Annals of Mathematics and Artificial Intelligence 0 (2001) ?–? 1

TALplanner: A Temporal Logic Based Forward Chaining Planner

Jonas Kvarnström and Patrick Doherty

Dept. of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden
E-mail: {jonkv,patdo}@ida.liu.se

This article has been accepted for publication (Nov. 2000) in the Annals of Mathematics and Artificial Intelligence, 2001.

We present TALplanner, a forward-chaining planner based on the use of domain-dependent search control knowledge represented as 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 and control formulas. The sequential version of TALplanner is presented. The expressivity of plan operators is then extended to deal with an interesting class of resource types. An algorithm for generating concurrent plans, where operators have varying durations and internal state, is also presented. All versions of TALplanner have been implemented. The potential of these techniques is demonstrated by applying TALplanner to a number of standard planning benchmarks in the literature.
Keywords: Planning, Temporal Logics, Action and Change, Knowledge Representation

1. Introduction

Recently, Bacchus and Kabanza et al. [9–11,22] have been investigating the use of modal temporal logics to express domain-specific search control knowledge for forward-chaining planners. This approach, implemented in the TLPLAN system, has demonstrated impressive improvements in efficiency when compared to many recent state-of-the-art planners such as BLACKBOX [25] and IPP [28], two of the leading competitors in the the AIPS'98 planning competition [1] (see [11] for comparisons).

TLPLAN uses modal temporal formulas in a first-order version of LTL (Linear Temporal Logic [17]) to express domain-specific search control knowledge. The forward chaining planner progresses the control formulas specified for a particular planning domain through each state generated by the operator sequence currently being investigated. Whenever progression returns the formula false, the control formula is guaranteed to be violated in all descendant search nodes, and the current search node and its descendants can immediately be pruned from the search tree (Figure 1). In many common benchmark domains, a small number of simple, intuitive control rules provides sufficient pruning that the remaining part of the search tree can simply be searched depth first, sometimes even without backtracking.

The present work on TALplanner is inspired by TLPLAN and may also be characterized as a forward chaining planner using domain-dependent knowledge to control search. On the other hand, there are also a number of significant differences between the planners.

- While TLPLAN uses a temporal logic only for specifying control knowledge, the formal basis for TALplanner originates from Temporal Action Logics (TAL) [14], a family of narrative-based logics for reasoning about action and change in dynamic and incompletely specified environments. TALplanner accepts a TAL goal narrative as input and generates a TAL plan narrative as output. Thus, TAL serves as a reference formalism for TALplanner, where the language used to represent narratives in TAL may be viewed as a rich plan representation language. All aspects of planning domains and problem instances are formally characterized in TAL and provided with a formal semantics, including goal statements, control formulas, operator definitions, and information regarding the initial state.
- Unlike LTL, TAL is not a modal logic. It is a non-monotonic first-order linear metric time logic originally developed for reasoning about action and change. It includes the use of circumscription and predicate completion techniques.
- Due to the use of modal temporal formulas in TLPLAN and the character of LTL semantics, the use of a formula progression algorithm as a basis for the forward chaining planner is a natural and powerful technique. A similar approach can be used with TALplanner: Modal formulas can be emulated by defining macros which are reduced to standard TAL formulas, and the formula progression algorithm can be extended to allow for TAL operators with duration and internal state changes within the duration. However, TALplan-

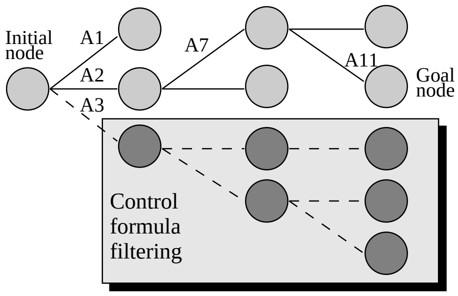

Figure 1. Pruning a Forward Chaining Search Space

ner can also use direct evaluation of TAL control formulas without formula progression, which enables the use of certain formula optimization techniques described briefly in Section 4.4.

- TALplanner has an extensive analysis phase where TAL control formulas and operator definitions are analyzed in order to increase performance during the planning phase. Some of the techniques being used will be discussed briefly in Section 4.4; the details will be presented in a forthcoming paper.
- Whereas TLPLAN generates sequential plans using single-step operators, TALplanner can generate sequential or concurrent plans using both single-step operators and operators with duration, and also allows internal state changes within the execution interval of an action. TALplanner also allows the use of an interesting class of resource types.

The development of TALplanner has followed an incremental methodology that has proven useful during the development of the logics in the TAL family.

Each TAL logic uses a different version of a high-level macro language $\mathcal{L}$(ND), which was initially quite restrictive but has been incrementally extended to allow increased expressivity as well as to provide new macros suitable to certain specific reasoning tasks. The formal specification for the different versions of $\mathcal{L}$(ND) is provided in terms of a translation into an expressive shared logical base language $\mathcal{L}$(FL). Making small incremental extensions facilitates the task of ensuring that all constructs provided by the language have a sound formal basis, while the use of a shared base language provides a common formal basis for all TAL logics and facilitates comparisons between different logics.

Using this methodology for TALplanner entails starting with a restricted version of a planning domain specification language called $\mathcal{L}(\mathrm{ND})^*$, implementing an efficient planner for this language, and then incrementally extending its expressivity. Extensions should always be grounded in the logical base language $\mathcal{L}$(FL). Each extension must also be tested empirically for efficiency, using existing benchmarks when possible and extending benchmarks when they lack features for testing the extensions.

Following this methodology, we first developed a restricted version of TALplanner and presented it in Doherty and Kvarnström [15]. Although very restricted compared to the full expressive power of recent logics in the TAL family, this version of TALplanner still allowed ADL-style actions [35] with a number of extensions. The paper presented both a modal emulation version of TALplanner, using a formula progression algorithm, and a non-modal version using formula evaluation. Compared to TLPLAN, both versions of TALplanner demonstrated significant speedups and considerably less use of memory on a suite of benchmarks from different planning domains.

In recent work, these two approaches have been unified into a single planner using both progressed modal control rules and evaluated non-modal control rules. This enables the user to take advantage of the relative strengths of each approach within any planning domain. TALplanner and its domain specification language have also been extended for the specification of resource constraints and the algorithms have been generalized for the use of concurrent actions [30]. Furthermore, a number of pre-processing and analysis techniques have been developed that improve the performance of the planner, especially when non-modal control formulas are used. These new enhancements have improved the performance of TALplanner reported in Doherty and Kvarnström [15] by many orders of magnitude for large problem instances.

The results of these and other extensions and improvements, and testing the extensions empirically via the use of standard and extended benchmarks, are the focus of this paper. To enhance the coherency and incremental flavor of the presentation, we will first present the unified sequential version of TALplanner and then add extensions for resources and concurrency.

1.1. The Structure of the Paper

In Section 2, the Temporal Action Logics (TAL) framework is presented. An example of a TAL narrative in the surface language $\mathcal{L}$(ND) and its translation into the logical base language $\mathcal{L}$(FL) is also provided. Section 3 contains an overview of the $\mathcal{L}$(ND) extensions required for the planning task, including goal statements, control statements, and a new, more concise syntax for action types (plan operators). It is also shown how the extended language $\mathcal{L}(\text{ND})^*$ is used in TALplanner, and a small example is presented using the well-known gripper domain. The unified sequential version of the TALplanner algorithm is presented in Section 4. In Section 5, the algorithm is modified and extended to generate concurrent plans, while Section 6 presents extensions for explicitly modeling resource consumption and production. Section 7 contains a discussion of soundness and completeness for TALplanner. In Section 8, the three different versions of TALplanner are empirically tested using a number of standard and extended benchmarks. The results are provided and discussed. Finally, Section 9 concludes with discussion and future work.

2. TAL: Temporal Action Logics

TAL, Temporal Action Logics [14], is a family of non-monotonic temporal logics developed for reasoning about action and change in dynamic and incompletely specified domains. The TAL logics originate from the Features and Fluents framework developed by Sandewall [36]. TAL is a narrative based formalism, where narratives are specified in a high-level macro language $\mathcal{L}$(ND) which can be translated into a logical base language $\mathcal{L}$(FL). The rich expressivity of the $\mathcal{L}$(ND) language includes the modeling of actions with duration, context-dependent actions, non-deterministic actions, delayed effects of actions [24], concurrency [23], incompletely specified timing of actions, and side-effects of actions [19]. It also provides robust solutions to the frame [13], ramification [19] and qualification [29] problems when used for reasoning in restricted domains in incompletely specified environments. All of these features have a corresponding formal semantics.

If action types in $\mathcal{L}$(ND) are viewed as plan operators, the language enables the representation of many types of plan operators such as those used in STRIPS, ADL and other planning formalisms. Due to the use of a surface macro language, it is straightforward to incrementally extend and generalize existing plan operator languages, or in cases where this is not adequate to simply use the rich language expressivity associated with TAL to define new plan operator languages. This approach gives a formal semantics to plan operators and thereby provides a very solid basis for experimenting with planners and formally verifying the correctness of generated plans.

In the remainder of this section, however, the main focus will be on the use of TAL in the area of reasoning about action and change. This will be explained using a fragment of TAL-C [14,23], one of the most recent members of the TAL family. We will also present a short narrative that provides concrete examples of certain statement classes in the TAL surface language $\mathcal{L}$(ND). This will provide a basis for understanding how TAL is used in TALplanner.

2.1. TAL Narrative Descriptions

When reasoning about action and change in TAL, a world domain is characterized in terms of a set of *fluents*, which are time dependent functions representing the change in the value of a specific property through time.

A *narrative type specification* defines a set of (currently finite) fluent value domains and contains type specifications for the fluents that describe the world and the actions that are available to an agent.

A *narrative background specification* consists of a narrative type specification together with other background information that is common to all narratives for a particular domain. This information is provided as a set of labeled narrative statements in the surface language $\mathcal{L}$(ND); examples will be given for each statement class in Section 2.2. Persistence statements (labeled per) allow each fluent to be specified as being of type *persistent* (normally retaining its value from the previous timepoint), *durational* (normally reverting to a default value), or *dynamic* (varying freely). Action type specifications (labeled acs) provide definitions of action types, while dependency constraints (labeled dep) characterize causal dependencies among fluents. Both action type specifications and domain constraints can provide exceptions to the persistence or default value assumptions for certain fluents at certain timepoints using the *reassignment macros* R, I and X, defined below. Finally, domain constraints (labeled dom) characterize information generally true in every state of the world.

A *narrative description*, or *narrative*, consists of a narrative background specification together with additional information specific to a particular reasoning problem. The additional information is also given as a set of labeled narrative statements in $\mathcal{L}$(ND). Action occurrence statements (labeled occ) specify which action instances occur and when, while observation statements (labeled obs) provide timing constraints as well as observations of fluent values at particular timepoints.

2.1.1. The Surface Language $\mathcal{L}(ND)$

Some brief comments about the most often used macros in $\mathcal{L}$(ND) are provided to assist in understanding the narratives presented in this article.

A *fixed fluent formula* $[\tau]\, f \mathrel{\hat{=}} v$ expresses the fact that the fluent f has the value v at the timepoint τ. For boolean fluents, the shorthand notation $[\tau]\, f \stackrel{\text{def}}{=} [\tau]\, f \mathrel{\hat{=}} \mathsf{true}$ and $[\tau]\, \neg f \stackrel{\text{def}}{=} [\tau]\, f \mathrel{\hat{=}} \mathsf{false}$ is allowed. Boolean connectives are allowed within a temporal scope (for example, $[\tau]\, f \mathrel{\hat{=}} v \wedge g \mathrel{\hat{=}} v'$), and closed, open, or semi-open intervals are permitted (for example, $[\tau, \tau']\, f \mathrel{\hat{=}} v$). The function $\mathit{value}(\tau, f)$ returns the value of the fluent f at the timepoint τ, where $[\tau]\ f \mathrel{\hat{=}} v$ iff $\mathit{value}(\tau, f) = v$. The expression $[\tau]\ f \mathrel{\hat{=}} g$, where f and g are fluents, is a shorthand notation for $[\tau]\ f \mathrel{\hat{=}} \mathit{value}(\tau, g)$.

An *occlusion expression* $X([\tau, \tau']\, \phi)$ expresses the fact that all fluents in ϕ are occluded (exempt from the persistence or default value assumptions) in the interval $[\tau, \tau']$. A *reassignment expression*, $R([\tau, \tau']\, \phi) \stackrel{\text{def}}{=} X([\tau, \tau']\, \phi) \wedge [\tau']\, \phi$, also requires ϕ to hold at the end of the interval, while a *durational reassignment expression* $I([\tau, \tau']\, \phi) \stackrel{\text{def}}{=} X([\tau, \tau']\, \phi) \wedge [\tau, \tau']\, \phi$ requires ϕ to hold throughout the interval. This is generalized to open, semi-open and singleton intervals.

An *atomic expression* is either any of the expressions defined above or a *feature*, *value* or *timepoint equality expression* ($f = f'$, $v = v'$, $\tau = \tau'$), a *temporal relational expression* ($\tau \otimes \tau'$, where $\otimes$ is a relation symbol in the temporal base structure), or an *action occurrence expression* ($[\tau, \tau']\, A(\overline{\omega})$, stating that the action A, with arguments $\overline{\omega}$, occurs during the interval $[\tau, \tau']$).

Statements in $\mathcal{L}$(ND) are formed from atomic expressions in a manner similar to the definition of well-formed formulas in a first-order logical language using the standard connectives, quantifiers and notational conventions.

2.1.2. The Base Language $\mathcal{L}(FL)$

In order to reason about a particular narrative, it is first mechanically translated into the base language $\mathcal{L}$(FL), an order-sorted classical first-order language with equality, using the translation function *Trans*() defined in Appendix A (see also Doherty et al. [14]). The base language uses the following predicates:

- $\mathit{Holds}(t, f, v)$ expresses that the fluent f has the value v at the timepoint t,
- $\mathit{Occlude}(t, f)$ expresses that the fluent f may change value at t (corresponding to the reassignment macros),
- $\mathit{Occurs}(t, t', a)$ expresses that the action a occurs between t and t',
- $\mathit{Per}(t, f)$ expresses that the fluent f is persistent at time t, and

- $Dur(t, f, v)$ expresses that f is durational with default value v at time t.

The logical theory which is the result of the translation is still under-constrained in the sense that a number of implicit assumptions about fluent change in the world remain to be characterized. In general, we want to encode the blanket assumption that fluent values do not change unless there is a good reason for this to happen. There are a number of legitimate reasons for fluents to change value, such as action occurrences where the effects of the action change fluent values, or causal dependencies between fluents where changes in some fluents force changes in others. In the TAL formalism, all such legitimate reasons for change are represented implicitly using the reassignment macros R, I and X in dependency constraints and action type definitions. When translated, these statements result in constraints on the *Occlude* predicate.

In the logical theory, we want to formally encode the assumption that these are the *only* reasons for fluents to be occluded. This is done by using a special form of circumscription [33] called filtered circumscription [16] which involves adding a second-order formula to the narrative logical theory. The *Occlude* predicate is circumscribed relative to the action definitions and dependency constraints with all other predicates fixed, and *Occurs* is circumscribed relative to the action occurrence formulas with all other predicates fixed. The results are combined and filtered with the $\mathcal{L}$(FL) translations of the persistence statements (forcing persistent and durational fluents to adhere to the persistence or default value assumptions), domain constraints, observations, and timing constraints, as well as the $\mathcal{L}$(FL) foundational axioms and temporal structure axioms (TAL uses a linear, discrete time structure with non-negative time). Except for the temporal structure axioms, the resulting second-order theory can be translated into a logically equivalent first-order theory, which is then used to reason about the narrative. In the remainder of the article, $Trans^{+}(\mathcal{N})$ will denote the result of translating the narrative $\mathcal{N}$ into $\mathcal{L}$(FL) and applying this filtered circumscription policy, which is formally defined in Appendix B.

The problems associated with building in such assumptions for fluent change are often called the frame, ramification and qualification problems. Although they are not directly related to the algorithmic mechanisms of TALplanner, it should be pointed out that most planning algorithms build in a form of closed world assumption (CWA) where fluents only change value due to action occurrences.

Refer to Doherty et al. [14] for a more detailed overview of TAL and the $\mathcal{L}$(ND) and $\mathcal{L}$(FL) languages.

2.2. A TAL Narrative Example

The following narrative is a variation of the well-known hiding turkey scenario [36]. In this variation, there is a turkey that is alive and not hiding at time 0 (obs1) and that may or may not be deaf. This turkey is afraid of sounds: If it is not deaf and there is some noise, it will hide (dep1). When there has been no noise for ten consecutive timepoints, it will finally stop hiding (dep2).

The scenario also involves a gun, which is not loaded in the initial state (obs1). There are two actions at our disposal: We can Load the gun, which ensures that the gun is loaded when the action has been executed but also makes some noise throughout the duration of the action (acs1). While the other fluents are persistent (per1), noise is durational with default value false (per2), since there is normally no noise unless someone is currently making noise. Thus, after the gun is loaded, noise will automatically revert to false. We can also Fire the gun, which results in the gun no longer being loaded – and if the gun was loaded and the turkey was not hiding, it will die (acs2).

In this particular scenario, the gun is Loaded the gun between 1 and 4 (occ1) and Fired between 5 and 6 (occ2). There are two sets of preferred models[1] of this narrative, one where the turkey is deaf and one where it is not. In the first model set, the turkey does not hide and ends up being shot, while in the second model set, it hears the noise, hides, and emerges from hiding ten timepoints later.

