEY and CRIKEY$_{SHE}$ in the Driverlog Shift Domain

domains with duration inequalities. CRIKEY looks at the specified plan 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 considers 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 required concurrency).

The goal in the Café domain (as introduced in Section 7.5.1 and in full in Appendix 13.1) 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 30 shows plan quality with respect to a metric for ten problems in the Café domain. Both graphs show results from exactly the same problems,



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

Fig. 30. Plan Quality in the Café Domain with CRIKEY

however in Figure 30(a), the metric is set to minimise the heat lost, and in Figure 30(b), the metric is set to minimise the delivery window. CRIKEY is trying to minimise either the delivery window or the heat-loss, as shown in the two different curves (the curves show the values of the plans for the different solutions produced by the planner according to the metric selected).

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 variable durations are available.

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

12 Related Work

Apart from the work on temporal planners designed to work with PDDL2 durative actions, there is older work on temporal planning that is relevant to the approach we adopt here of integrating techniques for planning and scheduling. Some of the most significant work includes IxTeT [31,10], which uses a flaw-resolution approach to planning, searching in a space of partial plans. IxTeT is equipped with both a plan-flaw resolution solver and a temporal constraint solver that reasons about the temporal relations introduced into the evolving plan and propagates any implications they might have. IxTeT should certainly be capable of solving problems that involve the required concurrency illustrated by problems such as the Match domain. However, IxTeT has not proved competitive in tackling current competition benchmarks and no public

copy is available for comparison with CRIKEY$_{SHE}$.

Tate's work on NONLIN [32], OPLAN [33] and, more recently INOVA [34] has also led to the development of planners capable of performing complex temporal reasoning. These planners are all hierarchical planners, relying on rich domain representations that encode a significant collection of advice to the planner on how to construct plans and resolve the conflicts that arise during their construction. These planners are not designed to work with the flat, purely action-centred, domains that PDDL is designed to encode.

A particularly interesting planner is the constraint-based system HSTS [35] and its successor, EUROPA [36]. These were designed for use in space mission operations planning. They perform temporal planning by constructing timelines and using constraints to tie together the elements on the timelines. These constraints are *compatibility* constraints that specify temporal relationships between the intervals occupied by activities on the timeline, such as ordering, separation, overlap and so on. These planners are capable of handling rich and complex problems, but rely on careful encoding of the domain models to exploit the constraint machinery that drives the solvers.

Aspen [37] is another example of an application planner, again targetted at space mission operations, capable of performing complex temporal reasoning. Aspen also exploits a rich representation language and procedural attachment to link code directly to the preconditions of actions. This machinery allows it to plan in complex domains, using guiding information carefully encoded in the domain model.

Vere's DEVISER planner [38] also uses a temporal model in which actions have duration, operating over intervals. The planner searches in a space of partial plans, developing plans by steady flaw resolution and integrated reasoning about temporal relations in the developing plan.

The planning systems TLPLAN [11] and TALPLANNER [12] are designed to perform temporal reasoning, using powerful expressive modal temporal logics to capture the constraints that must be satisfied, both in order to apply actions and also in order to have a valid completed plan. Both of these planners rely on carefully crafted domain models that capture control information to guide the planners in a search for a plan. The performance benefits from this are dramatic [39], but neither planner is effective when applied directly to typical PDDL2 domain models in which no advice is present.

ZENO [14] is one of the most ambitious temporal planners that predates the introduction of PDDL2. It works with domain models that express continuous change in metric resources. The planner is based on an integration of partial-order planning and a linear constraint solver. An important advantage of partial-order planning, as illustrated by the more recent VHPOP [3], is that

it is possible to handle problems that require concurrency without any modification of the standard planning algorithm. However, ZENO is restricted in its ability to handle concurrency when metric resources are changed by more than one action. Unfortunately, no partial-order planners are currently competitive with the most recent heuristic search planners for benchmark PDDL2 domains and ZENO was only ever able to solve extremely small problems.

13 Conclusion

Temporal planning comprises a mixture of two elements: 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 simplify the temporal reasoning problem by assuming that compressed durative actions will adequately model the behaviour of durative actions from the domain. This greatly simplifies how the problems can be coupled and does not permit the modelling of required concurrency, except as deadlines, preventing those planners from solving a wide range of interesting temporal problems.

