# Applying Domain Analysis Techniques for Domain-Dependent Control in TALplanner

**Jonas Kvarnström**
Dept. of Computer and Information Science, Linköping University, SE-581 83 Sweden
jonkv@ida.liu.se

### Abstract

A number of current planners make use of automatic domain analysis techniques to extract information such as state invariants or necessary goal orderings from a planning domain. There are also planners that allow the user to explicitly specify additional information intended to improve performance. One such planner is TALplanner, which allows the use of domain-dependent temporal control formulas for pruning a forward-chaining search tree.

This leads to the question of how these two approaches can be combined. In this paper we show how to make use of automatically generated state invariants to improve the performance of testing control formulas. We also develop a new technique for analyzing control rules relative to control formulas and show how this often allows the planner to automatically strengthen the preconditions of the operators, thereby reducing time complexity and improving the performance of TALplanner by a factor of up to 400 for the largest problems from the AIPS-2000 competition.

## Introduction

Although all the information necessary to solve a planning problem is implicitly available in the problem definition, there can be considerable advantages in making certain kinds of knowledge available to the planner in a more explicit form. Consequently, a wide range of automatic preprocessing and domain analysis techniques exist in the literature and have been implemented in various planning systems. These techniques include the automatic generation of state constraints (Fox & Long 1998; Gerevini & Schubert 1998; 2000; Scholz 2000; Rintanen 2000), the detection of symmetric objects (Fox & Long 1999), and the removal of facts and operator instances that turn out to be irrelevant to the solution of a particular problem instance (Nebel, Dimopoulos, & Koehler 1997; Haslum & Jonsson 2000).

There has also been a recent surge of interest in domain-dependent or hand-tailored planning systems, where various kinds of explicit knowledge can be given to a planner by the domain designer rather than being extracted automatically by the planner. Though this approach requires some more work by the user, it has the advantage of allowing the use of information that is immediately, intuitively apparent to a human but not easily extracted by a machine. Thus, the two approaches complement each other in a natural manner.

One relatively recent planner belonging to the hand-tailored category is TALplanner (Kvarnström & Doherty 2000; Doherty & Kvarnström 2001; Kvarnström & Doherty 2000b). Like TLPLAN (Bacchus & Kabanza 2000), TALplanner can accept domain-dependent control information in the form of control rules, logical formulas that serve to constrain the search process and allow the planners to prune significant parts of the search tree. In its initial implementation, however, TALplanner made no use of automatic domain analysis techniques.

This leads to two interesting questions: Which of the existing techniques would still be applicable to planners using domain-dependent control rules, given the additional complexities introduced by the rules? And perhaps more importantly, what *new* types of domain analysis are made possible through the addition of a new concept – control formulas, in addition to initial state, goal and operator definitions?

The main focus of this paper is on providing one answer to the second question. A technique is presented for extracting information from the set of operators in a planning domain. In addition to using preconditions, effects, and state constraints to generate a set of atomic facts that must hold at various points during the execution of the operator, a form of state transition analysis identifies which state variables may change and when this may occur. The information extracted from the operators is used in the context of a general formula optimizer intended to improve the performance of testing control formula violations.

The formula analysis often yields a set of conditions under which an operator will always violate a control rule. Such conditions can be used to automatically strengthen the preconditions of an operator, which leads to fewer actions being applied and fewer states being expanded. This, of course, yields further performance advantages.

Together, these two techniques have proven very effective in many domains, helping TALplanner win the "distinguished planner" award in the hand-tailored track of the AIPS-2000 planning competition (Bacchus 2001). As demonstrated by the benchmark tests at the end of the paper, performance is improved by a factor of 40 for the largest logistics problems from the AIPS-2000 competition and by a factor of 400 for the largest blocks world problems.

## Contents

This paper begins with an overview of TALplanner and the use of logic formulas as control rules, using the logistics domain as a source of concrete examples. Then, we discuss how TALplanner's preprocessor analyzes the control rules to extract so called pruning constraints. This allows the planner to test control formulas incrementally as new operators are added to a partial plan, in order to avoid duplicating the work done in previous stages of the planning process. The paper continues with a description of TALplanner's formula optimizer. The optimizer is used as a basis for adapting existing domain analysis techniques for generating state invariants, as well as for introducing a new domain analysis method where information about the context in which a pruning constraint will be tested is automatically extracted from the operator definitions in a planning domain. This considerably strengthens the optimizer and often results in the elimination of quantifiers. In many cases, TALplanner can also automatically move parts of the optimized incremental pruning constraints into the operator preconditions, automatically generating so called precondition control[1]. The effectiveness of these techniques is demonstrated by a set of benchmarks using well-known planning domains.

# TAL, TALplanner and the Logistics Domain

As for any planner, TALplanner requires a formal semantics for all the concepts involved in a planning domain definition. Even though the first version of the planner was limited to creating sequential plans with single-step deterministic actions, the semantics should allow for the modeling of more complex domains, so that it will not have to be replaced or patched whenever the planner is extended.

For this reason, the semantics of TALplanner is based on the use of TAL-C (Karlsson & Gustafsson 1999; Doherty *et al.* 1998), a non-monotonic linear discrete metric time logic for reasoning about action and change. The TAL (Temporal Action Logics) family of logics has been developed for modeling domains that may include the use of incomplete information, delayed effects of actions, finite or infinite chains of indirect effects, interacting concurrent actions, and independent processes not directly triggered by action invocations. Consequently, TAL-C was seen as an ideal choice not only for the initial version of TALplanner but also for most extensions that could conceivably be implemented in the foreseeable future.

TAL is a narrative-based formalism, where a narrative is specified as a set of labeled statements in a high-level macro language $\mathcal{L}(\mathrm{ND})$ designed to be easily extended for different tasks. The basic language has statement classes for observations (labeled obs), action descriptions (acs), action occurrences (occ), domain constraints (dom), and dependency constraints (dep). For the planning task, some of these standard classes are used together with several new types of statements described below. The formal semantics of a *goal narrative* in the extended language, denoted by $\mathcal{L}(\mathrm{ND})^*$, is defined by a translation into an order-sorted base language $\mathcal{L}(\mathrm{FL})$ together with a circumscription policy providing a solution to the frame and ramification problems (Doherty 1994; Gustafsson & Doherty 1996; Doherty *et al.* 1998)

In this section, we will attempt to provide an intuitive understanding of TAL and how it is used in domain specifications using concrete examples from the logistics planning domain. For a more complete description of TAL and its use in TALplanner, see (Doherty *et al.* 1998; Kvarnström & Doherty 2000).

## Types, Fluents, and the Initial State

In the standard logistics domain, a set of packages can be transported by truck between locations in the same city and by airplane between airports in different cities.

Since TAL is order-sorted, it is possible to use typed fluents (state variables) rather than representing types as unary predicates. In addition to the standard sort boolean $=$ $\{\mathsf{true}, \mathsf{false}\}$, a hierarchy of seven sorts is defined for the entities present in the domain: The type loc (location) has the subtypes airport and city, while thing has the subtypes obj and vehicle, the latter of which has the subtypes truck and plane. There will be two boolean fluents, at(thing,loc) and in(obj,vehicle), as well as a city-valued fluent city_of(loc) denoting the city containing the location loc.