The $\mathcal{L}$(ND) formalization of this scenario is as follows:

per1 $\forall t\, [\mathtt{true} \rightarrow Per(t+1, \mathsf{alive}) \wedge Per(t+1, \mathsf{deaf}) \wedge Per(t+1, \mathsf{hiding}) \wedge Per(t+1, \mathsf{loaded})]$
per2 $\forall t\, [\mathtt{true} \rightarrow Dur(t, \mathsf{noise}, \mathsf{false})]$
acs1 $[t_1, t_2]\ \mathsf{Load} \leadsto R((t_1, t_2]\ \mathsf{loaded}) \wedge I((t_1, t_2]\ \mathsf{noise})$
acs2 $[t_1, t_2]\ \mathsf{Fire} \leadsto ([t_1]\ \mathsf{loaded} \wedge \neg\mathsf{hiding} \rightarrow R((t_1, t_2]\ \neg\mathsf{alive})) \wedge$
$([t_1]\ \mathsf{loaded} \rightarrow R((t_1, t_2]\ \neg\mathsf{loaded}))$
dep1 $\forall t\, [[t]\ \neg\mathsf{hiding} \wedge \neg\mathsf{deaf} \wedge \mathsf{noise} \rightarrow R([t+1]\ \mathsf{hiding})]$
dep2 $\forall t\, [[t, t+9]\ \mathsf{hiding} \wedge \neg\mathsf{noise} \rightarrow R([t+10]\ \neg\mathsf{hiding})]$
obs1 $[0]\ \mathsf{alive} \wedge \neg\mathsf{loaded} \wedge \neg\mathsf{hiding}$
occ1 $[1, 4]\ \mathsf{Load}$
occ2 $[5, 6]\ \mathsf{Fire}$

The translation into $\mathcal{L}$(FL) is somewhat more complex, demonstrating some of the advantages in providing the macros in $\mathcal{L}$(ND). (Here, $\neg Holds(\tau, f, \mathsf{true})$ has been simplified into $Holds(\tau, f, \mathsf{false})$.)

per1 $\forall t\, [\mathtt{true} \rightarrow Per(t+1, \mathsf{alive}) \wedge Per(t+1, \mathsf{deaf}) \wedge Per(t+1, \mathsf{hiding}) \wedge Per(t+1, \mathsf{loaded})]$
per2 $\forall t\, [\mathtt{true} \rightarrow Dur(t, \mathsf{noise}, \mathsf{false})]$
acs1 $Occurs(t_1, t_2, \mathsf{Load}) \rightarrow$
$Holds(t_2, \mathsf{loaded}, \mathsf{true}) \wedge \forall t[t_1 < t \leq t_2 \rightarrow Occlude(t, \mathsf{loaded})] \wedge$
$\forall t[t_1 < t \leq t_2 \rightarrow Holds(t, \mathsf{noise}, \mathsf{true})] \wedge \forall t[t_1 < t \leq t_2 \rightarrow Occlude(t, \mathsf{noise})$
acs2 $Occurs(t_1, t_2, \mathsf{Fire}) \rightarrow ((Holds(t_1, \mathsf{loaded}, \mathsf{true}) \wedge Holds(t_1, \mathsf{hiding}, \mathsf{false}) \rightarrow$
$Holds(t_2, \mathsf{alive}, \mathsf{false}) \wedge \forall t[t_1 < t \leq t_2 \rightarrow Occlude(t, \mathsf{alive})]) \wedge$
$(Holds(t_1, \mathsf{loaded}, \mathsf{true}) \rightarrow$
$Holds(t_2, \mathsf{loaded}, \mathsf{false}) \wedge \forall t[t_1 < t \leq t_2 \rightarrow Occlude(t, \mathsf{loaded})]))$
dep1 $\forall t\, [Holds(t, \mathsf{hiding}, \mathsf{false}) \wedge Holds(t, \mathsf{deaf}, \mathsf{false}) \wedge Holds(t, \mathsf{noise}, \mathsf{true})) \rightarrow$
$Holds(t+1, \mathsf{hiding}, \mathsf{true}) \wedge Occlude(t+1, \mathsf{hiding})]$

[1] The *preferred models* of a narrative are the models of the *circumscribed* narrative theory.

dep2 $\forall t\,[\forall t'[t \leq t' \leq t+9 \rightarrow \mathit{Holds}(t', \mathsf{hiding}, \mathsf{true}) \wedge \mathit{Holds}(t', \mathsf{noise}, \mathsf{false})] \rightarrow$
$\mathit{Holds}(t+10, \mathsf{hiding}, \mathsf{false}) \wedge \mathit{Occlude}(t+10, \mathsf{hiding})$
obs1 $\mathit{Holds}(0, \mathsf{alive}, \mathsf{true}) \wedge \mathit{Holds}(0, \mathsf{loaded}, \mathsf{false}) \wedge \mathit{Holds}(0, \mathsf{hiding}, \mathsf{false})$
occ1 $\mathit{Occurs}(1, 4, \mathsf{Load})$
occ2 $\mathit{Occurs}(5, 6, \mathsf{Fire})$

The circumscription of the *Occurs* predicate in the action occurrences (occ) above (that is, $\mathit{Circ}_{SO}(\Gamma_{occ}(\mathit{Occurs}); \mathit{Occurs})$ as defined in Appendix B) is equivalent to the following first-order formula:

$\forall t, t', a[\mathit{Occurs}(t, t', a) \leftrightarrow (t = 1 \wedge t' = 4 \wedge a = \mathsf{Load}) \vee (t = 5 \wedge t' = 6 \wedge a = \mathsf{Fire})]$

The circumscription of the *Occlude* predicate in the action schemas (acs) and dependency constraints (dep) above (that is, $\mathit{Circ}_{SO}(\Gamma'(\mathit{Occlude}); \mathit{Occlude})$ as defined in Appendix B) is equivalent to the following set of first-order formulas:

$\forall t[\mathit{Occlude}(t, \mathsf{alive}) \leftrightarrow t = 6 \wedge \mathit{Holds}(5, \mathsf{loaded}, \mathsf{true}) \wedge \mathit{Holds}(5, \mathsf{hiding}, \mathsf{false})]$
$\forall t[\mathit{Occlude}(t, \mathsf{loaded}) \leftrightarrow 2 \leq t \leq 4 \vee t = 6 \wedge \mathit{Holds}(5, \mathsf{loaded}, \mathsf{true})]$
$\forall t[\neg \mathit{Occlude}(t, \mathsf{deaf})]$
$\forall t[\mathit{Occlude}(t, \mathsf{hiding}) \leftrightarrow$
$\exists t'[t = t' + 1 \wedge \mathit{Holds}(t', \mathsf{hiding}, \mathsf{false}) \wedge \mathit{Holds}(t', \mathsf{deaf}, \mathsf{false}) \wedge \mathit{Holds}(t', \mathsf{noise}, \mathsf{true})] \vee$
$\exists t'[t = t' + 10 \wedge \forall \tau[t' \leq \tau \leq t' + 9 \rightarrow \mathit{Holds}(\tau, \mathsf{hiding}, \mathsf{true}) \wedge \mathit{Holds}(\tau, \mathsf{noise}, \mathsf{false})]]]$
$\forall t[\mathit{Occlude}(t, \mathsf{noise}) \leftrightarrow 2 \leq t \leq 4]$

3. The TALplanner Framework

Clearly, the expressivity associated with recent logics in the TAL family already provides sufficient support for modeling many aspects of the specification of a planning problem, including common concepts such as operator definitions and specifications of initial states as well as less commonly used features such as non-deterministic operators, state constraints, and instantaneous and delayed indirect effects of actions.

However, when our work with TALplanner began, existing TAL logics had no provisions for specifying goals or domain-dependent control information. Following the TAL methodology of introducing new macros when needed but keeping the base language $\mathcal{L}(\mathrm{FL})$ and its circumscription policy unchanged, we therefore defined a new language called $\mathcal{L}(\mathrm{ND})^*$, a new version of the high-level surface language $\mathcal{L}(\mathrm{ND})$ which provides a number of new statement classes and macros. These extensions will be discussed in detail later in this section.

Given this extended narrative definition language, it is possible to view the planning task entirely in terms of TAL narratives (Figure 2). The input to the planner is an $\mathcal{L}(\mathrm{ND})^*$ *goal narrative* that contains a set of operator definitions and a description of the initial state, but that does not contain any action occurrences. TALplanner searches for a set of action occurrences (plan steps) that can be added to this narrative so that in the corresponding logical model, a goal state is achieved. If this succeeds, the output is a new TAL *plan narrative* in $\mathcal{L}(\mathrm{ND})$

where control rules and goal statements have been removed and the appropriate set of action occurrences has been added. In this process, the semantics of the goal and plan narratives is defined by the translation into the base language $\mathcal{L}(\mathrm{FL})$.

As an aid to understanding the concepts that will be presented in the remainder of this section, we will now show a concrete example using a variation of the well-known gripper domain, where a robot with a number of grippers can move objects between a number of rooms. Then, we will discuss the process of searching for plans given a TAL goal narrative, and explain how this process can be aided by allowing the domain designer to specify additional control knowledge in terms of temporal control formulas constraining the state sequence corresponding to an operator sequence. We will also discuss goal specifications and how the power of control formulas can be increased by allowing them to explicitly refer to the goal. A set of control formulas for the gripper domain will be provided. Finally, we will return once more to the abstract view of TALplanner, discussing Figure 2 in more detail.

3.1. A Robot Gripper Example

There are several different ways to model the robot gripper domain, partly depending on which aspects of the domain are considered important (for example, are balls interchangeable or is it important which ball ends up in which room?) and partly depending on the restrictions placed on the formalization by the planning formalism being used, such as whether resources and/or non-boolean properties can be modeled and whether object typing is explicitly supported or must be emulated through the use of unary predicates.

Here, balls are assumed not to be interchangeable, and the order-sorted type system of TAL will be used. The value domains used are obj for things (including the robot), ball for balls (a subtype of obj), room for rooms, and gripper for grippers. We use the fluents loc specifying the location of objects, free specifying whether a given gripper is free, and carry specifying whether the robot is carrying

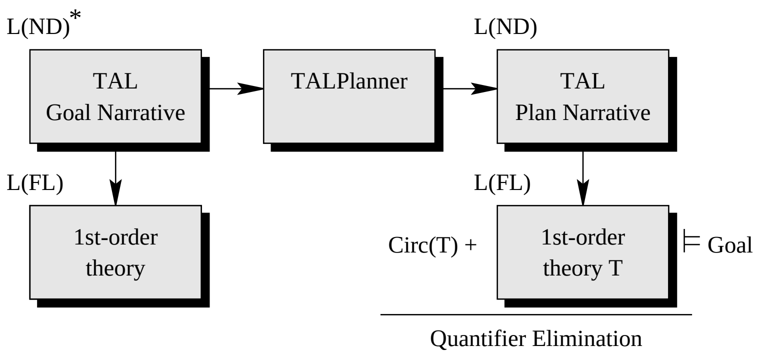

Figure 2. The relation between TAL and TALplanner

a certain object in a certain gripper. The goal narrative thus begins with the following labeled statements, which also specify the actual objects present in this specific problem instance.

```
#domain obj :elements { ball1, ball2, ball3, robot }
#domain ball :parent obj :elements { ball1, ball2, ball3 }
#domain room :elements { roomA, roomB }
#domain gripper :elements { left, right }

#feature loc(obj) :domain room
#feature free(gripper) :domain boolean
#feature carry(ball, gripper) :domain boolean
```

It is also necessary to define the three operators that are available to the robot. The robot can pick up a ball in any free gripper, as long as the ball is in the same location and the robot is not already carrying it. It can always drop a ball that it is carrying. Finally, it can move to another room, which changes not only the location of the robot but also the location of any ball that it is currently carrying. Note that in this formalization of the gripper domain, there is no "map" modeling connections between rooms. Instead, we make the common (although somewhat unrealistic) assumption that the robot may move directly from any room to any other room.

acs1 $[t_1, t_2]$ $\mathsf{pick}(ball, gripper) \leadsto$
$\quad [t_1]\ \mathsf{loc}(ball) \hat{=} \mathsf{loc}(\mathsf{robot}) \wedge \mathsf{free}(gripper) \wedge \neg\exists gripper'[\mathsf{carry}(ball, gripper')] \rightarrow$
$\quad\quad R([t_2]\ \mathsf{carry}(ball, gripper) \wedge \neg\mathsf{free}(gripper))$
acs2 $[t_1, t_2]$ $\mathsf{drop}(ball, gripper) \leadsto$
$\quad [t_1]\ \mathsf{carry}(ball, gripper) \rightarrow$
$\quad\quad R([t_2]\ \neg\mathsf{carry}(ball, gripper) \wedge \mathsf{free}(gripper))$
acs3 $[t_1, t_2]$ $\mathsf{move\text{-}to}(room) \leadsto$
$\quad [t_1]\ \mathsf{loc}(\mathsf{robot}) \hat{\neq} room \rightarrow$
$\quad\quad R([t_2]\ \mathsf{loc}(\mathsf{robot}) \hat{=} room) \wedge$
$\quad\quad \forall ball[[t_1]\ \exists gripper[\mathsf{carry}(ball, gripper)] \rightarrow R([t_2]\ \mathsf{loc}(ball) \hat{=} room)]$

Finally, the initial state and goal must be defined. In the initial state, all grippers are free and the robot and all balls are in room A. The goal requires all balls to be in room B and requires all grippers to be free but does not constrain the location of the robot. The exact goal statement syntax in $\mathcal{L}(\mathrm{ND})^*$ will be defined in Section 3.4.

```
#obs ∀gripper, ball [ [0] free(gripper) ∧ ¬carry(ball, gripper) ]
#obs ∀obj [ [0] loc(obj) =̂ roomA ]
#goal ∀ball [ loc(ball) =̂ roomB ] ∧ ∀gripper [ free(gripper) ]
```

Thus, there are initially three balls in room A, all of which should be moved to room B. Given the control knowledge that will be presented in Section 3.5, TALplanner will produce the following sequential plan:

```
#occ [0,1] pick(ball1, left)
#occ [1,2] pick(ball2, right)
#occ [2,3] move-to(roomB)
#occ [3,4] drop(ball1, left)
#occ [4,5] drop(ball2, right)
#occ [5,6] move-to(roomA)
#occ [6,7] pick(ball3, left)
#occ [7,8] move-to(roomB)
#occ [8,9] drop(ball3, left)
```

3.2. Searching for Plans

Although Figure 2 provides an abstract view of a planner for the TAL formalism, it provides no information as to *how* a suitable set of action occurrences should be generated, much less how it should be generated *efficiently.* The answer to this question depends both on the type of plans one is interested in (for example, whether one is interested in sequential or concurrent plans) and on the expressivity one wishes to allow (for example, whether indirect effects or non-deterministic operators should be allowed, and whether operators can have varying durations).

The current version of TALplanner places a number of restrictions on the TAL goal narratives that are allowed. As is the case with many existing planners, TALplanner currently requires complete information about the initial state, and operators must be deterministic. The planner does not allow for the use of dependency constraints or indirect effects of actions, and only allows restricted forms of domain constraints.

This restricts the expressivity of TALplanner operators considerably compared to what is allowed in the TAL logic. However, it is still possible to use ADL-style operators as well as operators with extended duration and state changes within the duration, as will be demonstrated in the extended logistics domain in Section 8.3. Thus, the current level of expressivity is still considerable compared to that allowed by many STRIPS- or ADL-based planners. Also, following the research methodology presented in the introduction, we intend to incrementally relax these restrictions in future versions of TALplanner, adding support for incomplete states, non-deterministic operators and restricted forms of indirect effects of actions.

Taking this into consideration, the forward-chaining paradigm, used in TLPLAN as well as many other recent planners such as HSP [12] and FF [21,20], appears to be a good choice for both the sequential and the concurrent version of TALplanner. Compared to other types of planners, forward-chaining planners have the advantage of always having access to a complete description of past and current states, which facilitates the use of operator types with complex preconditions and conditional effects as well as future extensions involving indirect effects.

The forward chaining search space is defined by the initial state and the set of operator instances applicable in any given state. An example for the gripper domain is shown in Figure 3. In the initial node, a number of operator instances

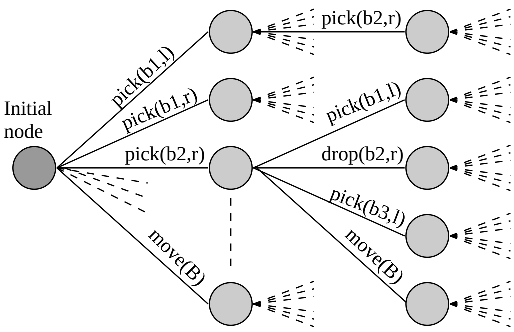

Figure 3. The Forward Chaining Search Space for the Gripper Domain

such as pick(ball1,left), pick(ball1,right) and move-to(roomB) are applicable. If the robot picks up ball2 in the right gripper, there are exactly four applicable actions in the resulting state: Pick up ball1 or ball3 in the remaining gripper, move to roomB, or drop the ball that was just picked up.

Here, nodes have been identified with operator sequences, which allows the search space to be represented as a tree. If nodes are instead identified with states, this structure would have to be generalized to a directed graph. For example, executing pick(ball1,left) followed by pick(ball2,right) leads to the same state as executing pick(ball2,right) followed by pick(ball1,left). The exact definitions of operator sequences, plans and search trees will be given separately for sequential plans in Section 4 and for concurrent plans in Section 5.

3.3. Pruning the Search Tree using Temporal Control Formulas

Although the forward-chaining search tree can be searched using standard algorithms such as depth first, breadth first, or iterative deepening, these algorithms are not goal-directed and would usually lead to very inefficient planners. Fortunately, there are a number of ways to alter these algorithms in order to improve performance. One can identify two main categories of techniques being used for this purpose in recent forward-chaining planners: A planner can use various forms of domain-dependent or domain-independent *heuristics* to guide the search process, as in the very successful planners HSP [12] and FF [21,20], or it can use domain-dependent or domain-independent *pruning* to remove search nodes completely, thereby decreasing the size of the search tree.

Naturally, these two techniques can easily be combined. Both TLPLAN and TALplanner allow the user to specify domain-dependent heuristics based on a single state as well as domain-dependent pruning rules in the form of *temporal control formulas*, which can refer to the entire state sequence corresponding to a plan as well as to the goal of a specific planning instance. If a state sequence generated by a planner is seen as a logical model, control formulas can be viewed as model filterers that can be used to rule out state sequences that cannot lead

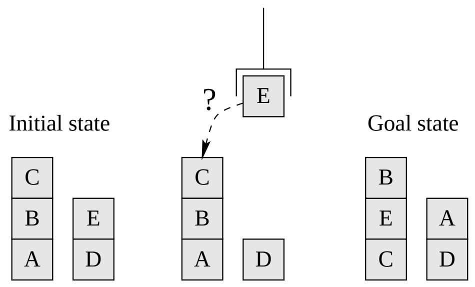

Figure 4. Pruning Nodes in the Blocks World

to plans or lead to suboptimal or redundant plans. In this paper, we will mainly concentrate on the use of such control formulas, leaving the detailed investigation of the combination of heuristics and pruning to future research.

The utility of domain-dependent pruning rules (in another form than the ones used by TLPLAN and TALplanner) was demonstrated at least as early as 1981, when Kibler and Morris [26] presented four intuitive control rules for the blocks world domain. One of these rules was "don't add to a pile containing a block that needs to be moved". For example, in Figure 4, the robot arm has just picked up block E. This block should be on top of C – but if it is placed there immediately, it will eventually have to be removed again in order to move blocks A and B. Informally, this rule can be expressed as a temporal formula of the form "if in any state x is not on top of y, and y belongs to a tower containing a block that must eventually be moved, then in the *next* state, x should still not be on top of y".

