Extending TALplanner with Concurrency and Resources

Jonas Kvarnström and **Patrick Doherty** and **Patrik Haslum**[1]

Abstract. We present TALplanner, a forward-chaining planner based on the use of domain-dependent search control knowledge represented as temporal formulas in the Temporal Action Logic (TAL). TAL is a narrative based linear metric time logic used for reasoning about action and change in incompletely specified dynamic environments. TAL is used as the formal semantic basis for TALplanner, where a TAL goal narrative with control formulas is input to TALplanner which then generates a TAL narrative that entails the goal formula. We extend the sequential version of TALplanner, which has previously shown impressive performance on standard benchmarks, in two respects: 1) TALplanner is extended to generate concurrent plans, where operators have varied durations and internal state; and 2) the expressiveness of plan operators is extended for dealing with several different types of resources. The extensions to the planner have been implemented and concurrent planning with resources is demonstrated using an extended logistics benchmark.

In W.Horn (ed.): ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence, IOS Press, Amsterdam, 2000.

1 INTRODUCTION

Recently, Bacchus and Kabanza et al. [3, 4, 10] have been investigating the use of modal temporal logics to express domain-specific search control knowledge for a forward-chaining planner called TLPLAN, an approach similar to the negative heuristics used by Kibler and Morris [12]. TLPLAN has demonstrated impressive improvements in efficiency when compared to many recent planners such as Blackbox [11] and IPP [14], the best performers in the AIPS'98 planning competition [1] (see [4] for comparisons).

In [8], we began exploring a somewhat different approach, representing not only control rules but also operators, goal statements, and domain constraints using TAL (Temporal Action Logics [7]), a family of narrative-based first-order temporal logics for reasoning about action and change in incompletely specified dynamic worlds. The resulting planner, TALplanner [8, 25], takes a TAL goal narrative as input and generates a new narrative where a set of TAL action occurrences (corresponding to plan steps) have been added.

In this process, TAL serves as a reference formalism providing a formal semantics for all parts of the specification of a planning problem, which provides a very solid basis for experimenting with planners and formally verifying the correctness of generated plans. In fact, TAL is a highly expressive formalism, allowing the modeling of STRIPS or ADL operators as well as actions with duration, non-deterministic effects, incompletely specified timing of actions, delayed effects, concurrent actions, qualifications to actions, side-effects of actions, state and domain constraints, and incomplete knowledge about the initial state. Currently, only part of this expressivity is allowed by TALplanner. The intention is to incrementally extend the planner to handle larger subsets of TAL, testing the extensions empirically using existing benchmarks when possible and extending benchmarks when necessary.

This paper reflects the methodology by extending the sequential version of TALplanner to a version which generates concurrent plans and by extending the expressivity of plan operators to allow the representation of a number of different resource types. The extensions are verified and tested for efficiency by extending a planning benchmark from the logistics domain, generating new domain-dependent search control formulas and testing the extensions against the unextended benchmark without use of concurrency or resources.

2 TAL: TEMPORAL ACTION LOGICS

The approach we use for reasoning about action and change is as follows. First, represent a narrative description as a set Υ of labeled statements in a surface language $\mathcal{L}(\mathrm{ND})$, a high-level language for representing observations, action types (plan operators), action occurrences, dependency constraints, and domain constraints. Second, generate the corresponding theory $\Delta = \Gamma \cup \Gamma_{\mathsf{fnd}}$ in $\mathcal{L}(\mathrm{FL})$, an order-sorted first-order language with a linear, discrete time structure. Γ is the translation of Υ into $\mathcal{L}(\mathrm{FL})$ using the *Trans* function [7], and Γ_{fnd} is a set of foundational axioms including unique names axioms and temporal structure axioms. Third, deal with the well known frame, qualification and ramification problems via a circumscription axiom which is easily reducible to a 1st-order formula via syntactic transformation. Let Γ' be the result of applying a circumscription policy to the dependency constraints and action types in Γ and let $\Delta' = \Gamma' \wedge \Gamma_{\mathsf{fnd}}$. Then, α is entailed by Υ iff $\Delta' \models \alpha$. (See [7] for details regarding TAL and [15] for an implementation.)

3 TALPLANNER

Since earlier TAL logics had no provision for goals or control rules, we define an extended surface language $\mathcal{L}(\mathrm{ND})^*$ which provides two new statement classes: Intended goal statements (labeled igoal) and goal control statements (labeled gctrl), described in the following two subsections. This is in line with the TAL philosophy of adding macros but keeping the underlying base language $\mathcal{L}(\mathrm{FL})$ unchanged.

