Using TALplanner

Jonas Kvarnström

March 3, 2009

Contents

1 Introduction

This is the user manual for TALplanner, a forward-chaining planner which allows the use of rich domain information in order to improve planning performance. This manual is mainly intended to explain the practical use of TALplanner: The reader is assumed to already be familiar with forward-chaining planning in general, and while the concept of domain-dependent control formulas will be discussed to some extent, the reader will be referred to other articles for a deeper view of the subject.

At the moment, this manual is definitely a work in progress and should be used in conjunction with the example domain definitions and problem definitions that are included in the TALplanner distribution. Please report any omissions, sections that are difficult to understand or other problems to jonkv@ida.liu.se. I will get back to you as soon as possible.

The history of TALplanner begins in 1999 [3], when a first prototype implementation was developed. This prototype was initially inspired by TLplan [1], but TALplanner soon began to diverge, leading to new features such as support for concurrency and resources [13]. Participation in two international planning competitions also led to new features and extended expressivity [12, 4, 14] as well as new domain analysis algorithms that improved the performance of the planner [10]. More recently, the planner has been extended with support for execution monitoring [15, 6]. The most complete and up-to-date description of the planner is available in my PhD thesis [11].

1.1 ALPHA TEST VERSION!

Throughout our research, the TALplanner implementation has served as a test bed where new ideas have continuously been tested and either validated or discarded. This is the main reason why no version of TALplanner has been released to the public before: It never seemed to be the right time to "freeze" and stabilize the current implementation. However, many have requested access to TALplanner, and at this point we have decided to begin releasing alpha versions.

I am currently in the process of making a series of architectural changes that will be required for future extensions of the planner, and while this is being done, some features of the planner are temporarily disabled. Additionally, some performance features and memory optimizations are temporarily disabled pending a review of the implementation, making these alpha versions considerably slower and more memory hungry than the final version will be – or than previous versions used to be.

We ask that any benchmarks or tests done using the alpha version of TALplanner are accompanied with a note explaining that several performance features are disabled and that the performance is not indicative of that of the final version of the planner, or even that of earlier versions!

2 Domain Definitions and Problem Instance Definitions

The input to TALplanner consists of two parts: A domain definition and a problem instance definition. Domain definitions are written using an extended version of TAL, Temporal Action Logic [2, 5]. Problem instance definitions can be written in TAL or in a simple subset of PDDL,

the Planning Domain Definition Language [9, 8, 7]. The latter syntax was implemented mainly for use in planning competitions and is less well-tested.

Some example domains and problem instances are provided in the TALplanner distribution. We recommend that you use these in parallel with reading the manual.

2.1 Types, Objects and Variables

TAL is a order-sorted logic, meaning that there is a hierarchy of sorts (types), or more accurately, a forest of sorts where there is not necessarily a single root. The temporal sort is always present and corresponds to the natural numbers (0, 1, 2, ...). The boolean value sort is also always present, and contains the values {true, false}. Other value sorts and their domains must be explicitly defined in the domain definition.

2.1.1 Enumerated Value Domains

A prototypical set of value domain definitions looks like this:

```
#domain vehicle :elements { mycar, othercar, somebike }
#domain car :parent vehicle :elements { mycar, othercar }
#domain bicycle :parent vehicle :elements { somebike }
#domain color :elements { red, green, blue }
```

In this case, we also defined a set of values that belong to the domains. This can be used to define values that will be present in all problem instances for a specific planning domain. If there are no such values, the domain declarations would instead look like this:

```
#domain vehicle :elements { }
#domain car :parent vehicle :elements { }
#domain bicycle :parent vehicle :elements { }
#domain color :elements { }
```

Then, a problem instance definition could contain the following statements, declaring a set of values that are present in this particular problem instance:

```
#objects :domain vehicle :elements { mycar, othercar, somebike }
#objects :domain car :elements { mycar, othercar }
#objects :domain bicycle :elements { somebike }
#objects :domain color :elements { red, green, blue }
```

Values that belong to a given domain must necessarily also belong to the corresponding superdomains, if any. In the example above, this was explicitly represented in the problem instance definition. However, it is sufficient to add values to a single domain. TALplanner will automatically add them to all corresponding superdomains. In other words, the following example is equivalent:

```
#objects :domain vehicle :elements { }
#objects :domain car :elements { mycar, othercar }
#objects :domain bicycle :elements { somebike }
#objects :domain color :elements { red, green, blue }
```

2.1.2 Numeric Value Domains

TALplanner also supports finite numeric value domains. The following declaration specifies the integer domain `int` consisting of the numbers 0 to 999 (lower bound included, upper bound excluded):

```
#domain :name int :integer :lb 0 :ub 1000
```

The following declaration specifies the fixed-point domain `fp` consisting of the numbers 0.000 to 999.999:

```
#domain :name fp :fixedpoint :lb 0 :ub 1000 :decimals 3
```

As shown in Section 2.3.3, TALplanner supports a number of standard arithmetic operations and relations for numeric value domains.