There are a number of different logics that are suitable for specifying temporal control formulas. TLPLAN uses a first-order version of the modal tense logic LTL, while TALplanner can either use TAL logic formulas or emulate the modal formulas used by TLPLAN. These alternatives will be discussed in more detail in the following subsections.

3.3.1. Control Formulas in TLPLAN

TLPLAN control formulas are defined in a first-order version of LTL (Linear Temporal Logic [17]), a linear modal tense logic. There are four temporal modalities: $\alpha \mathsf{U} \beta$ (α holds *until* β holds), $\Diamond \alpha$ (α will hold *eventually*), $\Box \alpha$ (α *always* holds), and $\bigcirc \alpha$ (α holds in the *next* state).

The modal control formula specified by the domain designer is progressed through each state generated by an operator sequence.[2] If the result is the formula false, the planner has proven that the node and all its descendants must necessarily violate the control formula, and the node can be pruned.

For efficiency reasons, TLPLAN stores a progressed formula in each search

[2] The progression algorithm is published in [9]. A variation is used in Section 4.

node. Each time a new action is examined, the planner only needs to progress a previously stored formula through the single new state generated by the action. This saves time, but can use considerable amounts of memory for large problem instances, as will be shown in the blocks world benchmark examples in Section 8.4.

It should be noted that TLPLAN does not ensure that control formulas are actually *satisfied* by the final plan, since the progression algorithm does not take into consideration what happens (or does not happen) after the end of the final operator in an operator sequence. For example, control formulas of the form $\Diamond \phi$ (eventually, ϕ will hold) will never cause a state sequence to be pruned. Since the progression algorithm never considers the "future", it always allows for the possibility that it *might* be possible to achieve ϕ in some future state. This is true even for a formula such as $\Diamond \mathsf{false}$.

Although this is rarely a problem, it is important to realize that TLPLAN control rules are not goals in themselves, but are intended to guide the planner towards a state satisfying the state-based goal.[3]

Consider once again the gripper domain. When using the standard formalization where a robot can move "instantly" between any two rooms without considering intervening rooms, there is clearly no point in allowing two consecutive move-to actions. This can be expressed using the following control rule, stating that if the robot has changed locations from one timepoint to the next, it must then remain at the same location in the following state:

$$\Box\, \neg\exists l[\mathsf{loc}(\mathsf{robot}) = l \wedge \bigcirc\, \mathsf{loc}(\mathsf{robot}) = l] \rightarrow \exists l[\bigcirc\, \mathsf{loc}(\mathsf{robot}) = l \wedge \bigcirc \bigcirc\, \mathsf{loc}(\mathsf{robot}) = l]$$

3.3.2. Using TAL Control Formulas in TALplanner

While TLPLAN is based on the use of modal control formulas and a progression algorithm, TALplanner is primarily based on the use of non-modal TAL control formulas (labeled control in the extended language $\mathcal{L}(\mathrm{ND})^*$) and a formula evaluation algorithm. These formulas also have a different semantics: The TAL control formulas are currently viewed as part of the goal specification, and must necessarily hold in any valid plan.[4]

Returning to the gripper domain, the rule that the robot should never move twice without picking up or dropping a ball can be formalized in TAL as follows:

$$\begin{aligned}\forall t.\mathit{value}(t, \mathsf{loc}(\mathsf{robot})) \neq \mathit{value}(t+1, \mathsf{loc}(\mathsf{robot})) \rightarrow \\ \mathit{value}(t+1, \mathsf{loc}(\mathsf{robot})) = \mathit{value}(t+2, \mathsf{loc}(\mathsf{robot}))\end{aligned}$$

[3] The TLPLAN implementation also allows the specification of temporally extended goals [8,10], modal temporal formulas that must be satisfied by the final plan. However, this cannot be used in combination with state-based goals, and also prevents the use of the goal modality (Section 3.4).

[4] This is not an integral part of the TALplanner algorithms: One could easily change the goal acceptance criterion so that non-modal control rules are used for pruning but are not seen as part of the goal.

However, the fact that TAL control formulas must hold in every valid plan does not necessarily imply that they must hold in every *prefix* of a valid plan. For example, any state goal can also be expressed as a control formula requiring that the state goal must eventually become true at some timepoint t and hold until infinity:

$$\exists t.\ [t, \infty)\ \forall ball[\mathsf{loc}(ball) \hat{=} \mathsf{roomB}] \wedge \forall gripper[\mathsf{free}(gripper)]$$

This condition will not be satisfied in the initial node, corresponding to the empty operator sequence – unless, of course, all balls were already in room B. Clearly, simply pruning every operator sequence that does not satisfy a control formula would make the planner incomplete. Instead, TALplanner analyzes the control formulas and automatically extracts a set of *pruning constraints*, formulas that do need to hold in every prefix of any valid plan.

The process of extracting pruning constraints is somewhat more complex than the use of progression in TLPLAN. This is especially true since in order to ensure that pruning constraints are as strong as possible, the extraction algorithms must take into account both the expressivity allowed in planning domain descriptions and the constraints that have been placed on the plans being generated (for example, whether sequential or concurrent plans are required).

On the other hand, the use of evaluation rather than progression also has several advantages. For example, there is no need to store a progressed control formula in each search node, which saves considerable amounts of memory. Also, as will be discussed briefly in Section 4.4, using evaluated pruning constraints facilitates certain kinds of formula analysis techniques that can be applied in order to take advantage of the domain-specific information inherent in the operator definitions for any given planning domain.

The process for extracting and optimizing pruning constraints will be discussed briefly in Section 4.4 for sequential TALplanner and Section 5.3 for concurrent TALplanner. A complete description will be presented in a future paper.

3.3.3. Using Emulated Modal Control Formulas in TALplanner

In addition to supporting TAL control formulas, TALplanner also allows the use of emulated modal control formulas (labeled mcontrol in the extended language $\mathcal{L}(\mathrm{ND})^*$) together with a progression algorithm shown in Section 4.5. However, due to the use of explicit time and operators with duration, the modal operators used by Kabanza et al. [22] have been adopted, where operators can be indexed with closed, open or semi-open temporal intervals.

There are four temporal operators, U (until), $\diamond$ (eventually), $\square$ (always), and $\circ$ (next), but all of them may be defined in terms of the U operator. As in Kabanza et al. [22], the first three temporal operators can be indexed with closed, open or semi-open intervals. The meaning of a formula containing a temporal operator is dependent on the timepoint n at which it is evaluated.

- $\phi \,\mathsf{U}_{[\tau,\tau']}\, \psi$ means that ϕ must hold from n until ψ is achieved at some timepoint in $[\tau + n, \tau' + n]$. We define $\phi \,\mathsf{U}\, \psi \equiv \phi \,\mathsf{U}_{[0,\infty]}\, \psi$.
- $\Diamond_{[\tau,\tau']}\, \phi \equiv \mathit{true}\,\mathsf{U}_{[\tau,\tau']}\, \phi$ means that eventually ϕ will be true at a timepoint in $[\tau + n, \tau' + n]$. We define $\Diamond\, \phi \equiv \Diamond_{[0,\infty]}\, \phi$.
- $\Box_{[\tau,\tau']}\, \phi \equiv \neg \Diamond_{[\tau,\tau']} \neg\phi$ means that ϕ must be true at all points in the interval $[\tau + n, \tau' + n]$. We define $\Box\, \phi \equiv \Box_{[0,\infty]}\, \phi$.
- $\bigcirc\, \phi \equiv \mathit{true}\,\mathsf{U}_{[1,1]}\, \phi$ means that ϕ must hold at $n + 1$.

For emulated modal control formulas, an *atomic expression* is either an *elementary fluent formula* of the form $f \hat{=} v$, expressing the fact that the fluent f has the value v, an equality expression or temporal relational expression as defined in Section 2.1, or a goal expression $\mathsf{goal}(\phi)$ expressing that the state goal entails ϕ (defined in Section 3.4). Modal control formulas in $\mathcal{L}(\mathrm{ND})^*$ are formed from atomic expressions in the standard manner using the four temporal modalities and the standard logical connectives, quantifiers and notational conventions.

From a semantic perspective, temporal modalities can be viewed as a special type of macro-operator in the extended surface language $\mathcal{L}(\mathrm{ND})^*$. The translation function TransModal takes a timepoint n and a modal control formula intended to be evaluated at n as input and returns a formula in $\mathcal{L}(\mathrm{ND})^*$ without temporal modalities as output. In the following, $\mathcal{Q}$ denotes a quantifier and $\otimes$ denotes a binary logical connective.

procedure $\mathsf{TransModal}(n, \gamma)$
if $\gamma = \mathcal{Q}x.\phi$ **then return** $\mathcal{Q}x.\mathsf{TransModal}(n, \phi)$
if $\gamma = \phi \otimes \psi$ **then return** $\mathsf{TransModal}(n, \phi) \otimes \mathsf{TransModal}(n, \psi)$
if $\gamma = \neg\phi$ **then return** $\neg\mathsf{TransModal}(n, \phi)$
if $\gamma = f(\overline{x}) \hat{=} v$ **then return** $[n]\ \gamma$
if γ contains no modalities **then return** γ
if $\gamma = \mathsf{goal}(\phi)$ **then return** $\mathsf{goal}(\phi)$
if $\gamma = (\phi \,\mathsf{U}_{[\tau,\tau']}\, \psi)$ **then return**
$\exists[t : n + \tau \leq t \leq n + \tau']\ (\mathsf{TransModal}(t, \psi) \wedge \forall[t' : n \leq t' < t]\mathsf{TransModal}(t', \phi))$

The algorithm TransModal provides the meaning of the temporal modalities in the linear, discrete temporal logic TAL, in correspondence with their intuitive meaning in a linear tense logic.

3.3.4. Converting Modal Control Formulas

From a knowledge engineering perspective, authors of control formulas may prefer to define control formulas using modalities without using a progression algorithm. In this case, modal control formulas can be automatically translated into the corresponding TAL control formulas.

However, in order to preserve the same pruning semantics used by TLPLAN when formula evaluation is used rather than progression, modal control formulas cannot simply be translated into *equivalent* TAL formulas using the TransModal algorithm. Instead, the translation procedure must automatically add the necessary contextualization to ensure that the value of the translated formula only depends on the states up to and including t_{fixed}, the last state that is guaranteed not to be changed by the addition of new operators. Therefore, an alternative algorithm called $\mathsf{TransModal}^+$ is defined. This algorithm takes a timepoint n and a modal control formula intended to be evaluated at n as input and returns a formula in $\mathcal{L}(\mathrm{ND})^*$ without temporal modalities, but with the additional fixed state time point constraint, as output. As in the definition of $\mathsf{TransModal}$, $\mathcal{Q}$ denotes a quantifier and $\otimes$ denotes a binary logical connective.

procedure $\mathsf{TransModal}^+(n, \gamma)$
if $\gamma = \mathcal{Q}x.\phi$ **then return** $n \leq t_{\mathsf{fixed}} \rightarrow \mathcal{Q}x.\mathsf{TransModal}^+(n, \phi)$
if $\gamma = \phi \otimes \psi$ **then return** $n \leq t_{\mathsf{fixed}} \rightarrow (\mathsf{TransModal}^+(n, \phi) \otimes \mathsf{TransModal}^+(n, \psi))$
if $\gamma = \neg\phi$ **then return** $n \leq t_{\mathsf{fixed}} \rightarrow \neg\mathsf{TransModal}^+(n, \phi)$
if $\gamma = f(\overline{x}) \hat{=} v$ **then return** $n \leq t_{\mathsf{fixed}} \rightarrow [n]\ \gamma$
if γ contains no modalities **then return** $n \leq t_{\mathsf{fixed}} \rightarrow \gamma$
if $\gamma = \mathsf{goal}(\phi)$ **then return** $n \leq t_{\mathsf{fixed}} \rightarrow \mathsf{goal}(\phi)$
if $\gamma = (\phi\ \mathsf{U}_{[\tau,\tau']}\ \psi)$ **then return** $(n + \tau' \leq t_{\mathsf{fixed}}) \rightarrow$
$\exists[t : n + \tau \leq t \leq n + \tau']\ (\mathsf{TransModal}^+(t, \psi) \wedge \forall[t' : n \leq t' < t]\ \mathsf{TransModal}^+(t', \phi))$

3.3.5. Comparing Progression and Evaluation of Control Formulas

Two ways of checking control formulas have now been discussed: Using progression and using formula evaluation. Both approaches have advantages and disadvantages in terms of computational complexity.

Although a naive progression algorithm would most likely do better than a naive formula evaluator, as demonstrated in earlier benchmark tests [15], we have found that evaluation enables certain optimizations for common classes of control formulas. These optimizations are considerably more difficult to apply to a progression algorithm. An additional advantage of not using progression is the significantly lower memory usage due to not needing to store a progressed control formula in each search node.

This, however, does not apply equally to all types of control formulas. For example, if a modal formula makes extensive use of the *until* operator, the evaluation of the corresponding TAL formula is more difficult to optimize, and the use of a progression algorithm is likely to improve performance.

Developing and analyzing optimization choices is currently being pursued as an active research issue.

3.4. Goals in $\mathcal{L}(ND)^*$

The state goal in a planning problem instance is specified using a set of *goal statements* in $\mathcal{L}(\mathrm{ND})^*$ (labeled goal) describing the desired properties of the goal state. Which restrictions should be placed on such formulas depends on the way in which the goal will be used by the planning algorithm.

Although any planning algorithm must be able to determine whether the final state resulting from a certain plan candidate entails the goal, TLPLAN has demonstrated the advantages of also being able to test whether the *goal* entails a certain formula. In the gripper domain, for example, one can intuitively see that if the robot is carrying one or more balls that should be in the current room – according to the goal – it should remain in the room until it has dropped them.

To allow such references to the goal, Bacchus and Kabanza [9] extend the first-order version of LTL used in TLPLAN by adding a goal modality, where $\mathsf{goal}(\phi)$ is true iff ϕ holds in every acceptable goal state, or, equivalently, iff the goal entails ϕ. Since TLPLAN is restricted to using conjunctive goals, this entailment check is very efficient. Using the goal modality, the gripper rule can be expressed as follows:

$$\Box\, \forall b, l, g[\mathsf{carry}(b, g) \wedge \mathsf{loc}(\mathsf{robot}) = l \wedge \mathsf{goal}(\mathsf{loc}(b) = l) \to \bigcirc\, \mathsf{loc}(\mathsf{robot}) = l].$$

While it would be possible to add a goal modality to TAL, this would require significant changes to the base language $\mathcal{L}$(FL). Instead, we prefer to once again follow the standard TAL methodology of adding new high-level macros and leaving the base language intact. This is achieved by introducing *goal expressions*, a new form of atomic expression in $\mathcal{L}$(ND)*.

Although allowing arbitrary (disjunctive and existential) goal statements and arbitrary goal expressions may require theorem proving, it is possible to enable the use of efficient formula evaluation techniques by placing suitable constraints on the allowed formulas. Such efficiency considerations must be weighed against the associated restrictions in expressivity. Therefore, two different alternatives are presented, one of which restricts expressivity but allows very efficient entailment checking and one of which allows more complex formulas. For each alternative, the translation into $\mathcal{L}$(FL) is provided, ensuring that the semantics of goal expressions is integrated with that of $\mathcal{L}$(FL).

3.4.1. Goals: A Straight-Forward Macro Translation

In the first alternative, *goal statements* are restricted to be conjunctions of expressions of the form $f \hat{=} v$, each of which states that a specific fluent (state variable) should take on a specific value. It should be noted that although explicit negations are not allowed, negative goals of the form $\neg\mathsf{at}(\mathsf{object}, \mathsf{location})$ can be written as $\mathsf{at}(\mathsf{object}, \mathsf{location}) \hat{=} \mathsf{false}$.

For each fluent $f : \mathsf{dom}_1 \times \cdots \times \mathsf{dom}_n \to \mathsf{dom}$, the translation into $\mathcal{L}$(FL) adds a corresponding goal fluent $\mathsf{goal}_f : \mathsf{dom}_1 \times \cdots \times \mathsf{dom}_n \times \mathsf{dom} \to \{\mathsf{true}, \mathsf{false}\}$. The intention is that $\mathsf{goal}_f(\overline{x}, v)$ should be true exactly when the goal entails that $f(\overline{x}) \hat{=} v$. This is achieved using TAL durational fluents. Each goal fluent is durational with default value false, meaning that each instance of the fluent is false whenever it is not explicitly forced to be true. What remains is to force the appropriate goal fluents to be true, which can be achieved by translating the goal statement $\wedge_{i=1}^{n} f_i(\overline{x}_i) \mathrel{\hat{=}} v_i$ into the dependency constraint $I(\forall t.[t]\ \wedge_{i=1}^{n} \mathsf{goal}_{f_i}(\overline{x}_i, v_i) \mathrel{\hat{=}} \mathsf{true})$.

Finally, *goal expressions* are restricted to be of the form $\mathsf{goal}(f(\overline{x}) \mathrel{\hat{=}} v)$. This expression can be translated into the $\mathcal{L}$(ND) formula $\forall t.[t]\ \mathsf{goal}_f(\overline{x}, v) \mathrel{\hat{=}} \mathsf{true}$. Due to the restrictions placed on goal statements, this notation can easily be extended to allow conjunctions, disjunctions, and quantification within the scope of the goal macro by observing the following equivalences:

- $\mathsf{goal}(\phi \wedge \psi) \equiv \mathsf{goal}(\phi) \wedge \mathsf{goal}(\psi)$
- $\mathsf{goal}(\phi \vee \psi) \equiv \mathsf{goal}(\phi) \vee \mathsf{goal}(\psi)$
- $\mathsf{goal}(\exists x.\phi) \equiv \exists x.\mathsf{goal}(\phi)$
- $\mathsf{goal}(\forall x.\phi) \equiv \forall x.\mathsf{goal}(\phi)$

Note that although some restrictions are placed on goal statements, this alternative still provides expressivity similar to or exceeding that of many existing planners, including TLPLAN.

3.4.2. Goals: Allowing Arbitrary Goals

Although the approach described above is sufficient for many planning problems, the planner should be able to use arbitrary goal statements and arbitrary formulas within the scope of the goal macro. This can be achieved with the help of some additional pre-processing.