TALplanner takes a goal narrative description $\mathcal{GN}$ in $\mathcal{L}(\mathrm{ND})^*$ as input and generates a plan narrative $\mathcal{N}_p$ in $\mathcal{L}(\mathrm{ND})$. If $\mathcal{GN}_{\mathsf{igoal}}$ and $\mathcal{GN}_{\mathsf{gctrl}}$ are the sets of intended goal statements and goal control statements in $\mathcal{GN}$, then (assuming the planner is successful) the generated plan narrative $\mathcal{N}_p$ is $(\mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{igoal}} \cup \mathcal{GN}_{\mathsf{gctrl}})) \cup \mathcal{GN}_{\mathsf{occ}}$, where $\mathcal{GN}_{\mathsf{occ}}$ is the set of action occurrences (plan steps) generated by the planner. Thus, $\mathcal{GN}_{\mathsf{igoal}}$ and $\mathcal{GN}_{\mathsf{gctrl}}$ are only used in the plan synthesis algorithm, and the output of the planner is a pure $\mathcal{L}(\mathrm{ND})$ narrative.

[1] Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, {jonkv,patdo,pahas}@ida.liu.se

The planning algorithms presented in this paper are sound in the sense that if a narrative description $\mathcal{N}_p$ is returned given $\mathcal{GN}$ as input, then $\Delta'_{\mathcal{N}_p} \models \mathit{Trans}([\mathbf{t}] \bigwedge \mathcal{GN}_{\mathsf{igoal}})$, where $\Delta'_{\mathcal{N}_p}$ is the circumscribed logical theory in $\mathcal{L}(\mathrm{FL})$ corresponding to $\mathcal{N}_p$ and **t** is the end timepoint of the last action occurrence in $\mathcal{GN}_{\mathsf{occ}}$. Completeness is a more difficult issue and depends on the control rules used for each domain.

3.1 Goals in $\mathcal{L}(\mathbf{ND})^*$

As demonstrated by TLPLAN, domain-dependent control rules are often more effective if they can refer to the intended goal. TLPLAN allows this by using a goal modality, but since TAL is not a modal logic, TALplanner handles it differently. Due to lack of space, we will present a restricted translation that requires conjunctive goals.

An *intended goal statement* in $\mathcal{L}(\mathrm{ND})^*$ is a conjunction of expressions of the form $f \hat{=} v$. This defines the set of goal states. Negative goals of the form $\neg f$ can be written as $f \hat{=} \mathsf{false}$.

For each fluent $f : \mathsf{dom}_1 \times \cdots \times \mathsf{dom}_n \rightarrow \mathsf{dom}$, we add a corresponding goal fluent $\mathsf{goal}_f : \mathsf{dom}_1 \times \cdots \times \mathsf{dom}_n \times \mathsf{dom} \rightarrow \{\mathsf{true}, \mathsf{false}\}$. The intention is that $\mathsf{goal}_f(\overline{x}, v)$ should be true exactly when the goal requires that $f(\overline{x}) \hat{=} v$. To achieve this, we make each goal fluent durational with default value false. By translating the intended goal statement $\wedge_{i=1}^{n} f_i(\overline{x}_i) \hat{=} v_i$ into the dependency constraint $\forall t.\ I([t] \ \wedge_{i=1}^{n} \mathsf{goal}_{f_i}(\overline{x}_i, v_i) \hat{=} \mathsf{true})$, we force the appropriate goal fluents to be true; all other goal fluents remain false. Finally, the set of atomic expressions in $\mathcal{L}(\mathrm{ND})^*$ is extended to include *goal expressions* of the form $\mathsf{goal}(f(\overline{x}) \hat{=} v)$, translated into $\forall t.[t]\ \mathsf{goal}_f(\overline{x}, v) \hat{=} \mathsf{true}$. Thus, goal expressions can be used in control rules as well as in domain constraints and operator preconditions and their semantics is integrated with that of TAL.

3.2 Control rules in $\mathcal{L}(\mathbf{ND})^*$

Unlike traditional search heuristics based on a single state, TLPLAN uses control formulas in a linear modal tense logic to place constraints on the entire state sequence induced by a plan prefix, and also allows formulas to refer to the given goal. There are four temporal modalities, U (until), $\Diamond$ (eventually), $\Box$ (always), and $\bigcirc$ (next), as well as a goal modality. A plan prefix can be seen as inducing an infinite state sequence, where the final state is repeated an infinite number of times. This sequence can be seen as a model, and control rules can be viewed as model filterers that rule out state sequences that cannot lead to plans or that lead to suboptimal or redundant plans.

As an example, consider the gripper domain, where a robot with a number of grippers moves objects between rooms. If some of the objects the robot is carrying should be in the current room, it should immediately drop one of those objects – it should not move to another room. For the special case of a robot with a single gripper, this can be expressed using the modal control rule, $\Box\,\forall o[\mathsf{carry}(o) \wedge \exists l[\mathsf{at}(\mathsf{robby}, l) \wedge \mathsf{goal}(\mathsf{at}(o, l))] \rightarrow \bigcirc\,\neg\mathsf{carry}(o)]$.