2.1.3 Value Variables

Like objects, variables are also typed in TAL: Each variable is associated with a given sort, and can only take on values belonging to that sort (including subsorts). Variables must be declared.

The temporal variables *t* and *t1* through *t5* are automatically declared.

Creating a value domain `dom` automatically declares the associated variables *dom*, *dom1*, *dom2*, *dom3*, *dom4*, and *dom5* (though this behavior can be suppressed by specifying the flag :novars before :elements or :parent in the domain declaration). For example, declaring the domain `vehicle` implicitly causes the variables *vehicle*, *vehicle1*, *vehicle2*, *vehicle3*, *vehicle4*, and *vehicle5* to be declared.

If more meaningful names are desired, additional variables can be declared as follows:

```
#timevar start, end
#valuevar from, to :domain location
```

Finally, value variables can be defined where there are used. The following operator declaration declares the variables *from* and *to* to be `location` variables. Note that the type of a given variable must be unique throughout a domain definition. In other words, given the following declaration, *from* and *to* cannot be used as variables of types other than `location` in other operator definitions or formulas.

```
#operator move(location from, location to) :at t
...
```

2.2 Features and Fluents

A feature, in TAL terminology, is a property of the world. A fluent is a representation of the development of a feature as a function of time (that is, what may sometimes be called a state variable). We may use these terms interchangeably.

Strictly speaking, what is declared in the domain definition is a *feature symbol* which may have arguments belonging to specific value sorts. Each instance, created by instantiating the arguments with concrete values from the corresponding value domains, is a separate feature.

However, we will often use the word feature to refer to either a feature symbol or a feature instance.

A feature symbol is declared as follows, providing a set of argument domains and a value domain:

```
#feature isLarge(vehicle) :domain boolean
#feature colorOf(vehicle) :domain color
#feature behind(vehicle, vehicle) :domain boolean
```

A number of additional flags can be added to specify certain properties of a feature. In many cases, this knowledge can improve both space and time requirements. Though it would be possible to extract such knowledge through automated domain analysis, we have chosen to focus on other aspects of planning and therefore require the information to be specified explicitly.

- `:timeless` states that the value of an instance of this feature does not vary over time. For example, in a domain where cars cannot be repainted, colorOf may be declared :timeless. Such features cannot be reassigned new values by actions.
- `:total-function` can be used for boolean features, and is a hint to the planner that given the way the planning domain is set up, the feature will always be a boolean representation of a total function. For example, the boolean feature at(obj,loc) may be used to represent the location of an object. If any object is always at exactly one location, the feature may be marked `:total-function`.

  In other words, the following definition:

  ```
  #feature at(obj, loc) :domain boolean :total-function
  ```

  implies that the operators you provide to the planner will always preserve the following properties:

  - $\forall t, obj, loc_1, loc_2.[t]\ \mathsf{at}(obj, loc_1) \wedge \mathsf{at}(obj, loc_2) \rightarrow loc_1 = loc_2$
  - $\forall t, obj \exists loc.[t]\ \mathsf{at}(obj, loc)$
- `:partial-function` can be used for boolean features, and is a hint to the planner that given the way the planning domain is set up, the feature will always be a boolean representation of a partial function. For example, the boolean feature `at(obj,loc)` may be used to represent the location of an object. If any object is always at *at most* one location, the feature may be marked `:partial-function`.

  In other words, the following definition:

  ```
  #feature at(obj, loc) :domain boolean :partial-function
  ```

  implies that the operators you provide to the planner will always preserve the following property:

  - $\forall t, obj, loc_1, loc_2.[t]\ \mathsf{at}(obj, loc_1) \wedge \mathsf{at}(obj, loc_2) \rightarrow loc_1 = loc_2$.
- `:double-partial-function` can be used for boolean binary features, and is a hint to the planner that given the way the planning domain is set up, the feature will always be a boolean representation of a partial function from the first argument to the second argument *and* a boolean representation of a partial function from the second argument to the first argument. In the blocks world, for example, on(block1,block2) can be marked `:double-partial-function`: a block can be on top of at most one other block and at most one block can be on top of any other block.