A *fluent formula* in $\mathcal{L}$(ND) is any formula formed from expressions of the form $f \mathrel{\hat{=}} v$ using quantification over values and the ordinary boolean connectives. In the second alternative, a *goal statement* in $\mathcal{L}$(ND)* can be any fluent formula, and *goal expressions* are of the form $\mathsf{goal}(\phi)$, where ϕ can be any fluent formula.

The translation into $\mathcal{L}$(FL) is as follows. Let $\mathcal{GN}$ be a goal narrative with n occurrences of goal expressions, and let ψ be the conjunction of all goal statements in $\mathcal{GN}$. For each goal expression $\mathsf{goal}(\phi_i(\overline{x}_i))$ occurring in $\mathcal{GN}$, where $\overline{x}_i$ are the free variables occurring in ϕ_i, create a new boolean durational goal fluent $\mathsf{goal}_i(\overline{x}_i)$ with default value false and replace the goal expression with the expression $\forall t.[t]\ \mathsf{goal}_i(\overline{x}_i)$. What remains is to ensure that $\mathsf{goal}_i(\overline{x}_i)$ holds exactly for those $\overline{x}_i$ for which $\mathsf{goal}(\phi_i(\overline{x}_i))$ should hold. For all instances $\langle v_1, \ldots, v_m \rangle$ of $\overline{x}_i$ such that $\mathit{Trans}^+(\forall t.[t]\ \psi) \models \mathit{Trans}(\forall t.[t]\ \phi(v_1, \ldots, v_n))$, add a dependency constraint $I(\forall t.[t]\ \mathsf{goal}_i(v_1, \ldots, v_n) \mathrel{\hat{=}} \mathsf{true})$. Since all value domains are finite, only a finite number of dependency constraints will be needed.

It should be noted that the main purpose of this translation procedure is to define the semantics of goal expressions. An implementation is free to use other, more efficient methods, such as using a theorem prover or pre-generating a suitable representation of all acceptable goal states and using formula evaluation, as long as the method being used follows the semantics defined here.

3.5. Control Rules for the Gripper Domain

A complete set of control formulas for the gripper domain can now be shown.

When writing control rules, it is always very important to take into account the range of possible goals that one wants to plan for. Here, the assumption is that the state goal will only require certain balls, and possibly the robot, to be in certain rooms – for example, goals that require certain grippers to be free or non-free will not be allowed. Under these assumptions, the following control rules are very useful for pruning the search tree.

First, if the robot is carrying a ball that should be in the current room, then it should remain in the same room at the next timepoint. This implies that the robot will never leave a room before it drops the relevant balls. For modularity reasons, it is not explicitly stated that the robot must drop a ball: There may be other relevant actions to perform in the current state, especially in an extended version of this domain.

#control :name "stay-if-should-drop"
∀t, room [[t] loc(robot) $\hat{=}$ room ∧ ∃ball, gripper [carry(ball, gripper) ∧ goal(loc(ball)$\hat{=}$room)] →
    [t+1] loc(robot) $\hat{=}$ room]

Second, if the robot is in a room that contains a ball that should be somewhere else, and there is a free gripper, then it should remain in the same room.

#control :name "stay-if-should-pick-up"
∀t, ball, room [[t] loc(robot) $\hat{=}$ room ∧ loc(ball) $\hat{=}$ room ∧ ∃gripper [free(gripper)] ∧
    ∃room' [goal(loc(ball) $\hat{=}$ room') ∧ room' ≠ room] →
    [t+1] loc(robot) $\hat{=}$ room]

Third, for any ball that the robot is not currently carrying, if there is no explicit goal that the ball should be in another room, there is no point in picking it up.

#control :name "only-pick-up-relevant-balls"
∀t, ball [[t] ¬∃gripper [carry(ball, gripper)] ∧
    ¬∃room [goal(loc(ball) $\hat{=}$ room) ∧ loc(ball) ≠room] →
    [t+1] ¬∃gripper [carry(ball, gripper)]

3.6. Operator Definitions in $\mathcal{L}(ND)^*$

Although the standard $\mathcal{L}$(ND) macros for specifying TAL operators are very powerful, the syntax is quite different from that normally used in the planning community. For this reason, as well as in order to facilitate future extensions to plan operator specifications, a new operator macro is introduced. Below, the syntax of this macro is defined using extended BNF notation. Extensions for resources are defined in Section 6.

```
opdef   ::= "operator" opname [ "(" argument ( "," argument )* ")" ]
            ":at" temporalVariable
            [ ":precond" logicFormula ]
```

```
            context ( "," context )*

context ::= ":context"
            [ ":forall" valueVariable ( "," valueVariable )* ]
            [ ":condition" logicFormula ]
            ":effects" effect ( "," effect )*

effect  ::= "[" "+" delay "]"                  fluentTerm ":=" valueTerm |
            "[" "+" delay "," "+" enddelay "]" fluentTerm ":=" valueTerm
```

Given some binding of the operator argument variables and the invocation timepoint variable (specified by :at), the operator is invocable with those arguments at that timepoint iff the global precondition specified by :precond holds. The exact effects may be context-dependent, and the :conditions and the value terms used in effects can depend on not only the invocation state but also any preceding state. For actions with only one context, the :context keyword can be omitted.

Since TALplanner allows operators with duration, it is necessary to specify not only the effects but also the delay between the invocation timepoint and each effect. It is possible to define effects that take place at a single timepoint as well as effects that take place during an interval. Since both the delay and the enddelay may be arbitrary temporal terms, the duration of the action may depend on the state in which it is invoked. However, the delays must be strictly positive: No operator is allowed to have effects in or before the state in which it is invoked.

The exact semantics of an operator definition is defined by its translation into the base language $\mathcal{L}$(FL):

- $\mathit{Trans}_{\mathsf{eff}}(t, [+\tau, +\tau']\ f := v) = I([t+\tau, t+\tau']\ f \hat{=} v)$
- $\mathit{Trans}_{\mathsf{eff}}(t, [+\tau]\ f := v) = I([t+\tau]\ f \hat{=} v)$
- $\mathit{Trans}_{\mathsf{con}}(t, \text{:forall } v_1, \ldots, v_n \text{ :condition } \phi \text{ :effects } \psi_1, \ldots, \psi_m) =$ $\forall v_1, \ldots, v_n[\phi \to \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{eff}}(t, \psi_i)]$
- $\mathit{Trans}_{\mathsf{op}}(\text{operator } \mathit{name}(v_1, \ldots, v_n) \text{ :at } t \text{ :precond } \phi \text{ :context } c_1 \ldots \text{ :context } c_n) =$ $\forall t, t'.\mathit{Occurs}(t, t', \mathit{name}(v_1, \ldots, v_n)) \to \mathit{Trans}(\phi \to \bigwedge_{i=1}^{n} \mathit{Trans}_{\mathsf{con}}(t, c_i))$.

This defines a basic syntax and semantics for $\mathcal{L}$(ND)* operator definitions. Section 6 presents an extension that facilitates the specification of resource usage. We are currently working on extending TALplanner for incomplete knowledge and non-deterministic operators, which will necessitate further additions.

It is now possible to provide an alternate definition of the gripper domain operators initially presented in Section 3.1.

```
#operator pick(ball, gripper) :at t
 :precond [t] loc(ball) ≙ loc(robot) ∧ free(gripper) ∧ ¬∃ gripper' [ carry(ball, gripper') ]
 :effects [+1] carry(ball, gripper) := true,
          [+1] free(gripper) := false

#operator drop(ball, gripper) :at t
 :precond [t] carry(ball, gripper)
```

```
 :effects [+1] carry(ball, gripper) := false,
          [+1] free(gripper) := true
#operator move-to(room) :at t
 :precond [t] loc(robot) ≠ room
 :context
  :effects [+1] loc(robot) := room
 :context :forall ball :condition [t] ∃ gripper [ carry(ball, gripper) ]
  :effects [+1] loc(ball) := room
```

The translation of these operators is equivalent to the $\mathcal{L}$(ND) operator definitions in Section 2.2.

3.7. TALplanner: An Abstract View

As shown in Figure 2, TALplanner takes a TAL goal narrative $\mathcal{GN}$ in $\mathcal{L}(\mathrm{ND})^*$ as input. The planner translates the goal narrative into a suitable internal representation and then searches for a plan, an executable operator sequence satisfying the goal statement and the control rules. If a plan exists, the result is a new plan narrative in $\mathcal{L}$(ND) where goals and control rules have been removed and a set of action occurrences (corresponding to plan steps) has been added.

More formally, let $\mathcal{GN}_{\mathsf{goal}}$, $\mathcal{GN}_{\mathsf{control}}$ and $\mathcal{GN}_{\mathsf{mcontrol}}$ be the sets of goal statements, TAL control statements and modal control statements in $\mathcal{GN}$, respectively. These statement classes are only used during the plan synthesis process and should not be included in the final plan narrative. Thus, assuming the planner succeeds in finding a plan, the plan narrative $\mathcal{N}_p$ is the $\mathcal{L}$(ND) narrative $(\mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})) \cup \mathcal{GN}_{\mathsf{occ}}$, where $\mathcal{GN}_{\mathsf{occ}}$ is the set of action occurrences (plan steps) generated by the planning algorithm.

Observe that one can use standard inference techniques or techniques specific to TAL to reason about both the input goal narrative and the output plan narrative. The plan narrative is always guaranteed to entail the state goal and the domain dependent control formulas in $\mathcal{GN}_{\mathsf{control}}$.

4. Sequential TALplanner

In previous papers [15,30], two separate TALplanner algorithms for generating sequential plans were proposed: TALplan/modal using emulated modal control formulas and a progression algorithm, and TALplan/non-modal using TAL control formulas and formula evaluation. Since experience has shown that both approaches have advantages and disadvantages, a unified sequential version of TALplanner has recently been developed. The new planner is called TALplan/seq and allows the use of both types of control rules.

The constraints placed on goal narratives by the current version of TALplan/seq will now be presented, followed by a definition of sequential plans and a description of the forward-chaining search tree induced by this definition.

The unified sequential TALplanner algorithm will then be described. This algorithm uses both formula progression and a pruning constraint extraction algorithm, discussed in the final two subsections.

4.1. Constraints on Goal Narratives

For this version of TALplan/seq, the following restrictions are placed on the goal narratives accepted as input:

- The initial state must be completely specified.
- Operators must be deterministic.
- Dependency constraints (and side effects of actions) are not allowed.
- Domain constraints must not relate to multiple states. More formally, domain constraints must be pure state constraints of the form $\forall t.[t]\ \phi(t)$, where the formula $\phi(t)$ is a fixed fluent formula that only refers to fluents at time t.
- Fluents must be persistent or dynamic, as defined in Section 2.1. Each dynamic fluent must be associated with a domain constraint that uniquely determines the value of that fluent. This provides the possibility of using defined predicates (essentially, boolean state variables defined in terms of formulas).

However, context-dependent operators and operators with duration and internal state changes within the execution of the operator are allowed, permitting TALplanner to handle full ADL-style operators with some additional extensions. Arbitrary (disjunctive and existential) goals are also permitted.

4.2. Sequential Plans

TALplanner takes a TAL goal narrative $\mathcal{GN}$ as input, and generates a new plan narrative $\mathcal{N}_p$ where a set of timed action occurrences has been added. Internally, however, TALplanner is basically a forward chaining planner, searching through the space of states reachable from the initial state. In order to be able to describe this search space more formally, a number of definitions are required.

Due to the use of actions with non-unit duration, plans cannot be just sequences of operators but must also contain timing information. This is provided as *action occurrences* or *timed operator instances* of the form $[\tau, \tau']\ o$, denoting the invocation of the operator instance o between times τ and τ', where $\tau < \tau'$.

An *executable operator sequence* is a tuple of timed operator instances with the following constraints. First, the empty tuple is an executable operator sequence. Second, given a sequence of n operators ending in $[t_{n-1}, t_n]\ o_n$, its successors are exactly those sequences adding one new timed operator instance $[t_n, t_{n+1}]\ o_{n+1}$ such that o_{n+1} is applicable at $[t_n, t_{n+1}]$ (where $t_n = 0$ if $n = 0$).

A *plan* is an executable operator sequence that satisfies all control formulas in $\mathcal{GN}_{\mathsf{control}}$ as well as the goal statements in $\mathcal{GN}_{\mathsf{goal}}$.

These definitions induce a search tree where the root is labeled with the empty operator sequence and the children of a node labeled with the sequence l are labeled with the successors of l. Clearly, this search tree must contain all plans. Therefore, a complete planner can be generated by using a complete search algorithm such as breadth first search or iterative deepening.

4.3. The Sequential TALplanner Algorithm

The following is a version of the sequential TALplanner algorithm using depth-first search with optional cycle checking. Naturally, the algorithm can easily be modified to use other search strategies.

Input: A goal narrative $\mathcal{GN}$.
Output: A plan narrative $\mathcal{N}_p$ which entails the goal and the non-modal control formulas.

1 **procedure** $\mathsf{TALplan/seq}(\mathcal{GN})$
2 $\gamma \leftarrow \bigwedge \mathcal{GN}_{\mathsf{goal}}$ *Conjunction of all goal statements*
3 $\mu \leftarrow \bigwedge \mathcal{GN}_{\mathsf{mcontrol}}$ *Conjunction of all modal control rules*
4 $\phi \leftarrow \bigwedge \mathcal{GN}_{\mathsf{control}}$ *Conjunction of all non-modal control rules*
5 $\pi \leftarrow \mathsf{generate\text{-}pruning\text{-}constraints}(\phi)$ *Generated pruning constraints*
6 $\mathsf{acc} \leftarrow \{\}$ *Visited (accepted) states for redundancy checking*
7 $\mathsf{Open} \leftarrow \langle\langle \mu, -1, 0, \mathcal{GN}\rangle\rangle$ *Stack (depth-first search)*
8 **while** $\mathsf{Open} \neq \langle\rangle$ **do**
9 $\langle \mu, \tau, \tau', \mathcal{GN}\rangle \leftarrow \mathbf{pop}(\mathsf{Open})$
10 $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})$
11 **if** $\mathit{Trans}^+(\mathcal{N} \cup \{t_{\mathsf{fixed}} = \tau'\}) \not\models \mathit{Trans}(\neg\pi)$ **then** *Check pruning constraints*
12 $\mu^+ \leftarrow \mathsf{Progress}(\mu, \tau+1, \tau'+1, \mathcal{N})$ *Progress modal control*
13 **if** $\mu^+ \neq \mathit{false}$ **then** *Check modal control*
14 $\mathsf{state} \leftarrow$ (state at time τ' for $\mathcal{N}$)
15 **if** (redundancy checking disabled) **or** $\mathsf{state} \notin \mathsf{acc}$ **then**
16 $\mathsf{acc} \leftarrow \mathsf{acc} \cup \{\mathsf{state}\}$
17 **if** $\mathsf{GoodPlan}(\mathcal{GN}, \tau')$ **then return** $\mathcal{N}$
18 **else** $\mathsf{Expand}(\mathcal{GN}, \tau', \mu^+, \mathsf{Open})$

Some explanations are in order. Each search node is associated with a modal control formula μ, which may be the constant true in the absence of modal control. Line 11 checks whether the negation of the non-modal pruning constraints π is entailed, in which case the search node should be pruned. If this is not the case, the planner then progresses the modal control formula μ through the new fixed state or states generated by the new action (the temporal interval $[\tau+1, \tau']$), in order to find a modal formula μ^+ that should hold at $\tau'+1$ (line 12). If $\mu^+ = \mathit{false}$, the search node should be pruned. Lines 14–16 perform redundancy checking. Line 17 checks whether a plan has been found, while line 18 pushes the successors of the current search node onto the stack using the Expand algorithm defined below.

4.3.1. What is a Good Plan?

The GoodPlan algorithm determines whether an operator sequence corresponding to a narrative $\mathcal{GN}$ is in fact a plan, where the $\overline{\tau}$ argument denotes the ending timepoint of the last operator instance added to the narrative $\mathcal{GN}$.

In the default implementation, GoodPlan ensures both that the state goal γ is entailed at time $\overline{\tau}$ and that the non-modal control formulas ϕ are entailed (recall that in order to emulate the semantics used for modal control formulas by TLPLAN, such formulas are only used for pruning purposes and are not considered to be part of the goal).

Different implementations could provide different criteria for whether a narrative satisfies a goal. For example, since TALplanner allows actions with duration and internal state, one may want to accept plans where a goal state is visited at some intermediate timepoint in the duration of an action, even if the final state does not satisfy the goal.

procedure $\mathsf{GoodPlan}(\mathcal{GN}, \overline{\tau})$
$\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})$
$\gamma \leftarrow \bigwedge \mathcal{GN}_{\mathsf{goal}}$ *Conjunction of all goal statements*
$\phi \leftarrow \bigwedge \mathcal{GN}_{\mathsf{control}}$ *Conjunction of all non-modal control rules*
if $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}(\phi \wedge [\overline{\tau}]\ \gamma)$ **return** *true*
else return *false*

4.3.2. What is a Successor?

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.

procedure $\mathsf{Expand}(\mathcal{GN}, s, \mu, \mathsf{Open})$
$\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})$
for all $\mathsf{a}(\overline{x}) \in \mathsf{ActionTypes}(\mathcal{N})$ **do** *For all action types (operators)*
  for all $[s,t]\ \mathsf{a}(\overline{c}) \in \mathsf{Instantiate}(s, \mathsf{a}(\overline{x}))$ **do** *For all instantiations*
    if $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}([s]\ \mathsf{precond}(\mathsf{a}(\overline{c})))$ **then** *If applicable at s*
      $\mathsf{Open} \leftarrow \mathsf{Open} \cup \{\langle \mu, s, t, \mathcal{GN} \cup \{[s,t]\ \mathsf{a}(\overline{c})\}\rangle\}$ *Add it to Open*

Note that in the implementation, a lazy version of Expand is used where successors are only generated as needed.

4.4. Extracting Pruning Constraints

As discussed in Section 3.3.2, the non-modal control rules defined by the domain designer provide constraints that must be satisfied by any valid plan. However, in order to prune the search tree, TALplanner requires constraints that must be satisfied by any *prefix* of a valid plan. In a pre-processing phase, TALplanner analyzes the control rules and automatically generates a set of such *pruning constraints*. If an operator sequence violates a pruning constraint, the corresponding search node and its descendants can immediately be discarded.

The process of extracting and analyzing pruning constraints can be quite complex for certain classes of control formulas, and also depends on the exact restrictions placed on goal narratives and on whether sequential or concurrent plans are being generated. A complete and formal definition of the algorithms being used for different formula classes and different restrictions is outside the scope of this article, and will be presented in a forthcoming paper. Here, an illustrative example will instead be provided for a very common formula class under the assumption of sequential planning with the restrictions given in Section 4.1.