A plan prefix should be retained unless it can be shown that *every* plan beginning with this prefix will violate some control rule. In TLPLAN, this is achieved by using formula progression and viewing a plan prefix as a state sequence whose last state is an idle state. The TALplan/modal algorithm [8] uses a similar approach, although the progression algorithm is somewhat different due to the use of actions with duration and internal state. In this approach, the temporal modalities can be viewed as special types of macro operators whose semantics is defined by the translation into plain $\mathcal{L}(\mathrm{ND})$ [8].

However, due to the use of explicit time in TAL, it is also possible to specify control rules in terms of formulas to be evaluated rather than progressed, as in the TALplan/non-modal algorithm [8]. We introduce a temporal constant t_* and ensure that it is always bound to the timepoint of the last *fixed* state – the last state that is guaranteed not to be changed by the addition of new operators. Then, control rules can be contextualized by referring to t_*, ensuring that potentially valid plan prefixes are not discarded prematurely. For example, the control formula above can be written as $\forall t, o[t < t_* \wedge [t]\ \mathsf{carry}(o) \wedge \exists l[\mathsf{at}(\mathsf{robby}, l) \wedge \mathsf{goal}(\mathsf{at}(o, l))] \rightarrow [t+1]\ \neg\mathsf{carry}(o)]$.

Each approach has advantages and disadvantages in terms of computational complexity. Although a naive progression algorithm might do better than a naive formula evaluator, as demonstrated in benchmark tests [8], we have found that evaluation enables certain optimizations that are more difficult to apply to a progression algorithm, making the current TALplanner implementation significantly faster when using evaluation. An additional advantage is the significantly lower memory usage due to not needing to store a progressed control formula in each search node. Developing and analyzing optimization choices is currently being pursued as an active research issue.

In the remainder of the paper, we will concentrate on the use of evaluated control formulas. The next section presents TALplan/non-modal, a variation of the algorithm presented in [8]. In the following sections, we extend this algorithm with resources and concurrency.

4 SEQUENTIAL TALPLANNER

We will now present a sequential planner, TALplan/non-modal, based on a combination of those found in Bacchus [4] and Kabanza [10]. Although there are many differences, the most important distinctions are that the algorithms are modified for the TAL family of logics and the notion of a narrative, and that formula evaluation is used in place of progression. The current algorithm has the following restrictions:

- The initial state must be completely specified.
- Actions must be deterministic (but can be context-dependent).
- Dependency constraints (and side effects) are not allowed.
- Domain constraints must be of the form $\forall t, \overline{x}\ [t \leq t_* \rightarrow ([t] f(\overline{x}) \leftrightarrow \phi)]$. This provides the possibility to use defined predicates (essentially, state variables defined in terms of formulas).

However, we do allow actions with duration and internal state changes. Also, although the goal formula translation from Section 3.1 only handles conjunctive goals, the algorithm itself and its current implementation allows arbitrary goals. Many existing planners have much less expressivity, even with these constraints.

Input: An initial goal narrative $\mathcal{GN}$.
Output: A plan narrative $\mathcal{N}_p$ which entails the goal $\bigwedge \mathcal{GN}_{\mathsf{igoal}}$.

```
1  procedure TALplan/non-modal($\mathcal{GN}$)
2  acc ← {}                      // Visited states for cycle checking
3  Open ← $\langle\langle 0, 0, \mathcal{GN}\rangle\rangle$    // Stack (depth-first search)
4  while Open ≠ ⟨⟩ do
5    $\langle\tau, \tau', \mathcal{GN}\rangle$ ← pop(Open)
6    $\mathcal{N} \leftarrow (\mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{igoal}} \cup \mathcal{GN}_{\mathsf{gctrl}}))$
7    if $\Delta'_{\mathcal{N}\cup\{t_*=\tau'\}} \not\models \mathit{Trans}(\neg \bigwedge \mathcal{GN}_{\mathsf{gctrl}})$ then
8      if $\Delta'_{\mathcal{N}} \models \mathit{Trans}([\overline{\tau}] \bigwedge \mathcal{GN}_{\mathsf{igoal}})$ then return $\mathcal{N}$
9      if (state at time $\tau'$ for $\mathcal{N}$) ∉ acc then
10       acc ← acc ∪ {(state at time $\tau'$ for $\mathcal{N}$)}
11       Expand($\mathcal{GN}, \tau'$, Open)
```

Some explanations may be in order. Line 7 checks whether the negation of the control rules is entailed, or, equivalently, whether any control rule is violated. Lines 9–11 are responsible for redundancy checking and expansion: If a plan prefix satisfying the control rules does not achieve the goal and is not redundant, then we push its successors on the stack. Naturally, the search strategy can easily be modified to use breadth-first search or various forms of heuristic search.