- `:secondary` can be used to indicate that the value of this feature will be uniquely determined by the values of the non-secondary ("primary") fluents. This helps the planner determine which feature values must be tested in cycle checking, and decreases memory consumption in certain cases. In the blocks world, for example, handempty is true if and only if there is no block b such that holding(b), which means that handempty could be marked as a secondary feature. The converse is not true, because knowing whether the hand is empty does not provide knowledge about which block is being held.

2.3 Terms and Formulas

TAL formulas are used to specify initial states, goals, action preconditions, and several other parts of a domain definition or problem instance definition. We again refer the reader to [2, 5] for a more formal specification of the syntax and semantics of TAL formulas. Below, we will briefly introduce the way formulas are written in TAL and how to translate logical connectives and other symbols into the ASCII-based input language used by TALplanner.

2.3.1 Feature Expressions

A *feature expression* denotes a feature instance and has the form $f(\overline{\omega})$, where f is a feature symbol and $\overline{\omega}$ is a sequence of value terms or feature expressions having types corresponding to the argument types of f. A feature expression is not a value term, since an external temporal context is required to determine the time at which the feature instance should be evaluated.

2.3.2 Terms

A *temporal term* denotes a non-negative integer timepoint and consists of a numeric temporal constant (0, 1, 2, ...), a symbolic temporal constant (currently not supported in TALplanner), a temporal variable, or a combination of temporal terms using the operators + and −. For example, if t is a temporal variable, then 3 and $t + 3$ are valid temporal terms.

Note that since non-negative time is used, terms that can result in negative values are forbidden. For example, $t - 1$ is not a legal temporal term, since it would be undefined for $t = 0$.

A *value term* denotes a value of a specific sort and consists of a value name (declared using `#objects` or `#domain`), a value constant (currently not supported in TALplanner), a value variable, or an expression of the form *value*(τ, f), where τ is a temporal term and f is a feature expression, denoting the value of the feature instance corresponding to f at the timepoint (temporal term) τ. For example, if *car* is a value variable and colorOf is a feature taking a single argument of domain `car`, then *value*(τ, colorOf(*car*)) is a valid value term.

Numeric value names (literals such as 1.0 or 42) may belong to multiple domains. For example, the value 42 may belong to either a domain `int` containing integers between 0 and 999 or to a fixed point domain `fp` containing numbers between 0.000 and 999.999. Unless the context provides enough information for the parser to disambiguate such values, you may have to manually disambiguate the value using an expression such as `fp` :: 42, where `fp` is the name of the intended domain.

2.3.3 Built-In Functions and Relations

The input language used by TALplanner has been extended to support a number of built-in functions and relations. These functions are used like feature expressiond, but are prefixed with a dollar sign to avoid name space collisions.

Most of the functions and operations below have standard definitions for infinite numeric domains. The exact semantics of these operations in *finite* numeric domains is defined in Kvarnström [11].

We also extend the use of these functions and operations to finite non-numeric domains by associating each value in a non-numeric domain with a non-negative number in the order they were declared, beginning with 0. In other words, given the domain $\langle a, b, c, d\rangle$, we have that $a < b$, $\$\mathsf{next}(b) = c$, and so on. Note also that non-numeric values can belong to multiple sorts in the sort hierarchy, and for some functions, the correct answer depends on which of these sorts is taken into consideration. For example, suppose that the sort `vehicle` consists of the values $\langle \mathsf{mycar}, \mathsf{somebike}, \mathsf{othercar}\rangle$, while the subsort `car` consists of the values $\langle \mathsf{mycar}, \mathsf{othercar}\rangle$. The next value after mycar is somebike in the `vehicle` domain, but othercar in the `car` domain. In all such cases, we use the *largest* domain to which the value belongs, as this is guaranteed to be uniquely defined.

Given two values v and v', the following arithmetic operations can be used.

- $\$\mathsf{plus}(v, v')$ – can also be written $v + v'$
- $\$\mathsf{minus}(v, v')$ – can also be written $v - v'$
- $\$\mathsf{times}(v, v')$ – can also be written $v * v'$
- $\$\mathsf{div}(v, v')$ – can also be written v/v'

Given two values v and v', the following relations can be used.