Consider again the gripper control rules presented in Section 3.5. These rules are of the form $\forall t.\phi(t)$, where $\phi(t)$ only depends on states in the interval $[t, t+c]$ for some constant c (for these three rules, $c = 1$).

Suppose an operator sequence is considered where $\phi(t)$ is false for some timepoint t (so that the control formula $\forall t.\phi(t)$ is violated), and suppose it can be guaranteed that the states in $[t, t+c]$ are *fixed*, that is, that they will remain unchanged in all descendants. Clearly, $\phi(t)$ will remain false in all descendants, and the control rule will always remain violated. Such an operator sequence cannot be the prefix of a plan, and can immediately be pruned.

A new temporal constant t_{fixed} is introduced that at any point in the search process denotes the ending timepoint of the current operator sequence. Clearly, since sequential plans are generated, all states up to and including t_{fixed} are fixed: New operators can be added, but they cannot affect the "past". The pruning constraint $\forall t.t + c \leq t_{\mathsf{fixed}} \rightarrow \phi(t)$ is generated and must be satisfied by any plan prefix. For example, the third gripper control rule, "only pick up relevant balls", generates a pruning constraint of the form

$$\forall t.t + 1 \leq t_{\mathsf{fixed}} \rightarrow \forall b.[t]\ \neg\exists g[\mathsf{carry}(b, g)] \land \psi(t, b) \rightarrow [t+1]\ \neg\exists g[\mathsf{carry}(b, g)]$$

where $\psi(t, b) \stackrel{\text{def}}{=} \neg\exists r[\mathsf{goal}(\mathsf{loc}(b) \hat{=} r) \land [t]\ \mathsf{loc}(b) \hat{\neq} r]$ expresses the fact that there is no reason for moving the ball b to another room.

4.4.1. Operator-Independent Pruning Constraint Analysis

TALplanner only generates successors for operator sequences that satisfy all pruning constraints. This fact can be used in order to optimize the evaluation of pruning constraints in the successors.

Consider once again a pruning constraint of the form $\forall t.t+c \leq t_{\mathsf{fixed}} \rightarrow \phi(t)$. If the planner has just added the timed operator instance $[\tau, \tau']\ o$, then t_{fixed} has just increased from τ to τ'. Thus, it is already ensured in the predecessor that $\forall t.t + c \leq \tau \rightarrow \phi(t)$, and since this condition only depended on fixed states, it remains true in the successor. In order to prove the constraint $\forall t.t+c \leq \tau' \rightarrow \phi(t)$ in the successor, only $\forall t.\tau < t + c \leq \tau' \rightarrow \phi(t)$ needs to be proved. By using this simpler pruning constraint, the planner avoids re-evaluating conditions that were

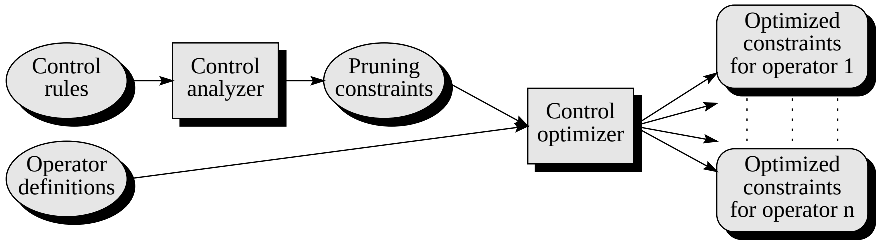

Figure 5. Optimization

already checked in a predecessor. This reduces the gripper pruning constraint above to the following formula:

$$\forall t.\tau < t+1 \leq \tau' \rightarrow \forall b.[t]\ \neg\exists g[\mathsf{carry}(b,g)] \wedge \psi(t,b) \rightarrow [t+1]\ \neg\exists g[\mathsf{carry}(b,g)]$$

4.4.2. Operator-Dependent Pruning Constraint Analysis

Pruning constraints can also be analyzed separately for each operator type in a domain, under the assumption that some instance of that operator has just been invoked (see also Figure 5). Although this is done during pre-processing, when exact arguments and invocation timepoints are not yet known, the operator definitions contain considerable information about the states in which the pruning constraints will eventually be evaluated: The preconditions must hold (or the operator would not have been invoked), the effects must have taken place, and since sequential planning is assumed and the use of dependency constraints is not allowed, no persistent state variables can have changed during the execution interval $[\tau,\tau']$ except those explicitly specified by the operator effects.

Let α denote the information present in the definition of an operator o. When proving a pruning constraint β immediately after adding an instance of o to an operator sequence, α is already known. If it is possible to find a simpler condition γ such that $\alpha \wedge \beta \equiv \alpha \wedge \gamma$, then proving γ is sufficient.

Consider the reduced gripper pruning constraint presented above under the assumption that $[\tau,\tau']$ $\mathsf{pick}(b_1,g_1)$ has just been invoked for some ball b_1 and some gripper g_1. Since the pick operator always uses one unit of time, it must be the case that $\tau' = \tau+1$. Simplifying the pruning constraint accordingly yields:

$$\forall b.[\tau]\ \neg\exists g[\mathsf{carry}(b,g)] \wedge \psi(\tau,b) \rightarrow [\tau+1]\ \neg\exists g[\mathsf{carry}(b,g)]$$

The quantified formula can only be false if $\mathsf{carry}(b,g)$ is false at τ but true at $\tau+1$. But this can only occur for $\mathsf{carry}(b_1,g_1)$, since no other instance of carry is altered by $\mathsf{pick}(b_1,g_1)$. Thus, the constraint can be simplified as follows:

$$[\tau]\ \neg\mathsf{carry}(b_1,g_1) \wedge \psi(\tau,b) \rightarrow [\tau+1]\ \neg\mathsf{carry}(b_1,g_1)$$

Due to the preconditions of $[\tau,\tau']$ $\mathsf{pick}(b_1,g_1)$, the formula $[\tau]$ $\neg\mathsf{carry}(b_1,g_1)$ must be false, and due to the direct effects, $[\tau+1]$ $\neg\mathsf{carry}(b_1,g_1)$ must be false. After

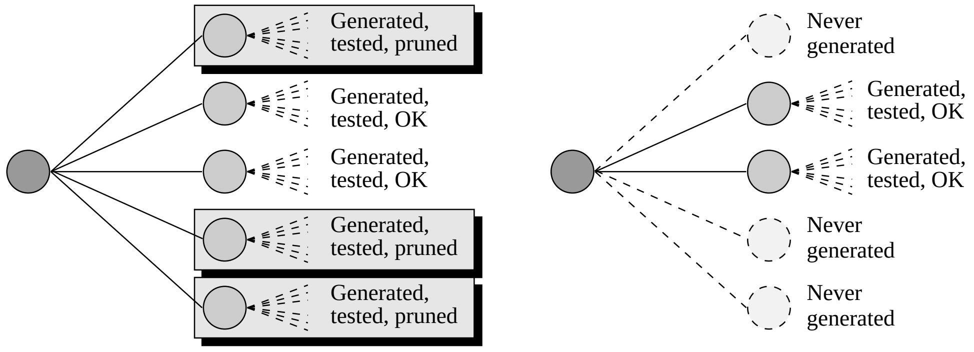

Figure 6. Fewer States Generated using Precondition Control

simplification, the remaining formula is $\neg\psi(\tau, b_1)$, where b_1 is bound to the actual argument of the pick operator just having been invoked.

In some cases, this process completely removes pruning constraints for certain operators. For example, TALplanner automatically detects that the constraint "do not move when there are objects to drop" cannot be violated by the drop or pick actions.

4.4.3. Generating Precondition Control

Often, the operator-specific pruning constraint analysis results in simplified pruning constraints where some conjuncts – or even the entire constraints – only refer to the invocation state of the operator. This was the case for the gripper example, where $\neg\psi(\tau, b_1)$ only refers to time τ. TALplanner moves such conjuncts into the precondition of the operator, automatically generating so called *precondition control* [7].

This is possibly one of the reasons why TALplanner is significantly faster than TLPLAN. In order to check whether a control rule is violated in a successor, TLPLAN (or TALplanner with modal control rules) must apply an action, generating one or more new states, and then progress the control rule through the new states (the left hand side of Figure 6). When a pruning constraint can be reduced to precondition control, however, it can be checked before the successor states are generated (the right hand side of Figure 6).

4.5. Progressing a Modal Control Formula

The Progress algorithm (abbreviated P below) will now be defined. Let ϕ be a modal control formula that should hold at τ in $\mathcal{N}$. Then, $\mathsf{Progress}(\phi, \tau, \tau', \mathcal{N})$ will be a modal control formula that should hold at τ' in $\mathcal{N}$. Usually, τ' should be the timepoint immediately after the last fixed state.

procedure $\mathsf{Progress}(\phi, \tau, \tau', \mathcal{N})$
if $\tau = \tau'$ **return** ϕ

if $\phi = f(\overline{x}) \hat{=} v$
  if $\mathit{Trans}^{+}(\mathcal{N}) \models \mathit{Trans}([\tau]\ \phi)$ **return** *true* **else return** *false*
if ϕ contains no temporal modalities
  if $\mathit{Trans}^{+}(\mathcal{N}) \models \mathit{Trans}(\phi)$ **return** *true* **else return** *false*
if $\phi = \neg\phi_1$ **return** $\neg\mathsf{P}(\phi_1, \tau, \tau', \mathcal{N})$
if $\phi = \phi_1 \otimes \phi_2$ **return** $\mathsf{P}(\phi_1, \tau, \tau', \mathcal{N}) \otimes \mathsf{P}(\phi_2, \tau, \tau', \mathcal{N})$
if $\phi = \phi_1 \mathsf{U}_{[\tau_1, \tau_2]} \phi_2$
  if $\tau_1 < 0 \wedge \tau_2 < 0$ **return** *false*
  elsif $0 \in [\tau_1, \tau_2]$
    return $\mathsf{P}(\phi_2, \tau, \tau', \mathcal{N}) \vee (\mathsf{P}(\phi_1, \tau, \tau', \mathcal{N}) \wedge \mathsf{P}(\phi_1 \mathsf{U}_{[\tau_1-1, \tau_2-1]} \phi_2, \tau+1, \tau', \mathcal{N}))$
  else return $\mathsf{P}(\phi_1, \tau, \tau', \mathcal{N}) \wedge \mathsf{P}(\phi_1 \mathsf{U}_{[\tau_1-1, \tau_2-1]} \phi_2, \tau+1, \tau', \mathcal{N})$

This algorithm is similar to the one used by Bacchus and Kabanza in [10]. However, there are some differences in the progression of the U operator, since TAL actions with duration can have internal state: Unlike the first-order version of MITL (metric interval temporal logic [3,4]) used in [10], there can be a sequence of state changes between the initiation state and the effect state of a TAL action.

Since $\bigcirc$, $\Box$ and $\Diamond$ can be defined in terms of U, the algorithm above suffices, although handling the additional modal operators explicitly might be useful for clarity and efficiency:

if $\phi = \Diamond_{[\tau_1, \tau_2]}\ \phi_1$
  if $[\tau_1, \tau_2] < 0$ **return** *false*
  elsif $0 \in [\tau_1, \tau_2]$ **return** $\mathsf{P}(\phi_1, \tau, \tau', \mathcal{GN}) \vee \mathsf{P}(\Diamond_{[\tau_1-1, \tau_2-1]} \phi_1, \tau+1, \tau', \mathcal{GN})$
  else return $\mathsf{P}(\Diamond_{[\tau_1-1, \tau_2-1]} \phi_1, \tau+1, \tau', \mathcal{GN})$
if $\phi = \Box_{[\tau_1, \tau_2]}\ \phi_1$
  if $[\tau_1, \tau_2] < 0$ **return** *false*
  elsif $0 \in [\tau_1, \tau_2]$ **return** $\mathsf{P}(\phi_1, \tau, \tau', \mathcal{GN}) \wedge \mathsf{P}(\Box_{[\tau_1-1, \tau_2-1]} \phi_1, \tau+1, \tau', \mathcal{GN})$
  else return $\mathsf{P}(\Box_{[\tau_1-1, \tau_2-1]} \phi_1, \tau+1, \tau', \mathcal{GN})$
if $\phi = \bigcirc \phi_1$
  if $\tau + 1 = \tau'$ **return** ϕ_1
  else return $\mathsf{P}(\phi_1, \tau+1, \tau', \mathcal{GN})$

The result of Progress is simplified using the rules $\neg \mathit{false} = \mathit{true}$, $(\mathit{false} \wedge \alpha) = (\alpha \wedge \mathit{false}) = \mathit{false}$, $(\mathit{false} \vee \alpha) = (\alpha \vee \mathit{false}) = \alpha$, $\neg \mathit{true} = \mathit{false}$, $(\mathit{true} \wedge \alpha) = (\alpha \wedge \mathit{true}) = \alpha$, and $(\mathit{true} \vee \alpha) = (\alpha \vee \mathit{true}) = \mathit{true}$.

5. Extending TALplanner for Concurrency

The sequential version of TALplanner will now be extended for generating concurrent plans. The concurrent planner will place the same constraints on goal narratives as the sequential planner (Section 4.1).

5.1. Concurrent Plans

Although the planner described in this section generates concurrent plans, a plan is still viewed as a sequence of timed operator sequences.

For concurrent TALplanner, an *executable operator sequence* is a tuple of timed operator instances. The empty tuple is an executable operator sequence, and given a sequence p of n operators ending in $[s_n, t_n]\ o_n$, its successors are those sequences adding one new timed operator instance $[s_{n+1}, t_{n+1}]\ o_{n+1}$ satisfying the following four constraints.

First, the new operator o_{n+1} must be applicable at $[s_{n+1}, t_{n+1}]$. This implies that its preconditions are satisfied, that its effects are not internally inconsistent, and that its effects do not contradict the effects of the operator instances already present in the sequence.

Second, the new operator should not be invoked before any of the operators already existing in the sequence. Therefore, it is required that $s_n \leq s_{n+1}$. This guarantees that 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.

Third, an upper bound will be placed on the invocation timepoint s_{n+1}. Let $\overline{\tau}$ be the maximum of all ending timepoints t_i of all actions in p. The states from s_n up to $\overline{\tau}$ may all be different, but since nothing can change after $\overline{\tau}$, successors with $s_{n+1} > \overline{\tau}$ are not considered. Thus, it must be the case that $s_{n+1} \leq \overline{\tau}$ (note that it is possible that $\overline{\tau} > t_n$, as in the operator sequence $\langle [0,7]\ o_1; [0,3]\ o_2 \rangle$).

Fourth, there is an additional difficulty associated with successors where $s_{n+1} = s_n$ (that is, where the new action has the same invocation timepoint as an existing action): 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, the existence of a total order $\succ$ on operator instances will be assumed, and if $s_{n+1} = s_n$, it must be the case that $o \succ o_n$.

Like the definition of sequential plans, 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. The algorithms presented below use depth-first search and rely on control rules to prune nodes that would not lead closer to the goal.

5.2. The Concurrent TALplanner Algorithm

The concurrent TALplanner algorithm will now be defined. The main difference between this algorithm and sequential TALplanner is that successors may be added at any timepoint between τ and $\overline{\tau}$, inclusive. Note that successors are pushed on the stack in reverse temporal order, since invoking operators as early as possible is preferred. The case where all states up to $\overline{\tau}$ are fixed must be treated separately (lines 11–18): Only here can a plan possibly be found, and only here can redundancy checking be performed.

Input: A goal narrative $\mathcal{GN}$.

Output: A plan narrative $\mathcal{N}_p$ which entails the goal and the non-modal control formulas.

procedure $\mathsf{TALplan/conc}(\mathcal{GN})$
$\gamma \leftarrow \bigwedge \mathcal{GN}_{\mathsf{goal}}$ *Conjunction of all goal statements*
$\mu \leftarrow \bigwedge \mathcal{GN}_{\mathsf{mcontrol}}$ *Conjunction of all modal control rules*
$\phi \leftarrow \bigwedge \mathcal{GN}_{\mathsf{control}}$ *Conjunction of all non-modal control rules*
$\pi \leftarrow \mathsf{generate\text{-}pruning\text{-}constraints}(\phi)$ *Generated pruning constraints*
$\mathsf{acc} \leftarrow \{\}$ *Visited (accepted) states for redundancy checking*
$\mathsf{Open} \leftarrow \langle\langle \mu, -1, 0, 0, \mathcal{GN}\rangle\rangle$ *Stack (depth-first search)*
while $\mathsf{Open} \neq \langle\rangle$ **do**
  $\langle \mu, \tau, \tau', \overline{\tau}, \mathcal{GN}\rangle \leftarrow$ **pop**(Open)
  $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})$
  if $\mathit{Trans}^+(\mathcal{N} \cup \{t_{\mathsf{fixed}} = \overline{\tau}\}) \not\models \mathit{Trans}(\neg\pi)$ **then** *Check pruning constraints*
    $\mu^+ \leftarrow \mathsf{Progress}(\mu, \tau+1, \overline{\tau}+1, \mathcal{N})$ *Progress modal control*
    if $\mu^+ \neq \mathit{false}$ **then** *Check modal control*
      $\mathsf{state} \leftarrow$ (state at time $\overline{\tau}$ for $\mathcal{N}$)
      if (redundancy checking disabled) **or** $\neg\exists \mathsf{s}' \in \mathsf{acc}$: better-or-equal[5]$(\mathsf{s}', \mathsf{state})$ **then**
        $\mathsf{acc} \leftarrow \mathsf{acc} \cup \{\mathsf{state}\}$
        if $\mathsf{GoodPlan}(\mathcal{GN}, \overline{\tau})$ **then return** $\mathcal{N}$
        else $\mathsf{Expand}(\mathcal{GN}, \tau, \overline{\tau}, \overline{\tau}, \mu^+, \mathsf{Open})$
  for s **from** $\overline{\tau} - 1$ **downto** τ **do**
    if $\mathit{Trans}^+(\mathcal{N} \cup \{t_{\mathsf{fixed}} = s\}) \not\models \mathit{Trans}(\neg\pi)$ **then** *Check pruning constraints*
      $\mu^+ \leftarrow \mathsf{Progress}(\mu, \tau+1, s+1, \mathcal{N})$ *Progress modal control*
      if $\mu^+ \neq \mathit{false}$ **then** *Check modal control*
        $\mathsf{Expand}(\mathcal{GN}, \tau, s, \overline{\tau}, \mu^+, \mathsf{Open})$

5.2.1. What is a Successor?

Although the sequential version of GoodPlan can still be used, the sequential version of the Expand algorithm must be modified somewhat to prevent the search tree from containing redundant pairs of plan prefixes (line 5). Also, $\overline{\tau}$ must be updated and stored in each search node ($\mathsf{max}(t, \overline{\tau})$ in line 7).