The Expand algorithm is responsible for finding all successors of a plan prefix. Here, this is done by finding all operator instances whose preconditions are satisfied at time s in $\mathcal{GN}$. Different implementations of Expand can provide different lookahead, decision-theoretic and filtering mechanisms for choice of actions.

```
1  procedure Expand($\mathcal{GN}, s$, Open)
2  $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\text{igoal}} \cup \mathcal{GN}_{\text{gctrl}})$
3  for all a($\overline{x}$) $\in$ ActionTypes($\mathcal{N}$) do
4    for all $[s,t]$ a($\overline{c}$) $\in$ Instantiate($s$, a($\overline{x}$)) do
5      if $\Delta'_{\mathcal{N}} \models Trans([s]$ precond(a($\overline{c}$))) then
6        Open $\leftarrow$ Open $\cup \{\langle s, t, \mathcal{GN} \cup \{[s,t]$ a($\overline{c}$)$\}\rangle\}$
```

5 TALPLANNER AND CONCURRENCY

In this section, we will show how TALplanner has been extended to generate concurrent plans.

Since TALplanner is fundamentally based on forward chaining, the definition of the set of successors for each plan prefix (or, equivalently, the definition of the search tree) is one of the most important features of the planning algorithm.

Generally, let $p = \langle [s_1, t_1]\ o_1; \ldots; [s_n, t_n]\ o_n \rangle$ be a (possibly empty) plan prefix. Then, any successor must be of the form $\langle [s_1, t_1]\ o_1; \ldots; [s_n, t_n]\ o_n; [s, t]\ o \rangle$, where the precondition of o is satisfied at its invocation timepoint $s \geq 0$. If $n > 0$, we also require that $s \geq s_n$: We never try to invoke a new operator earlier than an existing operator. Thus, all states up to and including s_n are *fixed*, and will never be modified in any successor of p, which is important for the efficiency of the implementation.

Sequential planning adds the condition that $s = t_n$ (or $s = 0$ if $n = 0$): A new action is always invoked exactly when the previous action finished executing. For concurrent planning, this condition must be relaxed. Let $\overline{\tau}$ be the maximum of all ending timepoints of all actions in p. The states from s_n up to $\overline{\tau}$ may all be different, due to actions having effects in their intermediate states, but since nothing can change after $\overline{\tau}$ there is no point in considering successors with $s > \overline{\tau}$. Thus, we require that $s_n \leq s \leq \overline{\tau}$.

There is an additional difficulty associated with successors where $s = s_n$: The search tree could contain redundant pairs of plan prefixes such as $\langle [0,3]\ o_1; [0,3]\ o_2 \rangle$ and $\langle [0,3]\ o_2; [0,3]\ o_1 \rangle$. To avoid this redundancy, we assume the existence of a total order $\succ$ on operator instances, and if $s = s_n$, we require that $o \succ o_n$.

This definition induces a possibly infinite search tree, which can be traversed using standard search strategies such as breadth first search or various forms of heuristic search methods. In the algorithms below, we will use depth-first search, relying on control rules to prune nodes that would not take us closer to the goal.

Input: An initial goal narrative $\mathcal{GN}$.
Output: A plan narrative $\mathcal{N}_p$ which entails the goal $\bigwedge \mathcal{GN}_{\text{igoal}}$.

```
1  procedure TALplan/conc($\mathcal{GN}$)
2  acc $\leftarrow \{\}$                      // Visited states for cycle checking
3  Open $\leftarrow \langle\langle 0, 0, 0, \mathcal{GN}\rangle\rangle$   // Stack (depth-first search)
4  while Open $\neq \langle\rangle$ do
5    $\langle \tau, \tau', \overline{\tau}, \mathcal{GN}\rangle \leftarrow$ pop(Open)
6    $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\text{igoal}} \cup \mathcal{GN}_{\text{gctrl}})$
7    if $\Delta'_{\mathcal{N}\cup\{t_*=\overline{\tau}\}} \not\models Trans(\neg \bigwedge \mathcal{GN}_{\text{gctrl}})$ then
8      state $\leftarrow$ (state at time $\overline{\tau}$ for $\mathcal{N}$)
9      if not exists state$'$ $\in$ acc: better-or-equal(state$'$, state) then
10       acc $\leftarrow$ acc $\cup$ {state}
11       if $\Delta'_{\mathcal{N}} \models Trans([\overline{\tau}]\ \bigwedge \mathcal{GN}_{\text{igoal}})$ then return $\mathcal{N}$
12       else Expand($\mathcal{GN}, \tau, \overline{\tau}, \overline{\tau}$, Open)
13   for $s$ from $\overline{\tau} - 1$ downto $\tau$ do
14     if $\Delta'_{\mathcal{N}\cup\{t_*=s\}} \not\models Trans(\neg \bigwedge \mathcal{GN}_{\text{gctrl}})$ then
15       Expand($\mathcal{GN}, \tau, s, \overline{\tau}$, Open)
```