- $\$\mathsf{less}(v, v')$ – can also be written $v < v'$
- $\$\mathsf{lessequal}(v, v')$ – can also be written $v <= v'$
- $\$\mathsf{greater}(v, v')$ – can also be written $v > v'$
- $\$\mathsf{greaterequal}(v, v')$ – can also be written $v >= v'$

The standard equality operators are part of TAL:

- $\$\mathsf{equals}(v, v')$ – can also be written $v = v'$
- $\$\mathsf{notequals}(v, v')$ – can also be written $v == v'$

Note the difference between the equality operator comparing two value terms ($v = v'$) and the "isvalue" operator stating that a feature takes on a given value ($f == v$).

A number of functions on values are also available.

- $\$\mathsf{prev}(v)$ – the value before v (in the largest domain it belongs to, as discussed previously)
- $\$\mathsf{next}(v)$ – the value after v.
- $\$\mathsf{sqrt}(v)$ – the square root of v.
- $\$\mathsf{index}(v)$ – the index of v. To use this, exactly one integer domain must have been declared.

- $\$\mathsf{abs}(v)$ – the absolute value of v.

The following functions are used for geometric reasoning.

- $\$\mathsf{sin}(v), \$\mathsf{cos}(v), \$\mathsf{tan}(v)$ – trigonometric functions.
- $\$\mathsf{asin}(v), \$\mathsf{acos}(v), \$\mathsf{atan}(v)$ – inverse trigonometric functions.
- $\$\mathsf{distance2d}(x, y, x', y')$ – the distance between (x, y) and (x', y') in the plane.

The following expressions are used to calculate values by iterating over all instantiations of a sequence of variables.

- The expression $\$\mathtt{sum}(< v_1, ..., v_n >, \phi, \omega)$ iterates over all possible instantiations of $\langle v_1, ..., v_n \rangle$, returning the sum, for all instantiations instantiations that satisfy the logic formula ϕ, of the value term ω.

  Example: $\$\mathtt{sum}(< pile, crate >, [t]\ \mathsf{relevant}(pile)\ \&\ \mathsf{in}(crate, pile), value(t, \mathsf{weight}(crate)))$.

  This can also be used to *count* values, by letting $\omega = 1$.
- The expression $\$\mathtt{mmax}(< v_1, ..., v_n >, \phi, \omega)$ iterates over all possible instantiations of $\langle v_1, ..., v_n \rangle$, returning the maximum, for all instantiations that satisfy the logic formula ϕ, of the value term ω.
- The expression $\$\mathtt{mmax}(< v_1, ..., v_n >, \phi, \omega)$ iterates over all possible instantiations of $\langle v_1, ..., v_n \rangle$, returning the minimum, for all instantiations that satisfy the logic formula ϕ, of the value term ω.

2.3.4 Fluent Formulas

A *fluent formula* may contain feature expressions that lack a temporal context specifying at which timepoint, (or equivalently, in which state) they should be evaluated. For this reason, fluent formulas are generally only used as parts of logic formulas that provide a suitable temporal context, as shown in the next section, or as goals, where the temporal context is implicit.

An *elementary fluent formula* has the form $f \mathrel{\hat{=}} \omega$, where f is a feature expression and ω is a value term whose type corresponds to the type of f. For example, $\mathsf{colorOf}(mycar) \mathrel{\hat{=}} red$ is a fluent formula. The formula $f \mathrel{\hat{=}} \mathsf{true}$ can be abbreviated as f. Other fluent formulas are formed from these formulas using the standard connectives together with quantification over values.

The input language for TALplanner uses the following syntax for operators, connectives and quantifiers in fluent formulas:

```
==  (feature value: ≙)

!   (negation)
&   (conjunction)
|   (disjunction)
->  (implication)
<-> (equivalence)

forall var1, var2, ... [ fluent formula ]
exists var1, var2, ... [ fluent formula ]
```

Note again the difference between the equality operator comparing two value terms ($v = v'$) and the "isvalue" operator stating that a feature takes on a given value ($f == v$).

The following shorthand notation can also be used.

- `forall var1 = term1, ..., varn = termn [ formula ]` is equivalent to
  `forall var1, ..., varn [ var1 = term1 & ...  & varn = termn -> formula ]`
- `exists var1 = term1, ..., varn = termn [ formula ]` is equivalent to
  `exists var1, ..., varn [ var1 = term1 & ...  & varn = termn & formula ]`

2.3.5 Logic Formulas

In contrast to fluent formulas, logic formulas are "self-contained" in the sense that no external temporal context is required for evaluation. A logic formula in TAL is one of the following:

- $\tau = \tau'$, $\tau < \tau'$ or $\tau \leq \tau'$, where τ and τ' are temporal terms.
- $\omega = \omega'$, where ω and ω' are value terms.
- $[\tau, \tau']\ \alpha$, $(\tau, \tau']\ \alpha$, $[\tau, \tau')\ \alpha$, $(\tau, \tau')\ \alpha$, $[\tau, \infty)\ \alpha$, $(\tau, \infty)\ \alpha$ or $[\tau]\ \alpha$, where τ and τ' are temporal terms and α is a fluent formula.
- A combination of logic formulas using the standard logical connectives together with quantification over values and time.