Given these fluents, the initial state of a logistics problem instance can be defined using TAL observation statements:

$$\mathsf{obs}\ [0]\ \mathsf{city\_of}(\mathsf{pos1}) \mathrel{\hat{=}} \mathsf{cit1} \wedge \mathsf{city\_of}(\mathsf{pos2}) \mathrel{\hat{=}} \mathsf{cit2} \wedge \ldots$$
$$\mathsf{obs}\ [0]\ \mathsf{at}(\mathsf{obj11}, \mathsf{pos1}) \wedge \mathsf{at}(\mathsf{tru1}, \mathsf{pos1}) \wedge \ldots$$

These observations use *fixed fluent formulas*, formulas of the form $[\tau]\ \phi$ denoting the fact that the *fluent formula* $\phi$ holds at time $\tau$. A fluent formula is a boolean combination of elementary fluent formulas of the form $f \mathrel{\hat{=}} v$, denoting the fact that the fluent $f$ takes on the value $v$. For boolean fluents, as in the second observation, the shorthand notation $f$ or $\neg f$ is allowed. The notation is also extended for open, closed, and semi-open temporal intervals. In addition to these formulas, the function *value*$(\tau, f)$ denotes the value of $f$ at time $\tau$.

## Goals

A new statement class for goals (labeled goal) is added to $\mathcal{L}(\mathrm{ND})^*$. A goal statement consists of a fluent formula that must hold in the goal state:

$$\mathsf{goal}\ \mathsf{at}(\mathsf{obj11}, \mathsf{apt1}) \wedge \mathsf{at}(\mathsf{obj23}, \mathsf{pos1}) \wedge \ldots$$

The ability to test whether a formula is entailed by the goal is very useful in the domain-dependent control rules. Therefore, a new macro is added: The *goal expression* goal$(\phi)$ holds iff the goal of this problem instance (equivalently, the conjunction of all goal statements) entails the fluent formula $\phi$. The translation into $\mathcal{L}(\mathrm{FL})$ is somewhat complex; see (Kvarnström & Doherty 2000) for further information.

## Operator Definitions and Plans

Since TAL-C is a logic for reasoning about action and change, it has a notion of actions that can be used for modeling planning operators. For example, the following statement defines a load-truck operator for the logistics domain:

[1] The use of *manually* generated precondition control has been discussed independently by Bacchus and Ady (Bacchus & Ady 1999) in the context of TLPLAN.

$$\begin{aligned}&\textsf{acs}\ [s,t]\ \textsf{load-truck}(\textit{obj},\textit{truck},\textit{loc}) \rightsquigarrow \\ &\quad [s]\ \textsf{at}(\textit{obj},\textit{loc}) \wedge \textsf{at}(\textit{truck},\textit{loc}) \rightarrow \\ &\quad I([t,t]\ \textsf{at}(\textit{obj},\textit{loc}) \hat{=} \textsf{false}) \wedge \\ &\quad I([t,t]\ \textsf{in}(\textit{obj},\textit{truck}) \hat{=} \textsf{true}) \wedge t = s+1\end{aligned}$$

This action definition uses the $I$ reassignment macro, defined by $I([\tau,\tau']\, f \hat{=} v) \stackrel{\text{def}}{=} [\tau,\tau']\, f \hat{=} v \wedge X([\tau,\tau']\, f \hat{=} v)$, where the first conjunct denotes the fact that $f$ must take on the given value throughout the given interval and the second conjunct (the $X$ macro) states that this change in fluent values should be allowed by the TAL circumscription policy. The last conjunct ($t = s + 1$) constrains this to be a single-step action.

To facilitate the addition of resource constraints and other new concepts not present in standard TAL-C, a new operator macro has been introduced. The semantics of this macro is defined by a translation into standard TAL action schemas. Using this macro, the six operators in the logistics domain can be defined as follows:

operator load-truck(*obj, truck, loc*) :at s
:precond [s] at(*obj, loc*) ∧ at(*truck, loc*)
:effects [s+1] at(*obj, loc*) := false, [s+1] in(*obj, truck*) := true

operator load-plane(*obj, plane, loc*) :at s
:precond [s] at(*obj, loc*) ∧ at(*plane, loc*)
:effects [s+1] at(*obj, loc*) := false, [s+1] in(*obj, plane*) := true

operator unload-truck(*obj, truck, loc*) :at s
:precond [s] in(*obj, truck*) ∧ at(*truck, loc*)
:effects [s+1] in(*obj, truck*) := false, [s+1] at(*obj, loc*) := true

operator unload-plane(*obj, plane, loc*) :at s
:precond [s] in(*obj, plane*) ∧ at(*plane, loc*)
:effects [s+1] in(*obj, plane*) := false, [s+1] at(*obj, loc*) := true

operator drive(*truck, loc1, loc2*) :at s
:precond [s] at(*truck, loc1*) ∧ city_of(*loc1*) ≙ city_of(*loc2*) ∧ *loc1* ≠ *loc2*
:effects [s+1] at(*truck,loc1*) := false, [s+1] at(*truck,loc2*) := true

operator fly(*plane, airport1, airport2*) :at s
:precond [s] at(*plane, airport1*) ∧ *airport1* ≠ *airport2*
:effects [s+1] at(*plane,airport1*) := false, [s+1] at(*plane,airport2*) := true

We denote the formal invocation timepoint of an operator $o$, as specified by :at, by $\textsf{inv}(o)$. The fact that an operator $o$ is invoked with arguments $\overline{\omega}$ in the interval $[\tau,\tau']$ is denoted by the *action occurrence expression* $[\tau,\tau']\, o(\overline{\omega})$. A concrete example would be $[0,1]\ \textsf{load-truck}(\textsf{obj11},\textsf{tru1},\textsf{loc1})$.

An *operator sequence* is a tuple of timed action occurrences. In a *valid operator sequence*, all preconditions are satisfied and no operator has inconsistent effects. A *plan* is a valid operator sequence whose final state satisfies the goal.

### The Semantics of a Goal Narrative

As shown in Figure 1, the semantics of a goal narrative in $\mathcal{L}(\text{ND})^*$ is defined by a translation *Trans*() into the standard TAL base language $\mathcal{L}(\text{FL})$, which is an order-sorted first-order language with the predicates $\textit{Holds}(\tau, f, v)$ expressing that the fluent (time-dependent state variable) $f$ takes on the value $v$ at time $\tau$ and $\textit{Occlude}(\tau, f)$ expressing that $f$ is allowed to change values at $\tau$.

Given a goal narrative $\mathcal{GN}$ and its translation $\textit{Trans}(\mathcal{GN})$ into $\mathcal{L}(\text{FL})$, a circumscription policy minimizes *Occlude* relative to action descriptions and dependency constraints; see (Kvarnström & Doherty 2000; Doherty *et al.* 1998) for a definition of this policy and the foundational axioms used by TAL. Due to structural constraints on $\mathcal{L}(\text{ND})$ statements, the resulting second-order theory can be translated into a logically equivalent first-order theory (denoted by $\textit{Trans}^+(\mathcal{GN})$) which is used to reason about the narrative.