Successors may be added at any timepoint between τ and $\overline{\tau}$. Note that they are pushed on the stack in reverse temporal order, since we prefer invoking operators as early as possible. The case where *all* states up to $\overline{\tau}$ are fixed must be treated separately (lines 7–12): Only here can a plan possibly be found, and only here can redundancy (cycle) checking be performed. The ordering relation better-or-equal is described in Section 5.2.

As discussed above, the Expand algorithm must be modified somewhat to prevent the search tree from containing redundant pairs of plan prefixes (line 5 below). Also, $\overline{\tau}$ must be updated and stored in each search node (max($t, \overline{\tau}$) in line 7).

```
1  procedure Expand($\mathcal{GN}, \tau, s, \overline{\tau}$, Open)
2  $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\text{igoal}} \cup \mathcal{GN}_{\text{gctrl}})$
3  for all a($\overline{x}$) $\in$ ActionTypes($\mathcal{N}$) do
4    for all $[s,t]$ a($\overline{c}$) $\in$ Instantiate($s$, a($\overline{x}$)) do
5      if $s \neq \tau$ or a($\overline{c}$) $\succ$ lastact($\mathcal{N}$) then
6        if $\Delta'_{\mathcal{N}} \models Trans([s]$ precond(a($\overline{c}$))) then
7          push $\langle s, t, \max(t, \overline{\tau}), \mathcal{GN} \cup \{[s,t]$ a($\overline{c}$)$\}\rangle$ on Open
```

5.1 Concurrency and Control Rules

Control rules should normally only be applied to fixed states, since if a violation depends on a non-fixed state, it might be possible to "repair" it by adding a new concurrent action that modifies the non-fixed state. However, there are control rules for which this can never be the case. Consider the gripper rule discussed in Section 3.2. Clearly, once the robot has picked up an object that was already in its goal location, adding actions to the plan prefix cannot possibly undo this violation.

Fortunately, many control rules do have this property. Although this could sometimes be detected automatically, especially in trivial cases such as the gripper control rule discussed above, this has currently been left as a topic for future research. Instead, TALplanner allows rules to be marked as *unrepairable*. For such rules, the planner always uses $t_* = \overline{\tau}$ rather than $t_* = s$, which increases performance by allowing the planner to detect violations earlier.

5.2 Concurrency and Resources

Although resources can be modeled in plain TAL, the formalization can be quite complex, especially when concurrency is involved. To facilitate the use of resources, we therefore introduce two new macros in $\mathcal{L}(\text{ND})^*$. The resource macro is used for declaring resource fluents and their maximum and minimum allowed values, while the operator macro allows for a more structured way of defining operators and their preconditions, effects, and resource usage.

There are five kinds of resource effects. At any delay t from its invocation timepoint, an operator can *produce* resources (which cannot be consumed until $t+1$). Resources can be *consumed* at t, which leaves room for more production at $t+1$. Resources can be *borrowed* , either *exclusively* or *non-exclusively* (shared with other non-exclusive borrowers). Finally, resources can be *assigned* a new value.

Resources and resource effects are translated into fluents, plain action effects, and domain constraints. The translation also uses control rules to ensure that resource values are within the allowed range at all times. Due to lack of space, the translation is not shown here.

With resources, plain cycle checking is generally too weak. For example, if moving consumes fuel, moving from a to b to a leads to a new state where less fuel is available. Therefore, each resource can be associated with a preference: *more*, *less*, or *none*. Here, we would always prefer to have more fuel. This induces a partial order on states, better-or-equal, used in the planner. Since resources can be used in preconditions, effects, control rules, and goals, generating preferences automatically can be quite complex for non-trivial domains and has currently been left as a future research topic.

6 THE LOGISTICS DOMAIN

In the standard logistics domain, a number of packages can be transported by truck between locations in the same city and by airplane between cities. The goal is normally to deliver each package from its initial location to its destination.

This domain is naturally concurrent. For example, different vehicles can be moved and different packages loaded and unloaded, relatively independent of one another. This kind of concurrency exists in many standard domains, and has motivated approaches such as partial-order planning and Graphplan's use of parallel actions [18, 6].

However, few planners have considered the duration of concurrent actions. To create efficient plans, a planner must be able to plan a sequence of several "short" actions, like loading, driving and unloading a truck, in parallel with a "long" action, like flying an airplane between distant cities. Assuming all actions to have unit duration is very restrictive.