The symbol ∞ is represented as `$inf` in the TALplanner input language.

2.4 Initial State

The initial state must currently be completely defined in each problem instance definition. There are two ways of describing the initial state: Using PDDL syntax or using TAL syntax.

2.4.1 Initial States using TAL syntax

In TAL, observed facts such as those facts that hold in the initial state are specified using observation statements, labeled `#obs`, or domain constraints, labeled `#dom`. Since observation statements are intended to correspond to direct observations of specific values, they only support simple conjunctions. Domain constraints have no such limitation and can be used for arbitrary formulas constraining the initial state. In the TALplanner context, these are logically equivalent.

Because TAL intrinsically supports incomplete information (though this is not currently implemented in TALplanner), the closed world assumption is *not* applied to the set of positive facts specified in the problem instance definition. In other words, negative facts must also be explicitly specified. Additionally, the TAL logic permits the specification of facts at any timepoint, and each observation statement must therefore specify a temporal context. For TALplanner, this context must currently always be 0.

Observation statements and domain constraints are logical formulas in TAL, as described above. At this point, we only provide a set of illustrative examples.

```
#obs [0] isLarge(mycar) & !isLarge(othercar) & !isLarge(somebike)
#obs [0] colorOf(mycar) == red & colorOf(othercar) == red
#obs [0] colorOf(somebike) == blue
...
#dom [0] forall vehicle, loc [ at(obj,loc) <->
    (vehicle=mycar & loc=athome) |
    (vehicle=othercar & loc=road42)
]
```

2.4.2 Initial States using PDDL syntax

TALplanner uses the standard syntax for PDDL problem definitions. Domain names, requirements and metrics are ignored.

```
(define (problem NAME)
   (:domain NAME)
   [(:requirements :FOO :BAR :...)]
   (:objects LIST)
   [(:init INIT)]
   (:goal GOAL)*
   [(:metric IGNORED...)]
)
```

2.5 Goal

The goal of a PDDL planning problem instance is specified using the standard PDDL syntax.

The goal of a TAL planning problem instance is specified using goal statements. Each goal statement consists of a TAL fluent formula that must be true in any valid goal state. Since arbitrary fluent formulas are allowed, goals can be disjunctive and existential quantifiers can be used. However, it should be noted that TALplanner has almost exclusively been used with conjunctive goals and that the code paths related to using existential and disjunctive goals together with the goal operator need further testing and code review.

```
#goal behind(othercar, mycar)
#goal forall vehicle [ isLarge(vehicle) -> colorOf(vehicle) == red ]
...
```

2.5.1 The goal operator

Control rules use a goal operator to test whether or not the goal of the current problem instance requires a certain formula to hold: If γ is the conjunction of all goal formulas, then the formula goal(ϕ) holds iff $\gamma \models \phi$, where ϕ is a fluent formula that may contain variables bound outside the goal modality. This is key to the pruning power of control rules.

The following is a simple example from the logistics domain, stating that if a package is at a given location and the goal requires it to be in that location, it must remain there at the next timepoint:

```
#control :name "never-destroy-at-goals"
   forall t, package, location [
          ([t] at(package, location)) & goal(at(package, location)) ->
          [t+1] at(package, location)
   ]
```

A translation of the goal operator to standard TAL is provided in Kvarnström [11].

2.6 Operator Definitions

The following syntax is used for operator definitions in TALplanner. Concrete examples are provided in the TALplanner distribution.

```
#operator opname(var1, ..., varn)
  :at start,
      t2 [= temporalterm],
      ...,
      tn [= temporalterm],
      end
  :duration temporalterm [:as var]
  :precond [:name "name"] logicformula
  ...
  :prevail [:name "name"] logicformula
  ...
  :control [:name "name"] logicformula
  :resources global-resource-usage
  :context context
  ...
```

The operator is given a name, and its formal parameters are specified. Each formal parameter is a value variable that is either declared previously or declared inline.

Since TAL uses explicit time, a formal invocation timepoint variable is also required, specified using :at *varname*. Here, the invocation timepoint is called *start*. Since TALplanner allows temporally extended actions having effects at multiple timepoints, additional temporal variables can also be defined for use in effects. Example: :at start, arrival = start + 20, end.