1 **procedure** $\mathsf{Expand}(\mathcal{GN}, \tau, s, \overline{\tau}, \mu, \mathsf{Open})$
2 $\mathcal{N} \leftarrow \mathcal{GN} \setminus (\mathcal{GN}_{\mathsf{goal}} \cup \mathcal{GN}_{\mathsf{control}} \cup \mathcal{GN}_{\mathsf{mcontrol}})$
3 **for all** $\mathsf{a}(\overline{x}) \in \mathsf{ActionTypes}(\mathcal{N})$ **do** *For all action types (operators)*
4   **for all** $[s,t]\ \mathsf{a}(\overline{c}) \in \mathsf{Instantiate}(s, \mathsf{a}(\overline{x}))$ **do** *For all instantiations*
5   **if** $s \neq \tau$ **or** $\mathsf{a}(\overline{c}) \succ \mathsf{lastact}(\mathcal{N})$ **then** *If not redundant ordering*
6   **if** $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}([s]\ \mathsf{precond}(\mathsf{a}(\overline{c})))$ **then** *If applicable at s*
7   **push** $\langle \mu, s, t, \mathsf{max}(t, \overline{\tau}), \mathcal{GN} \cup \{[s,t]\ \mathsf{a}(\overline{c})\}\rangle$ **on** Open *Add it to Open*

5.3. Extracting Pruning Constraints

The algorithms for generating and analyzing pruning constraints for sequential plans are based on the assumption that all states up to and including the

[5] The better-or-equal relation is explained in Section 6.1.

maximum effect timepoint in an operator sequence will remain unmodified in every successor. Although this does not hold for concurrent plans, there is a weaker condition that TALplanner can take advantage of in order to strengthen its pruning capabilities.

Let $p = \langle [s_1, t_1]\ o_1, \ldots, [s_n, t_n]\ o_n \rangle$ be an executable operator sequence. If sequential plans are being generated, all states up to and including time t_n are fixed, while states after t_n may be modified in successors. If concurrent plans are being generated, all states up to and including s_n are fixed. States after s_n may be modified in successors, *except* for those state variables that have already been explicitly assigned a value at timepoints within the interval $[s_n + 1, t_n]$. By considering this during the generation and analysis of pruning constraints for concurrent plans, TALplanner can detect control violations earlier than would have been possible with a pure progression algorithm.

6. Extending TALplanner with Resources

Many planning domains involve the use of limited resources which can be consumed, produced, borrowed, or used in various other ways. For example, vehicles such as trucks and airplanes can have limited carrying capacities and a limited amount of fuel available.

For sequential planning, such properties can usually be modeled quite easily in TALplanner by using plain action effects updating the values of non-boolean fluents: A drive action would decrease the amount of fuel available by assigning a new value to the fuel fluent. However, the use of concurrency adds considerably to the complexity of modeling resources in this manner, mainly due to the possibility of multiple actions using the same resource in the same state, and even for sequential planning, adding explicit built-in support for resources often facilitates the writing of domain definitions. Therefore, three new extensions to TALplanner are provided: Resource declaration statements, an extended form of operator statements, and a set of fluent macros for use with resources.

Resource declaration statements have the following form:

$$\mathsf{resource}\ \mathsf{r}(\overline{x})\ \mathsf{:domain}\ \mathit{domain}$$

For example, in a problem domain where vehicles have limited space, a space resource can be declared as follows:

$$\mathsf{resource\ space(vehicle)\ :domain\ integer}$$

Operator statements are extended in order to provide a structured way of declaring the resource usage of an operator.

```
context ::= ":context"
            [ ":forall" valueVariable ( "," valueVariable )* ]
            [ ":condition" logicformula ]
```

```
            [ ":resources" resUse ( "," resUse )* ]
            ":effects" effect ( "," effect )*

resUse  ::= "[" "+" delay   [ "," "+" enddelay ]  "]"
            ( ":produce" | ":consume" |
                ":borrow" | ":borrow-nonex" |  ":assign" )
            resourceFluent [ "(" valueTerm ( "," valueTerm )* ")" ]
            ":amount" amountExpression
```

Like plain effects, resource effects can refer to a single timepoint (a strictly positive delay from the invocation timepoint) or an interval of time (also strictly positive: no operator is allowed to use resources in its invocation state). However, while plain effects always specify a new value to be assigned to a fluent, there are five different kinds of resource effect:

- :produce produces a certain amount of the given resource. What has been produced in a certain state cannot be used (consumed or borrowed) in the same state.
- :consume consumes a certain amount of the given resource. Consumption in one state leaves room for more production in the following state.
- :borrow borrows a certain amount of resources at the specified timepoint or during the specified interval. After the specified timepoint or interval, the resources are returned, and are immediately ready to be borrowed or consumed.
- :borrow-nonex is similar to borrow. However, the resources are borrowed non-exclusively, so that multiple concurrent instances of :borrow-nonex can share the same resources.
- :assign assigns a new value to the resource. This effect is incompatible with the other effects in the sense that it is impossible to borrow, produce or consume some units of a certain resource and to assign a new value to the same resource at the same timepoint. TALplanner ensures that plans satisfy this constraint by automatically generating a suitable control rule for each resource.

Each resource has a number of different aspects modeled as fluent macros.

First, in any state, there is an *initial* amount of resources – the amount that would be available if no resource effects took place in that state. Given a resource res, this is modeled as a fluent macro init(res).

Second, given a state and a resource res, certain amounts of that resource have been produced, consumed, borrowed non-exclusively, and borrowed exclusively. These amounts may arise from cumulative effects of a number of concurrent operators, and can be referred to using the fluent macros produced(res), consumed(res), borrowed-nonex(res), and borrowed(res), respectively.

Third, it is often useful to be able to refer to the amount of resources actually available for consumption in any given state. This amount can be referred to as available(res), and is equivalent to init(res) − consumed(res) − borrowed-nonex(res) −

$\mathsf{borrowed(res)}$ (recall that resources produced in a state are not available for consumption in the same state). Similarly, one can refer to the amount of resources transferred to the next state as $\mathsf{transferred(res)}$; this is equivalent to $\mathsf{init(res)} + \mathsf{produced(res)} - \mathsf{consumed(res)}$.

Finally, the minimum and maximum amounts allowed can be specified using the macros $\mathsf{minimum(res)}$ and $\mathsf{maximum(res)}$. In states where a resource has been explicitly assigned a new value, it is sufficient to ensure that the new value is within the allowed range. When no assignment has taken place, however, there may be a number of concurrent resource effects affecting the same resource. When enforcing resource constraints, TALplanner assumes that multiple resource effects might not in fact occur exactly simultaneously in the real world even though they are modeled as taking place in the same state, and therefore takes the pessimistic view. For the $\mathsf{minimum}$ constraint, this entails the assumption that all consumption in any given state might take place before any production, leading to the constraint that $\mathsf{init(res)} - \mathsf{consumed(res)} - \mathsf{borrowed\text{-}nonex(res)} - \mathsf{borrowed(res)} \geq \mathsf{minimum(res)}$ in all states, or, equivalently, that $\mathsf{available(res)} \geq \mathsf{minimum(res)}$ in all states. Similarly, to ensure that the amount of resources never exceeds the specified maximum regardless of the actual order between production and consumption within a state, the planner requires that $\mathsf{init(res)} + \mathsf{produced(res)} \leq \mathsf{maximum(res)}$ at all timepoints.

For every resource, it is necessary to specify the initial, minimum and maximum values in the initial state. For example, the fact that it is possible to have between 0 and 60 units of fuel, and that 30 units are available in the initial state, can be expressed as follows:

$$[0]\ \mathsf{init(fuel)} \,\hat{=}\, 30 \wedge \mathsf{minimum(fuel)} \,\hat{=}\, 0 \wedge \mathsf{maximum(fuel)} \,\hat{=}\, 60$$

Like all aspects of TALplanner, the formal semantics of resources is defined in terms of translations into $\mathcal{L}$(FL). Each resource definition gives rise to a number of TAL fluents, some of which are dynamic (non-inert), while each resource effect is translated into an ordinary operator effect. At any given timepoint, domain constraints determine the values of the non-inert resource fluents associated with each resource – the total amount produced, consumed, borrowed, etc. The implicit condition that the amount of any resource should never be outside its allowed range of values is translated into a set of control rules. The exact translation is straightforward, and will therefore be omitted and presented it in a forthcoming technical report.

Note that since resource macros can be used in control rules, one is not limited to simple minimum/maximum constraints on resources. It is easy to state that no more than 5.5 units of fuel can be consumed at any given timepoint:

$$\forall t. value(t, \mathsf{consumed(fuel)}) \leq 5.5$$

One can also state that it is impossible to produce and consume units of the same resource at the same time, that equal amounts of two resources must always be available, or even a complex constraint such that whenever some condition ϕ holds, a certain resource may not be consumed during the following 6 timepoints:

$$\forall t.[t]\ \mathsf{consumed}(\mathsf{res}) \hat{=} 0 \vee \mathsf{produced}(\mathsf{res}) \hat{=} 0$$
$$\forall t.value(t, \mathsf{available}(\mathsf{res}_1)) = value(t, \mathsf{available}(\mathsf{res}_2))$$
$$\forall t.\ [t]\ \phi \rightarrow [t+1, t+6]\ \mathsf{consumed}(\mathsf{res}) \hat{=} 0$$

6.1. Resources and Redundancy Checking

When resources are involved, ordinary cycle checking is generally too weak. Suppose, for example, that moving consumes fuel. An operator sequence where one moves from a to b and immediately back to a is intuitively redundant, but does not lead to a cycle, since less fuel is available after moving.

Therefore, each resource can be associated with a preference: more, less, or none. The syntax for resource declarations is extended accordingly:

resource r($\overline{x}$) :domain *domain* [:preference (:more | :less | :none)]

This induces a partial order on states, better-or-equal, used in TALplan/conc: better-or-equal(s, s') holds iff (1) s and s' are equal wrt. ordinary fluents and resources with preference none, (2) for every resource with preference more, there is at least as much available in s as in s', and (3) for every resource with preference less, there is at least as much available in s' as in s. Thus, declaring the fuel resource with preference more allows TALplanner to prune action sequences such as the one mentioned above.

Since resources can be used not only in resource effects but also in preconditions, control rules, and goals, generating these preferences automatically can be quite complex for non-trivial domains, and is a topic for future research.

6.2. Concurrency and Resources in the Gripper Domain

The obvious use of resources and concurrency is of course the ability to model and solve problems that can not be represented in a more restrictive formalism such as STRIPS. Perhaps less obvious, but equally important, is that explicit representations of resources lead to more efficient encodings of some standard STRIPS problems.

Consider once more the gripper domain. In the case where the robot has more than one gripper, one difficulty for a planner is recognizing that the grippers are functionally identical; if a plan can not be completed using the left gripper to pick up a certain object, using the right gripper instead will not fix the problem (this was pointed out in [18]). This difficulty can be avoided by modeling the grippers as a resource of bounded capacity. Picking up and dropping objects "consumes" and "produces" grippers, respectively. This also facilitates the definition of problem instances: Rather than having to name each gripper, it suffices to specify the number of grippers that are available.

The value domains obj for things (including the robot), ball for balls (a subtype of obj), and room for rooms are unchanged from the previous formalization of this domain. The integer domain is added and the gripper domain is no longer used. The fluent loc is unchanged, while the fluent free is replaced with the resource gripper and the fluent carry no longer has a gripper argument.

```
#domain obj :elements { ball1, ball2, ball3, robot }
#domain ball :parent obj :elements { ball1, ball2, ball3 }
#domain room :elements { roomA, roomB }
#resource gripper :domain integer

#feature loc(obj) :domain room
#feature carry(obj) :domain boolean

#obs [0] init(gripper) ≙ 2 ∧ minimum(gripper) ≙ 0 ∧ maximum(gripper) ≙ 2
```

Given these definitions, the gripper domain operators from Section 3.6 can be redefined as follows:

```
#operator pick(ball) :at t
 :precond [t] loc(ball) ≙ loc(robot) ∧ ¬ carry(ball)
 :resources [+1] :consume gripper :amount 1
 :effects [+1] carry(ball) := true

#operator drop(ball) :at t
 :precond [t] carry(ball)
 :resources [+1] :produce gripper :amount 1
 :effects [+1] carry(ball) := false

#operator move-to(room) :at t
 :precond [t] loc(robot) ≙̸ room
 :context
   :effects [+1] loc(robot) := room
 :context :forall ball :condition [t] carry(ball)
   :effects [+1] loc(ball) := room
```

7. Soundness and Completeness

The issues of soundness and completeness can only be considered relative to the exact restrictions being placed on a plan. This is especially true for planners that allow the specification of control rules, due to the question of whether or not such rules are part of the requirements that must be satisfied by a plan.

The sequential and concurrent TALplanner algorithms are sound given the definitions of a plan in Sections 4.2 and 5.1, respectively: If a narrative description $\mathcal{N}_p$ is returned given $\mathcal{GN}$ as input, then $\mathit{Trans}^{+}(\mathcal{N}_p) \models \mathit{Trans}(\bigwedge \mathcal{GN}_{\mathsf{control}} \wedge [\mathbf{t}]\ \bigwedge \mathcal{GN}_{\mathsf{goal}})$, where $\mathbf{t}$ is the end timepoint of the last action occurrence in $\mathcal{GN}_{\mathsf{occ}}$. (Recall that modal control rules are not viewed as part of the goal that must be satisfied by a final plan.)

Completeness is a somewhat more complex issue, and depends on whether or not cycle checking is used and whether or not modal control rules are present in the goal narrative.

7.1. Completeness Without Modal Control Rules

In the absence of modal control rules, TALplanner prunes a search node only if it can prove that all descendants must violate some non-modal control formula. Since non-modal control formulas must necessarily be satisfied in any valid plan, this pruning does not render TALplanner incomplete, as long as a complete search strategy such as breadth first or iterative deepening is used.

In most cases, however, TALplanner is run using depth-first search and cycle checking, as shown in the TALplan/seq and TALplan/conc algorithms above.

Since non-modal control rules in TALplanner are in effect temporally extended goals, it may be natural to view cycle checking as one very specific form of temporally extended goal: A valid plan must not contain a cycle. Given this perspective, TALplanner is complete even when cycle checking is activated.

However, if one takes a more traditional view of cycle checking, considering it merely an aid to the planner during the search process, then TALplanner is *not* complete in the presence of cycle checking, since it is technically possible to write control formulas that can only be satisfied by plans containing cycles. It should still be possible to provide restricted completeness results for specific types of control formulas, but we leave this for future research.

7.2. Completeness With Modal Control Rules

Emulated modal control rules in TALplanner are not considered part of the goal to be achieved. Therefore, careless use of modal control rules can easily render the planner incomplete, in the sense that the control rules specified by the domain designer could explicitly disallow all the paths that could lead to a goal state. For example, adding the modal control rule false will prevent TALplanner from finding any plan.[6]

8. Example Domains and Empirical Test Results

In this section, a number of example domains will be presented with empirical test results from running TALplanner on a number of problem instances within each domain. Although the main focus will be on a number of variations of the well-known logistics domain, the blocks world will also be considered in addition to a number of results from the AIPS 2000 Planning Competition [2]. TALplanner received the Outstanding Performance award in the hand-tailored (domain-dependent) track of the planning competition and first place in the ADL-plus-resources track of the Miconic 10 elevator control domain competition [27] sponsored by Schindler Lifts Ltd.

[6] The *non-modal* control rule false is viewed as a goal; if it is present, there *is* no plan.

Rather than showing the complete TALplanner domain definitions, only the operator definitions and control rules will be provided. The complete domain definitions will soon be available at http://www.ida.liu.se/∼jonkv/vital.html.

The test results from the AIPS 2000 Planning Competition were generated on a 500 MHz Pentium III machine with 1 GB of memory. Results will be presented for the first three domains used in the hand-tailored track of the competition: The blocks world, the logistics domain, and the schedule domain. The complete PDDL domain definitions and problem instances are available from the AIPS 2000 home page [2], together with the raw data files from which the graphs in this section were created. TALplanner's results are compared with SHOP [34] (a hierarchical task network (HTN) planner), System R [32] (a regression-based planner where domain-dependent control information is used to order subgoals, prune subgoals, and determine the way a subgoal is solved by regressing it to a new conjunctive goal), PbR [5,6] (Planning by Rewriting, a planner first generating a plan quickly and then optimizing the plan using domain-specific rewriting rules), and BDDPlan [37] (a planner using Binary Decision Diagrams to support reasoning in the Fluent Calculus, where a model checking algorithm is used to do an implicit breadth first search).

The remaining test results, for the logistics and blocks world domains, were generated by the authors. Test results for TLPLAN and TALplanner were generated on a 333 MHz Pentium II computer running Windows NT 4.0 SP3, with 256 MB of memory. In the logistics domain, TALplanner was also compared with SHOP [34], a hierarchical task network (HTN) planner, which ran on a 440 MHz Sun Ultra 10 with 256 MB of memory.

The TLPLAN tests were performed using the precompiled C version that can be downloaded from `http://www.cs.toronto.edu/~fbacchus/`. TALplanner is written in Java, and TALplanner 2.741 was used together with the Java Development Kit 1.2.2-001 and the HotSpot virtual machine (2.0rc2), both of which can be downloaded from http://java.sun.com. SHOP is written in Lisp, and SHOP 1.6.1 was used together with Allegro Common Lisp Enterprise Edition 5.0.

8.1. 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. In the worst case, each object may have to be transported by truck from its original location to an airport, by airplane to another airport, and then by truck from that airport to its final location, thus requiring up to nine actions per package when loading and unloading actions are included.

Three variations of this domain will be considered: The plain logistics domain, the logistics domain modified to allow the generation of concurrent plans by TALplan/conc, and an extended domain with variable duration of actions, where packages use variable amounts of space and resources are used to model the carrying capacities of vehicles.