The logistics domain is also easily extended to include resources. For example, the carrying capacity of different vehicles is an example of a property most naturally modelled as a resource.

6.1 Plan Operators, Resources and Control Rules

In keeping with the standard formulation, the following features, resources and operators describe the domain. Note that TAL is an order-sorted logic, and all variables are typed; the types used are loc, with subtype airport, city, and thing. The type thing has subtypes obj and vehicle (which in turn has subtypes truck and plane). The following is written using macros in $\mathcal{L}(\mathrm{ND})^*$ and can be translated into TAL formulas. Note that $[t]\ \alpha$ means that α holds at time t.

```
#feature at(thing, loc), in(obj, vehicle), moving(vehicle, loc): boolean
#feature city_of(loc): city
#feature dist(loc, loc), size(obj): integer

#resource use_of(thing) :domain integer :min 0 :max 1 :preference :none
#resource space(vehicle) :domain integer
 :min 0 :max capacity(vehicle) :preference :more

#operator load(obj, vehicle, loc) :at t
 :precond [t] at(obj, loc) & at(vehicle, loc)
 :resources [+1] :borrow-nonex use_of(vehicle) :amount 1,
               [+1] :borrow use_of(obj) :amount 1
               [+1] :consume space(vehicle) :amount size(obj)
 :effects [+1] at(obj, loc) := false, [+1] in(obj, vehicle) := true

#operator unload(obj, vehicle, loc) :at t
 :precond [t] in(obj, vehicle) & at(vehicle, loc)
 :resources [+1] :borrow-nonex use_of(vehicle) :amount 1,
               [+1] :borrow use_of(obj) :amount 1
               [+1] :produce space(vehicle) :amount size(obj)
 :effects [+1] in(obj, vehicle) := false, [+1] at(obj, loc) := true

#operator drive(truck, loc1, loc2) :at t
 :precond [t] at(truck, loc1) & city_of(loc1) == city_of(loc2) & loc1 != loc2
 :resources [+1,+dist(loc1,loc2)/2] :borrow use_of(truck) :amount 1,
 :effects [+1] at(truck,loc1) := false,
           [+1,+dist(loc1,loc2)/2-1] moving(truck,loc2) := true,
           [+dist(loc1,loc2)/2] at(truck,loc2) := true
           [+dist(loc1,loc2)/2] moving(truck,loc2) := false

#operator fly(plane, airport1, airport2) :at t
 :precond [t] at(plane, airport1) & airport1 != airport2
 :resources [+1,+dist(airport1,airport2)/5] :borrow use_of(plane) :amount 1,
 :effects [+1] at(plane,airport1) := false,
           [+1,+dist(airport1,airport2)/5-1] moving(plane,airport2) := true,
           [+dist(airport1,airport2)/5] at(plane,airport2) := true
           [+dist(airport1,airport2)/5] moving(plane,airport2) := false
```

The city_of, dist and size features define parameters of the problem. The use_of resource ensures that an object is never used in conflicting concurrent actions (in other words, it provides a form of mutual exclusion). Loading and unloading packages into or from a vehicle borrows the use_of resource non-exclusively, allowing several loading or unloading actions involving the vehicle to take place concurrently. Actions that move the vehicle borrow the resource exclusively, so that it can never be moved to two different destinations at the same time, or moved during loading or unloading. Vehicles also have limited carrying capacity, modeled using the space resource. Unloading produces space, while loading consumes space; the precondition that there must be enough space is implicit.

The following domain constraints define abbreviations used in the control rules: An object needs to be moved by plane from loc1 if its destination loc2 is in another city, and it needs to be unloaded at loc1 if loc1 is in the same city as its destination.

```
#dom forall t [ [t] move-by-plane(obj, loc1) <->
 exists loc2 [ goal(at(obj, loc2)) & ([t] city_of(loc1) !== city_of(loc2)) ]

#dom forall t [ [t] unload-from-plane(obj, loc1) <->
 exists loc2 [ goal(at(obj, loc2)) & ([t] city_of(loc1) == city_of(loc2)) ]
```