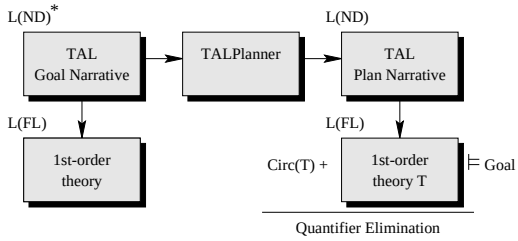


Figure 1: TAL/TALplanner relation

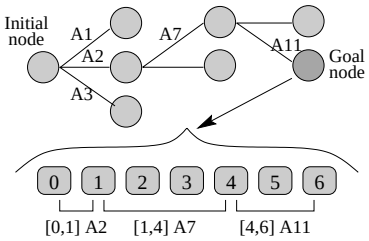


Figure 2: Search Space

A goal narrative can also be used as the input to TALplanner, which then generates a *plan narrative* $\mathcal{N}$ where a set of timed action occurrences (corresponding to a plan) has been added. If $\phi$ is the goal and $\tau$ the final timepoint in the plan, then it is guaranteed that $\textit{Trans}^+(\mathcal{N}) \models [\tau]\ \phi$: The goal must hold in the final state.

**Notational Conventions:** All free variables will be assumed to be implicitly universally quantified. We will say that an operator sequence $p$ entails a formula $\phi$ iff $\textit{Trans}^+(\mathcal{N} \cup p) \models \textit{Trans}(\phi)$, where the narrative $\mathcal{N}$ is often to be understood from the context.

This concludes the description of planning domain definitions in TAL. The following sections will show TALplanner's forward-chaining search tree and how the search process is constrained using control rules.

## Search in Sequential TALplanner

Like any forward-chaining planner, TALplanner searches for a plan in a tree where the root corresponds to the initial state and where each outgoing edge corresponds to one of the operators applicable in its source node (Figure 2).

Usually, each node in this tree is considered to be a single state. However, since the evaluation of a domain-dependent control rule in a node may require access to the entire history beginning in the initial state, it is more convenient to view each node as consisting of a state sequence, or (equivalently) a logical model, as shown in the figure.

A simple forward-chaining planner can be implemented by searching this tree using a standard search algorithm, such as iterative deepening or depth first. But although using a complete search algorithm is clearly enough to make the planner complete, it is equally clear that a certain degree of

goal-directedness is required to make the search process *efficient*. The following section shows how domain-dependent control rules can be used for pruning less interesting parts of the search tree and guiding the planner towards the goal.

## Using Domain-Dependent Control Rules

Most planners allow the specification of a set of goal states, often in the shape of propositional (or first-order) formulas that must hold in a goal state.

There are many ways to permit more detailed control over the search process as well as more complex constraints on the plans to be generated; see for example SHOP (Nau *et al.* 1999; 2001), System R (Lin 2001), and PbR (Ambite 1998; Ambite, Knoblock, & Minton 2000), all of which participated in the AIPS-2000 planning competition.

TALplanner uses domain-dependent control rules in the form of first-order TAL formulas that must be entailed by the final plan generated by the planner. Thus, the definition of a plan must be amended: A *plan* is a valid operator sequence *which satisfies the control rules* and whose final state satisfies the goal.

This serves two separate purposes. First, it allows the specification of complex temporally extended goals such as safety conditions that must be upheld throughout the execution of a plan, and second, the additional constraints on the final plan often allow the planner to prune entire branches of the search tree, since it can be proven that any leaf on the branch will violate at least one control rule.

### Control Rules for the Logistics Domain

The following three simple control rules (inspired by TLPLAN) will later be used as concrete examples.

A package should only be loaded onto a plane if a plane is required to move it: If the goal requires it to be at a location in another city. If we have unloaded a package from a plane, it must be the case that the package should be in the current city. If a package is at its destination, it should not be moved.

control :name "only-load-when-necessary"
[t] $\neg$in(*obj*, *plane*) $\wedge$ at(*obj*, *loc*) $\wedge$ [t+1] in(*obj*, *plane*) $\rightarrow$
$\exists$*loc*′ [ goal(at(*obj*, *loc*′)) $\wedge$ [t] city_of(*loc*) $\neq$ city_of(*loc*′) ]
control :name "only-unload-when-necessary"
[t] in(*obj*, *plane*) $\wedge$ at(*plane*, *loc*) $\wedge$ [t+1] $\neg$in(*obj*, *plane*) $\rightarrow$
$\exists$*loc*′ [ goal(at(*obj*, *loc*′)) $\wedge$ [t] city_of(*loc*) $\hat{=}$ city_of(*loc*′) ]
control :name "objects-remain-at-destinations"
[t] at(*obj*, *loc*) $\wedge$ goal(at(*obj*, *loc*)) $\rightarrow$ [t+1] at(*obj*, *loc*)

### Using Control Rules for Pruning

Consider the objects-remain-at-destinations rule above. We can see that if an operator sequence moves an object which is already at its final destination, then that operator sequence cannot be a plan and cannot possibly be *extended* into a plan. No matter what actions are added, there will always remain a timepoint where an object is moved unnecessarily, and in the end the control rule will not be satisfied.

We would now like the planner to automatically detect such control rule violations. This requires conditionalizing the control rules and generating *pruning constraints* that only constrain the fixed "past" in an operator sequence. More formally, let $t_*$ be the end timepoint of the last operator in a search node. Then, the constraints must only constrain the fixed "past" states in $[0, t_*]$. The infinite sequence of "future" states in $(t_*, \infty)$ must not be constrained, since even if a violation were to be detected there, this violation might disappear when additional actions are added.

For example, objects-remain-at-destinations results in the pruning constraint $t + 1 \leq t_* \wedge [t]$ at(*obj*, *loc*) $\wedge$ goal(at(*obj*, *loc*)) $\rightarrow [t + 1]$ at(*obj*, *loc*), where $t + 1 \leq t_*$ ensures that states after $t_*$ are not constrained.

### Incremental Evaluation of Pruning Constraints

Although it would be possible to evaluate the complete pruning constraints at each node in the search tree, we immediately take the analysis one step further and note two important ways of improving performance.

First, if the planner needs to evaluate the pruning constraints in a node, the constraints must have been satisfied in its immediate ancestor – otherwise, the ancestor would have been pruned and this node would not have been expanded. This can be taken advantage of by generating *incremental* pruning constraints that only check the new states generated by the last operator to be added to the plan.[2] For example, after adding the operator $[4, 6]$ A11 in Figure 2, only the two new states at time 5 and 6 should have to be checked.

Second, separate incremental pruning constraints are generated for each operator type. In the logistics domain, this means there will be six incremental constraints for each control rule: One for load-plane, one for drive-truck, and so on. These operator-specific constraints will only be evaluated immediately after an operator of the corresponding type has been added to the plan, which is necessary in order to take into account the fact that different operator types may have different durations.