The duration of the operator can be explicitly specified, and possibly assigned to a temporal variable that can then be used in effect definitions. If no duration is specified, a duration of 1 is assumed.

To aid readability, each operator can have multiple preconditions. Preconditions can also be named, which can be useful when tracing the operation of the planner. Preconditions must only refer to the state of the world at the invocation time of the operator.

Prevail conditions are a generalization of preconditions, and can refer to the state of the world at arbitrary times – including *after* the invocation of the operator. For this reason, prevail conditions cannot in general be checked before attempting to invoke an operator. However, the planner backtracks if it detects the violation of a prevail condition, and the final plan is guaranteed to satisfy all prevail conditions.

Operator-specific control formulas can also be specified. Whereas preconditions or prevail conditions specify constraints on the physical executability of an action, operator-specific control formulas are intended to express when an action *should not* be used despite being executable. These conditions can refer to the invocation state of the operator as well as to future states. Just like preconditions and prevail conditions, operator-specific control formulas can refer to all parameters of an operator.

TALplanner's support for resources and resource constraints is explained in Section 2.8.

Context-dependent effects are specified as a set of *contexts*. Each context uses the following syntax:

```
:context
  [:forall var1, ..., varn]
  [:precond logicformula]
  [:resources resource-effect-list]
  :effects eff1, ..., effn
```

Each effect has one of the following forms:

```
[time] feature-expression := value,
[time1,time2] feature-expression := value
[+offs] feature-expression := value,
[+offs1,+offs2] feature-expression := value
```

The first case corresponds to the TAL reassignment formula $R([\mathsf{time}]\ \mathsf{feature\text{-}expression} \mathrel{\hat{=}} \mathsf{value})$, which assigns the given feature a new value (feature expression or value term) at the time specified by the temporal term time.

The second case corresponds to the TAL interval reassignment formula $R([\mathsf{time})\ \mathsf{feature\text{-}expression} \mathrel{\hat{=}} \mathsf{value}$, which assigns the given feature a specific value, and forces the feature to take on this value, throughout a temporal interval.

The final two cases are similar, but times are specified as offsets from the invocation time of the operator rather than as absolute timepoints.

2.7 Control Formulas

Given a uniquely identified initial state, any complete plan consisting of deterministic actions results in a single TAL interpretation, corresponding to the sequence of states that would be generated if the plan were to be executed. Control formulas are TAL formulas that provide explicit constraints on the state sequences that are allowed by the planner.

Control formulas can be used for several different but related purposes: To describe (physical) constraints that are inherent in the planning domain but are more difficult to describe using operator preconditions, to describe temporally extended goals that are not part of the domain itself but that the plan must achieve in order to be valid, or to constrain the search procedure so that redundant parts of the search space are not explored. Regardless of the purpose for which a control formula is used, it is still simply an arbitrary TAL formula which will be entailed by any plan returned by TALplanner.

Control formulas are common to all instances of a planning domain, and therefore form part of the domain specification.

This section mainly describes how control formulas are declared in TALplanner. See for example Kvarnström [11] for additional background information about control formulas and their use. See the domain definitions included in the TALplanner distribution for concrete examples.

Control formulas can be declared as follows:

- Using the statement `#control` ϕ in a domain definition. This declares a global control formula that must be satisfied by any plan for any problem instance in the given domain. The formula ϕ can be an arbitrary logic formula, but TALplanner performs considerably better with specific types of control formulas that are triggered by state transitions [10, 11].

  Global control formulas can be named using the syntax `#control :name "name"` ϕ. This can be used when tracing the planning process, allowing TALplanner to succinctly identify the reason for backtracking.

- Using a :control clause to declare operator-specific control formulas, as explained in Section 2.6.

- Using the statement `#precontrol :operator op(var1,...,varn) [:name "name"]` ϕ to declare an operator-specific control formula. This statement has the same effect as the :control clause and can be used if you want to keep control information separate from operator definitions.

TALplanner also supports control formulas using a modal syntax and progression, similar to that used in TLplan. This will be described in a future version of the manual.

2.8 Resources

TALplanner's support for resource constraints will be described in a future version of this manual.

2.9 Sequential vs. Concurrent Domains

TALplanner supports both sequential and concurrent plans. Since operators may have non-unit duration, each plan consists of a set of TAL *action occurrences*, each of which provides an explicit invocation time and end time for each action in the plan. For sequential plans, action execution intervals are disjoint. For concurrent plans, action execution intervals are permitted to overlap.