8.1.1. The Standard Logistics Domain

In keeping with the standard formulation, the following fluents, resources and operators describe the logistics domain. Note that TAL is an order-sorted logic, and all variables are typed. The type loc (location) has the subtypes airport, city, while the type thing has the subtypes obj and vehicle, the latter of which has the subtypes truck and plane. The fluent city_of demonstrates the use of non-boolean state variables: At any timepoint, city_of(loc) $\hat{=}$ $city$ means that the location loc is in the city $city$.

```
#feature at(thing, loc), in(obj, vehicle) :domain boolean
#feature city_of(loc) :domain city

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

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

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

#operator unload-plane(obj, plane, loc) :at t
 :precond [t] in(obj, plane) ∧ at(plane, loc)
 :effects [+1] in(obj, plane) := 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
 :effects [+1] at(truck,loc1) := false, [+1] at(truck,loc2) := true

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

The following domain constraints define abbreviations used in the control rules and are similar to the defined predicates used by TLPLAN. For example, an object needs to be moved by truck from loc if its destination is in another city and the object is not already at an airport, or if the destination is another location in the same city.

```
#feature use-truck(obj, loc), unload-from-truck(obj, loc) :domain boolean :defined
#feature use-plane(obj, loc), unload-from-plane(obj, loc) :domain boolean :defined

#dom ∀t [ [t] use-truck(obj, loc) ↔
            ∃loc' [ goal(at(obj, loc')) ∧ city_of(loc) ≙̸ city_of(loc') ∧ ¬is_airport(loc) ] ∨
            ∃loc' [ goal(at(obj, loc')) ∧ city_of(loc) ≙ city_of(loc') ∧ loc ≠ loc') ]]

#dom ∀t [ ([t] unload-from-truck(obj, loc) ↔
            goal(at(obj, loc)) ∨
            is_airport(loc) ∧ ∃loc' [ goal(at(obj, loc')) ∧ city_of(loc) city_of(loc') ]]
```

```
#dom ∀t [ [t] use-plane(obj, loc) ↔ ∃loc' [ goal(at(obj, loc')) ∧ [t] city_of(loc) ≠̂ city_of(loc') ]]
#dom ∀t [ [t] unload-from-plane(obj, loc) ↔ ∃l' [ goal(at(obj, l')) ∧ [t] city_of(loc) =̂ city_of(l') ]]
```

The control rules below are inspired by, but not identical to, those used by TLPLAN. Briefly, an airplane should remain where it is until all packages that should be moved by the plane have been loaded. If it does move, it should move to a location where it needs to deliver packages, or to an airport where there are packages to be picked up and where no other airplane is present. 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 but are omitted here.

```
#control :name "airplanes-stay-until-everything-is-loaded"
∀t, plane, loc [ [t] at(plane, loc) ∧ ∃obj [ (at(obj, loc) ∧ use-plane(obj, loc)) ∨
                    at(plane, loc) ∧ ∃obj [ (in(obj, plane) ∧ unload-from-plane(obj, loc)) ] →
                 [t+1] at(plane, loc) ]
#control :name "airplanes-move-to-relevant-locations"
∀t, plane, loc [
  [t] at(plane, loc) →
  [t+1] at(plane, loc) ∨
       ∃loc', obj [ at(plane, loc') ∧ in(obj, plane) ∧ unload-from-plane(obj, loc') ] ∨
       ∃loc', obj [ at(plane, loc') ∧ at(obj, loc') ∧ use-plane(obj, loc') ]]
#control :name "only-load-when-necessary"
∀t, obj, plane, loc [ [t] ¬in(obj, plane) ∧ at(obj, loc) ∧ ¬use-plane(obj, loc) →
                      [t+1] ¬in(obj, plane) ]
#control :name "only-unload-when-necessary"
∀t, obj, plane, loc [ [t] in(obj, plane) ∧ at(plane, loc) ∧ ¬unload-from-plane(obj, loc) →
                      [t+1] in(obj, plane) ]
#control :name "objects-remain-at-destination-locations"
∀t, obj, loc [ [t] at(obj, loc) ∧ goal(at(obj, loc)) → [t+1] at(obj, loc) ]
```

8.1.2. Test Results

TALplanner and TLPLAN were tested and compared using the 30 logistics problems from the AIPS'98 planning competition [1]. See Table 1 for the complete list of results; times are in seconds. (The table also contains results for the extended domains that will be presented in the following sections.)

For two of the problems, 256 MB of memory was not sufficient for TLPLAN; the others required between 0.4 seconds and 17 hours to complete. TALplanner proved to be considerably more efficient: The longest plan (for problem 29) contained 330 operators and was created in approximately 0.3 seconds, while the most complex problem (problem 28) resulted in 274 operators and required 0.63 seconds. TALplanner allocated approximately 1.1 megabytes of memory, to which the runtime size of the Java environment (5–7 MB) must be added.

Table 1
Test Results for the Logistics Domain

	Standard Logistics Domain				Concurrency	Duration
			TALplan/seq		TALplan/conc	
	TLPLAN	SHOP	progression	evaluation	evaluation	evaluation
1	0.421	0.060	0.050	0.040	0.050	0.270
2	1.712	0.090	0.060	0.040	0.060	0.811
3	19.398	0.210	0.170	0.060	0.070	2.063
4	54.338	0.250	0.250	0.060	0.070	5.889
5	0.310	0.060	0.040	0.030	0.080	0.541
6	84.191	0.480	0.581	0.070	0.110	6.729
7	5.568	0.120	0.111	0.041	0.060	1.061
8	97.310	0.360	0.511	0.061	0.070	5.658
9	218.644	0.420	0.651	0.080	0.080	9.594
10	167.581	1.260	0.611	0.080	0.090	5.738
11	5.167	0.100	0.120	0.040	0.100	0.911
12	286.021	0.760	0.430	0.050	0.090	14.781
13	1073.263	1.160	1.041	0.070	0.130	16.524
14	802.824	1.000	3.284	0.100	0.100	6.800
15	24.675	0.260	0.340	0.080	0.080	1.512
16	168.002	0.390	0.381	0.060	0.090	10.004
17	90.460	0.210	0.822	0.100	0.080	2.895
18	4358.367	2.690	1.773	0.120	0.190	21.080
19	2685.021	0.960	1.713	0.110	0.180	18.466
20	3414.089	1.420	1.762	0.121	0.231	37.815
21	2102.643	0.860	1.402	0.100	0.161	39.436
22		11.570	20.149	0.330	0.490	71.402
23	116.798	0.810	0.871	0.080	0.090	2.434
24	695.780	0.320	0.761	0.070	0.110	39.096
25	11724.910	2.530	4.166	0.220	0.330	146.921
26	9976.946	19.200	12.338	0.230	0.440	83.960
27	14994.551	1.290	6.209	0.220	0.420	72.814
28		6.720	32.446	0.631	0.871	670.284
29	60874.834	15.510	15.422	0.291	0.431	34.550
30	14070.923	3.860	6.229	0.211	0.361	312.099
	Pentium II-333	UltraSparc-440	Pentium II-333			

In Nau et al. [34], the HTN planner SHOP was found to be considerably faster than TLPLAN. Nau et al. believe the most important reason to be the fact that SHOP in effect allows the user to design a planning algorithm, rather than prune a search space, and that SHOP's use of problem reduction can be more efficient than the state space search used by TLPLAN. However, even though the SHOP results in Table 1 were generated using a newer, considerably faster version of SHOP than the one used in [34], TALplanner is still faster by almost two orders of magnitude for some of the larger problem instances, despite running on a slower computer.

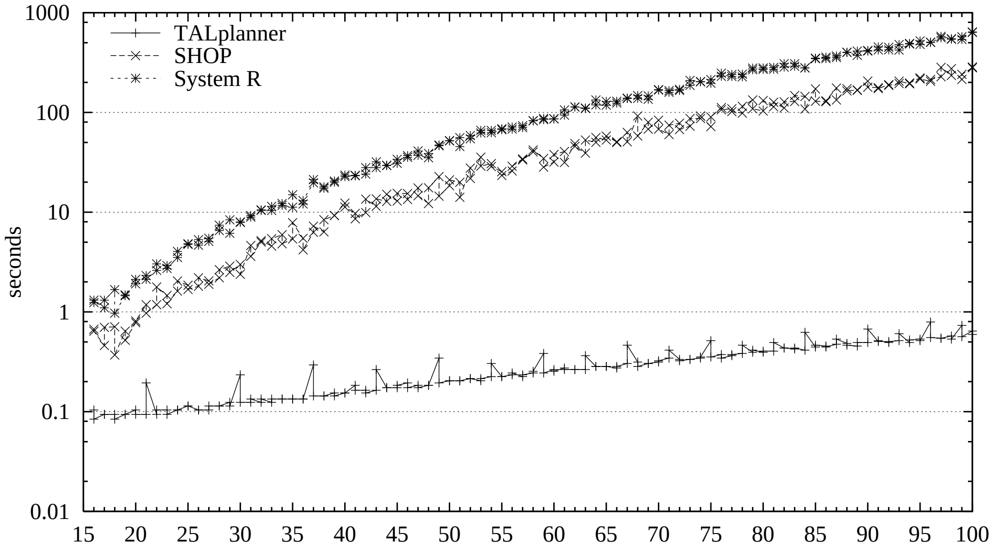

Figure 7. Logistics Problems from the AIPS 2000 Planning Competition

Figure 7 contains the logistics results from the domain-dependent track of the AIPS 2000 Planning Competition. The x axis indicates the number of packages to be moved; since more than one problem instance had the same number of packages, there is more than one result at each x coordinate. Note also the spikes in the curve for TALplanner, resulting from Java garbage collection.

8.2. Extending the Logistics Domain for Concurrency

Like many planning domains, the logistics domain is naturally concurrent. For example, different vehicles can be moved and different packages loaded and unloaded, relatively independent of one another.

However, there are other actions that should *not* be performed concurrently. For example, one cannot load packages into a truck and simultaneously drive that truck to another location. Although this is naturally true for any plan generated by the sequential planner, the concurrent planner requires such action combinations to be explicitly prevented by preconditions and resource constraints. In the logistics domain, a new resource use_of(thing) is introduced for mutual exclusion:

```
#resource use_of(thing) :domain integer :preference :none
#obs [0] ∀thing [ init(use_of(thing)) ≙ 0 ∧ minimum(use_of(thing)) ≙ 0 ∧
                  maximum(use_of(thing)) ≙ 1 ]
```

The use_of resource ensures that an object or vehicle is never used in conflicting concurrent actions. When packages are loaded into or unloaded from a vehicle, the corresponding use_of resources are borrowed 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.

8.2.1. Test Results

TALplanner was tested once more for the 30 logistics problems from the AIPS'98 planning competition [1], using the concurrent version of the planner with the new operator definitions described above but the same control rules as for the sequential planner. The resulting plans made good use of the concurrency inherent in the domain, requiring approximately the same number of plan steps but considerably fewer time steps. However, due to the additional overhead of resources and concurrency, the planner now required almost 0.9 seconds to complete the most complex problem (see Table 1, column Concurrency).

8.3. Extending the Logistics Domain for Resources and Actions with Duration

In a domain such as logistics, it is unreasonable to expect all actions to have the same duration. Therefore, to create efficient plans it is not sufficient to plan actions concurrently, but the planner must also 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.

Here, the logistics domain is extended to make use of actions with variable duration. The new integer-valued fluent dist(loc, loc) corresponds to the distance between two locations, and the drive and fly actions are modified to take distances into account. The domain is extended further by modeling the sizes of packages using an integer-valued fluent size(obj) and the carrying capacities of vehicles using an integer-valued resource space(vehicle). Since driving and flying no longer moves a vehicle instantaneously between locations, a boolean fluent moving(vehicle, loc) is introduced, which holds whenever vehicle is moving towards loc but has not yet arrived. This requires the following additions and changes to the goal narrative. Note that in this formalization, four operators are used rather than six.

```
#feature moving(vehicle, loc) :domain boolean
#feature dist(loc, loc), size(obj) :domain integer

#resource space(vehicle) :domain integer :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(object) :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(object) :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
```

Two of the control rules also need to be 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 in the control rule. We must also take into account the fact that vehicles may be moving for some time before they arrive at their destinations, and ensure that not every available airplane flies to the same destination whenever packages can be picked up.