For these reasons, TALplanner generates from the original set of control rules control (1) one set of initial pruning constraints init, (2) for each operator type $o^i$ with formal invocation timepoint $s$ and formal arguments $x^i_1, \ldots, x^i_{m_i}$ a set $\mathsf{incr}_i(s, x^i_1, \ldots, x^i_{m_i})$ of incremental pruning constraints for that operator (where the variables indicated in parentheses may be free in the constraints), and (3) one set of final pruning constraints final, such that control is entailed by a plan $\langle [\tau_1, \tau'_1] o^{i_1}_1(c^1_1, \ldots, c^{m_i}_1), \ldots, [\tau_n, \tau'_n] o^{i_n}_n(c^1_n, \ldots, c^{m_i}_n) \rangle$ iff:

1. The initial constraints hold in the root node: $\langle\rangle \models$ init,
2. Whenever a new operator $o^{i_k}_k$ is added, its incremental pruning constraints $\mathsf{incr}_{i_k}$ hold: for all $1 \leq k \leq n$, $\langle [\tau_1, \tau'_1]\ o^{i_1}_1(c^1_1, \ldots, c^{m_i}_1), \ldots, [\tau_k, \tau'_k]\ o^{i_k}_k(c^1_k, \ldots, c^{m_i}_k) \rangle \models \mathsf{incr}_{i_k}[\tau_k, c^1_k, \ldots, c^{m_i}_k]$
3. The final pruning constraints hold in the complete plan: $\langle [\tau_1, \tau'_1]\ o^{i_1}_1(c^1_1, \ldots, c^{m_i}_1), \ldots, [\tau_n, \tau'_n]\ o^{i_n}_n(c^1_n, \ldots, c^{m_i}_n) \rangle \models$ final.

[2] TLPLAN uses a progression algorithm, which automatically ensures that only fixed states are constrained. This avoids the need to generate pruning constraints and automatically provides an incremental evaluation, while the use of formula evaluation in TALplanner facilitates certain kinds of optimizations and results in a considerably lower memory consumption.

### Generating Pruning Constraints

Clearly, TALplanner can handle any control formula $\phi \in$ control simply by placing it in final. This serves as a fallback allowing the planner to handle arbitrary control formulas, while the most common classes of formulas can be analyzed in more detail to improve performance. We first consider two common classes under the assumption that all operators take constant time.

A *state constraint* is a formula $\forall t.\phi(t)$ where $\phi$ does not refer to states at any other time than $t$. For such formulas, let $\psi$ be $\phi$ with all variables except $t$ replaced with fresh variables of the same sort. Then, $\psi[t \mapsto 0]$ is added to init; for each operator type $o^i$ with duration $\tau$, we add $\bigwedge_{k=1}^{\tau} \psi[t \mapsto \mathsf{inv}(o^i) + k]$ to $\mathsf{incr}_i$; and nothing is added to final.

A *state transition constraint* is a formula $\forall t.\phi(t)$ where $\phi$ only refers to states in $[t, t+1]$. For such formulas, let $\psi$ be $\phi$ with all variables except $t$ replaced with fresh variables of the same sort. Nothing is added to init; for each operator type $o^i$ with duration $\tau$, we add the formula $\bigwedge_{k=0}^{\tau-1} \psi[t \mapsto \mathsf{inv}(o^i) + k]$ to $\mathsf{incr}_i$; and $\psi[t \mapsto t_*]$ is added to final.

These two classes are very common and are in fact sufficient for many planning domains. TALplanner handles several additional classes of varying complexity. Although an understanding of these classes is not essential for the domain-dependent analysis techniques presented in this paper, we will show how to handle one more class of formulas for operators with arbitrary, possibly variable duration.

Let $\forall t.\phi(t)$ be a control formula, where $\phi$ only refers to states in $[t, t+d]$ and $d$ is independent of $t$. Let $\psi$ be $\phi$ with all variables except $t$ replaced with fresh variables of the same sort. Then, the formula $d = 0 \rightarrow \phi[t \mapsto 0]$ is added to init. For each operator type $o^i$ with duration $\tau$, the formula $\forall k.1 \leq k \leq \tau \rightarrow \phi[t \mapsto \mathsf{inv}(o^i) + k - d]$ should be added to $\mathsf{incr}_i$, *but* since $\mathsf{inv}(o^i) + k - d$ could be negative and TAL currently uses non-negative time, the formula has to be rewritten as $\forall k.1 \leq k \leq \tau \rightarrow (\forall t.t+d = \mathsf{inv}(o^i)+k \rightarrow \phi(t))$. Finally, $\forall k.1 \leq k \leq d \rightarrow (\forall t.t + d = t_* + k \rightarrow \phi(t))$ is added to final.

In the logistics domain, all operators have duration 1 and formal invocation timepoint $\mathsf{inv}(o^i) = s$. For objects-remain-at-destinations, the formula $[s]\ \mathsf{at}(obj_1, loc_1) \wedge \mathsf{goal}(\mathsf{at}(obj_1, loc_1)) \rightarrow [s+1]\ \mathsf{at}(obj_1, loc_1)$ is added to each $\mathsf{incr}_i$, while $[t_*]\ \mathsf{at}(obj_1, loc_1) \wedge \mathsf{goal}(\mathsf{at}(obj_1, loc_1)) \rightarrow [t_*+1]\ \mathsf{at}(obj_1, loc_1)$ is added to final (note that variables have been renamed).

## Optimizing Formulas

Once control rules have been split into initial, incremental and final pruning formulas, the preprocessor performs three distinct kinds of optimizations intended to generate formulas that can be evaluated more efficiently.

First, it makes use of well-known logical equivalences and type analysis techniques to generate simpler but equivalent formulas. Second, it makes use of the *context* in which a formula will be evaluated in order to generate simpler formulas that are equivalent given the context. Third, it generates for each formula a set of necessary variable bindings intended to permit the optimization or elimination of quantifiers.

This section will describe some of these optimizations, while extensions related to domain analysis techniques will be discussed in the next two sections.

### Equivalence Optimization

It is often the case that a formula $\alpha$ can be rewritten on a simpler form $\beta$ such that $\alpha \equiv \beta$. Although this is not the focus of this paper, it should still be mentioned that TALplanner implements a number of such standard optimizations, making use of well-known logical equivalences such as $\phi \wedge (\phi \vee \psi) \equiv \phi$ as well as a form of type analysis.

We denote this optimization procedure by $\mathsf{optimize}(\alpha)$, where the argument $\alpha$ is the formula to be optimized and the return value $\beta$ is logically equivalent to $\alpha$.

### Context-Dependent Optimization

Given some information regarding the context in which a certain formula will be evaluated, some considerably stronger optimizations can be applied. Formally, suppose that $\alpha$ will only be evaluated when $\gamma$ is known to hold. Under these conditions, $\alpha \equiv (\alpha \wedge \gamma)$, since we can trivially conjoin anything that is known to be true. TALplanner therefore attempts to find the simplest possible $\beta$ such that $(\alpha \wedge \gamma) \equiv (\beta \wedge \gamma)$.

The optimizer is extended to accept both a formula $\alpha$ to optimize and an *optimization context* $\Phi$, a set of formulas known to hold during the evaluation of $\alpha$.

**Using Context Information** The context information is used in the optimization of atomic formulas, where an entailment checker attempts to determine whether a formula $\alpha$ is entailed by the context (in which case it can be optimized to `true`) or whether its negation is entailed (`false`). It should be noted that although this entailment checker must be sound it need not be complete. Incompleteness weakens the optimizer but does not affect correctness.

**Generating Context Information** In the initial call to the formula optimizer, no context information is available and the empty set $\emptyset$ is provided as an optimization context.

The context given to $\mathsf{optimize}()$ is generally passed on unmodified when the optimizer makes a recursive call to optimize a subformula. However, for a conjunction $\bigwedge_{i=0}^{n} \phi_i$, the value of any single conjunct $\phi_k$ is irrelevant if any other conjunct is false. Thus, the optimizer recursively optimizes $\phi_k$ in the context that all other conjuncts hold: $\bigwedge_{0 \leq i \leq n, i \neq k} \phi_i$. A dual optimization is applied to disjunctions.