Generating concurrent plans usually requires additional domain information to enforce the necessary mutual exclusion. For example, one might have to ensure that one cannot travel from A to both B and C concurrently. Whether this is done is a property of the domain, and therefore each planning domain should be annotated as being either a sequential domain or a concurrent domain:

```
#sequential
#concurrent
```

By default, sequential planning is used.

Note that the semantics of TALplanner domain definitions is based on TAL rather than on PDDL, which has consequences for the modeling of mutual exclusion. For example, two actions are permitted to modify the same feature instance at the same time, as long as they assign the same new value.

2.10 Redundancy Checking

TALplanner supports cycle checking as well as extended forms of redundancy checking. Which level of redundancy checking should be used is a property of the domain, and can therefore be declared in the planning domain specification.

```
#redundandycheck n
```

Five levels of redundancy checking are supported:

- 0: No redundancy checking.
- 1: Immediate cycle checking. Backtracks if after applying an action ending at time t, the state at time t is identical to, or worse than, the state at time $t-2$. For example, in the blocks world, picking up a block from the table and immediately placing it back on the table would lead to this type of cycle. This is currently only useful for single-step actions.
- 2: Standard cycle checking. Backtracks if applying an action results in a new stable state which is equivalent to, or worse than, a stable state occurring earlier in the same plan candidate. A state is considered stable if the planner has reached the state in a way that allows it to remain at that state indefinitely. For example, actions can have multiple intermediate effect states before the final effect state that ends the execution of the action. Such intermediate effect states are not stable, since one cannot halt the action in the middle of its execution.
- 3: Global optimal redundancy checking. Backtracks if applying an action results in a new stable state which is equivalent to, or worse than, a stable state occurring earlier in the same plan candidate or some other plan candidate already visited in the search tree.
- 4 (default): Global non-optimal redundancy checking. Backtracks if applying an action results in a new stable state which is equivalent to, or worse than, a stable state occurring at any point in the same plan candidate or some other plan candidate already visited in the search tree.

A state s is considered "worse than" a state s' if the states agree on the values of all features and s' is preferred over s' in terms of resource usage, as defined in Section 2.8.

2.11 Assertions

Assertions are formulas that are guaranteed to follow from the domain definition and can be used to provide hints to TALplanner's formula optimizer. This may help TALplanner resolve conditions before even applying an action, which may in turn allow control formulas into preconditions, which results in considerable speedups. In the blocks world, for example, you might use the following assertions to help TALplanner optimize control formulas:

```
#assert forall t, block [ [t] ontable(block) -> !holding(block) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !clear(block2) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !ontable(block1) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !holding(block1) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !holding(block2) ]
#assert forall t, block1, block2 [ [t] holding(block2) -> !clear(block2) ]
#assert forall t, block1, block2 [ [t] holding(block2) -> !handempty ]
```

Though it would be possible to extract such knowledge through automated domain analysis, we have chosen to focus on other aspects of planning and therefore require the information to be specified explicitly.

2.12 Solvable and Unsolvable Instances

The following statements can be used to tell TALplanner that an instance is known to have a solution, or that it is known not to have a solution. After solving a set of problem instances, TALplanner will indicate whether any supposedly solvable problem instances could not be solved, or whether any supposedly unsolvable problem instances did have a solution, which can indicate errors in control rules or other parts of your domain model.

```
#solvable true
#solvable false
```

3 Running TALplanner

TALplanner is written in Java 6 (also known as Java 1.6) and requires a Java Runtime Environment, which can be downloaded from http://java.sun.com for most platforms.

The planner is generally started from the command line, as follows:

```
java -cp talplanner.jar talplanner.Planner
     -domain mydomain.tal -problems p1.tal p2.tal ...
```

There are a number of command line options that can be used to modify the behaviour of the planner, generate trace output, and so on. Some of these options are supported through standard command line options, similar to `-domain` and `-problems` above, and can be placed at any point after the Java class name talplanner.Plan. Other options are experimental and have not yet been integrated into the standard set of arguments, and are instead specified using the standard Java syntax -Dvar or -Dvar=value *before* the class name:

```
java -cp talplanner.jar -Dvar=value talplanner.Planner
     -domain mydomain.tal -problems p1.tal p2.tal ...
```

3.1 Controlling Output

The following options are used for controlling the output of TALplanner.