The following control rules are inspired by those used by TLPLAN, but have been modified for concurrency and the use of vehicles with limited space. Briefly, an airplane should remain where it is until all packages that should be moved by the plane, and that actually fit into the plane, have been loaded; note the explicit reference to resources. It should only move to locations where it needs to deliver or pick up packages. If a package is at its destination, it should not be moved. A package should only be loaded onto a plane if a plane (rather than a truck) is needed to move it, and should only be unloaded if it is in its destination city. Similar control rules are needed for trucks.

```
#control :unrepairable   forall t, plane, loc [
   [t] at(plane, loc) & exists obj [
     (at(obj, loc) & move-by-plane(obj, loc) & size(obj) <= space(plane)) |
     (in(obj, plane) & unload-from-plane(obj, loc)) ]
   -> [t+1] at(plane, loc) ]

#control :unrepairable   forall t, plane, loc [
   ([t] at(plane, loc)) ->
   ([t+1] at(plane, loc))
   | exists loc2 [
     ([t+1] (at(plane, loc2) | moving(plane, loc2))) &
     ([t+1] exists obj [ in(obj, plane) & unload-from-plane(obj, loc2) ])]
   | exists loc2 [
     ([t+1] (at(plane, loc2) | moving(plane, loc2))) &
     ([t+1] exists obj [ at(obj, loc2) & move-by-plane(obj, loc2) &
       size(obj) <= space(plane) ]) &
     (forall p2 [[t+1] (at(p2, loc2) | moving(p2,loc2)) -> p2 = plane])]]

#control :unrepairable   forall t, obj, loc [
   [t] at(obj, loc) & goal(at(obj, loc)) -> [t+1] at(obj, loc) ]

#control :unrepairable   forall t, obj, plane [
   [t] !in(obj, plane) & forall loc [ at(obj, loc) -> !move-by-plane(obj, loc)] ->
   [t+1] !in(obj, plane) ]

#control :unrepairable   forall t, obj, plane [
   [t] in(obj, plane) & forall loc [at(plane, loc)->!unload-from-plane(obj, loc)] ->
   [t+1] in(obj, plane) ]
```

7 TEST RESULTS

We have tested both TLPLAN and TALplanner in a number of standard planning domains using suitable control rules.[2] Here, we will concentrate on the 30 logistics problems from the AIPS'98 planning competition [1]. For three of those problems, TLPLAN ran out of memory; the others required between 0.4 seconds and 17 hours to

[2] All tests were performed using a Pentium II-333 PC with 256 MB of memory, running Windows NT.

complete. TALplan/non-modal proved to be considerably more efficient, creating a 274-operator plan for the most complex problem in under 0.7 seconds using less than 10 MB of memory.

With a use_of resource for mutual exclusion, TALplan/conc required around 0.9 seconds for the most complex problem.

Finally, we have tested TALplan/conc on extended logistics problems based on the same 30 problems but using the operators and control rules presented above. Trucks had 5 units of space, while planes had 25 units. Package sizes were between 1 and 3, and distances varied between 1 and 25. Due to certain optimizations not yet being implemented for actions with variable duration, TALplanner now needed around 11 minutes to complete the most complex problem. Once these optimizations have been implemented, we expect performance to improve by at least an order of magnitude.

TALplanner also had a very successful showing at the AIPS-2000 Planning Competition, where it won the "Distinguished Planner" award in the domain dependent planning track and first place in the Schindler Miconic-10 Intelligent Elevator Control planning competition. For the results of the competition and a view of comparative graphs, see Bacchus [2].

Further test results for TALplanner and complete domain specifications for a number of planning domains will be available on the WWW [25].

8 RELATED WORK

Planning with domain-dependent control information, and "knowledge-based planning" in general [30], has for a long time been investigated in the context of HTN planning [29, 26], case-based planning, and some kinds of reactive planning [5]. The idea of planning with domain information in the form of control rules in the "classical" state-space setting in fact dates back to 1981 [12], although it is only more recently that the idea has re-emerged and been applied to forward-chaining [4] and SAT-based [9] planners.

Time and resource reasoning has been integrated in several HTN planners (*e.g.* SHOP [20]). In classical planning, though many planners form parallel plans (*e.g.* Graphplan [6] and descendants), not so many treat operators with non-unit durations and internal state. Several approaches in this direction, *e.g.* Deviser [27], Zeno [21], IxTeT [16] and TripTic [23], are based on partial-ordered planning combined with temporal constraint reasoning. A more recent approach is Temporal Graphplan [24].

Resources, or rather continuous state variables, have been integrated in several different planning approaches, for example in the Graphplan based RIPP [13] planner, constraint propagation [22], integer-linear programming [28] and a combination of LP and SAT encoding [31]. HSTS [19] and parcPlan [17], both constraint-based planners, integrate time and resources in a coherent framework.

9 CONCLUSIONS

We have presented a forward-chaining planner, TALplanner, with a formal semantics based on the use of a temporal action logic (TAL). TALplanner has been extended to generate concurrent plans and to deal with a number of different resource types. The planner is fully implemented and has been demonstrated and tested using an extended logistics benchmark. Empirical results have previously shown that TALplanner is one of the fastest and most memory efficient domain-dependent planners currently being developed. The new extensions increase the suite of application domains where the planner can be used. We are currently working on relaxing the closed world assumption built into the planner and introducing limited types of side effects and nondeterministic plan operators.