### Quantifier Optimization

The third type of optimization performed by TALplanner relates to quantifiers. Since TAL uses finite value domains, the evaluation of a universally quantified formula $\forall x.\phi(x)$ can be implemented simply by iterating over each possible value of $x$. However, if it can be determined that $\phi(x)$ is definitely true for all $x \notin X$, then it is sufficient to check $\forall x \in X.\phi(x)$. If $X = \{v\}$ (a single value term), then $\phi(x)$ can be optimized to $\phi[x \mapsto v]$, given that $v$ has a suitable sort so that type correctness is preserved. A dual optimization can be applied to existential quantifiers.

To permit this type of optimization, the optimizer is extended to return a tuple $\langle \psi, \mathsf{necNeg}, \mathsf{necPos} \rangle$, where $\psi$ is equivalent to $\phi$ in the given context, necNeg (corresponding to $X$ in the example) is a set of bindings necessary for $\neg\psi$ to hold in the given context, and necPos is a set of bindings necessary for $\psi$ to hold in the given context.

**Generating Necessary Variable Bindings** Variable bindings can be generated by equality expressions: $\mathit{var} = \omega$ generates the binding $\{\mathit{var} \mapsto \omega\}$ to be added to necPos, and $\mathit{var} \neq \omega$ generates the binding $\{\mathit{var} \mapsto \omega\}$ to be added to necNeg. Similarly, a fixed fluent formula $[\tau]\ f \mathrel{\hat{=}} \mathit{var}$ generates a positive binding $\{\mathit{var} \mapsto \mathit{value}(\tau, f)\}$, and the optimization of $[\tau]\ f \mathrel{\hat{=}} \omega$ with a known formula $[\tau]\ f \mathrel{\hat{=}} \mathit{var}$ generates a positive binding $\{\mathit{var} \mapsto \omega\}$.

When optimizing a conjunction $\bigwedge_{i=0}^{n} \phi_i$, each conjunct is recursively optimized. Denote the return values by $\langle \psi_i, \mathsf{necNeg}_i, \mathsf{necPos}_i \rangle$ for $0 \leq i \leq n$. For the conjunction to hold, the conjunction of all $\mathsf{necPos}_i$ must hold; for the conjunction to be false, the disjunction of all $\mathsf{necNeg}_i$ must hold. A dual optimization is applied to disjunctions. For $\neg\phi$, necPos and necNeg are swapped; for a quantified formula $\forall x.\phi$ or $\exists x.\phi$, the bindings generated for the inner formula are returned after removing the bindings for $x$.

It may be the case that no binding at all is possible (for example, because two conjuncts require bindings that cannot belong to the same value domain). In this case, a formula may be immediately optimized to `true` or `false`.

This concludes the description of the formula optimizer, which will be used as a basis for the domain analysis techniques that will be discussed below.

## Using Existing Domain Analysis Techniques

As mentioned in the introduction, there are two interesting questions to be answered regarding the use of domain analysis techniques for planners that utilize domain-dependent control: Which existing techniques for domain-independent planners can be reused, and what new opportunities are opened by the addition of control formulas? This section will focus on the first question, while the next section will concentrate on the second one.

There are several potential difficulties associated with reusing existing domain analysis techniques in TALplanner.

The control rules used by TALplanner are essentially temporally extended goals. These rules constrain the possible ways a goal state can be reached, but several analysis techniques depend on the fact that only the final state is constrained and that the way this state is reached is unimportant.

Also, one of TALplanner's design goals is the ability to plan for domains with large numbers of objects and operator instances. Even if an operator could have billions of operator instances or more, this should not be a major problem as long as sufficiently strong control rules can be written to guide the planner towards choosing "good" instances to be applied. For this reason, techniques that rely on generating all ground instances of operators or predicates are less likely to be useful in conjunction with TALplanner. This includes techniques such as RIFO (Nebel, Dimopoulos, & Koehler 1997) and (Haslum & Jonsson 2000).

Finally, another important design goal is permitting the use of more complex types of operators, including operators with extended duration and (eventually) non-deterministic effects, as well as the use of resources and concurrency (Kvarnström, Doherty, & Haslum 2000). Any techniques depending on the use of single-step operators would require extensions in order to be used in TALplanner.

### Using State Invariants

Given these restrictions, the most promising type of domain analysis technique to be integrated into TALplanner has been the automatic extraction of state constraints or state invariants. This involves analyzing the operator definitions in a domain, possibly together with the initial state of a specific planning instance, and finding a set of invariants that are guaranteed to hold in any state generated by a valid operator sequence (Fox & Long 1998; Gerevini & Schubert 1998; 2000; Scholz 2000; Rintanen 2000).

For the logistics domain, one such invariant would be $[t]\ \mathsf{in}(\mathit{obj}, \mathit{vehicle}) \rightarrow \neg\mathsf{at}(\mathit{obj}, \mathit{loc})$: An object in a vehicle is not at any location. (While this may appear counter-intuitive, it does follow from the way the logistics domain is usually modeled.)

There are two steps involved in integrating such a technique into the planner: The technique must be adapted to work with TALplanner's operator definitions (and possibly extended to handle operators with extended duration), and the planner must be altered to actually use the state invariants once they have been generated. We have chosen to begin with the second step, extending TALplanner to make use of *manually* specified state invariants. This will provide the opportunity to test carefully whether the use of the invariants has a sufficient impact on the planner's performance to warrant following through with the implementation of the automatic domain analysis.

State invariants are provided to the formula optimizer using an extended form of the optimization context introduced in the previous section. The extended context consists of a tuple $\langle \Phi, \Psi \rangle$, where $\Phi$ (like before) is a set of formulas known to hold and $\Psi$ is a set of state invariants.

Whenever new facts are added to $\Phi$, as is done (for example) in conjunctions where each conjunct is optimized given the assumption that all other conjuncts hold, the facts are combined with the state invariants with limited use of a resolution algorithm. This may yield further facts to be added to $\Phi$, significantly strengthening the information available to the optimizer.

Below, the inference procedure will be denoted by $\mathsf{infer}(\Phi, \Psi)$, where $\Phi$ is a set of known formulas, $\Psi$ is a set of state invariants, and the return value is a set of formulas containing $\Phi$ and possibly additional formulas that are entailed by $\mathit{Trans}^{+}(\Phi \wedge \Psi)$.

As can be seen in the benchmark tests later in this paper, the use of state invariants can indeed have a significant impact on the performance of the planner, decreasing the time required for some blocks world problems by a factor of 3. Consequently, a future version of TALplanner will be integrated with one of the existing automatic analysis methods.

## New Domain Analysis Techniques for Domain-Dependent Control

For most planning domains, TALplanner spends a significant amount of time testing incremental pruning constraints – in fact, this often accounts for more than 99% of the time used by the planner. Clearly, any technique that allows the incremental constraints to be tested more quickly will have a considerable impact on performance.