- `-version`: Show version information and exit.
- `-noversion`: Don't show version information (without this, TALplanner will automatically show version information before generating a plan).
- `-hideplan`: Don't show the generated plan. This may be useful for timing purposes, where the generated plan is not of interest.
- `-verbose` *n*: Determines the degree of detail used when tracing plan generation. Level 0 corresponds to no tracing. Level 3 shows one line of information for each action instance

that is applied, together with the reason for any backtracking that takes place. Levels 1, 2, 4 and 5 have not been used or tested recently.

- `-interval` ***n*** can be used together with `-verbose 0` to show a progress indicator every n times an action is applied.
- `-Dshowcontrolpruning` avoids integrating control rules into preconditions even when this is possible. This means that control rule violations are detected only after an action is applied. This is bad for performance but gives the user the opportunity to see the reason why an action is rejected, which is useful when debugging control rules. Should be used with a non-zero verbosity level.

A number of additional output options exist and will eventually be documented.

3.2 Controlling Search

By default, TALplanner uses depth first search in a search tree pruned by control formulas. Search can be limited using the following parameters.

- `-maxdepth` n determines the maximum depth to which the tree is searched.
- `-maxtime` n determines the maximum number of milliseconds to spend on a single plan.
- `-maxvisited` n determines the maximum number of search nodes that are visited, or equivalently, the maximum number of actions that are applied.
- `-maxaccepted` n determines the maximum number of search nodes corresponding to accepted actions.

The latter two options require an explanation.

Before an action is applied, TALplanner checks preconditions and any other conditions that can be moved into preconditions using domain analysis and formula optimization. If an action satisfies these conditions, it will be applied, which counts as visiting one search node.

Immediately after an action is applied, TALplanner may check a set of resource constraints, control formulas, redundancy constraints, and other constraints that could not be moved into preconditions. If the planner can determine that one of these constraints is violated, TALplanner will immediately discard the action and backtrack. Otherwise, the action is considered to be "accepted". Naturally, the planner might later determine that the accepted action leads to a dead end and backtrack over the action. In this case, the action is still considered to have been accepted.

Other search algorithms. TALplanner also supports A* search and branch-and-bound search, which will be documented in a future version of this manual.

References

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKTlplan.ps.

[2] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. TAL: Temporal Action Logics – language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2(3–4):273–306, September 1998.

[3] Patrick Doherty and Jonas Kvarnström. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In Claire Dixon and Michael Fisher, editors, *Proceedings of the 6th International Workshop on Temporal Representation and Reasoning (TIME)*, pages 47–54, Orlando, Florida, USA, May 1999. IEEE Computer Society Press.

[4] Patrick Doherty and Jonas Kvarnström. TALplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95–102, August 2001.

[5] Patrick Doherty and Jonas Kvarnström. *Handbook of Knowledge Representation*, chapter 16, Temporal Action Logics. Elsevier, December 2007.

[6] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 2009. Published and available online February 2009; print publication forthcoming.

[7] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the fourth international planning competition. Technical Report 195, Albert Ludwigs Universität, Institut für Informatik, Freiburg, Germany, January 2004. Available at http://www.mpi-sb.mpg.de/~hoffmann/publications.html.

[8] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, December 2003. Available at http://www.jair.org/contents/v20.html.

[9] Malik Ghallab, Adele E. Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela M. Veloso, Daniel S. Weld, and David Wilkins. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, USA, 1998.

[10] Jonas Kvarnström. Applying domain analysis techniques for domain-dependent control in TALplanner. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 101–110, Toulouse, France, April 2002. AAAI Press, Menlo Park, California, USA.

[11] Jonas Kvarnström. *TALplanner and Other Extensions to Temporal Action Logic*. PhD thesis, Linköpings universitet, April 2005. Linköping Studies in Science and Technology, Dissertation no. 937.

[12] Jonas Kvarnström and Patrick Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, June 2000.

[13] Jonas Kvarnström, Patrick Doherty, and Patrik Haslum. Extending TALplanner with concurrency and resources. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, volume 54 of *Frontiers in Artificial Intelligence and Applications*, pages 501–505, Berlin, Germany, August 2000. IOS Press, Amsterdam, The Netherlands.

[14] Jonas Kvarnström and Martin Magnusson. TALplanner in the Third International Planning Competition: Extensions and control rules. *Journal of Artificial Intelligence Research*, 20:343–377, December 2003.

[15] Jonas Kvarnström, Fredrik Heintz, and Patrick Doherty. A temporal logic-based planning and execution monitoring system. In J. Christopher Beck Eric Hansen Jussi Rintanen, Bernhard Nebel, editor, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, Sydney, Australia, September 2008.