```
#control :name "airplanes-stay-until-everything-is-loaded"
 ∀t, plane, loc [
   [t] at(plane, loc) ∧
   [t] ∃obj [ (at(obj, loc) ∧ use-plane(obj, loc) ∧ size(obj) ≤ available(space(plane))) ∨
             (in(obj, plane) ∧ unload-from-plane(obj, loc)) ] →
   [t+1] at(plane, loc) ]
#control :name "airplanes-move-to-relevant-locations"
 ∀t, plane, loc [
   [t] at(plane, loc) →
   [t+1] at(plane, loc) ∨ ∃loc', obj [ (at(plane, loc') ∨ moving(plane, loc')) ∧
                                       (in(obj, plane) ∧ unload-from-plane(obj, loc')) ] ∨
                          ∃loc', obj [ (at(plane, loc') ∨ moving(plane, loc')) ∧ at(obj, loc') ∧
                                       (use-plane(obj, loc') ∧ size(obj) ≤ available(space(plane))) ∧
                                       ∀p' [ at(p', loc') ∨ moving(p',loc') → p' = plane ]]]
```

8.3.1. Test Results

The concurrent planner has been tested on extended logistics problems based on the 30 problems from AIPS-98 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 (randomly generated), and distances between locations varied between 1 and 25.

The operator-specific analysis of pruning constraints has not yet been implemented for operators with variable duration. Mostly for this reason, TALplanner now needed approximately 11 minutes to solve the most complex problem (Table 1, column Duration). However, it was still well over 100 times faster on many of the extended problems compared to TLPLAN running the corresponding non-extended problems with single-step actions and no space constraints. Once the appropriate extensions to the operator-specific analysis algorithms have been implemented, performance should improve by at least an order of magnitude.

8.4. The Blocks World

We now briefly turn our attention to the standard blocks world with the operators pickup(block) which picks up a block from the table, putdown(block) which places a block on the table, unstack(block,block) which picks up a block previously placed on top of another block, and stack(block,block) which places a block on top of another block. Domain definitions, control rules and problems for this domain will soon be online at http://www.ida.liu.se/~jonkv/vital.html; see also the control rules published in [15].

We created a number of different test problems using between 25 and 5000 blocks and tested them in TLPLAN and TALplanner. For TLPLAN, the world definition and control rules from `domains/Blocks/4OpsBlocksWorld.tlp` in the TLPLAN distribution were used together with an additional control rule ensuring that blocks are not placed on the table if their final destinations are ready. This additional rule resulted in shorter plans as well as improved performance. For TALplanner, the same control rules were translated into TAL.

Table 2 contains the results (times are in seconds). TLPLAN was tested on the first ten problems; for the larger problems, 256 MB of memory was not sufficient. TALplanner did somewhat better using modal control rules, providing better performance as well as lower memory usage (approximately 65 MB for problem 38). When evaluated control rules were used, TALplanner solved the entire set of problem instances in less than one minute and required approximately 70 MB of memory for the largest problem.[7]

Figure 7 shows the blocks world results from the domain-dependent track of the AIPS 2000 Planning Competition, where the x axis indicates the number of blocks.

8.5. The Schedule Domain

The schedule domain was the third domain to be used in the domain-dependent track of the AIPS 2000 Planning Competition. In this domain, there is a collection of parts and a number of operators that operate on these parts:

[7] TALplanner can also use a more efficient representation of states, which decreases performance by between 5% and 25% but decreases the memory requirements for problem 43 to 35 MB.

Table 2
Test Results for the Blocks World

	Blocks	Plan length	TLPLAN	TALplan/seq progression	TALplan/seq evaluation
16	25	16	0.100	0.070	0.090
24	50	68	1.843	0.681	0.090
25	70	86	5.528	1.713	0.090
26	70	104	7.060	1.943	0.110
27	100	158	35.321	5.748	0.110
28	140	230	175.893	17.024	0.130
29	200	350	734.546	38.135	0.151
30	280	350	2918.847	83.941	0.160
31	280	470	3067.261	122.807	0.190
32	460	470	20745.280	1735.135	0.210
33	460	794		775.535	0.331
34	640	1118		2249.375	0.461
35	820	1478		3739.057	0.601
36	1000	1802		7174.557	0.771
37	1400	2450		34337.405	1.111
38	1400	2630		16771.947	1.192
39	2000	3278			1.733
40	2000	3710			1.922
41	5000	3710			6.920
42	5000	9326			8.702
43	5000	15314			18.327

polish, roll, lathe, grind, punch, drill-press, spray-paint, and immersion-paint. Each operator has a number of effects, some of which may undo the effects of other operators; for example, if a part has been painted, lathing it will have the side effect of removing the paint. The goal is for each part to have a certain shape, surface condition, and/or color.

The results for the schedule domain are shown in Figure 9, where the x axis indicates the number of parts to be scheduled. Here, TALplanner is slower than SHOP for the smallest problem instances. The reason for this is mainly that the startup time for the Java Virtual Machine and the Just-In-Time compilation of TALplanner (4–6 seconds) has been distributed evenly over all problem instances. Due to the relatively small number of instances, this contributes almost a tenth of a second to each instance, which is significant due to the small size of each instance.

8.6. Discussion

Having presented benchmark results for TALplanner in a number of domains, there are two points that should be mentioned.

In each of the benchmark domains used in this article, the performance of

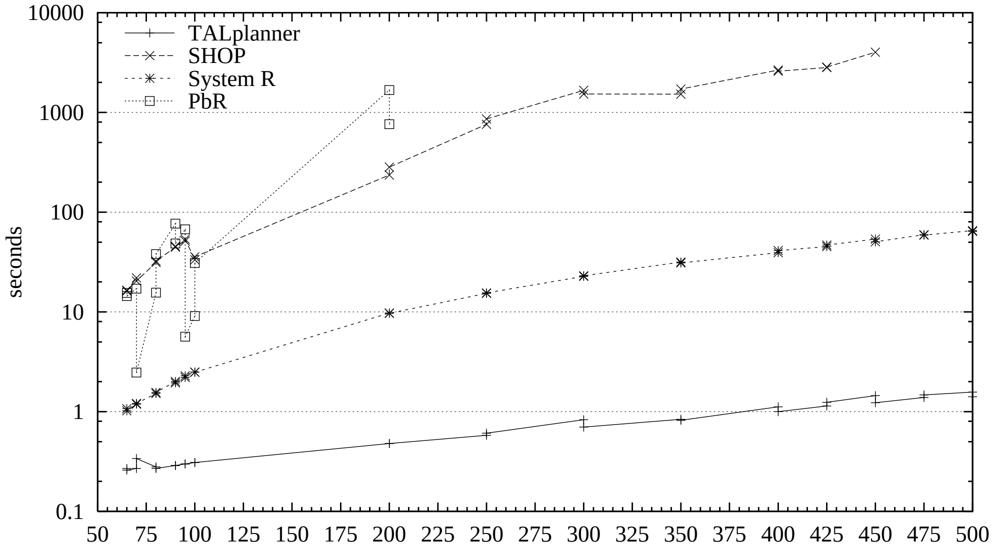

Figure 8. Blocks World Problems from the AIPS 2000 Planning Competition

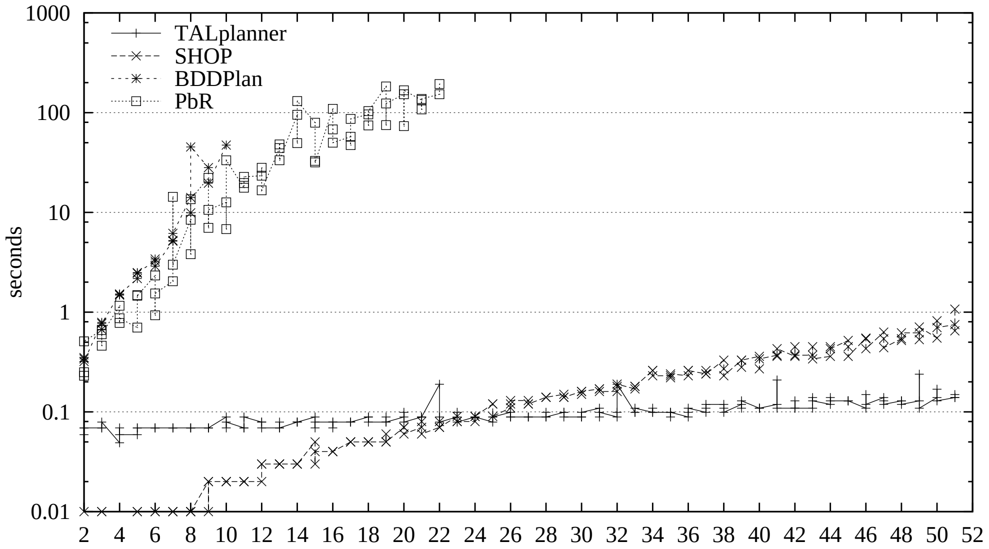

Figure 9. Schedule World Problems from the AIPS 2000 Planning Competition

TALplanner has been shown to be significantly better than that of its competitors. As a counterpoint, it should also be mentioned that the performance of TALplanner in the FreeCell domain used in the AIPS 2000 Planning Competition was not as good as we would have liked it to be. This domain appears to be better suited for domain-specific or domain-independent heuristics than for hard pruning rules. At the time of the competition, TALplanner did not yet support the specification of domain-specific heuristics, and there was very little time to develop control rules. As a result, although TALplanner still solved all problems and generated plans reasonably quickly, the plans were in some cases thousands of operators long, tens of times as long as necessary.

Obviously, the performance of TALplanner depends on the quality of the control rules being used, and the ease of writing good control rules depends on the nature of the domain. For some domains, temporal control rules may not be a suitable solution at all, or may be helpful to a certain extent but require the assistance of other techniques for optimum performance.

The second point relates to the differences in performance between TALplanner with progressed modal control rules and TALplanner with evaluated non-modal control rules. Although this performance difference is largely due to additional optimizations being possible when one is not constrained to using a purely progression-based planner, it should also be mentioned that our efforts have mainly been concentrated on optimizing the evaluation-based planner, and some of these optimizations could most likely be adapted to the use of modal formulas by defining an extended, more complex progression algorithm.

9. Conclusions

We have presented a forward-chaining planner, TALplanner, coupled with a formal semantics based on the use of a temporal action logic (TAL). Two versions of the planner were considered, one generating sequential plans and one generating concurrent plans. An interesting class of resource types was integrated into plan operator descriptions. All versions of TALplanner described in the paper have been implemented. The performance of these versions has been empirically tested using a number of different benchmarks from different planning domains. Based on the data, TALplanner has demonstrated impressive performance in comparison to a number of state-of-the-art planners in the literature. We believe the techniques described here can be extended in a number of different directions and in fact must be in order to scale up and be used in challenging domains such as cognitive robotics.

One of the most limiting factors in the current versions of TALplanner is the dual combination of restricting the initial state to be completely specified and for action types to be deterministic. We are currently investigating a number of techniques which will result in a relaxation of these restrictions.

Acknowledgements

This research is supported by the Wallenberg Foundation and the ECSEL/ENSYM graduate studies program. We would also like to thank Patrik Haslum for contributing to the extensions for concurrency and resources.

Appendix

A. The *Trans* Function

The *Trans* function defines the translation from $\mathcal{L}(\mathrm{ND})$ to $\mathcal{L}(\mathrm{FL})$. In the following, $\mathcal{Q}$ is a quantifier and $\otimes$ is a binary logical connective. All variables occurring only on the right-hand side are assumed to be previously unused variables.

$$
\begin{aligned}
\mathit{Trans}([\tau]\ f(\overline{\omega}) \hat{=} v) &= \mathit{Holds}(\tau, f(\overline{\omega}), v) \\
\mathit{Trans}([\tau]\ \neg\alpha) &= \neg\,\mathit{Trans}([\tau]\ \alpha) \\
\mathit{Trans}([\tau]\ \alpha \otimes \beta) &= \mathit{Trans}([\tau]\ \alpha) \otimes \mathit{Trans}([\tau]\ \beta) \\
\mathit{Trans}([\tau]\ \mathcal{Q}v[\alpha]) &= \mathcal{Q}v[\mathit{Trans}([\tau]\ \alpha)] \\
\mathit{Trans}([\tau,\tau']\ \alpha) &= \forall t[\tau \le t \le \tau' \to \mathit{Trans}([t]\ \alpha)] \\
\mathit{Trans}((\tau,\tau']\ \alpha) &= \forall t[\tau < t \le \tau' \to \mathit{Trans}([t]\ \alpha)] \\
\mathit{Trans}(X([\tau]\ f(\overline{\omega}) \hat{=} v)) &= \mathit{Occlude}(\tau, f(\overline{\omega})) \\
\mathit{Trans}(X([\tau]\ \neg\alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \\
\mathit{Trans}(X([\tau]\ \alpha \otimes \beta)) &= \mathit{Trans}(X([\tau]\ \alpha)) \wedge \mathit{Trans}(X([\tau]\ \beta)) \\
\mathit{Trans}(X([\tau]\ \mathcal{Q}v[\alpha])) &= \forall v[\mathit{Trans}(X([\tau]\ \alpha))] \\
\mathit{Trans}(X([\tau,\tau']\ \alpha) &= \forall t[\tau \le t \le \tau' \to \mathit{Trans}(X([t]\ \alpha))] \\
\mathit{Trans}(X((\tau,\tau']\ \alpha) &= \forall t[\tau < t \le \tau' \to \mathit{Trans}(X([t]\ \alpha))] \\
\mathit{Trans}(R([\tau]\ \alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \wedge \mathit{Trans}([\tau]\ \alpha) \\
\mathit{Trans}(R([\tau,\tau']\ \alpha) &= \mathit{Trans}(X([\tau,\tau']\ \alpha) \wedge \mathit{Trans}([\tau']\ \alpha) \\
\mathit{Trans}(R((\tau,\tau']\ \alpha) &= \mathit{Trans}(X((\tau,\tau']\ \alpha) \wedge \mathit{Trans}([\tau']\ \alpha) \\
\mathit{Trans}(I([\tau]\ \alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \wedge \mathit{Trans}([\tau]\ \alpha) \\
\mathit{Trans}(I([\tau,\tau']\ \alpha) &= \mathit{Trans}(X([\tau,\tau']\ \alpha) \wedge \mathit{Trans}([\tau,\tau']\ \alpha) \\
\mathit{Trans}(I((\tau,\tau']\ \alpha) &= \mathit{Trans}(X((\tau,\tau']\ \alpha) \wedge \mathit{Trans}((\tau,\tau']\ \alpha) \\
\mathit{Trans}(C_T([\tau]\ \alpha)) &= \forall t[\tau = t+1 \to \mathit{Trans}([t]\ \neg\alpha)] \wedge \mathit{Trans}([\tau]\ \alpha) \\
\mathit{Trans}([\tau,\tau']\ \Psi(\overline{\omega})) &= \mathit{Occurs}(\tau, \tau', \Psi(\overline{\omega})) \\
\mathit{Trans}(\mathit{Per}(t,f)) &= \mathit{Per}(t,f) \\
\mathit{Trans}(\mathit{Dur}(t,f,v)) &= \mathit{Dur}(t,f,v) \\
\mathit{Trans}(\mathit{value}(t,f)) &= \mathit{value}(t,f)
\end{aligned}
$$

B. Circumscription in TAL

The second-order circumscription of a number of predicates $\overline{P} = P_1, \ldots, P_n$ in the theory $\Gamma(\overline{P})$ is denoted $\mathit{Circ}_{SO}(\Gamma(\overline{P}); \overline{P})$ (see Lifschitz [31]). Intuitively, $\mathit{Circ}_{SO}(\Gamma(\overline{P}); \overline{P})$ represents a (second-order) theory containing $\Gamma(\overline{P})$ where the extensions of the predicates $\overline{P}$ are minimal.

To transform a narrative from $\mathcal{L}$(ND) to $\mathcal{L}$(FL):

1. Let *dom*, *acs*, *dep*, *obs*, *occ*, and *per* be the sets of statements with labels dom, acs, dep, obs, occ, and per respectively, completed with universal quantification for variables occurring freely.

2. Let $\Gamma_{dom} = \mathit{Trans}(dom)$, $\Gamma_{acs} = \mathit{Trans}(acs)$, $\Gamma_{dep} = \mathit{Trans}(dep)$, $\Gamma_{obs} = \mathit{Trans}(obs)$, $\Gamma_{occ} = \mathit{Trans}(occ)$, and $\Gamma_{per} = \mathit{Trans}(per)$

3. Let $\Gamma' = \Gamma_{acs} \cup \Gamma_{dep}$ and let $\Gamma = \mathit{Circ}_{SO}(\Gamma'(\mathit{Occlude}); \mathit{Occlude}) \cup \Gamma_{dom} \cup \mathit{Circ}_{SO}(\Gamma_{occ}(\mathit{Occurs}); \mathit{Occurs}) \cup \Gamma_{obs} \cup \Gamma_{fl} \cup \Gamma_{fnd}$. Γ is the theory that is used for proofs in TAL.

The set Γ_{fl} consists of the following formulas relating to Per and Dur:

$$\begin{aligned}\Gamma_{fl} = \{&\forall t, f, v[\mathit{Dur}(t, f, v) \wedge \neg \mathit{Occlude}(t, f) \rightarrow \mathit{Holds}(t, f, v)],\\ &\forall t, f, v[\mathit{Per}(t, f, v) \wedge \neg \mathit{Occlude}(t{+}1, f) \rightarrow (\mathit{Holds}(t{+}1, f, v) \leftrightarrow \mathit{Holds}(t, f, v))],\\ &\forall t, f, v, w[\mathit{Dur}(t, f, v) \wedge \mathit{Dur}(t, f, w) \rightarrow v = w],\\ &\forall t, f[\mathit{Per}(t, f) \oplus \exists v[\mathit{Dur}(t, f, v)]\}\end{aligned}$$

Finally, the set Γ_{fnd} consists of foundational axioms for unique names for actions, fluents and values, constraints that a fluent has exactly one value at each timepoint, and the temporal structure axioms.

References

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

[2] AIPS-2000. Artificial Intelligence Planning Systems: 2000 Planning Competition. http://www.cs.toronto.edu/aips2000, 2000.

[3] R. Alur, T. Feder, and T. A. Henzinger, 'The benefits of relaxing punctuality', in *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing (PODC'91)*, pp. 139–152, Montréal, Canada, (August 1991). Available at http://www.cis.upenn.edu/~alur/Podc91.ps.gz.

[4] R. Alur and T. A. Henzinger, 'Back to the future: Towards a theory of timed regular languages', in *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 177–186, Pittsburgh, Pennsylvania, USA, (October 1992). IEEE Computer Society Press. Updated version available at http://www-cad.eecs.berkeley.edu/~tah/Publications/back_to_the_future.ps.

[5] J. L. Ambite, *Planning by Rewriting*, Ph.D. dissertation, University of Southern California, 1998. Available at http://www.isi.edu/~ambite/thesis.ps.gz.

[6] J. L. Ambite, C. A. Knoblock, and S. Minton, 'Learning plan rewriting rules', in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS-2000)*, eds., S. Chien, S. Kambhampati, and C. A. Knoblock, pp. 3–12, Breckenridge, Colorado, USA, (April 2000). AAAI Press, Menlo Park, USA. Available at http://www.isi.edu/~ambite/2000-aips.ps.

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

[8] F. Bacchus and F. Kabanza, 'Planning for temporally extended goals', in *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pp. 1215–1222,

Portland, Oregon, USA, (August 1996). AAAI Press / The MIT Press. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKAAAI96.ps.gz.

[9] F. Bacchus and F. Kabanza, 'Using temporal logic to control search in a forward chaining planner', in *New Directions in AI Planning*, eds., M. Ghallab and A. Milani, 141–153, IOS Press, (1996). Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKEWSP96.ps.gz.

[10] F. Bacchus and F. Kabanza, 'Planning for temporally extended goals', *Annals of Mathematics and Artificial Intelligence*, **22**, 5–27, (1998). Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKAMAI98.ps.gz.

[11] F. Bacchus and F. Kabanza, 'Using temporal logics to express search control knowledge for planning', *Artificial Intelligence*, **116**, 123–191, (2000). Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKTlplan.ps.

[12] B. Bonet and H. Geffner. HSP: Heuristic search planner, 1998. Available at http://www.ldc.usb.ve/~hector/.

[13] P. Doherty, 'Reasoning about action and change using occlusion', in *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, ed., A. G. Cohn, pp. 401–405, Amsterdam, The Netherlands, (August 1994). John Wiley and Sons, Ltd., England. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/ecai94.ps.gz.

[14] 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.

[15] P. Doherty and J. Kvarnström, 'TALplanner: An empirical investigation of a temporal logic-based forward chaining planner', in *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (TIME'99)*, eds., C. Dixon and M. Fisher, pp. 47–54, Orlando, Florida, USA, (May 1999). IEEE Computer Society Press. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time99-final.ps.gz.

[16] P. Doherty and W. Łukaszewicz, 'Circumscribing Features and Fluents: A fluent logic for reasoning about action and change', in *Proceedings of the Eighth International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, eds., Z. W. Ras and M. Zemankova, pp. 521–530, Charlotte, North Carolina, USA, (October 1994).

[17] E. A. Emerson, 'Temporal and modal logic', in *Handbook of Theoretical Computer Science*, ed., J. van Leeuwen, volume B, chapter 16, 997–1072, MIT, (1990).

[18] M. Fox and D. Long, 'The detection and exploitation of symmetry in planning problems', in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, ed., T. Dean, pp. 956–961, Stockholm, Sweden, (1999). Morgan Kaufmann Publishers, San Francisco. Available at http://www.dur.ac.uk/~dcs0www/research/stanstuff/symm.ps.gz.

[19] J. Gustafsson and P. Doherty, 'Embracing occlusion in specifying the indirect effects of actions', in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96)*, eds., L. C. Aiello, J. Doyle, and S. Shapiro, pp. 87–98. Morgan Kaufmann Publishers, San Francisco, (1996). Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/final-kr96.ps.gz.

[20] J. Hoffmann. Fast-Forward home page. http://www.informatik.uni-freiburg.de/~hoffmann/ff.html.

[21] J. Hoffmann and B. Nebel, 'The FF planning system: Fast planning generation through heuristic search'. Submitted to Journal of Artificial Intelligence Research. Available at http://www.informatik.uni-freiburg.de/~hoffmann/papers/jair2000b.ps.gz.

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

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

[24] L. Karlsson, J. Gustafsson, and P. Doherty, 'Delayed effects of actions', in *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98)*, ed., H. Prade, pp. 542–546, Brighton, UK, (August 1998). John Wiley and Sons, Ltd., England. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/ecai98.ps.gz.

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

[26] D. Kibler and P. Morris, 'Don't be stupid', in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, ed., A. Drinan, pp. 345–347, Vancouver, B.C., Canada, (August 1981).

[27] J. Koehler. Miconic 10 elevator domain home page. http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html.

[28] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, 'Extending planning graphs to an ADL subset', in *Proceedings of the Fourth European Conference on Planning (ECP'97)*, ed., S. Steel, pp. 273–285, Toulouse, France, (September 1997). Springer. Available at http://www.informatik.uni-freiburg.de/~hoffmann/papers/ecp97.ps.gz.

[29] J. Kvarnström and P. Doherty, 'Tackling the qualification problem using fluent dependency constraints', *Computational Intelligence*, **16**(2), 169–209, (May 2000).

[30] J. Kvarnström, P. Doherty, and P. Haslum, 'Extending TALplanner with concurrency and resources', in *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, ed., W. Horn, pp. 501–505, Berlin, Germany, (August 2000). IOS Press, The Netherlands. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/www-ecai.ps.gz.

[31] V. Lifschitz, 'Computing circumscription', in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 121–127, Los Angeles, California, USA, (August 1985). Morgan Kaufmann Publishers, Los Altos, CA.

[32] F. Lin. A planner called R. Submitted to AI Magazine. Available at http://www.cs.ust.hk/faculty/flin/papers/r-aim.ps.

[33] J. McCarthy, 'Circumscription – a form of non-monotonic reasoning', *Artificial Intelligence*, **13**, 27–39, (1980). Available at http://www-formal.stanford.edu/jmc/circumscription.ps.

[34] D. Nau, Y. Cau, A. Lotem, and H. Muños-Avila, 'SHOP: Simple hierarchical ordered planner', in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, ed., T. Dean, pp. 968–973, Stockholm, Sweden, (1999). Morgan Kaufmann Publishers, San Francisco. Available at http://www.cs.umd.edu/~nau/papers/shop-ijcai99.pdf.

[35] E. P. D. Pednault, 'ADL: Exploring the middle ground between STRIPS and the Situation Calculus', in *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, eds., R. J. Brachman, H. J. Levesque, and R. Reiter, pp. 324–332, Toronto, Ontario, Canada, (1989). Morgan Kaufmann Publishers, San Mateo.

[36] E. Sandewall, *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems*, Oxford University Press, 1994.

[37] H.-P. Störr. BDDPlan home page. http://pikas.inf.tu-dresden.de/~stoerr/bddplan.html.