The incremental pruning constraints mainly depend on the state or states generated by the latest operator invocation, and although the preprocessor cannot know in advance which operator instance was invoked, it *can* know which operator *type* was invoked (such as drive or fly) – there is a separate set of constraints $\mathsf{incr}_i$ for each operator type $o^i$. This leads to the idea of attempting to extract some information from the operator definitions regarding the states in which the constraints will be evaluated, and then using this context information in the formula optimizer.

Current versions of TALplanner make use of two different kinds of context information automatically extracted from operator definitions.

First, the preconditions must hold (otherwise the operator would not have been invoked) and the effects must hold (since if they were inconsistent the planner would already have backtracked). This can be used to augment the set $\Phi$ of known formulas provided to the optimizer in the optimization context.

Second, many control rules are only triggered by certain specific state transitions, and the only state transitions that are possible during the execution of an operator are those that are explicitly specified in the effects. Analyzing these transitions makes further optimizations possible.

Before the operator analysis is described in detail, Figure 3 provides an overview of the complete formula optimization process. The planner analyzes each operator definition to extract a set of facts that must hold at various points during the execution of any instance of the operator. These facts are combined with the state invariants (provided by the user or an automatic domain analyzer) to generate a set of context facts $\Phi$ which is given to the formula optimizer as part of the optimization context. The optimizer also makes use of state transition information automatically extracted from the operator definitions.

### Operator Analysis: Extracting Facts

Let $o^i$ be an operator type (for example, drive). When an instance of this operator type is invoked (for example, $[0, 1]$ drive(tru1, pos1, pos2)), the formal invocation timepoint is bound to the actual invocation timepoint, and the formal arguments are bound to their actual values ($s$ is bound to 0, *truck* to tru1, and so on).

If the precondition is not satisfied, the operator is never applied. If it is satisfied, the effects are applied, and if they are inconsistent, the planner backtracks. In other words, the incremental pruning constraints in $\mathsf{incr}_i$ are only tested if both the precondition and the effects hold. Consequently, a set of known facts can be extracted from the operator quite easily. Let $\phi$ be the precondition of $o^i$, $\phi'$ be a conjunction

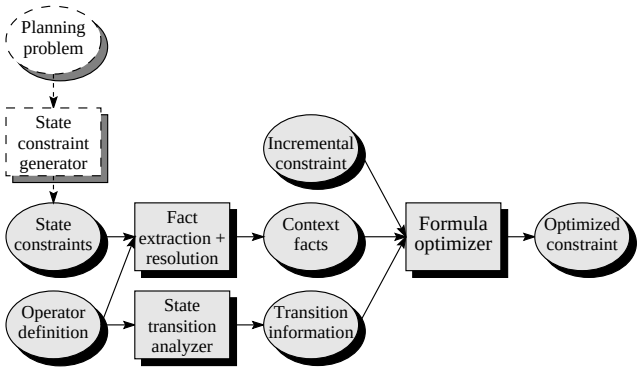


Figure 3: Control Analysis and Optimization

of fixed fluent formulas extracted from the unconditional effects of the action (for example, the effect $[s + 3]$ at$(o, l) :=$ false generates the formula $[s + 3]$ at$(o, l) \mathrel{\hat{=}}$ false), and $\Psi$ be the set of state invariants specified in the domain definition. Then, the initial set of known formulas is $\Phi =$ infer$(\phi \wedge \phi', \Psi)$.

For example, the *drive* operator yields the formulas $\phi =$ $[s]$ at(*truck*, *loc1*) $\wedge$ city_of(*loc1*) $\mathrel{\hat{=}}$ city_of(*loc2*) $\wedge$ *loc1* $\neq$ *loc2* and $\phi' = [s + 1]$ at(*truck*, *loc1*) $\mathrel{\hat{=}}$ false $\wedge$ $[s + 1]$ at(*truck*, *loc2*) $\mathrel{\hat{=}}$ true. Additional facts may be inferred from this using state invariants and resolution, and these facts can then be used in the optimizer in order to determine that a certain formula must or cannot hold.

Note that both the formal invocation timepoint of the operator $o^i$ and its formal arguments can occur as free variables in $\phi$ or $\phi'$. When the constraints in $\mathsf{incr}_i$ are tested, the formal arguments will be bound to the values used during the latest operator invocation, as stated in the definition of $\mathsf{incr}_i$. In this way, an incremental pruning constraint can refer directly to the arguments of the corresponding operator invocation. (Since all variables in pruning constraints and state invariants have been renamed and replaced with fresh variables, there is no risk of mistaking one instance of a variable for another.)

### Generating Bindings using State Transition Analysis

Many control rules can only be violated if certain state transitions take place. This is a natural consequence of the fact that many control rules follow a certain pattern, where a property true in one state should either be preserved to the next state or violated in a very specific way.

For example, the incremental pruning constraints generated from only-load-when-necessary state that a package should only be loaded into a plane if a plane is needed to move it. This can also be stated in another way: If a package is *not* in a plane in a certain state, then it should remain *not* in that plane in the next state, unless a plane is needed in order to move it. As long as the property $\neg$in$(obj_1, plane_1)$ is preserved from $s$ to $s + 1$, the constraint cannot be violated.

As another example, the incremental pruning constraints generated by objects-remain-at-destinations state that if a package is at a certain location at $s$, it must be at that lo-

cation at $s+1$, unless there is no goal that it should remain there. If the property $\mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$ is preserved, the constraint cannot be violated.

Clearly, it would be a major advantage if the preprocessor could determine in advance that these state transitions cannot take place – then, the entire incremental constraints would necessarily be true, and would not need to be tested. Failing this, it would be of almost equal benefit to the planner if it could be determined that the state transitions can only take place for certain specific instances of a fluent, thereby reducing the number of instances to be tested.

In fact, this *can* be detected in advance, as will be demonstrated using a few examples. Returning to objects-remain-at-destinations, the incremental pruning constraints generated by this rule can only be violated if $\mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$ is made false between $s$ and $s+1$. But $s$ is the invocation timepoint of the latest operator, and $s+1$ is the effect state. The unload-truck operator never makes an instance of at false at $s+1$, and therefore this incremental constraint is never violated for unload-truck. Although drive makes $\mathsf{at}(\mathit{truck}, \mathit{loc1})$ false, this instance refers to a truck rather than an object and cannot be unified with $\mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$. Therefore, the incremental constraint can never be violated by drive. Finally, the load-truck action makes $\mathsf{at}(\mathit{obj}, \mathit{loc})$ false, and unifying this with $\mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$ yields the variable bindings $\{\mathit{obj}_1 \mapsto \mathit{obj}, \mathit{loc}_1 \mapsto \mathit{loc}\}$. These bindings must necessarily hold if the disjunction should be false, and can therefore be added to necNeg when the disjunction is analyzed.

These insights can be used to improve the formula optimizer.

**Extending the Optimizer** The optimization context is extended to a tuple $\langle \Phi, \Psi, o \rangle$ where $\Phi$ is a set of formulas known to hold, $\Psi$ is a set of state invariants, and $o$ is an operator type. The intention is that the optimized formula will only be evaluated in a context where $\Phi$ and $\Psi$ are known to hold, and where an instance of $o$ has just been added to the operator sequence. In addition, it is guaranteed that when the formula is evaluated, the formal invocation timepoint and formal argument variables of $o$ will still be bound to the actual arguments used in the latest operator invocation.