In this work we have presented a planner that is designed to solve problems that require concurrency, involving significant and complex interactions between actions. We have shown that it is possible to achieve this in combination with reasoning about numeric resources, particularly where these interact in certain ways with variable durations of actions. We have also shown that it can be achieved while retaining competitive performance across benchmark domains. Our solution, implemented in CRIKEY, links well-known techniques from planning and scheduling to achieve a composite solution that integrates these elements in an efficient way, restricting the communications between the two parts by identifying the parts of a planning problem in which scheduling choices are significant.

Two versions of CRIKEY have been presented: the first, CRIKEY itself, solves problems with a wide range of temporal interactions, while the second, CRIKEY$_{SHE}$, reasons only about a specific common form of required concurrency, the single hard envelope. The benefit of this restriction is, of course, a gain in efficiency.

CRIKEY$_{SHE}$ performed competitively in IPC4. CRIKEY$_{SHE}$ does not depend on the assumption that problems will have no required concurrenct, while the other competing planners exploited this assumption. In assuming compressed durative actions can capture the behaviours of the durative actions in the problems with which they are presented, other planners effectively tackle 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 if the planning choices create a scheduling problem for which there is no good quality solution.

An important limitation of CRIKEY (and CRIKEY$_{SHE}$) is that it cannot solve problems where an action A must necessarily occur alongside a copy of *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 [40], CRIKEY only has one 'counter' for each ground action to mark its execution. However, recent analysis by Fox and Long [41] 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 10) 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 *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, relying on peculiar action formulations that do not occur in any of the domains presented in this paper or in 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.

13.1 Future Work

Work is underway investigating combining the approach to temporal planning adopted in CRIKEY with the problem decomposition framework used in TSGPLAN [42]. 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 TSGPLAN, the aim is to produce a system with the same temporal reasoning capabilities as CRIKEY but with better overall performance.

We are also exploring the interactions between numeric resource management and temporal reasoning, including continuous resources affected by exogenous processes. CRIKEY provides a useful platform on which to construct these more complex forms of reasoning, in order to handle rich and expressive domain features.

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, Journal of Artificial Research (JAIR) 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] M. Ghallab, H. Laruelle, Representation and control in IxTeT, a temporal planner, in: Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94), AAAI Press, Menlo Park, CA, 1994, pp. 61–67.

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

[12] 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.

[13] 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.

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

[15] K. Halsey, D. Long, M. Fox, CRIKEY - A Planner Looking at the Integration of Scheduling and Planning, in: Proceedings of the Workshop on Integration Scheduling Into Planning at 13th International Conference on Automated Planning and Scheduling (ICAPS'03), 2004, pp. 46–52.

[16] 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.

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

[18] P. Laborie, M. Ghallab, IxTeT: an integrated approach for plan generation and scheduling, in: Proceedings of INRIA/IEEE Symposium on Emerging Technologies and Factory Automation, 1995, pp. 485–495.

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

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

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

[22] 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.

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

[24] 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.

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

[26] P. Laborie, D. Godard, Self-adapting large neighbourhood search: Application to single-mde scheduling problems, in: Proceedings of MISTA-07, 2007.

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

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

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

[30] 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.

[31] T. Vidal, M. Ghallab, Constraint-based temporal management in planning: the IxTeT approach, in: Proc. of 12th European Conference on AI, 1996.

[32] A. Tate, Generating project networks, in: Proceedings of IJCAI-77, 1977.

[33] B. Drabble, A. Tate, The use of optimistic and pessimistic resource profiles to inform search in an activity based planner, in: Proc. of 2nd Conference on AI Planning Systems (AIPS), AAAI Press, 1994.

[34] A. Tate, The <I-N-OVA> Constraint Model of Plans, in: Proceedings of the Third International Conference on Artificial Intelligence Planning Systems, 1996, pp. 221–228.

[35] N. Muscettola, HSTS: Integrating planning and scheduling, in: M. Zweben, M. Fox (Eds.), Intelligent Scheduling, Morgan Kaufmann, San Mateo, CA, 1994, pp. 169–212.

[36] M. Ai-Change, J. Bresina, L. Charest, J. Hsu, A. Jónsson, R. Kanefsy, P. Maldegue, P. Morris, K. Rajan, J. Yglesias, MAPGEN: Mixed intitive activity planning for the Mars Exploratory Rover mission, in: Proceedings of Demonstration Systems Track, ICAPS'03, 2003.

[37] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee, Iterative repair planning for spacecraft operations in the ASPEN system, in: International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS), 1999.

[38] S. Vere, Planning in time: Windows and durations for activities and goals, IEEE Trans. on Pattern Analysis and MI 5 (1983) 246–267.

[39] 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.

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

[41] 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.

[42] 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)))
```