REFERENCES

[1] AIPS98. Artificial Intelligence Planning Systems: 1998 Planning Competition. http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html.

[2] F. Bacchus. AIPS 2000 planning competition results. Presentation available at http://www2.cs.toronto.edu/aips2000.

[3] F. Bacchus and F. Kabanza, 'Planning for temporally extended goals', *Annals of Mathematics and Artificial Intelligence*, **22**, 5–27, (1998).

[4] F. Bacchus and F. Kabanza, 'Using temporal logics to express search control knowledge for planning', *Artificial Intelligence*, (1998). Submitted for publication.

[5] M. Beetz and D. McDermott, 'Improving robot plans during their execution', in *Proc. Artificial Intelligence Planning Systems*, (1994).

[6] A. L. Blum and M. L. Furst, 'Fast planning through graph analysis', *Artificial Intelligence*, **90**(1–2), 281–300, (1997).

[7] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström, 'TAL: Temporal Action Logics – language specification and tutorial', *Linköping Electronic Articles in Computer and Information Science*, **3**(15), (September 1998). Available at http://www.ep.liu.se/ea/cis/1998/015.

[8] P. Doherty and J. Kvarnström, 'TALplanner: An empirical investigation of a temporal logic-based forward chaining planner', in *Proc. TIME'99*, (1999).

[9] Y. Huang, B. Selman, and H. Kautz, 'Control knowledge in planning: Benefits and tradeoffs', in *Proc. 16th National Conference on Artificial Intelligence*, (1999).

[10] F. Kabanza, M. Barbeau, and R. St-Denis, 'Planning control rules for reactive agents', *Artificial Intelligence*, **95**, 67–113, (1997).

[11] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. www.research.att.com/∼kautz.

[12] D. Kibler and P. Morris, 'Don't be stupid', in *Proc. IJCAI'81*, (1981).

[13] J. Koehler, 'Planning under resource constraints', in *Proc. ECAI'98*, (1998).

[14] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, 'Extending planning graphs to an ADL subset', in *European Conference on Planning*, (1997). http://www.informatik.uni-freburg.de/∼koehler/ipp.html.

[15] J. Kvarnström and P. Doherty. VITAL. An on-line system for reasoning about action and change using TAL, 1997–2000. Available at http:// http://www.ida.liu.se/∼jonkv/vital.html.

[16] P. Laborie and M. Ghallab, 'Planning with sharable resource constraints', in *Proc. IJCAI'95*, (1995).

[17] J. M. Lever and B. Richards, '*parc*PLAN: A planning architecture with parallel actions, resources and constraints', in *Proc. 9th International Symposium on Methodologies for Intelligent Systems*, (1994).

[18] D. McAllester and D. Rosenblitt, 'Systematic nonlinear planning', in *Proc. 9th National Conference on Artificial Intelligence*, (1991).

[19] N. Muscettola, 'Integrating planning and scheduling', In Zweben and Fox [32].

[20] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, 'SHOP: Simple hierarchical ordered planner', in *Proc. IJCAI'99*, (1999).

[21] J. S. Penberthy and D. S. Weld, 'Temporal planning with continous change', in *Proc. 12th National Conf. on Artificial Intelligence*, (1994).

[22] J. Rintanen and H. Jungholt, 'Numeric state variables in constraint-based planning', in *Proc. 5th European Conf. on Planning*, (1999).

[23] E. Rutten and J. Hertzberg, 'Temporal planner = nonlinear planner + time map manager', *AI Communications*, **6**(1), 18–26, (1993).

[24] D. E. Smith and D. S. Weld, 'Temporal planning with mutual exclusion reasoning', in *Proc. IJCAI'99*, (1999).

[25] TALplanner home page. http://www.ida.liu.se/labs/kplab/talplanner.

[26] A. Tate, B. Drabble, and R. Kirby, 'O-Plan2: An open architecture for command, planning and control', In Zweben and Fox [32], 213 – 239.

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

[28] T. Vossen, M. Ball, A. Lotem, and D. Nau, 'On the use of integer programming models in AI planning', in *Proc. IJCAI'99*, (1999).

[29] D. E. Wilkins, 'Can AI planners solve practical problems?', *Computational Intelligence*, **6**(4), 232 – 246, (1990).

[30] D. E. Wilkins and M. desJardins, 'A call for knowledge-based planning', in *Proc. AIPS Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, (2000).

[31] S. A. Wolfman and D. S. Weld, ‘The LPSAT engine & its application to resource planning’, in *Proc. IJCAI’99*, (1999).
[32] *Intelligent Scheduling*, eds., M. Zweben and M. Fox, Morgan-Kaufmann, 1994.