The following algorithm is called from the optimizer when analyzing a disjunction, given the disjunction and an optimization context as arguments. The return value is a set of variable bindings necNeg that are necessary for the disjunction to be false: If any binding does not hold, the disjunction will be satisfied. Explanations will be provided below.

```
1  procedure findpp(⋁_{i=1}^n φ_i, ⟨Φ, Ψ, o⟩)
2  let conjuncts = infer(Φ ∧ ⋀_{i=1}^n ¬φ_i, Ψ)
3  let necNeg = ∅
4  for all [τ] f ≙ v in conjuncts do
5    for all [τ'] f ≙ v' in conjuncts do
6      if can prove τ < τ' then
7        if can prove that v ≠ v' then
8          if can prove that τ ≥ t* then
9            return an impossible binding
10         if sequential operator type o given then
11           let b = analyzeST([τ] f ≙ v, [τ'] f ≙ v', ⟨Φ, Ψ, o⟩)
12           let necNeg = conjoin(necNeg, b)
13 return necNeg
```

The pruning constraint for objects-remain-at-destinations relative to load-plane is $\forall \mathit{obj}_1, \mathit{loc}_1.[s]\ \mathsf{at}(\mathit{obj}_1, \mathit{loc}_1) \wedge \mathsf{goal}(\mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)) \rightarrow [s+1]\ \mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$, where the implication inside the universal quantifier prefix can also be written as a disjunction $\bigvee_{i=1}^n \alpha_i$. This disjunction can be analyzed using the algorithm above.

For the disjunction to be false, it must clearly be the case that $\bigwedge_{i=1}^n \neg\alpha_i$. There is also a set of formulas $\Phi$ that are known to hold regardless of whether the disjunction holds or not, so for the disjunction to be false, we must have $\Phi \wedge \bigwedge_{i=1}^n \neg\alpha_i$. The resolution inference algorithm can be used together with the state invariants to infer additional facts: For the disjunction to be false, all formulas in $\mathsf{infer}(\Phi \wedge \bigwedge_{i=1}^n \neg\alpha_i, \Psi)$ must hold (for example, since we must have $[s+1]\ \mathsf{at}(\mathit{obj}_1, \mathit{loc}_1)$, we can infer $\forall \mathit{vehicle}.[s+1]\ \neg\mathsf{in}(\mathit{obj}_1, \mathit{vehicle})$). The conjunction of the formulas returned by infer will be denoted by $\bigwedge_{i=1}^m \beta_i$.

Now, suppose that $\beta_i$ is $[\tau]\ f \hat{=} v$ and that $\beta_j$ is the formula $[\tau']\ f \hat{=} v'$. Suppose further that it can be proven[3] that $\tau < \tau'$, so the second formula refers to a later timepoint, and that $v \neq v'$. Due to $\beta_i$, the fluent could not have taken on the value $v'$ at $\tau$, but due to $\beta_j$, it must take on that value at $\tau'$. The value of $f$ must have changed in the interval $(\tau, \tau']$.[4]

What remains is trying to find a set of variable bindings that are necessary for $f$ to be able to change in $(\tau, \tau']$, or in the best case, to determine that $f$ in fact must remain constant. If any such bindings are found, they can be conjoined to necNeg, since the bindings are necessary for the disjunction to be false. TALplanner uses two different types of state transition analysis for finding bindings.

First, if $\tau \geq t_*$, then the entire interval $(\tau, \tau']$ is strictly after $t_*$. But no effects can take place after $t_*$, so no fluents can change there. Therefore, it is impossible that the disjunction does not hold, and an impossible variable binding is returned. This is useful for analyzing final constraints.

Second, if an operator type is specified, the disjunction will be evaluated immediately after an operator of that type is invoked, and the transitions possible during the execution interval can be analyzed. This analysis is useful for constraints in $\mathsf{incr}_i$, and is described in detail below.

**State Transition Analysis for Operators** The state transition analysis algorithm for sequential operators is as follows.

```
1  procedure analyzeST([τ] f ≙ v, [τ'] f ≙ v', ⟨Φ, Ψ, o⟩)
2  if we can prove τ' > inv(o) then
3    let eff = all conditional and unconditional effects of o
4    for all [τ̄] g := w in eff do
5      if can prove f ≠ g then remove this from eff
```

[3] Whenever we say "if we can prove $\phi$" rather than "if $\phi$ is the case", failing to prove this fact may lead to a decrease in performance but is always safe. For example, the attempt to prove that $\tau < \tau'$ could be a test whether $\tau'$ is of the syntactic form $\tau + n$ for some positive $n$, or could be a stronger test involve more complex temporal reasoning.

[4] TALplanner also handles negated formulas $\neg[\tau]\ f \hat{=} v$ and $\neg[\tau]\ f' \hat{=} v'$. The extension is trivial and is omitted to improve the clarity of the presentation.

```
    elsif can prove τ̄ ∉ (τ, τ'] then remove this from eff
    elsif can prove w ≠ v' then remove this from eff
  if {free variables in eff} ⊆ {arguments of o} then
    let necNeg = an impossible binding
    for all [τ̄] g := w in eff do
      let necNeg = disjoin(necNeg, unify(g, f))
    return necNeg
return ∅
```

This algorithm returns a set of bindings that are required for $f$ to change values from $v$ to $v'$ between $\tau$ and $\tau'$, given that an instance of $o$ is the last operator to be invoked in the current search node. Note that $f$ might be a fluent expression with arguments, such as $\mathsf{at}(obj_1, loc_1)$.

Since $o$ is (currently) the last operator, the only changes that can take place in $(\mathsf{inv}(o), \infty)$ are those explicitly caused by $o$. No information is provided about what might have happened in $[0, \mathsf{inv}(o)]$, though, so if it cannot be proven that $\tau' > \mathsf{inv}(o)$, the analysis is aborted.

Otherwise, consider every effect of the operator, conditional as well as unconditional. For load-truck, this would be $[s+1]\,\mathsf{at}(obj, loc) := \mathsf{false}$ and $[s+1]\,\mathsf{in}(obj, truck) := \mathsf{true}$.

If an effect cannot affect $f$, it is irrelevant and can be discarded. If it might affect $f$ but not at an interesting timepoint (in $(\tau, \tau']$, when the change must take place) it can be discarded. Finally, if the effect assigns a value $w$ that is different from $v'$, then it definitely cannot cause a transition from $v$ to $v'$, and can be discarded.

The remaining effects might cause $f$ to change values from $v$ to $v'$ between $\tau$ and $\tau'$. If they contain free variables that are not arguments of $o$, then those variables must have been bound in quantified effects, and the analysis is aborted. Otherwise, it is safe to claim that $f$ must be equal to one of the effect fluents. This means it must be unified with one of them for the desired state transition to occur, so the disjunction of all $\mathsf{unify}(g_i, f)$ can be returned.

## Generating Precondition Control

After optimizing a formula in $\mathsf{incr}_i$ for an operator $o^i$, the result often turns out to have conjuncts that only refer to the invocation timepoint of $o^i$. Clearly, those conjuncts can be moved from $\mathsf{incr}_i$ into the precondition, removing the need to actually invoke the operator before the conditions are tested. This has proven to drastically reduce the number of states generated by TALplanner, significantly increasing the performance of the planner.

In the logistics world, for example, the precondition $\exists loc'[\mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [s]\ \mathsf{city\_of}(loc) \neq \mathsf{city\_of}(loc')]$ is generated for the load-plane operator by the only-load-when-necessary control rule: There must be a goal that the object *obj* to be loaded into the plane should be in another city.

But if a control rule can be expressed as a precondition, why not simply write it that way? There are several reasons why the use of control rules is often better, perhaps the most important of which is that it allows a more modular specification of the control knowledge: Each constraint is specified as a single control rule, rather than as a number of (possibly different) preconditions in each operator. Allowing an automatic analyzer to generate preconditions wherever possible should also be less error-prone, especially for more complex rules where interdependencies between multiple actions must be taken into account. This is done by TALplanner.

## Benchmark Tests and Analysis

The techniques described in this paper have proven very effective for many standard benchmark domains. However, the four additions to the planner – using state invariants, generating context facts from operators, analyzing state transitions in operators, and generating precondition control – cannot easily be studied in isolation, since there are several types of optimizations that could be generated by more than one technique. A complete analysis would require testing all the 16 variations made possible by turning individual extensions on or off, for a large number of domains and problems.

Nevertheless, a certain pattern appears in most domains. The operator-specific control rule analysis is absolutely essential to the performance of the planner, to the extent that removing it generally makes the generation of precondition control impossible (since it requires the reduction and removal of control rule disjuncts referring to the "future", which can only be done with an operator-specific analysis) and makes the use of state invariants ineffective.

When the operator-specific analysis is added, the generation of precondition control has a significant effect. Finally, when precondition control has been introduced, far fewer states are expanded and the speed of testing preconditions becomes paramount to the performance of the planner. This makes the use of state invariants more significant.

This is demonstrated in a set of benchmark tests using problems from the AIPS-2000 competition. These tests were run on an 800 MHz Pentium III machine with 512 MB of RAM, running Red Hat Linux 7.1 and Java 1.3.

For logistics (Figure 4), the topmost line indicates the time used without the new techniques. Adding operator-specific analysis improves performance by a factor of up to 4 for the largest problems. Adding precondition control yields another factor of 8, and finally, adding state invariants reduces the amount of time used by a factor of 1.3.

In the blocks world (Figure 5), operator-specific analysis results in an 8-fold speedup for the largest problems with 500 blocks, after which adding precondition control results reduces the time by a factor of 16 and the use of state invariants yields another factor of 3. In total, the new analysis techniques make TALplanner up to 400 times faster for the largest problem instances.[5]

It should be noted that these improvements are partly due to the elimination of quantifiers and therefore do not result in a constant factor speedup but a reduction in time complexity. Thus, larger problems are affected to a greater degree.

## Conclusions and Future Work

We have presented a new domain analysis technique used for extracting context information from operator definitions.

[5] Early versions of these techniques were implemented in the version competing in AIPS-2000.

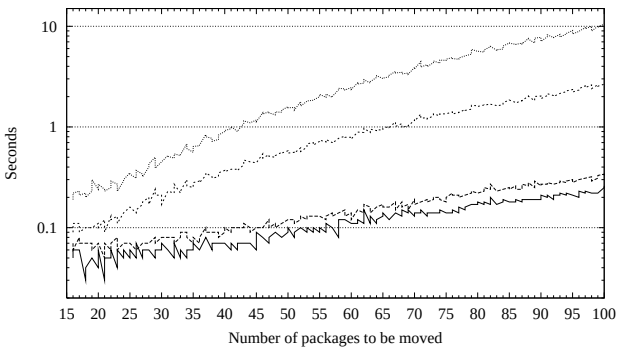


Figure 4: Logistics Problems from AIPS-2000

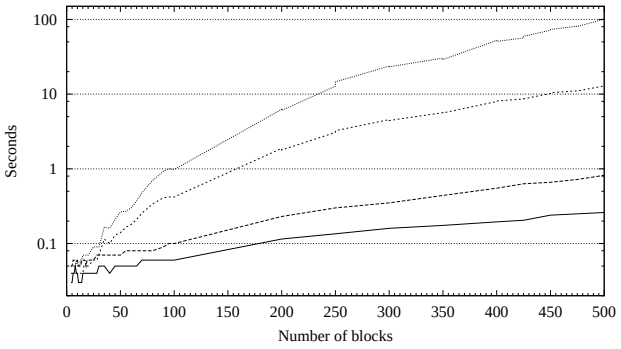


Figure 5: Blocks World Problems from AIPS-2000

This technique has been combined with the use of state invariants and the generation of precondition control in order to increase the performance of testing whether domain-dependent control rules are satisfied. Benchmark results show an improvement up to a factor of 400 for the blocks world and up to a factor of 40 for the logistics domain. Similar trends are present in all domains tested so far, and additional empirical testing is in progress.

## References


Ambite, J. L.; Knoblock, C. A.; and Minton, S. 2000. Learning plan rewriting rules. In *Proc. AIPS-2000*, 3–12. http://www.isi.edu/∼ambite/2000-aips.ps.

Ambite, J. L. 1998. *Planning by Rewriting*. Ph.D. Dissertation, University of Southern California. http://www.isi.edu/∼ambite/thesis.ps.gz.

Bacchus, F., and Ady, M. 1999. Precondition control. ftp://newlogos.uwaterloo.ca/pub/bacchus/BApre.ps.gz.

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

Bacchus, F. 2001. The AIPS'00 planning competition. *AI Magazine* 22(3):47–106. Competition web page at http://www.cs.toronto.edu/aips2000/.

Doherty, P., and Kvarnström, J. 2001. TALplanner: A temporal logic-based planner. *AI Magazine* 22(3):95–102.

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

Doherty, P. 1994. Reasoning about action and change using occlusion. In *Proc. ECAI-94*, 401–405.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning domains. In *Proc. IJCAI-99*. http://www.dur.ac.uk/∼dcs0www/research/stanstuff/papers.html.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. AAAI-98*.

Gerevini, A., and Schubert, L. 2000. Discovering state constraints in DISCOPLAN: Some new results. In *Proc. AAAI-00*.

Gustafsson, J., and Doherty, P. 1996. Embracing occlusion in specifying the indirect effects of actions. In *Proc. KR-96*.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In *Proc. AIPS-2000*. AAAI Press.

Karlsson, L., and Gustafsson, J. 1999. Reasoning about concurrent interaction. *Journal of Logic and Computation* 9(5):623–650.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

Kvarnström, J.; Doherty, P.; and Haslum, P. 2000. Extending TALplanner with concurrency and resources. In *Proc. ECAI-2000*.

Kvarnström, J., and Doherty, P. 2000b. TALplanner project page. Accessible via the KPLAB web page, http://www.ida.liu.se/∼patdo/kplabsite/html/external/.

Lin, F. 2001. A planner called R. *AI Magazine* 22(3):73–76.

Nau, D.; Cau, Y.; Lotem, A.; and Muños-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *Proc. IJCAI-99*, 968–973. San Francisco: Morgan Kaufmann Publishers.

Nau, D.; Cao, Y.; Lotem, A.; and Muños-Avila, H. 2001. The SHOP planning system. *AI Magazine* 22(3):91–94.

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

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. AAAI-00*. http://www.informatik.uni-freiburg.de/∼rintanen/CV.html.

Scholz, U. 2000. Extracting state constraints from PDDL-like planning domains. In *Proc. AIPS Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning*, 43–48.